Dan C. Marinescu

CLOUD COMPUTING

Theory and Practice

MORGAN KAUFMANN

Second Edition

Cloud Computing

Cloud Computing
Theory and Practice

Second Edition

Dan C. Marinescu

ELSEVIER

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-812810-7

Publisher: Katey Birtcher
Acquisition Editor: Steve Merken
Developmental Editor: Nate McFadden
Production Project Manager: Punithavathy Govindaradjane
Designer: Christian J. Bilbow

Typeset by VTeX

To Vera Rae and Luke Bell

Foreword

Everyone who uses computers these days has heard of cloud computing, and most of us utilize the cloud in some fashion, whether it is for highly-sophisticated calculations for science and engineering or just to store the family's photographs safely. What is a computational cloud? Why is it so ubiquitous? Who provides cloud computing services? What services are available? Why is it cost-effective? How are computational clouds built and programmed? How are they accessed? How do they store and process huge collections of information? How can they be made to be secure? This book answers these questions and teaches you about the design, implementation, use, and advantages of cloud computing. The book is the definitive textbook and reference on cloud computing, written for researchers and educators from academia, industry, and government.

I have known the author of this book, Dr. Dan Marinescu, for over 30 years. Based on many years of working on research problems in the computer science and computer engineering with Dan, I can state with certainty that Dan is a scholar, an intellectual, a dedicated researcher, and a diligent author. As people who want to learn about cloud computing, we are fortunate that Dan has applied his intellect and energy toward compiling into one coordinated volume extensive knowledge of numerous aspects of cloud computing.

The text of the book is written in a straightforward and understandable way, drawing from hundreds of sources. The material and jargon from these sources are organized and presented in a common terminology, context, and framework to make them coherent.

This book is a comprehensive encyclopedia for cloud computing. In addition to the topics mentioned above, here is a sampling of other topics readers will learn about from this book, all in the context of cloud computing: application development, big data, containers, control theory for optimizing system use, data storage, data streaming, deadlock prevention, energy efficiency, graphics processing units, hypervisors, interclouds, interconnection networks, internet communications, MapReduce programming, mobile computing, network-centric computing and network-centric content, parallelism (data-level, thread-level, and task-level), performance analysis, process coordination, resource management and scheduling, security, service level agreements (SLAs), trust, virtualization, and warehouse-scale computers. Also in the book are surveys and comparisons of existing cloud computing hardware and software systems.

Each chapter is followed by a set of thought-provoking exercises and problems. Many of these problems refer the reader to specific other references for additional information. The problems guide students to apply and build on the knowledge in the book to explore other systems and expand and deepen their understanding. Furthermore, there is an appendix with research projects for students in a cloud computing class.

There is an extensive bibliography of over 500 publications. These are cited under each of the topics in the book and leads readers to more details. The references are often also used in the problems and exercise to help the reader learn more.

The book is well-organized, consisting of three main sections, each with multiple chapters. The book has great technical depth, addressing both underlying theoretical concepts and practical real-world issues. Even though the book is technically deep, the writing style and organization make it easy to follow and understand. There are over 180 figures, with meaningful lengthy captions, that are constructed in a way that is clear and comprehensible to support the information presented in the text.

Cloud computing infrastructures are large-scale complex systems that need to be designed, analyzed, deployed, and made secure in a manner that supports users' needs and provides a financial profit to the cloud services providers. These are very difficult problems, and approaches to handling these problems are presented throughout the book.

In conclusion, Dr. Dan Marinescu has applied his vast experience as an author of books and as an outstanding researcher to successfully take an extensive and complicated body of knowledge and present it in an organized, informative, and understandable way. This book is an authoritative textbook and reference on cloud computing for researchers, practitioners, system designers and implementers, and application specialists using cloud computing.

H.J. Siegel
Fellow of the IEEE
Fellow of the ACM
Professor Emeritus at Colorado State University, where he was
the George T. Abell Endowed Chair Distinguished Professor of Electrical
and Computer Engineering, Professor of Computer Science, and
Director of the university-wide Information Science and Technology Center

Preface to the Second Edition

Almost half a century after the dawn of the computing era, an eternity in the age of silicon, the disruptive multicore technology forced the computational science community and the application developers to realize the need to understand and exploit concurrency. There is no point now to wait for faster clock rates, we better design algorithms and applications able to use the multiple cores of a modern processor.

The thinking changed again when cloud computing showed that there are new applications that can effortlessly exploit concurrency and, in the process, generate huge revenues. A new era in parallel and distributed systems began, the time of Big Data hiding nuggets of useful information and requiring massive amounts of computing resources. In this era "coarse" is good and "fine" is not good, at least as far as the granularity of parallelism is concerned. The new challenge is harnessing the power of millions of multicore processors and allowing them to work in concert effectively.

The pace of technological developments in computing and information processing is truly breathtaking and often exceeds the expectations and the predictions of the most optimistic experts and forecasters. For example, in early 1990s the SBSS (Science-Based Stockpile Stewardship) program of the DOE to transition from underground testing of the nuclear stockpile to science-based, computer-driven testing, required an increase of supercomputer speed by a factor of 10 000 over a period of ten years. The goal of 100 Tflops was exceeded by a factor of 20 [417].

The last decades have reinforced the idea that information processing can be done more efficiently on large farms of computing and storage systems accessible via the Internet. Advancements in networking, processor architecture, storage technology, and software technology are responsible for the acceptance of new computing models.

In early 1990s, the Grid computing movement initiated by US National Laboratories and universities for the benefit of the world-wide scientific community, captivated the attention of scholars and funding agencies. Then, a decade later, the cloud computing era targeting enterprise applications began.

In 2006 Amazon introduced Amazon Web Services (AWS). The first cloud computing services offered were EC2 (Elastic Cloud Computing) and S3 (Simple Storage Service). Today, S3 has surpassed two trillion objects and, routinely, runs more than 1.1 million peak requests per second; its year-over-year gross rate is 132% [232]. Elastic MapReduce launched 5.5 million clusters since the start of the service in May 2013.

AWS has more than one million customers who have access to 28+ data centers; one data center is powered by 50 000–80 000 servers, has a network capacity of 10^2 Tbps and uses 25–30 megawatts of energy [220]. In 2015 Amazon had the largest cloud infrastructure. Fourteen other cloud providers combined have 1/5th of the aggregate capacity of AWS [232].

An unofficial estimate puts the number of servers used by Google in January 2012 close to 1.8 millions. Today there are more than 200 Cloud Service Providers (CSPs) and some 120 of them support the IaaS (Infrastructure as a Service) and DBaaS (Database as a Service) cloud delivery models. While in the past it took years for an IT company to reach one million customers, it took only weeks to Instagram to reach this milestone.

Cloud computing promoted by IT companies such as Amazon, Google, Microsoft, IBM, Oracle, and others has effectively democratized computing. In 2015, 2.6 billion out of the 7.2 billion inhabitants of the planet Earth use Email services[1] such as Gmail. Hundreds of millions use online services to buy all imaginable goods, to rent apartments in far away places, or to find a raid home. Millions of computer experts along with neophytes need only a credit card to access computer resources previously offered by the supercomputers operated by government institutions and only available to a select few.

Computer clouds ushered us into the Big Data age. According to an IBM post: “Every day, we create 2.5 quintillion bytes of data, so much that 90% of the data in the world today has been created in the last two years alone. This data comes from everywhere: sensors used to gather climate information, posts to social media sites, digital pictures and videos, purchase transaction records, and cell phone GPS signals to name a few. This data is Big Data” [251].

The complexity of the software and the hardware infrastructure supporting cloud service is astounding. Two billion lines of code are maintained by Google and drive applications such as Google Search, Google Maps, Google Docs, Google+, Google Calendar, Gmail, YouTube, and every other Google Internet service. By comparison Windows operating system developed by Microsoft since 1980’s has some 50 million lines of code, 40 times less than what Google has developed in its 19 years of existence.[2]

Warehouse-scale computers (WSCs) with tens of thousands processors are no longer a fiction, but serve millions of users, and are analyzed in computer architecture textbooks [56,228] and in recent research papers such as [262]. WSC’s processor throughput is more important than single-threaded peak performance, because no single processor can handle the full workload of modern applications [239]. As the number of parallel threads increases, reducing serialization and communication overheads is increasingly more difficult thus, *brawny-core* systems, whose single-core performance is fairly high, are preferable to more energy efficient *wimpy-core* ones.

In the early days of network-centric computing it was postulated that web searching is the “killer application” that will drive the software and hardware of large-scale systems for the next decades [54]. It turns out that the applications running on computer clouds are very diverse. For example, at Google the top 50% most frequently used applications account for only 50% of the CPU cycles used [262].

The broad spectrum of cloud applications adds to the challenges faced by cloud infrastructure. For example, controlling tail latency of workloads consisting of a mix of time-critical and batch jobs is far from trivial [131]. Some critical system requirements are contradictory, e.g., multiplexing resources to increase efficiency and lowering the response time, while supporting performance and security isolation.

The effort to build one layer of abstraction removed from the underlaying hardware, a sort of operating system for Internet-scale jobs is underway. Systems such as Dryad [253], DryadLinq [539], Mesos [237], Borg [502], Omega [446], and Kubernets [82] attempt to bridge the gap between a clustered infrastructure, and the assumptions made by applications about their environments. These systems manage a virtual computer aggregating resources of a physical cluster with a very large number of independent servers.

The software stack for cloud computing has evolved in the quest to provide a unified higher-level view of the system, rather than a large collection of individual machines. Virtualization and

[1] See http://www.radicati.com/wp/wp-content/uploads/2015/02/Email-Statistics-Report-2015-2019-Executive-Summary.pdf.
[2] Google was formally incorporated in September 1998.

containerization are ubiquitous abstractions that allow easier access to the increasingly larger and diverse population of cloud users. Distributed and semi-structured storage systems such as Google's BigTable [96] or Amazon's Dynamo [134] are widely used. The pleiad of systems supporting higher level abstractions include FiumeJava [92], Mesa [212], Pig [187], Spark [541], Spark Streaming [543], Tachyon [301], and others.

Cloud computing had and will continue to have a profound influence not only on the large number of individuals and institutions who are now empowered to process huge amounts of data, but also on the research community. Computer clouds operate in an environment characterized by the *variability of everything* and by *conflicting requirements*. Such disruptive qualities of computer clouds ultimately demand a new thinking in system design.

The scale of clouds amplifies unanticipated benefits, as well as the nightmares of system designers. Even a slight improvement of server performance and/or of the algorithms for resource management could lead to huge cost savings and rave reviews. At the same time, the failure of one of the millions of hardware and software components can be amplified, propagate throughout the entire system and have catastrophic consequences. When engineering large scale systems an important lesson is to prepare for the unexpected, as low probability events occur and can cause major disruptions.

The very fast pace of developments in cloud computing in the last few years are reflected in the second edition of this book, a major revision of the first edition. We attempt to sift through the large volume of information and present the main ideas related to cloud computing. Chapter 1 is an informal introduction to computer clouds, network-centric computing and network-centric content, to the entities involved in cloud computing, the paradigms and the services, and the ethical issues. Chapter 2 gives an overview of services offered by the Big Three CSPs (Cloud Service Providers), Amazon, Google, and Microsoft and of responsibility sharing between a CSP and the cloud users. The rest of the material is organized in four sections.

Section I introduces important theoretical and practical concepts related to parallel and distributed computing. Chapter 3 presents subjects ranging from computational models, the global state of a process group to causal history, atomic actions, concurrency, modeling concurrency with Petri nets, atomic actions, consensus protocols, load balancing and consensus protocols. Chapter 4 covers data, thread-level, task-level parallelism, parallel computer architecture and distributed systems, an introduction to virtualization, and a discussion on how to address the complexity of modern system through modularization, layering, and hierarchical organization.

Section II presents two critical elements of the cloud infrastructure. Chapter 5 is dedicated to communication and cloud access and presents the network organization, the cloud networking infrastructure, named data networks, software defined networks, interconnection networks such as InfiniBand and Myrinet, storage area networks, scalable data center communication architectures, content delivery networks, and vehicular ah-hoc networks. Chapter 6 covers storage models, file systems, NoSQL databases, a locking service, Google's Bigtable and Megastore, storage reliability at scale, and database services.

Section III covers cloud applications and cloud resource management and scheduling and includes Chapters 7, 8, 9, and 10. After a brief review of workflows Chapter 7 analyzes coordination using ZooKeeper, the MapReduce programming model, the frameworks supporting processing of large data sets in a distributed computing environment including Hadoop, Hive, Yarn, Tez, Pig, and Impala, followed by the applications of cloud computing in science and engineering, biology research, and social computing. Chapter 8 discusses cloud infrastructure including Warehouse Scale Computers (WSC) and WSC performance, the components of the software stack, cloud resource management, execution engines for coarse-grain data-parallel applications, in-memory cluster computing for Big Data, and containerization software including Docker and Kubernetes. Chapter 9 is dedicated to resource management and scheduling. A utility model for cloud-based web services, the applications of control theory to scheduling, two-level resource allocation strategies, and coordination of multiple autonomic performance mangers, delay scheduling, data-aware scheduling, several scheduling algorithms including start-time fair queuing and borrowed virtual time are some of the topics covered in this chapter. Chapter 10 covers resource virtualization including performance and security isolation, hardware support for virtualization, analysis of widely used hypervisors Xen and KVM, nested virtualization, and the performance penalties and the risks associated with virtualization.

Section IV presents research topics in cloud computing and includes of Chapters 11, 12, and 13. Chapter 11 is dedicated to cloud security; after a general discussion of cloud security risks, privacy, and trust it analyzes the security of virtualization and the security risks posed by shared images and by the management operating system. The implementation of a hypervisor based on micro kernel design principles and a trusted virtual machine monitor are then presented. Chapter 12 is focused on the challenges posed by Big Data, data streaming, and mobile applications. The analysis of the Big Data revolution is followed by a presentation of MapReduce successors including Pig, Hive and Impala. OLTP (Online Transaction Processing) databases and in-core databased are then presented. Mobile computing applications, the energy consumption of mobile applications, and the limitation of mobile cloud computing are analyzed. Chapter 13 is dedicated to more advanced topics such as the impact of scale on efficiency, cloud scheduling subject to deadlines, self-organization, and combinatorial auctions for cloud resources.

Two appendices provide information useful to users who plan to use AWS services and to students enrolled in cloud computing classes. Appendix 1 covers cloud application development and Appendix 2 presents several cloud projects in large-scale simulations and cloud services. Applications to system design when multiple design alternatives are evaluated concurrently and Big Data applications in computational sciences are also presented.

The history notes at the end of several chapter present the milestones for the science and the technology discussed in the chapter. These history notes serve as reminders of how important concepts, now considered classical, have been developed in the short time since the cloud computing era began. They also show the impact of technological advances and the radical changes they have triggered in our society and in our thinking.

Some 550 references are cited in the text. Many references present recent research results in several areas related to cloud computing, others are classical references on major topics in parallel and distributed systems. A glossary covers important concepts and terms used in the text. A list of abbreviations is also provided.

The author is grateful to many colleagues and collaborators who have shared their wisdom and knowledge with him along the years. Special thanks to Professors H. J. Siegel from Colorado State University, John Patrick Morrison from University College Cork in Ireland, and Stephan Olariu from the Old Dominion University. We are grateful to Professor Stephan Olariu and to Gabriela Marinescu for commenting on some 600 pages of text. Many thanks to Nate McFadden and Steve Merken from Elsevier for guidance and help.

Abbreviations

ABI	Application Binary Interface
ACID	Atomicity, Consistency, Isolation, Durability
ACL	Access Control List
API	Application Program Interface
AMI	Amazon Machine Image
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
AVX	Advanced Vector Extensions
AWS	Amazon Web Services
AWSLA	Amazon Web Services Licensing Agreement
BASE	Basically Available, Soft state, Eventually consistent
BCE	Basic Core Equivalent
BIOS	Basic Input Output System
BPD	Bootstrap Performance Diagnostic
BSP	Bulk Synchronous Parallel
CCN	Content Centric Network
CDN	Content Delivery Network
CFS	Colossus File System
CISC	Complex Instruction Set Computer
COMA	Cache Only Memory Access
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CPI	Cycles per Instruction
CRM	Custom Relation Management
CSP	Cloud Service Provider
CUDA	Compute Unified Device Architecture
DAT	Dynamic Address Translation
DBaaS	Database as a Service
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
EBS	Elastic Block Store
EC2	Elastic Cloud Computing
ECS	EC2 Container Service
EMR	Elastic Map Reduce
EPIC	Explicitly Parallel Instruction Computing
FC	Fiber Channel
FCFS	First Come First Serve
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GFS	Google File System
GiB	Gibi Byte
GIMP	GNU Manipulation Program
GPFS	General Parallel File System

GPU	Graphics Processing Unit
HDD	Hard Disk Drive
HDFS	Hadoop File System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Drive Electronics
IoT	Internet of Things
IPC	Instructions per Clock Cycle
IRTF	Internet Research Task Force
ISA	Instruction Set Architecture
JDBC	Java Database Connectivity
JMS	Java Message Service
JSON	Javascript Object Notation
KVM	Kernel-based Virtual Machine
MAC	Medium Access Control
MFLOPS	Million Floating Point Operations per Second
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions per Second
MLMM	Multi-level Memory Manager
MMX	Multi Media Extension
MPI	Message Passing Interface
MSCR	Map Shuffle Combine Reduce
NAT	Network Address Translation
NDN	Named Data Networks
NFS	Network File System
NTFS	New Technology File System
NUMA	Non-Uniform Memory Access
NV-RAM	Non-Volatile Random Access Memory
OCCI	Open Cloud Computing Interface
OGF	Open Grid Forum
OLTP	On Line Transaction Processing
OLAP	On Line Analytical Processing
PaaS	Platform as a Service
PHP	recursive acronym for PHP: Hypertext Preprocessor
PN	Petri Net
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RAR	Read After Read
RAW	Read After Write
RDD	Resilient Distributed Dataset
RDS	Relational Database Service
RDBMS	Relational Database Management System
REST	Representational State Transfer
RFC	Remote Frame Buffer
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RTT	Round Trip Time
SaaS	Software as a Service
SAN	Storage Area Network
SDK	Software Development Kit

SDN	Software Defined Network
SHA	Secure Hash Algorithm
SLA	Service Level Agreement
SIMD	Single Instruction Multiple Data
SNS	Simple Notification Service
SOAP	Simple Object Access Protocol
SPMD	Same Program Multiple Data
SQL	Structured Query Language
SQS	Simple Queue Service
SSD	Solid State Disk
SSE	Streaming SIMD Extensions
S3	Simple Storage System
SWF	Simple Workflow Service
TCP	Transport Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
UFS	Unix File System
UMA	Uniform Memory Access
vCPU	Virtual CPU
VLIW	Very Long Instruction Word
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMM	Virtual Machine Monitor
VMM	Virtual Memory Manager
VNC	Virtual Network Computing
VPC	Virtual Private Cloud
VPN	Virtual Private Network
WAN	Wide Area Network
WAW	Write After Write
WAR	Write After Read
WSC	Warehouse Scale Computer
WWN	World Wide Name
XML	Extensible Markup Language
YARN	Yet Another Resource Negotiator

CHAPTER

INTRODUCTION

1

Conceptually, computing can be viewed as another utility, like electricity, water, or gas, accessible to every household in many countries of the world. Computer clouds are the utilities providing computing services. In utility computing the hardware and the software resources are concentrated in large data centers. The users of computing services pay as they consume computing, storage, and communication resources. While utility computing often requires a cloud-like infrastructure, the focus of cloud computing is on the business model for providing computing services.

More than half a century ago, at the centennial anniversary of MIT, John McCarthy, the 1971 Turing Award recipient for his work in Artificial Intelligence, prophetically stated: “If computers of the type I have advocated become the computers of the future, then computing may someday be organized as a public utility, just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry.” The prediction of McCarthy is now a technological and social reality.

Cloud computing is a disruptive computing paradigm and, as such, it required major changes in many areas of computer science and computer engineering including data storage, computer architecture, networking, resource management, scheduling, and last but not least, computer security. The chapters of this book cover the most significant challenges posed by the scale of the cloud infrastructure and the very large population of cloud users with diverse applications and requirements.

The Internet made cloud computing possible; we could not even dream of using computing and storage resources from distant data centers without fast communication. The evolution of cloud computing is organically tied to the future of the Internet. The Internet of Things (IoT) has already planted some of its early seeds in computer clouds. For example, Amazon already offers services such as Lambda and Kinesis discussed in Section 2.4.

The number of Internet users has increased tenfold from 1999 to 2013; the first billion was reached in 2005, the second in 2010, and the third in 2014. This number is even larger now, see Figure 1.1. Many Internet users have discovered the appeal of cloud computing either directly or indirectly through a variety of services, without knowing the role the clouds play in their life. In the years to come the vast computational resources provided by the cloud infrastructure will be used for the design and engineering of complex systems, scientific discovery, education, business, analytics, art, and virtually all other aspects of human endeavor. Exabytes of data stored in the clouds are streamed, downloaded, and accessed by millions of cloud users.

This chapter introduces basic cloud computing concepts in Section 1.1. The broader context of network-centric computing and network-centric content is discussed in Section 1.2. Why cloud computing became a reality in the last years after a long struggle to design large-scale distributed systems and computational grids? This question is addressed in Section 1.3, while Section 1.4 coveres the defining attributes of computer clouds and the cloud delivery models. Ethical issues and cloud vulnerability are discussed in Sections 1.5 and 1.6, respectively.

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00001-7

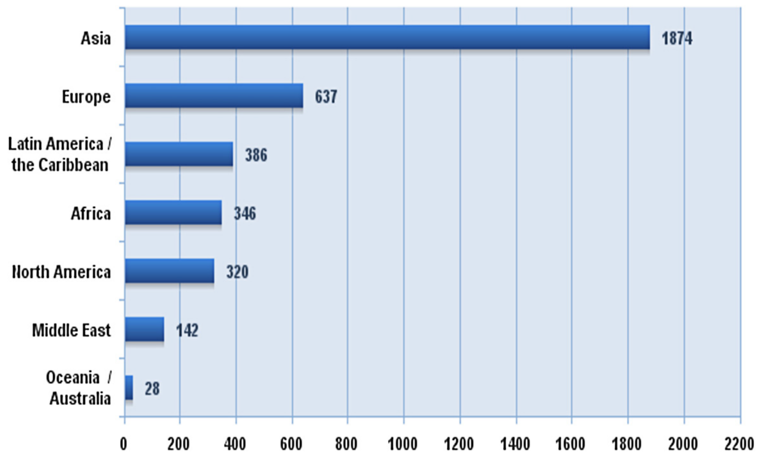


FIGURE 1.1

The number of Internet users in different regions of the world as of March 25, 2017 (in millions), according to http://www.internetworldstats.com/stats.htm.

1.1 CLOUD COMPUTING

In 2011, NIST, the US National Institute of Standards and Technology, defined cloud computing as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

Cloud computing is characterized by five attributes: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. DBaaS – database as a service is a more recent addition to the three cloud service delivery models: SaaS – software as a service, PaaS – platform as a service, and IaaS – infrastructure as a service. Private cloud, community cloud, public cloud, and hybrid cloud are the four deployment models shown in Figure 1.2, Section 1.4.

Cloud computing era started in 2006 when Amazon offered the Elastic Cloud Computing (EC2) and the Simple Storage Service (S3), the first services provided by Amazon Web Services (AWS). Five years later, in 2012, EC2 was used by businesses in 200 countries. S3 has surpassed two trillion objects and routinely runs more than 1.1 million peak requests per second. The Elastic MapReduce has launched 5.5 million clusters since the start of the service in May 2010 (ZDNet 2013). The range of services offered by Cloud Service Providers (CSPs), and the number of cloud users have increased dramatically during the last few years.

The cloud computing movement is motivated by the idea that data processing and storage can be done more efficiently on large farms of computing and storage systems accessible via the Internet. Computer clouds support a paradigm shift from local to network-centric computing and network-centric content where distant data centers provide the computing and storage resources. In this new paradigm users relinquish control of their data and code to Cloud Service Providers.

Cloud computing offers scalable and elastic computing and storage services. The resources used for these services can be metered and the users can be charged only for the resources they used. Cloud computing is a business reality as a large number of organizations have adopted this paradigm.

Cloud computing is cost-effective because of resource multiplexing. Application data is stored closer to the site where it is used in a manner that is device and location-independent; potentially, this data storage strategy increases reliability, as well as security. The maintenance and the security are ensured by service providers. Organizations using computer clouds are relieved of supporting large IT teams, acquiring and maintaining costly hardware and software, and paying large electricity bills. CSPs can operate more efficiently due to economy of scale.

Data analytics, data mining, computational financing, scientific and engineering applications, gaming and social networking, as well as other computational and data-intensive activities benefit from cloud computing. Storing information on the cloud has significant advantages. Content previously confined to personal devices such as workstations, laptops, tablets, and smart phones need no longer be stored locally. Data stored on computer clouds can be shared among all these devices and it is accessible whenever a device is connected to the Internet. For example, in 2011 Apple announced the *iCloud,* a network-centric alternative for content including music, videos, movies, and personal information. In February 2017 iCloud had 782 million subscribers according to http://appleinsider.com/.

Cloud computing represents a dramatic shift in the design of systems capable of providing vast amounts of computing cycles and storage space. Computer clouds use off-the shelf, low-cost components. During the previous four decades powerful, one-of-a-kind supercomputers, were built at a high cost, with the most advanced components available at the time.

In early 1990s Gordon Bell argued that one-of-a-kind systems are not only expensive to build, but the cost of rewriting applications for them is prohibitive. He anticipated that sooner or later massively parallel computing will evolve into computing for the masses [59].

There are virtually no bounds on composition of digital systems controlled by software, so we are tempted to build increasingly more complex systems including systems of systems [335]. The behavior and the properties of such systems are not always well understood. We should not be surprised that computing clouds will occasionally exhibit an unexpected behavior and large-scale systems will occasionally fail.

The architecture, the coordination mechanisms, the design methodology, and the analysis techniques for large-scale complex systems such as computing clouds will evolve in response to changes in technology, the environment, and the social impact of cloud computing. Some of these changes will reflect changes in communication, in the Internet itself in terms of speed, reliability, security, capacity to accommodate a larger addressing space by migration to IPv6, and so on.

Cloud computing reinforces the idea that computing and communication are deeply intertwined. Advances in one field are critical for the other. Indeed, cloud computing could not emerge as a feasible alternative to the traditional paradigms for high-performance computing before the Internet was able to support high-bandwidth, low-latency, reliable, low-cost communication. At the same time, modern networks could not function without powerful computing systems to manage the network. High performance switches are critical elements of both networks and computer clouds.

The complexity of the cloud computing infrastructure is unquestionable and raises questions such as: How can we manage such systems? Do we have to consider radically new ideas, such as self-management and self-repair for future clouds consisting of millions of servers? Should we migrate from a strictly deterministic view of such complex systems to a non-deterministic one? Answers to these questions provide a rich set of research topics for the computer science and engineering community.

The cloud movement is not without skeptics and critics. The critics argue that cloud computing is just a marketing ploy, that users may become dependent on proprietary systems, that the failure of a large system such as the cloud could have significant consequences for a very large group of users who depend on the cloud for their computing and storage needs. Security and privacy are major concerns for cloud computing users.

A very important question is if under pressure from the user community the current standardization efforts will succeed. The alternative, the continuing dominance of proprietary cloud computing environments is likely to have a negative impact on the field. The cloud delivery models, SaaS, PaaS, IaaS, together with DBaaS discussed in depth in Chapter 2 will continue to coexist for the foreseeable future.

Services based on SaaS will probably be increasingly more popular because they are more accessible to lay people, while services based on the IaaS will be the domain of computer savvy individuals, large organizations, and the government. If the standardization effort succeeds, then we may see IaaS designed to migrate from one infrastructure to another and overcome the concerns related to vendor lock-in. The popularity of DBaaS services is likely to grow.

1.2 NETWORK-CENTRIC COMPUTING AND NETWORK-CENTRIC CONTENT

Network-centric computing and network-centric content concepts reflect the fact that data processing and data storage takes place on remote computer systems accessed via the ubiquitous Internet, rather than locally. The term *content* refers to any type or volume of media, be it static or dynamic, monolithic or modular, live or stored, produced by aggregation, or mixed.

The two network-centric paradigms share a number of characteristics:

- Most network-centric applications are data intensive. For example, data analytics allow enterprises to optimize their operations; computer simulation is a powerful tool for scientific research in virtually all areas of science from physics, biology, and chemistry, to archeology. Sophisticated tools for computer-aided design such as Catia (Computer Aided Three-dimensional Interactive Application) are widely used in aerospace and automotive industries. The widespread use of sensors generate a large volume of data. Multimedia applications are increasingly more popular; the larger footprint of the media data increases the load placed on storage, networking, and processing systems.
- Virtually all applications are network-intensive. Transferring large volumes of data requires high-bandwidth networks. Parallel computing, computation steering, and data streaming are examples of applications that can only run efficiently on low latency networks. Computation steering in numerical simulation means to interactively guide a computational experiment towards a region of interest.
- Computing and communication resources (CPU cycles, storage, network bandwidth) are shared and resources can be aggregated to support data-intensive applications. Multiplexing leads to a higher resource utilization; indeed, when multiple applications share a system their peak demands for resources are not synchronized and the average system utilization increases.
- Data sharing facilitates collaborative activities. Indeed, many applications in science, engineering, as well as industrial, financial, governmental applications require multiple types of analysis

of shared data sets and multiple decisions carried out by groups scattered around the globe. Open software development sites are another example of such collaborative activities.
- The systems are accessed using *thin clients* running on systems with limited resources. In June 2011 Google released Google Chrome OS designed to run on primitive devices and based on the browser with the same name.
- The infrastructure supports some form of *workflow management*. Indeed, complex computational tasks require coordination of several applications; composition of services is a basic tenet of Web 2.0.

There are sources of concern and benefits of the paradigm shift from local to network-centric data processing and storage:

- The management of large pools of resources poses new challenges as such systems are vulnerable to malicious attacks that can affect a large user population.
- Large-scale systems are affected by phenomena characteristic to complex systems such as phase transitions when a relatively small change of environment could lead to an undesirable system state [328]. Alternative resource management strategies, such as self-organization, and decisions based on approximate knowledge of the system state must be considered.
- Ensuring Quality of Service (QoS) guarantees is extremely challenging in such environments, as total performance isolation is elusive.
- Data sharing poses not only security and privacy challenges but also requires mechanisms for access control for authorized users and for detailed logs of the history of data changes.
- Cost reduction. Concentration of resources creates the opportunity to pay-as-you-go for computing and thus, eliminates the initial investment and reduces significantly the maintenance and operation costs of the local computing infrastructure.
- User convenience and elasticity, the ability to accommodate workloads with very large peak-to-average ratios.

The creation and consumption of audio and visual content is likely to transform the Internet. It is expected that the Internet will support increased quality in terms of resolution, frame rate, color depth, stereoscopic information. It seems reasonable to assume that the Future Internet[1] will be *content-centric*. *Information* is the result of functions applied to content.

The content should be treated as having meaningful semantic connotations rather than a string of bytes; the focus will be on the information that can be extracted by content mining when users request named data and content providers publish data objects. Content-centric routing will allow users to fetch the desired data from the most suitable location in terms of network latency or download time. There are also some challenges, such as providing secure services for content manipulation, ensuring global rights-management, control over unsuitable content, and reputation management.

[1]The term “Future Internet” is a generic concept referring to all research and development activities involved in development of new architectures and protocols for the Internet.

1.3 CLOUD COMPUTING, AN OLD IDEA WHOSE TIME HAS COME

It is hard to point out a single technological or architectural development that triggered the movement towards network-centric computing and network-centric content. This movement is the result of a cumulative effect of developments in microprocessor, storage, and networking technologies coupled with architectural advancements in all these areas and last but not least, with advances in software systems, tools, programming languages and algorithms supporting distributed and parallel computing.

Along the years we have witnessed the breathtaking evolution of solid state technologies which led to the development of multicore processors. Quad-core processors such as AMD Phenom II X4, Intel i3, i5, and i7, and hexacore processors such as AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X are now used to build the servers populating computer clouds. The proximity of multiple cores on the same die allows cache coherency circuitry to operate at a much higher clock rate than it would be possible if signals were to travel off-chip.

Storage technology has also evolved dramatically. For example, solid state disks such as RamSan-440 allow systems to manage very high transaction volumes and larger numbers of concurrent users. RamSan-440 uses DDR2 (double-data-rate) RAM to deliver 600 000 sustained random IOPS (Input/output operations per second) and over 4 GB/second of sustained random read or write bandwidth, with latency of less than 15 microseconds and it is available in 256 GB and 512 GB configurations. The price of memory has dropped significantly; at the time of this writing the price of 1 GB module for a PC is around $5. Optical storage technologies and flash memories are widely used nowadays.

The thinking in software engineering has also evolved and new models have emerged. A software architecture and a software design pattern, the *three-tier model* has emerged. Its components are:

1. *Presentation tier,* the topmost level of the application. Typically, it runs on a desktop, PC, or workstation, uses a standard graphical user interface (GUI), and displays information related to services e.g., browsing merchandize, purchasing, and shopping cart contents. The presentation tier communicates with other tiers.
2. *Application/logic tier* controls the functionality of an application and may consist of one or more separate modules running on a workstation or application server. It may be multi-tiered itself and then the architecture is called an *n-tier architecture*.
3. *Data tier* controls the servers where the information is stored; it runs a relational database management system on a database server or a mainframe and contains the computer data storage logic. The data tier keeps data independent from application servers or processing logic and improves scalability and performance.

Any tier can be replaced independently; for example, a change of operating system in the presentation tier would only affect the user interface code.

Once the technological elements were in place it was only a matter of time until the economical advantages of cloud computing became apparent. Due to the economy of scale large data centers, centers with more than 50 000 systems, are more economical to operate than medium size centers which have around 1 000 systems. Large data centers equipped with commodity computers experience a five to seven times decrease of resource consumption, including energy, compared to medium size data centers [37].

The networking costs, in dollars per Mbit/sec/month, are $95/13 = 7.1$ larger for medium size data centers. The storage costs, in dollars per GB/month, are $2.2/0.4 = 5.7$ larger for medium size centers.

Medium size data centers have a larger administrative overhead, one system administrator for 140 systems versus one for 1 000 systems for large centers.

Data centers are very large consumers of electric energy used to keep the servers and the networking infrastructure running and heating and cooling the data centers. In 2006 the data centers reportedly consumed 61 billion kWh, 1.5% of all electric energy in the U.S., at a cost of $4.5 billion. We have seen a 4% increase in total data center energy consumption from 2010 to 2014.

In 2014 there were about three million data centers in the U.S. The data centers consumed about 70 billion kWh, representing about 2% of the total energy consumption in the U.S. This is the equivalent of the electric energy consumed by about 6.4 million average homes in the U.S. that year. The energy costs differ from state to state, e.g., one kWh costs 3.6 cents in Idaho, 10 cents in California, and 18 cents in Hawaii. This explains why cloud data centers are placed in regions with lower energy cost.

A natural question to ask is: Why cloud computing could be successful when other paradigms have failed? The reasons why cloud computing could be successful can be grouped in several general categories: technological advances, a realistic system model, user convenience, and financial advantages. A non-exhaustive list of reasons for the success of cloud computing includes:

- Cloud computing is in a better position to exploit recent advances in software, networking, storage, and processor technologies. Cloud computing is promoted by large IT companies where these new technological developments take place and these companies have a vested interest to promote the new technologies.
- A cloud consists of a mostly homogeneous set of hardware and software resources in a single administrative domain. In this setup security, resource management, fault-tolerance, and quality of service are less challenging than in a heterogeneous environment with resources in multiple administrative domains.
- Cloud computing is focused on enterprise computing [160,164]; its adoption by industrial organizations, financial institutions, healthcare organizations and so on, has a potentially huge impact on the economy.
- A cloud provides the illusion of infinite computing resources; its elasticity frees the applications designers from the confinement of a single system.
- A cloud eliminates the need for up-front financial commitment and it is based on a pay-as-you-go approach; this has the potential to attract new applications and new users for existing applications fomenting a new era of industry-wide technological advancements.

In spite of the technological breakthroughs that have made cloud computing feasible, there are still major obstacles for this new technology; these obstacles provide opportunity for research. We list a few of the most obvious obstacles:

- Availability of service; what happens when the service provider cannot deliver? Can a large company such as GM move its IT activities to the cloud and have assurances that its activity will not be negatively affected by cloud overload? A partial answer to this question is provided by Service Level Agreements (SLA)s discussed in Section 2.9. A temporary fix but with negative economical implications is *overprovisioning*, i.e., having enough resources to satisfy the largest projected demand.

- Vendor lock-in; once a customer is hooked to one cloud service provider it is hard to move to another. The standardization efforts at NIST attempt to address this problem.
- Data confidentiality and auditability; this is indeed a serious problem analyzed in Chapter 11.
- Data transfer bottlenecks critical for data-intensive applications. Transferring 1 TB of data on a 1 Mbps network takes 8 000 000 seconds or about 10 days; it is faster and cheaper to use courier service and send data recoded on some media than to send it over the network. Very high speed networks will alleviate this problem in the future, e.g., a 1 Gbps network would reduce this time to 8 000 seconds, or slightly more than 2 hours.
- Performance unpredictability; this is one of the consequences of resource sharing. Strategies for performance isolation are discussed in Section 10.1.
- Elasticity, the ability to scale up and down quickly. New algorithms for controlling resource allocation and workload placement are necessary. Autonomic computing based on self-organization and self-management seems to be a promising avenue.

There are other perennial problems with no clear solutions at this time, including software licensing and dealing with systems bugs.

1.4 CLOUD DELIVERY MODELS AND DEFINING ATTRIBUTES

Cloud computing delivery models, deployment models, defining attributes, resources, and organization of the infrastructure discussed in chapter are summarized in Figure 1.2. The cloud delivery models, SaaS, PaaS, IaaS, and DBaaS can be deployed as public, private, community, and hybrid clouds.

The defining attributes of the new philosophy for delivering computing services are:

- Cloud computing uses Internet technologies to offer elastic services. The term "elastic computing" refers to the ability of dynamically acquiring computing resources and supporting a variable workload. A cloud service provider maintains a massive infrastructure to support elastic services.
- The resources used for these services can be metered and the users can be charged only for the resources they used.
- The maintenance and security are ensured by service providers.
- Economy of scale allows service providers to operate more efficiently due to specialization and centralization.
- Cloud computing is cost-effective due to resource multiplexing; lower costs for the service provider are passed on to the cloud users.
- The application data is stored closer to the site where it is used in a device and location-independent manner; potentially, this data storage strategy increases reliability and security and, at the same time, it lowers communication costs.

The term "computer cloud" is overloaded as it covers infrastructures of different sizes, with different management, and a different user population. Several types of clouds are envisioned:

- Private Cloud – the infrastructure is operated solely for an organization, It may be managed by the organization or a third party and may exist on or off the premises of the organization.

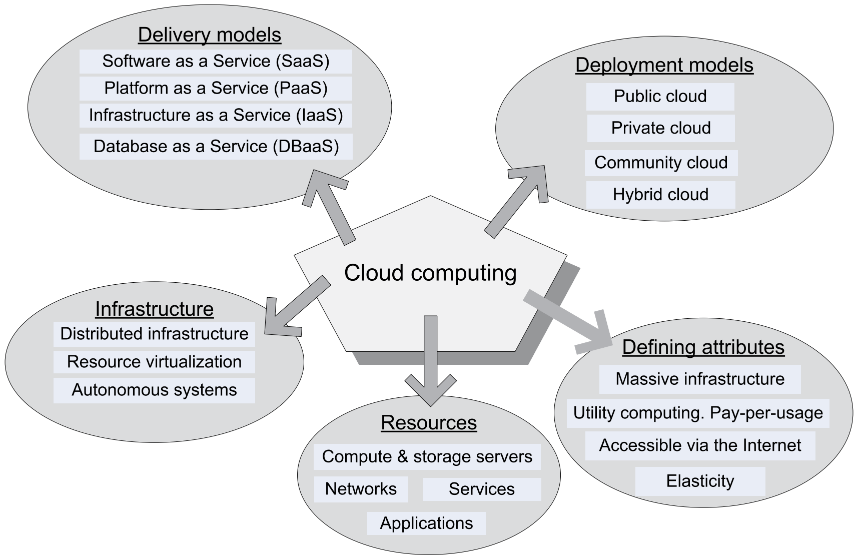


FIGURE 1.2

Cloud computing: delivery models, deployment models, defining attributes, resources, and organization of the infrastructure.

- Community Cloud – the infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premises or off premises.
- Public Cloud – the infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.
- Hybrid Cloud – the infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds).

A private cloud can provide the computing resources needed by a large organization, e.g., a research institution, a university, or a corporation. The argument that a private cloud does not support utility computing is based on the observation that an organization has to invest in the infrastructure and the user of a private cloud does pays as it consumes resources [37]. Nevertheless, a private cloud could use the same hardware infrastructure as a public one; its security requirements will be different from those for a public cloud and the software running on the cloud is likely to be restricted to a specific domain.

Cloud computing is a technical and social reality and an emerging technology. At this time, one can only speculate how the infrastructure for this new paradigm will evolve and what applications will migrate to it. The economical, social, ethical, and legal implications of this shift in technology, when the users rely on services provided by large data centers and store private data and software on systems they do not control, are likely to be significant.

Scientific and engineering applications, data mining, computational financing, gaming and social networking, as well as many other computational and data-intensive activities can benefit from cloud computing. A broad range of data from the results of high energy physics experiments to financial or enterprise management data, to personal data such as photos, videos, and movies, can be stored on the cloud.

The obvious advantage of network centric content is the accessibility of information from any site where one could connect to the Internet. Clearly, information stored on a cloud can be shared easily, but this approach raises also major concerns: Is the information safe and secure? Is it accessible when we need it? Do we still own it?

In the next years, the focus of cloud computing is expected to shift from building the infrastructure, today's main front of competition among the vendors, to the application domain. This shift in focus is reflected by Google's strategy to build a dedicated cloud for government organizations in the United States. The company states that: "We recognize that government agencies have unique regulatory and compliance requirements for IT systems, and cloud computing is no exception. So we've invested a lot of time in understanding government's needs and how they relate to cloud computing."

In a discussion of the technology trends, Jim Gray emphasized that the cost of communication in a wide area network has decreased dramatically and will continue to do so. Thus, it makes economical sense to store the data near the application [202], in other words, to store it in the cloud where the application runs. This insight leads us to believe that several new classes of cloud computing applications could emerge in the next few years [37].

As always, a good idea has generated a high level of excitement translated into a flurry of publications, some of a scholarly depth, others with little merit, or even bursting with misinformation. In this book we attempt to sift through the large volume of information and dissect the main ideas related to cloud computing. We first discuss applications of cloud computing and then analyze the infrastructure for cloud computing.

Several decades of research in parallel and distributed computing have paved the way for cloud computing. Through the years we have discovered the challenges posed by the implementation, as well as the algorithmic level and the ways to address some of them and avoid the others. Thus, it is important to look back at the lessons we learned along the years from this experience. This is the reason we discuss concurrency in Chapter 3 and parallel and distributed systems in Chapter 4.

1.5 ETHICAL ISSUES IN CLOUD COMPUTING

Cloud computing is based on a paradigm shift with profound implications on computing ethics. The main elements of this shift are:

1. The control is relinquished to third party services.
2. The data is stored on multiple sites administered by several organizations.
3. Multiple services interoperate across the network.

Unauthorized access, data corruption, infrastructure failure, and service unavailability are some of the risks related to relinquishing the control to third party services; moreover, whenever a problem occurs it is difficult to identify the source and the entity causing it. Systems can span the boundaries of multiple organizations and cross the security borders, a process called *de-perimeterisation*. As a result of de-perimeterisation “not only the border of the organizations IT infrastructure blurs, also the border of the accountability becomes less clear” [485].

The complex structure of cloud services make it difficult to determine who is responsible for each action. Many entities contribute to an action with undesirable consequences and no one can be held responsible, the so-called “problem of many hands.”

Ubiquitous and unlimited data sharing and storage among organizations test the self-determination of information, the right and/or the ability of individuals to exercise personal control over the collection, the use, and the disclosure of their personal data by others. This tests the confidence and trust in today’s evolving information society. Identity fraud and theft are made possible by the unauthorized access to personal data in circulation and by new forms of dissemination through social networks. All these factors could also pose a danger to cloud computing.

Cloud service providers have already collected petabytes of sensitive personal information stored in data centers around the world. The acceptance of cloud computing will be determined by the effort dedicated by the CSPs and the countries where the data centers are located to ensure privacy. Privacy is affected by cultural differences; while some cultures favor privacy, other cultures emphasize community and this leads to an ambivalent attitude towards privacy in the Internet which is a global system.

The question of what can be done proactively about ethics of cloud computing does not have easy answers as many undesirable phenomena in cloud computing will only appear in time. However, the need for rules and regulations for the governance of cloud computing are obvious. Governance means the manner something is governed or regulated, the method of management, the system of regulations. Explicit attention to ethics must be paid by governmental organizations providing research funding; private companies are less constraint by ethics oversight and governance arrangements are more conducive to profit generation.

Accountability is a necessary ingredient of cloud computing; adequate information about how data is handled within the cloud and about allocation of responsibility are key elements for enforcing ethics rules in cloud computing. Recorded evidence allows us to assign responsibility; but there can be tension between privacy and accountability and it is important to establish what is being recorded, and who has access to the records.

Unwanted dependency on a cloud service provider, the so-called *vendor lock-in*, is a serious concern and the current standardization efforts at NIST attempt to address this problem. Another concern for the users is a future with only a handful of companies which dominate the market and dictate prices and policies.

1.6 CLOUD VULNERABILITIES

Clouds are affected by malicious attacks and failures of the infrastructure, e.g., power failures. Such events can affect the Internet domain name servers and prevent access to a cloud or can directly affect the clouds. For example, an attack at Akamai on June 15, 2004 caused a domain name outage and a major blackout that affected Google, Yahoo, and many other sites. In May 2009, Google was the target of a serious denial of service (DNS) attack which took down services like Google News and Gmail for several days.

Lightning caused a prolonged down time at Amazon on June 29–30, 2012; the AWS cloud in the East region of the US which consists of ten data centers across four availability zones, was initially troubled by utility power fluctuations, probably caused by an electrical storm. Availability zones are locations within data center regions where public cloud services originate and operate. A June 29, 2012 storm on the East Coast took down some of Virginia based Amazon facilities and affected companies using systems exclusively in this region. Instagram, a photo sharing service, was one of the victim of this outage according to http://mashable.com/2012/06/30/aws-instagram/.

The recovery from the failure took a very long time and exposed a range of problems. For example, one of the ten centers failed to switch to backup generators before exhausting the power that could be supplied by UPS units. AWS uses “control planes” to allow users to switch to resources in a different region and this software component also failed. The booting process was faulty and extended the time to restart EC2 and EBS services.

Another critical problem was a bug in the Elastic Load Balancer (ELB), used to route traffic to servers with available capacity. A similar bug affected the recovery process of the Relational Database Service (RDS). This event brought to light “hidden” problems that occur only under special circumstances.

The stability risks due to interacting services are discussed in [177]. A cloud application provider, a cloud storage provider, and a networks provider could implement different policies and the unpredictable interactions between load-balancing and other reactive mechanisms could lead to dynamic instabilities. The unintended coupling of independent controllers which manage the load, the power consumption, and the elements of the infrastructure could lead to undesirable feedback and instability similar with the one experienced by the policy-based routing in the Internet BGP (Border Gateway Protocol).

For example, the load balancer of an application provider could interact with the power optimizer of the infrastructure provider. Some of these couplings may only manifest under extreme condition and be very hard to detect under normal operating condition, but could have disastrous consequences when the system attempts to recover from a hard failure, as in the case of the AWS 2012 failure.

Clustering resources in data centers located in different geographical areas lowers the probability of catastrophic failures. This geographic dispersion of resources could have additional positive side effects such as reduction of communication traffic, lowering energy costs by dispatching the computations to sites where the electric energy is cheaper, and improving performance by an intelligent and efficient load balancing strategy.

Sometimes, a user has the option to decide where to run an application; we shall see in Section 2.3 that an AWS user has the option to choose the regions where the instances of his/her applications will run, as well as the regions of the storage sites. System’s objective, maximize throughput, resource utilization, and financial benefits have to be carefully balanced with the user needs, low cost and response time and maximum availability.

The price to pay for any system optimization is an increased system complexity. For example, the latency of communication over a Wide Area Network (WAN) is considerably larger than the one over a Local Area Network (LAN) and requires the development of new algorithms for global decision making.

Chapter 2 takes a closer look at the cloud ecosystem as of late 2016. The next two chapters, Chapters 3 and 4 discuss concurrency concepts and parallel and distributed computing principles concepts relevant to cloud computing. Both subjects are very broad and we only cover aspects particularly relevant to cloud computing. An in depth analysis of networking access to computer clouds and cloud data storage are the subjects of the Chapters 5 and 6, respectively.

CHAPTER

CLOUD SERVICE PROVIDERS AND THE CLOUD ECOSYSTEM 2

This chapter presents the cloud ecosystem as of mid 2017. The major players in this ecosystem are Amazon, Google, and Microsoft. Each one of the big three cloud service providers support one or more of the cloud computing delivery models: SaaS, PaaS, IaaS, and DBaaS. Amazon is a pioneer in IaaS, Google's efforts are focused on SaaS and PaaS delivery models, Microsoft is mostly involved in PaaS, while Amazon, Oracle, and many other CSPs offer DBaaS services.

Several other IT companies are also involved in cloud computing. IBM offers a cloud computing platform, IBMSmartCloud, consisting of servers, storage and virtualization components for building private and hybrid cloud computing environments. In October 2012 it was announced that IBM teamed up with AT&T to give customers access to IBM's cloud infrastructure over AT&T's secure private lines.

In 2011 HP announced plans to enter the cloud computing club. Oracle announced its entry to enterprise computing in the early 2012. The Oracle Cloud is based on Java, SQL standards, and software systems such as Exadata, Exalogic, WebLogic, and Oracle Database. Oracle plans to offer application and platform services. Some of these services are Fusion HCM (Human Capital Management), Fusion CR (Customer Relation Management), and Oracle Social Network. The platform services are based on Java and SQL.

The chapter starts with an overview of the cloud ecosystem followed by an in-depth discussion of cloud delivery models and services and then by the analysis of cloud computing at Amazon, Google, and Microsoft in Sections 2.1, 2.2, 2.3, 2.4, 2.5, and 2.6, respectively. Sections 2.7 and 2.8 cover the pervasive issue of vendor lock-in and the prospects of cloud interoperability.

The presentation of Service Level Agreements (SLAs) in Section 2.9 is followed by a discussion of the responsibility sharing between the cloud users and the cloud service providers in Section 2.10. User experience is analyzed in Section 2.11, while Section 2.12 is dedicated to a discussion of software licensing. Section 2.13 presents an analysis of cloud energy consumption and the ecological impact of cloud computing. The major challenges faced by cloud computing are discussed in Section 2.14. The chapter concludes with further readings and exercises and problems in Sections 2.15 and 2.16, respectively.

2.1 THE CLOUD ECOSYSTEM

Hundreds of millions of individuals use online services. Computer clouds store and process every day a large fraction of the 2.5 quintillion bytes of data from sensors gathering climate data, from millions and millions of individuals taking digital pictures and videos, from cell phone GPS signals, and from a

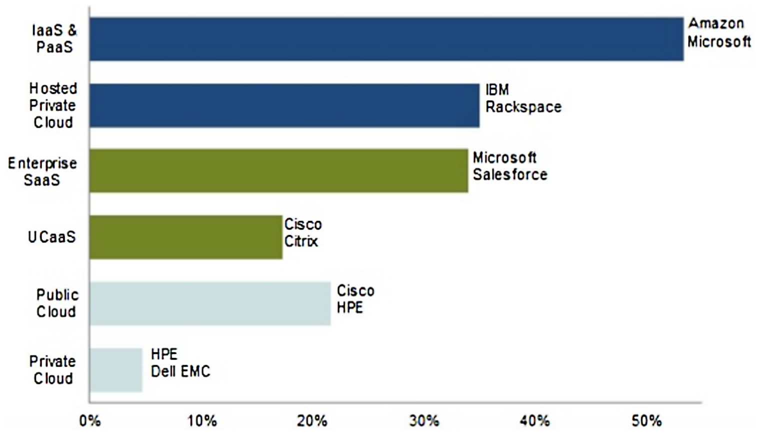


FIGURE 2.1

Annualized cloud computing revenues growth in the third quarter of 2016 by segment. The market leaders in each segment according to Synergy Research Group.

multitude of other sources. Cloud computing promoted by IT companies including Amazon, Google, Microsoft, IBM, and Oracle has effectively democratized computing.

Some 350 billion power-meter readings are analyzed every year to predict the electric power consumption. Every day 5 million trade transactions are scrutinized to detect fraud. Twitter processes 12 TB of data every day. Instagram uses AWS to process more than 25 photos and close to 100 likes a second. Forbes predicts that worldwide spending on public cloud services will grow at a 19.4% compound annual growth rate from nearly $70B in 2015 to more than $141B in 2019, see http://www.forbes.com/.

The economic implications of cloud computing cannot be overestimated. The number of enterprises using public and hybrid clouds for data analytics, product design, and a broad range of other applications increases significantly from year to year. While in the past it took years for an IT company to reach one million customers, it took only weeks to *Instagram* to reach this milestone.

The revenues generated by cloud computing increase dramatically from year to year. For example, according to the Synergy Research Group the revenues jumped 25% in 2016 as shown in Figure 2.1 from http://www.geekwire.com/2017/cloud-computing-revenues-jumped-25-2016-strong-growth-ahead-researcher-says/.

The cloud computing landscape is still dominated by Amazon. In 2013 the next fourteen other cloud providers combined had 1/5 the aggregate capacity of AWS [232]. A 2016 survey [420] reports "AWS is used by 57% of respondents; enterprise adoption of AWS grew from 50 to 56% while adoption by smaller businesses fell slightly from 61 to 58%," see Figure 2.2.

The same survey reports that "The number of enterprises running more than 1 000 virtual machines (VMs) in public cloud increased from 13 to 17%, while those running more than 1 000 VMs in private cloud grew from 22 to 31%." Docker has shown growth year-over-year, from 13 to 27% of respondents using it. The popularity of hybrid clouds is on the rise as many users of public clouds are developing their own private cloud infrastructure.

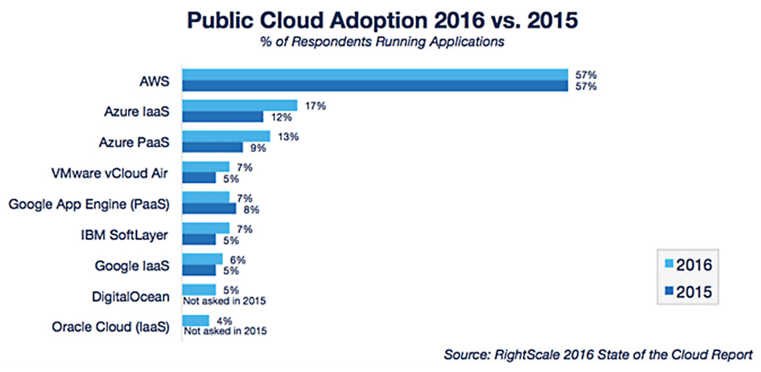


FIGURE 2.2

The 2016 state of public cloud adoption [420].

Some of the challenges faced by cloud users are: security, compliance, managing costs, and lack of expertise. Cloud users attempt to reduce the cloud costs through a range of measures including:

- Careful resource utilization monitoring.
- Avoiding peak hours and shutting down temporary workloads.
- Purchasing AWS reserved instances along with effective use of spot instances.
- Moving workloads to regions of lower costs.

The National Science Foundation (NSF) supports research community access to two cloud facilities, CloudLab and Chameleon. *CloudLab* is a testbed allowing researchers to experiment with cloud architectures and new applications. Some 15,000 cores, at three sites in Utah, Wisconsin, and South Carolina, are available for such experiments. *Chameleon* is an OpenStack KVM experimental environment for large-scale cloud research.

2.2 CLOUD COMPUTING DELIVERY MODELS AND SERVICES

According to the NIST reference model in Figure 2.3 [362], the entities involved in cloud computing are: *service consumer* – entity that maintains a business relationship with, and uses service from, service providers; *service provider* – entity responsible for making a service available to service consumers; *carrier* – the intermediary that provides connectivity and transport of cloud services between providers and consumers; *broker* – an entity that manages the use, performance and delivery of cloud services, and negotiates relationships between providers and consumers; *auditor* – a party that can conduct independent assessment of cloud services, information system operations, performance and security of the cloud implementation. An *audit* is a systematic evaluation of a cloud system by measuring how well it conforms to a set of established criteria. For example, a security audit evaluates cloud security, a privacy-impact audit evaluates cloud privacy assurance, while a performance audit evaluates cloud performance.

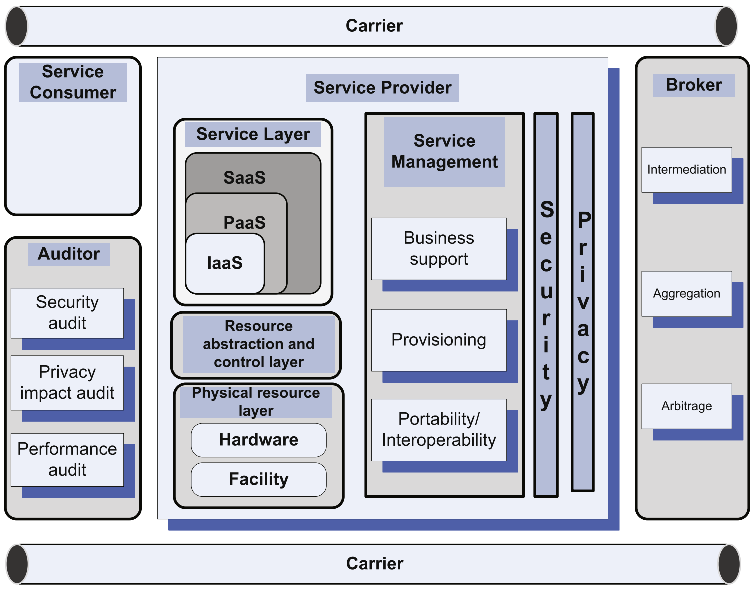


FIGURE 2.3

The entities involved in a service-oriented computing and, in particular, in cloud computing according to the NIST chart. The carrier provides connectivity between service providers, service consumers, brokers, and auditors.

It is difficult to distinguish the services associated with cloud computing from those that any computer operations center would include [463]. While many of the services discussed in this section could be provided by a cloud architecture, they are available in non-cloud architectures as well.

Figure 2.4 presents the structure of the three delivery models, SaaS, PaaS, and IaaS, according to the Cloud Security Alliance [124]. User's degrees of freedom and the complexity of its interaction with the cloud infrastructure vary from extremely limited in case of SaaS, to modest for PaaS, and significant for IaaS.

Software as a Service. The SaaS cloud infrastructure only runs applications developed by the service provider. A wide range of stationary and mobile devices allow a large population of clients to access the services provided by these applications using a thin client interface such as a web browser (e.g., web-based email). The users of services do not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings. Services offered include:

(a) Enterprise services such as: workflow management, group-ware and collaborative, supply chain, communications, digital signature, customer relationship management (CR), desktop software, financial management, geo-spatial, and search.

(b) Web 2.0 applications such as: metadata management, social networking, blogs, wiki services, and portal services.

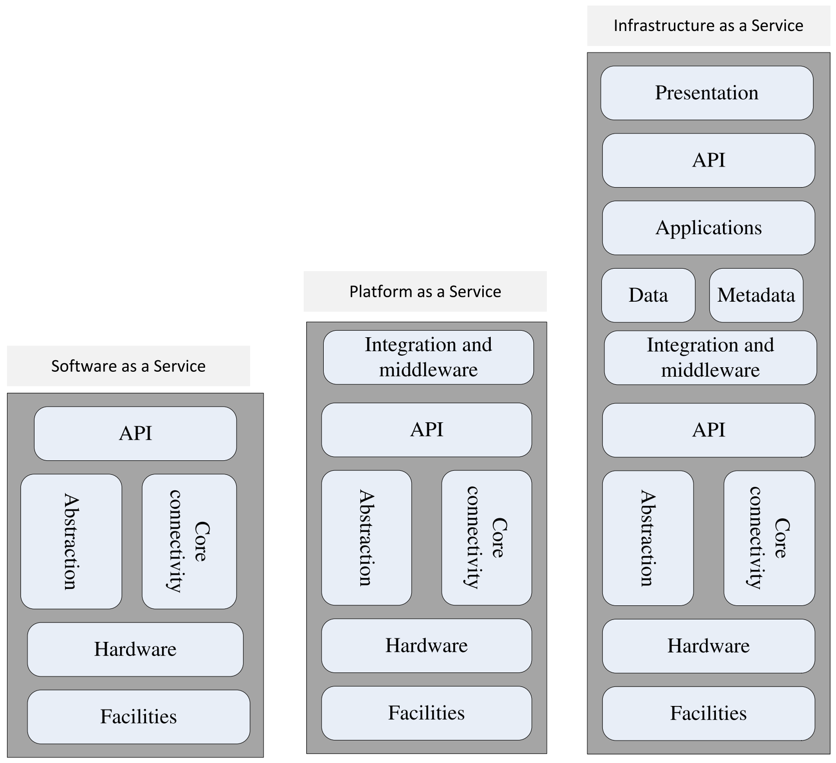


FIGURE 2.4

The structure of the three delivery models, SaaS, PaaS, and IaaS. SaaS gives the users capability to use applications supplied by the service provider but allows no control of the platform or the infrastructure. PaaS gives the capability to deploy consumer-created or acquired applications using programming languages and tools supported by the provider. IaaS allows the user to deploy and run arbitrary software, which can include operating systems and applications.

The SaaS is not suitable for applications which require real-time response or those when data is not allowed to be hosted externally; the most likely candidates for *SaaS* are applications when:

- Many competitors use the same product, such as Email;
- Periodically there is a significant peak in demand, such as billing and payroll;
- There is a need for the web or mobile access, such as mobile sales management software;
- There is only a short-term need, such as collaborative software for a project.

Platform as a Service. PaaS offers the capability to deploy consumer-created or acquired applications using programming languages and tools supported by the provider. The user does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage. The user has control over the deployed applications and, possibly, application hosting environment configurations. Such services include: session management, device integration, sandboxes, instrumentation and testing, contents management, knowledge management, and Universal Description, Discovery and Integration (UDDI), a platform-independent, Extensible Markup Language (XML)-based registry providing a mechanism to register and locate web service applications.

PaaS is not particularly useful when the application must be portable, when proprietary programming languages are used, or when the underlaying hardware and software must be customized to improve the performance of the application. Its major application areas are in software development when multiple developers and users collaborate and the deployment and testing services should be automated.

Infrastructure as a Service. IaaS has the capability to provision processing, storage, networks, and other fundamental computing resources; the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of some networking components, e.g., host firewalls. Services offered by this delivery model include: server hosting, web servers, storage, computing hardware, operating systems, virtual instances, load balancing, Internet access, and bandwidth provisioning.

The IaaS cloud computing delivery model has a number of characteristics such as: the resources are distributed and support dynamic scaling, it is based on a utility pricing model and variable cost, and the hardware is shared among multiple users. This cloud computing model is particularly useful when the demand is volatile and a new business needs computing resources and it does not want to invest in a computing infrastructure or when an organization is expanding rapidly.

Figure 2.3 shows that a number of activities are necessary to support the three delivery models. They include:

1. Service management and provisioning, such as: virtualization, service provisioning, call center, operations management, systems management, QoS management, billing and accounting, asset management, SLA management, technical support, and backups.
2. Security management, such as: ID and authentication, certification and accreditation, intrusion prevention, intrusion detection, virus protection, cryptography, physical security, incident response, access control, audit and trails, and firewalls.
3. Customer services, such as: customer assistance and on-line help, subscriptions, business intelligence, reporting, customer preferences, and personalization.
4. Integration services, such as: data management and development.

This shows that a service-oriented architecture involves multiple subsystems and complex interactions among these subsystems. Individual subsystems can be layered; for example, we see that the service layer sits on top of a resource abstraction layer which controls the physical resource layer.

Database as a Service. DBaaS is a cloud service where the database runs on the service provider's physical infrastructure. Compared with on-site physical server and storage architecture, a cloud database service offers distinct advantages:

- Instantaneous scalability.
- Performance guarantees.
- Specialized expertise.
- Latest technology.

- Failover support.
- Declining pricing.

Some of the most relevant features of the DBaaS model are:
1. Self-service – service provisioning without major deployment or configuration and without performance and cost penalties.
2. Device and location-independent abstract database resources without concern for hardware utilization.
3. Elasticity and scalability – automated and dynamic scaling.
4. Pay-as-you-go model – metered use of resources and cost reflecting the resources used.
5. Agility – the applications adapt seamlessly to new technology or additional requirements.

The cloud DBaaS uses a layered architecture. The *user interface layer* supports access to the service via the Internet. The *application layer* is used to access software services and storage space. The *database layer* provides efficient and reliable database service; it saves time for querying and loading data by reusing the query statements residing in the storage. The *data storage layer* encrypts the data when stored without user involvement; backup management and disk monitoring is also provided by this layer.

Multi-tenancy is an integral part of the DBaaS model. In spite of its advantages multi-tenancy poses resource management as well as security challenges as discussed in Section 11.6.

2.3 AMAZON WEB SERVICES

Amazon changed the face of computing in the last decade. First, it installed a powerful computing infrastructure to sustain its core business, selling online a variety of goods ranging from books and CDs to gourmet foods and home appliances. Then the company discovered that this infrastructure can be further extended to provide affordable and easy to use resources for enterprise computing, as well as computing for the masses.

In mid 2006 Amazon introduced AWS based on the IaaS delivery model. In this model the cloud service provider offers an infrastructure consisting of compute and storage servers interconnected by high-speed networks and supports a set of services to access these resources. An application developer is responsible to install applications on a platform of his choice and to manage the resources provided by Amazon.

It is reported that in 2012 businesses in 200 countries used the AWS. This shows the international appeal of this computing paradigm. A significant number of large corporations, as well as start-ups take advantage of computing services supported by the AWS infrastructure. For example, a start-up reports that its monthly computing bills at Amazon are in the range of \$100 000, while it would spend more than \$2 000 000 to compute using its own infrastructure, without the benefit of the speed and the flexibility offered by AWS; the start-up employs 10 engineers rather than 60 that would need to support its own computing infrastructure (“Active in cloud, Amazon reshapes computing” in the New York Times, August 28, 2012).

A March 28 2017, New York Time article with the title “Amazon’s Cloud Business Lifts Its Profit to a Record” reports: “The biggest source of the company’s profits is Amazon Web Services, the cloud computing business that started just over a decade ago and is now on track to bring in more than

$10 billion a year in revenue.....Amazon is the rare technology company of its size to still deliver double-digit revenue growth."

AWS computing, storage, and communication services. Amazon was the first provider of cloud computing; it announced a limited public beta release of its Elastic Computing platform called EC2 in August 2006. AWS released 24 services in 2008, 48 in 2009, 61 in 2010, 82 in 2011, 159 in 2012, 280 in 2013, and 449 new services and major features were released in 2014 [232]. Figure 2.5 shows the palette of AWS services accessible via the *Management Console* as of late 2011.

Elastic Compute Cloud (EC2)[1] is a web service with a simple interface for launching instances of an application under several operating systems, such as several Linux distributions, Microsoft Windows Server 2003 and 2008, OpenSolaris, FreeBSD, and NetBSD.

An *instance* is a virtual server; the user chooses the region and the availability zone where this virtual server should be placed and also selects from a limited menu of instance types the one which provides the resources, CPU cycles, main memory, secondary storage, communication and I/O bandwidth needed by the application.

When launched, an instance is provided with a *DNS name*; this name maps to a *private IP address* for internal communication within the internal EC2 communication network and a *public IP address* for communication outside the internal Amazon network, e.g., for communication with the user that launched the instance. Network Address Translation (NAT) maps external IP addresses to internal ones.

The public IP address is assigned for the lifetime of an instance and is returned to the pool of available public IP addresses when the instance is either stopped or terminated. An instance can request an *elastic IP address*, rather than a public IP address. The elastic IP address is a static public IP address allocated to an instance from the available pool of the availability zone; an elastic IP address is not released when the instance is stopped or terminated and must be released when no longer needed.

An instance is created either from a predefined Amazon Machine Image (AMI) digitally signed and stored in S3, or from a user-defined image. The image includes the operating system, the run-time environment, the libraries, and the application desired by the user. AMI images create an exact copy of the original image but without configuration-dependent information such as *hostname* or MAC address. A user can: (i) launch an instance from an existing AMI and terminate an instance; (ii) start and stop an instance; (iii) create a new image; (iv) add tags to identify an image; and (v) reboot an instance.

EC2 is based on the Xen virtualization strategy discussed in detail in Section 10.5. In EC2 each virtual machine or instance functions as a virtual private server. An instance specifies the maximum amount of resources available to an application, the interface for that instance, as well as the cost per hour. A server may run multiple virtual machines or instances, started by one or more users; an instance may use storage services, S3, EBS, and Simple DB, as well as other services provided by AWS, see Figure 2.6.

A user can interact with EC2 using a set of SOAP messages, see Section 7.1 and can list available AMI images, boot an instance from an image, terminate an image, display the running instances of a user, display console output, and so on. The user has root access to each instance in the elastic and secure computing environment of EC2. The instances can be placed in multiple locations in different Regions and Availability Zones.

[1]Amazon EC2 was developed by a team led by C. Pinkham including C. Brown, Q. Hoole, R. Paterson-Jones, and W. Van Biljon, all from Cape Town, South Africa.

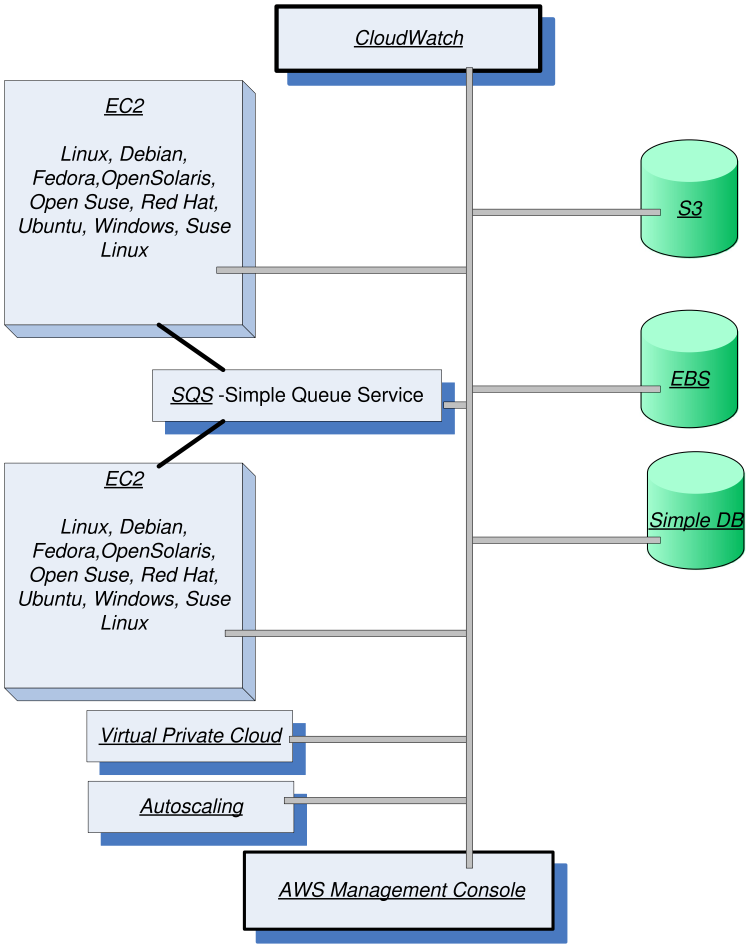

CloudWatch
EC2
Linux, Debian, Fedora,OpenSolaris, Open Suse, Red Hat, Ubuntu, Windows, Suse Linux
S3
SQS -Simple Queue Service
EBS
EC2
Linux, Debian, Fedora,OpenSolaris, Open Suse, Red Hat, Ubuntu, Windows, Suse Linux
Simple DB
Virtual Private Cloud
Autoscaling
AWS Management Console

FIGURE 2.5

Services offered by AWS are accessible from the *AWS Management Console*. Applications running under a variety of operating system can be launched using EC2. Multiple EC2 instances can communicate using *SQS*. Several storage services are available, such as S3, Simple DB, and EBS. The Cloud Watch supports performance monitoring and the *Auto Scaling* supports elastic resource management. The Virtual Private Cloud allows direct migration of parallel applications.

EC2 allows the import of Virtual Machine (VM) images from the user environment to an instance through a facility called *VM import*. It also distributes automatically the incoming application traffic among multiple instances using the *elastic load balancing* facility. EC2 associates an *elastic IP address* with an account; this mechanism allows a user to mask the failure of an instance and re-map a public IP address to any instance of the account, without the need to interact with the software support team.

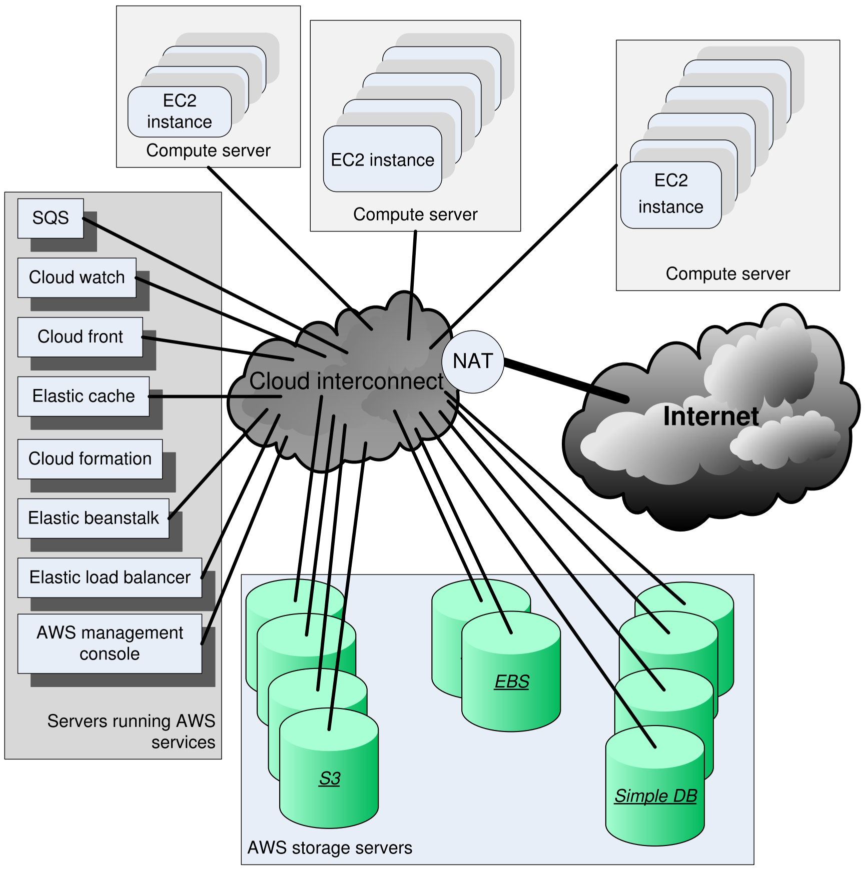


FIGURE 2.6

The configuration of an availability zone supporting AWS services. A cloud interconnect supports high-speed communication among compute and storage servers in the zone; it also supports communication with servers in other availably zones and with cloud users via a Network Address Translation. NAT maps external IP addresses to internal ones. Multi-tenancy increases server utilization and lowers costs.

Simple Storage System (S3) is a storage service designed to store large objects. It supports a minimal set of functions: write, read, and delete. S3 allows an application to handle an unlimited number of objects ranging in size from one byte to five terabytes. An object is stored in a *bucket* and retrieved via a unique, developer-assigned key; a bucket can be stored in a Region selected by the user.

S3 maintains for each object: the name, modification time, an access control list, and up to four kilobytes of user-defined metadata; the object names are global. Authentication mechanisms ensure that data is kept secure; objects can be made public, and rights can be granted to other users. S3 supports `PUT`, `GET`, and `DELETE` primitives to manipulate objects, but does not support primitives to copy, to rename, or to move an object from one bucket to another. Appending to an object, requires a read followed by a write of the entire object.

S3 computes the MD5[2] of every object written and returns it in a field called *ETag*. A user is expected to compute the *MD5* of an object stored or written and compare this with the *ETag*; if the two values do not match, then the object was corrupted during transmission or storage. The S3 SLA guarantees reliability. S3 uses standards-based REST and SOAP interfaces, see Section 7.1; the default download protocol is HTTP, but BitTorrent[3] protocol interface is also provided to lower costs for high-scale distribution.

Elastic Block Store. EBS provides persistent block level storage volumes for use with EC2 instances. A volume appears to an application as a raw, unformatted and reliable physical disk; the size of the storage volumes ranges from one gigabyte to one terabyte. The volumes are grouped together in Availability Zones and are automatically replicated in each zone. An EC2 instance may mount multiple volumes, but a volume cannot be shared among multiple instances. EBS supports the creation of snapshots of the volumes attached to an instance and then uses them to restart an instance. The storage strategy provided by EBS is suitable for database applications, file systems, and applications using raw data devices.

Simple DB is a non-relational data store that allows developers to store and query data items via web services requests; it supports store and query functions traditionally provided only by relational databases. Simple DB creates multiple geographically distributed copies of each data item and supports high performance web applications; at the same time, it manages automatically the infrastructure provisioning, hardware and software maintenance, replication and indexing of data items, and performance tuning.

Simple Queue Service. SQS is a hosted message queue. SQS is a system for supporting automated workflows; it allows multiple EC2 instances to coordinate their activities by sending and receiving SQS messages. Any computer connected to the Internet can add or read messages without any installed software or special firewall configurations.

Applications using SQS can run independently and asynchronously, and do not need to be developed with the same technologies. A received message is “locked” during processing; if processing fails, the lock expires and the message is available again. The timeout for locking can be changed dynamically via the *ChangeMessageVisibility* operation. Developers can access SQS through standards-based SOAP and Query interfaces. Queues can be shared with other AWS accounts and Anonymously; queue sharing can also be restricted by IP address and time-of-day. An example showing the use of message queues is presented in Section 7.6.

CloudWatch is a monitoring infrastructure used by application developers, users, and system administrators to collect and track metrics important for optimizing the performance of applications and for increasing the efficiency of resource utilization. Without installing any software a user can monitor approximately a dozen pre-selected metrics and then view graphs and statistics for these metrics.

When launching an Amazon Machine Image (AMI) the user can start the CloudWatch and specify the type of monitoring. The Basic Monitoring is free of charge and collects data at five-minute intervals

[2]MD5 (Message-Digest Algorithm) is a widely used cryptographic hash function; it produces a 128-bit hash value. It is used for checksums. SHA-i (Secure Hash Algorithm, $0 \leq i \leq 3$) is a family of cryptographic hash functions; SHA-1 is a 160 bit hash function resembling MD5.

[3]BitTorrent is a peer-to-peer (P2P) communications protocol for file sharing.

for up to ten metrics, while the Detailed Monitoring is subject to a charge and collects data at one minute interval. This service can also be used to monitor the latency of access to EBS volumes, the available storage space for RDS DB instances, the number of messages in SQS, and other parameters of interest for applications.

Virtual Private Cloud. VPC provides a bridge between the existing IT infrastructure of an organization and the AWS cloud; the existing infrastructure is connected via a Virtual Private Network (VPN) to a set of isolated AWS compute resources. VPC allows existing management capabilities such as security services, firewalls, and intrusion detection systems to operate seamlessly within the cloud.

Auto Scaling exploits cloud elasticity and provides automatic scaling of EC2 instances. The service supports: grouping of instances, monitoring of the instances in a group, and defining *triggers*, pairs of CloudWatch alarms and policies, which allow the size of the group to be scaled up or down. Typically, a maximum, a minimum, and a regular size of the group are specified.

An Auto Scaling group consists of a set of instances described in a static fashion by launch configurations. When the group scales up, new instances are started using the parameters for the *runInstances* EC2 call provided by the launch configuration; when the group scales down, the instances with older launch configurations are terminated first. The monitoring function of the Auto Scaling service carries out health checks to enforce the specified policies; for example, a user may specify a health check for elastic load balancing and then Auto Scaling will terminate an instance exhibiting a low performance and start a new one. Triggers use CloudWatch alarms to detect events and then initiate specific actions; for example, a trigger could detect when the CPU utilization of the instances in the group goes above 90% and then scale up the group by starting new instances. Typically, triggers to scale up and down are specified for a group.

Several AWS services introduced in 2012 include:

- Route 53 – a low-latency DNS service used to manage user's DNS public records.
- Elastic MapReduce (EMR) – a service supporting processing of large amounts of data using a hosted Hadoop running on EC2 and based on the MapReduce paradigm discussed in Section 7.5.
- Simple Workflow Service (SWS) – supports workflow management and allows scheduling, management of dependencies, and coordination of multiple EC2 instances.
- ElastiCache – a service enabling web applications to retrieve data from a managed in-memory caching system rather than a much slower disk-based database.
- DynamoDB – a scalable and low-latency fully managed NoSQL database service.
- CloudFront – a web service for content delivery.
- Elastic Load Balancer – a cloud service to automatically distribute the incoming requests across multiple instances of the application.

Two other services, the CloudFormation and the Elastic Beanstalk are discussed next.

CloudFormation allows the creation of a stack describing the infrastructure for an application. The user creates a template, a text file formatted as in Javascript Object Notation (JSON), describing the resources, the configuration values, and the interconnection among these resources. The template can be parameterized to allow customization at run time, e.g., to specify the types of instances, database port numbers, or RDS size. Here is a template for the creation of an EC2 instance:

```
{
 "Description" : "Create instance running Ubuntu Server 12.04 LTS 64 bit AMI"
  "Parameters" : {
        "KeyPair" : {
            "Description" : "Key Pair to allow SSH access to the instance",
            "Type" : "String"
        }
  },
  "Resources" : {
        "Ec2Instance" : {
            "Type" : "AWS::EC2::Instance",
            "Properties" : {
                "KeyName" : { "Ref" : "KeyPair" },
                "ImageId" : "aki-004ec330"
            }
        }
  },
  "Outputs" : {
        "InstanceId" : {
             "Description" : "The InstanceId of the newly created instance",
             "Value" : { "Ref" : "Ec2InstDCM" }
         }
  },
  "AWSTemplateFormatVersion" : "2012-03-09"
}
```

Elastic Beanstalk interacts with other AWS services including EC2, S3, SNS, Elastic Load Balance, and AutoScaling. ElasticBeanstalk handles automatically the deployment, capacity provisioning, load balancing, auto-scaling, and application monitoring functions [495]. The service automatically scales the resources as required by the application, either up, or down based on default Auto Scaling settings. Some of the management functions provided by the service are:

1. Deploy a new application version or rollback to a previous version.
2. Access the results reported by CloudWatch monitoring service.
3. Email notifications when application status changes or application servers are added or removed.
4. Access server log files without needing to login to the application servers.

The Elastic Beanstalk service is available to developers using either a Java platform, the PHP server-side description language, or *.NET* framework. For example, a Java developer can create an application using an Integrated Development Environment, such as Eclipse and package the code into Java Web Application Archive file of type ".war". The ".war" file should then be uploaded to the Elastic Beanstalk using the Management Console and then deployed and in a short time the application will be accessible via an URL.

Users have several choices to interact and manage AWS resources either from a web browser or from a system running Linux or Microsoft Windows:

1. The Web Management Console; not all options may be available in this mode.
2. Command-line tools, see http://aws.amazon.com/developertools.

3. AWS SDK libraries and toolkits provided for several programming languages including Java, PHP[4], C#, and Obj C.
4. Raw REST requests as shown in Section 7.1.

The Amazon Web Services Licensing Agreement (AWSLA) allows the cloud service provider to terminate service to any customer at any time for any reason and contains a covenant not to sue Amazon or its affiliates for any damages that might arise out of the use of AWS. As noted in [186], AWSLA prohibits the use of "other information obtained through AWS for the purpose of direct marketing, spamming, contacting sellers or customers." It prohibits AWS from being used to store any content that is "obscene, libelous, defamatory or otherwise malicious or harmful to any person or entity;" it also prohibits S3 from being used "in any way that is otherwise illegal or promotes illegal activities, including without limitation in any manner that might be discriminatory based on race, sex, religion, nationality, disability, sexual orientation or age."

An early evaluation of Amazon Web Services. A 2007 evaluation of AWS [186] reports that EC2 instances are fast, responsive, and very reliable; a new instance could be started in less than two minutes. During the year of testing, one unscheduled reboot and one instance freeze were experienced, no data was lost during the reboot, but no data could be recovered from the virtual disks of the frozen instance.

To test the S3 service, a bucket was created and loaded with objects in sizes of 1 byte, 1 KB, 1 MB, 16 MB, and 100 MB. The measured throughput for the 1-byte objects reflected the transaction speed of S3 because the testing program required that each transaction be successfully resolved before the next was initiated. The measurements showed that a user could execute at most 50 non-overlapping S3 transactions. The 100 MB probes measured the maximum data throughput the S3 system could deliver to a single client thread. The measurements showed that the data throughput for large objects was considerably larger than for small objects, most likely due to a high transaction overhead. The write bandwidth for 1 MB data was roughly 5 MB/s while the read bandwidth was 5 times lower, 1 MB/s.

Another test was designed to see if concurrent requests could improve the throughput of S3. The experiment involved two virtual machines running on two different clusters and accessing the same bucket with repeated 100 MB `GET` and `PUT` operations. The virtual machines were coordinated, with each one executing 1 to 6 threads for 10 minutes and then repeating the pattern for 11 hours. As the number of threads increased from 1 to 6, the bandwidth received by each thread was roughly cut in half and the aggregate bandwidth of the six threads was 30 MB/s, about three times the aggregate bandwidth of one thread. In 107 556 tests of EC2, each one consisting of multiple read and write probes, only 6 write retries, 3 write errors, and 4 read retries were encountered.

2.4 THE CONTINUING EVOLUTION OF AWS

Amazon is one of the most important forces driving the spectacular evolution of cloud computing in recent years. The AWS infrastructure has benefited from a wealth of new technologies. Today AWS is

[4]PHP evolved from a set of Perl scripts designed to produce dynamic web pages called "Personal Home Page Tools" into a general-purpose server-side scripting language. The code embedded into an HTML source document is interpreted by a web server with a PHP processor module which generates the resulting web page.

Table 2.1 The resources offered by M4, C4, and G2 instances; the number of vCPUs, the amount of memory, the data rates for disk access, and the cost per hour.

Instance type	vCPU	Memory (GiB)	EBS throughput (Mbps)	Cost ($/hour)
m4.large	2	8	450	0.12
m4.xlarge	4	16	750	0.239
m4.2xlarge	8	32	1 000	0.479
m4.4xlarge	16	64	2 000	0.958
m4.10xlarge	40	160	4 000	2.394
c4.large	2	3.75	500	0.105
c4.xlarge	4	7.5	750	0.209
c4.2xlarge	8	15	1 000	0.419
c4.4xlarge	16	30	2 000	0.838
c4.8-xlarge	36	60	4 000	1.675
g2.2xlarge	8	15	–	0.65
g2.4xlarge	32	60	–	2.60

probably the most attractive and cost-effective cloud computing environment, not only for enterprise applications, but also for computational science and engineering applications [24].

The massive effort to continually expend the hardware and the software of the AWS cloud infrastructure is astounding. Amazon has designed its own storage racks; such a rack holds 864 disk drives and weighs over a ton. The company has designed and built their own power substations. Three of its regions, US West (Oregon), AWS GovCloud (US), and EU (Frankfurt) are 100% carbon neutral.

EC2 instances. AWS offers several types of EC2 instances targeting different classes of applications:

- T2 – provide a baseline CPU performance and the ability to exceed the baseline.
- M3 & M4 – provide a balance of compute, memory, and network resources.
- C4 – use high performance processors and have the lowest price/compute performance.
- R3 – are optimized for memory-intensive applications.
- G2 – target graphics and general-purpose GPU applications.
- I2 – are storage optimized.
- D2 – deliver high disk throughput.

Each instance packages a different combination of processors, memory, storage, and network bandwidth. The number of vCPUs as well as the type of processor, its architecture, and clock speed are different for different instance types. A vCPU is a virtual processor assigned to one virtual machine.

AWS does not specify if a vCPU corresponds to a core of a multi-core processor, though this is likely. The amount of memory per vCPU is the same for low and high-end instances. The memory is sometimes measured in Gibibytes, 1 GiB $= 2^{30}$ bytes or 1 073 741 824 bytes while 1 GB $= 10^9$ bytes.

The processors used by instances in Table 2.1 are Intel Xeon E5-2670 v3 running at 2.5 GHz for M4 instances, Intel Xeon E5-2666 v3 running at 2.9 GHz for C4 instances, and E5-2670 for G2 instances. The first two processors support Advanced Vector Extensions AVX and AVX2. The two are extensions to the x86 ISA. In AVX2 the width of the SIMD register file is increased from 128 bits to 256 bits along with several additional features including: expansion of most vector integer SSE and AVX instructions to 256 bits; three-operand general-purpose bit manipulation and multiply; three-operand fused multiply-accumulate support; gather support, enabling vector elements to be loaded from non-contiguous memory locations; DWORD- and QWORD-granularity any-to-any permutes; and vector shifts. An NVIDIA GPU for a G2 instances has 1 536 CUDA cores and 4 GB of video memory.

Several operating systems including Apple OS, Linux, Windows, FreeBSD, OpenBSD, and Solaris support AVX. Recent releases of the GCC compiler support AVX. Unfortunately, there are no recent benchmarks comparing the performance of systems on the top 500 list with some of the 2016 AWS instances presented in Table 2.1.

The floating point performance of C4 instances is impressive. For example, a c4.8xlarge instance with 2 Intel Xeon E5-2666 v3 processors running at 3.50 GHz, 18 cores, and 36 threads, with 32 KB × 9 L1 instruction and data caches, 256 KB × 9 L2 cache, 26.3 MB L3 cache, and 60 GB main memory delivers more than 61 Gflops from a multi-core configuration according to http://browser.primatelabs.com/geekbench3/1694602.

The performance of G2 instances with attached GPUs is even more impressive. Results reported in [125] show the performance of CUDA 7.0 for several libraries including *cuFFT*, *cuBLAS*, *cuSPARSE*, *cuSOLVER*, *cuRAND*, *cuDNN*. For example, *cuBLAS* supports all 152 standard routines and distributed computations across multiple GPUs with out-of-core streaming to CPU and no upper limits on matrix size supporting more than three Tflops on single-precision and more than one Tflops in double-precision.

AWS Lambda Service. For many of us cloud computing is associated with Big Data applications and long lasting computations rather than real-time applications and short bursts of computing triggered by some external events. Consistent to its reputation as the leader in cloud computing AWS strives to offer new services in anticipation of future needs of the cloud computing community.

Anticipating services relevant to the Internet of Things AWS introduced a few years ago a *servless* computer service where applications are triggered by conditions and/or events specified by the user. For example, an application may run for a brief period of time at midnight to check the daily energy consumption of an enterprise, may be activated weekly to check the sales of chain, or to turn on the alarm system of a home triggered by an event generated by the smartphone of the owner.

In stark contrast to EC2 when customers are billed on an hourly basis, e.g., if a C4 instance is used for one hours and ten minutes the billing is for two hours, the *Lambda* service is billed for the actual time with a resolution of milliseconds. The service seems relatively easy to use. "First you create your function by uploading your code (or building it right in the Lambda console) and choosing the memory, timeout period, and AWS Identity and Access Management (IAM) role. Then, you specify the AWS resource to trigger the function, either a particular Amazon S3 bucket, Amazon DynamoDB table, or Amazon Kinesis stream. When the resource changes, Lambda will run your function and launch and manage the compute resources as needed in order to keep up with incoming requests" according to https://aws.amazon.com/lambda/details/. Amazon Kinesis is a data streaming platform.

Regions and availability zones. Amazon offers cloud services through a network of data centers on several continents. In mid 2017 Amazon had 28+ data centers. An *availability zone* (AZ) is a data center with 50 000–80 000 servers using 25–30 MW of power. In each *region* there are several *availability zones* interconnected by high-speed networks. Regions do not share resources and communicate through the Internet. All regions have at least two availability zones. The AWS regions as of January 2017 are:

Table 2.2 AWS services with limited availability in the Europe/Middle East/Africa regions.

Service	Ireland	Frankfurt	London
Amazon AppStream 2.0	Yes		
Amazon CloudSearch	Yes	Yes	
Amazon Cognito	Yes	Yes	
Amazon GameLift	Yes	Yes	
Amazon Cloud Directory	Yes	Yes	
Amazon EC2 Systems Manager	Yes	Yes	
Amazon Elastic Transcoder	Yes		
Amazon Elastic File System (EFS)	Yes		
Amazon Machine Learning	Yes		
Amazon SimpleDB	Yes		
Amazon WorkSpaces	Yes	Yes	
AWS IoT	Yes	Yes	

Table 2.3 The billing rates from several regions.

Region	Location	Availability zones	Cost
US West	Oregon	us-west-2a/2b/2c	Low
US West	North California	us-west-1a/1b/1c	High
US East	North Virginia	us-east-1a/2a/3a/4a	Low
Europe	Ireland	eu-west-1a/1b/1c	Medium
South America	São Paulo, Brazil	sa-east-1a/1b	Very high
Asia Pacific	Tokyo, Japan	ap-northeast-1a/1b	High
Asia Pacific	Singapore	ap-southeast-1a/1b	Medium

- Americas: Northern Virginia, Ohio, Oregon, Northern California, Montreal, São Paulo, and the GovCloud.
- Asia Pacific: Singapore, Tokyo, Sydney, Seul, Mumbai, Beijing.
- Europe/Middle East/Africa: Ireland, Frankfort, London.

Storage is automatically replicated within a region. S3 buckets are replicated within an availability zone and between the availability zones of a region, while *EBS* volumes are replicated only within the same availability zone. Critical applications are advised to replicate important information in multiple regions to be able to function when the servers in one region are unavailable due to catastrophic events.

Most AWS services are available in all regions, though some are not. The list of services in the eight Americas regions includes: CloudWatch; CloudWatch Logs; DynamoDB; ElastiCache; Elastic MapReduce; Glacier; Kinesis Streams; Redshift; Relational Database Service (RDS); Simple Notification Service (SNS); Simple Queue Service (SQS); Simple Storage Service (S3); Simple Workflow Service (SWF); Virtual Private Cloud (VPC); Auto Scaling; CloudFormation; CloudTrail; Config; Direct Connect; Key Management Service; Shield Standard; Elastic Load Balancing; and VM Import/Export. Table 2.2 lists services with limited availability in the three Europe/Middle East/Africa regions as of January 2017.

The billing rates differ from one region to another and can be roughly grouped into four categories, low, medium, high, and very high, see Table 2.3. These rates are determined by the components of the operating costs including energy, communication, and maintenance costs. Thus, the choice of the region is motivated by the desire to minimize costs, reduce the communication latency, and increase reliability and security.

AWS networking. Each AWS region has redundant *transit centers* connecting private links to other AWS regions, private links to AWS Direct Connect customers, and to the Internet through peering and paid transit. Most major regions are interconnected by private fiber channels and this avoids peering issues, buffering problems, and capacity limitations that may occur on public links.

Peak traffic between availability zones of up to 25 Tbps is supported. The communication latency between availability zones is in the 1 to 2 milliseconds range. The communication latency between two servers has three components:

1. Application $\mapsto$ guest OS $\mapsto$ hypervisor $\mapsto$ Network Interface (NIC) – in the milliseconds range.
2. Through the NIC – in the microseconds range.
3. Over the fiber – in the nanoseconds range.

Single Root I/O Virtualization virtualizes the NICs, each guest gets its own virtual NIC [232].

2.5 GOOGLE CLOUDS

Google's effort is concentrated in several areas of Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) [198]. It was estimated that the number of servers used by Google in January 2012 was close to 1.8 million and that number was expected to reach close to 2.4 million in the early 2013 [399]. Google maintains two billion lines of code [5] related to its cloud infrastructure.

Services such as Gmail, Google Drive, Google Calendar, Picasa and Google Groups are free of charge for individual users and available for a fee for organizations. These services are running on a cloud and can be invoked from a broad spectrum of devices including mobile ones including smart phones, tablets, and laptops. The data for these services is stored at data centers on the cloud.

AppEngine. Google is a leader in the Platform-as-a-Service (PaaS) space. AppEngine (AE) is an infrastructure for building web and mobile applications and run these application on Google servers. Initially, it supported only Python and support for Java was later added. The database for code development can be accessed with GQL (Google Query Language) with a SQL-like syntax.

The AppEngine is an ensemble of computer, storage, search, and networking services. The *Compute Engine* (CE) supports the creation of VMs with resources tailored to the application needs. The CE configurations range from micro instances to the ones with 32 vCPUs or 208 GB of memory. Up to 64 TB of network storage can be attached to a VM. Always-encrypted local solid-state drive (SSD) block storage and automatic scaling are also supported. Several operating systems including Debian, CentOS, CoreOS, SUSE, Ubuntu, Red Hat, FreeBSD, or Windows 2008 R2 and 2012 R2 are supported.

[5] This repository is available only to Google's 25,000 developers; Google has its own version control system called Piper. GitHub is a public open source repository where individuals share enormous amounts of code.

The *Container Engine* (CntE) is a cluster manager and orchestration system for Docker containers built on the Kubernetes system. The *Container Registry* stores private Docker images. CntE schedules and manages containers automatically according to user specifications. JSON config files are used to specify the amount of CPU/memory, the number of replicas, and other relevant information. The Cloud Container Engine SLA commitment is a monthly uptime of at least 99.5%. *Cloud Functions* (CF) is a lightweight, event-based, asynchronous system to create single-purpose functions that respond to cloud events. CFs are written in Javascript and execute in a Node.js runtime environment. *Cloud Load Balancing* supports scalable load balancing on the Google Cloud Platform.

Cloud Storage is a unified object storage allowing a multi-region operation for applications including video streaming and frequently accessed web and images sites. *Cloud SQL* is a fully-managed database service, *Cloud Bigtable* is a high performance NoSQL database service for large analytical and operational workloads, and *Cloud Datastore* is a highly-scalable NoSQL database for web and mobile applications.

Big Data applications are supported by several services. *Bigquery* is a fully-managed enterprise data warehouse for large-scale data analytics. *Cloud Dataflow* supports stream and batch execution of pipelines. The *Cloud Dataproc* manages Spark and Hadoop service. *Cloud Datalab* is an interactive tool for large-scale data exploration, analysis, and visualization. *Cloud Pub/Sub* is a global service for real-time reliable messaging and data streaming.

Network functionality is managed by the *Cloud Virtual Network* (CVN). AppEngine users can connect resources to each other and isolate them from one another in a Virtual Private Cloud using the CVN. Cloud routers maintain virtual routers enabling Border Gateway Protocol (BGP) to route updates between a user Compute Engine network and user non-Google network. CVN supports Virtual Private Networks (VPNs) and distributed firewalls. *Cloud CDN* is a low-latency, low-cost content delivery network. *Cloud DNS* is a resilient, low-latency DNS for Google's worldwide network.

Several AppEngine services support security. *Cloud Identity and Access Management* (IAM) provides the tools to manage resource permissions and to map job functions within a company to groups and roles. *Cloud KMS* is a key management service allowing users to manage encryption and generate, use, rotate and destroy AES256 key encryption keys. *Cloud Security Scanner* is used to scan for common vulnerabilities in Google App Engine applications.

App Engine provides cloud development tools. *Cloud SDK* is a set of tools including *gcloud, gsutil*, and *bq*, to access the Compute Engine, the Cloud Storage and other services. The *Cloud Source Repositories* provides Git version control to support collaborative development for applications or services. *Android Studio*. *PowerShell*, *IntelliJ*, *Eclipse* and *Visual Studio* tools are also available.

The array of management tools include the *Stackdriver* which supports monitoring, logging, and diagnostics tools along with the *Stackdriver Monitoring* tool which provides performance, uptime, and overall health information on cloud applications. The *Stackdriver Debugger* lets users inspect the state of an application without logging statements and without stopping or slowing down your application. The *Stackdriver Error Reporting* counts, analyzes and aggregates the crashes and provides a cleaned exception stack trace.

A range of services supporting machine learning are also provided. The *Cloud Machine Learning* is a managed service based on the TensorFlow model to build machine learning models, that work on any type of data. The *Cloud Natural Language API* is a text analysis tool that can be used to extract information about people, places, events, and so on while the *Cloud Speech API* allows developers to

Table 2.4 The features of the AppEngine standard environment (AE-STD) and of the flexible environment (AE-FLX).

Feature	AE-STD	AE-FLX
Instance startup time	Milliseconds	Minutes
Maximum request timeout	60 Seconds	60 Minutes
Background threads	Yes, with restrictions	Yes
Background process	No	Yes
Writing to local disk	No	Yes
SSH debugging	No	Yes
Scaling	Manual, Basic, Automatic	Manual, Automatic
Network access	Only via AppEngine services	Yes
Support installing third party binaries	No	Yes
Location/Availability	North America, Europe, Asia Pacific	North America Asia Pacific

Table 2.5 Pricing and service quotas for Google's App Engine for US and Europe.

App Engine Service	Free Access Daily Limits	Price per Unit Above Free Access
Instances	28 instance hours	$0.05 per instance hour
Cloud Datastore Ops reads/writes/deletes	50k/25k/25k	$0.06/$0.18/$0.02 per 100k ops in each category
Cloud Datastore Storage	1 GB	$01.8 per GB/month
Outgoing Network Traffic	1 GB	$0.12/GB
Incoming Network Traffic	1 GB	Free
Cloud Storage	5 GB	$0.026 per GB/month
Memcache – Dedicated Pool	0	$0.06 per GB/hour
Memcache – Shared Pool	Free	Free
Searches	100	$0.50 per 10 k searches
Search – Indexing Documents	0.01 GB	$2.0 per GB
Task Queue	5 GB	$0.026 GB/month
SSL Virtual IPs	–	$39.0 Virtual IP/month

convert audio to text by applying powerful neural network models and the *Cloud Vision API* is used to understand the content of an image by encapsulating powerful machine learning models.

An AE application can run in the *standard environment* and/or the *flexible environment*. The features of the two environments are summarized in Table 2.4. The pricing and service quota are summarized in Table 2.5. After a 10-minute minimum charge the users are charged in minute-level increments.

Gmail. The service hosts emails on Google servers, provides a Web interface to access them and tools for migrating from Lotus Notes and Microsoft Exchange.

Google Docs is a web-based software for building text documents, spreadsheets and presentations. It supports features such as tables, bullet points, basic fonts and text size; it allows multiple users to edit and update the same document, to view the history of document changes, and it provides a spell checker. The service allows users to import and export files in several formats including Microsoft Office, PDF, text, and OpenOffice extensions.

Google Calendar is a browser-based scheduler. It supports multiple calendars for a user, the ability to share a calendar with other users, the display of daily/weekly/monthly views, to search events, and to synchronize with the Outlook Calendar. The calendar is accessible from mobile devices; event reminders can be received via SMS, desktop pop-ups, or emails. It is also possible to share your calendar with other Google calendar users.

Picasa is a tool to upload, share, and edit images; it provides 1 GB of disk space per user free of charge. Users can add tags to images and attach locations to photos using *Google Maps*. *Google Groups* allows users to host discussion forums to create messages online or via email.

Google Co-op allows users to create customized search engines based on a set of *facets* or categories; for example, the facets for a search engine for the database research community available at http://data.cs.washington.edu/coop/dbresearch/index.html are: `professor, project, publication, jobs.`

Google Base is a service allowing the users to load structured data from different sources to a central repository which is a very large, self-describing, semi-structured, heterogeneous database. It is self-describing because each item follows a simple schema: (item type, attribute names). Few users are aware of this service, thus *Google Base* is accessed in response to keyword queries posed on *Google.com*, provided that there is relevant data in the database. To fully integrate Google Base, the results should be ranked across properties. Also, the service needs to propose appropriate refinements with candidate values in select-menus; this is done by computing histograms on attributes and their values during query time.

Google Drive is an online service for data storage available since April 2012. It gives users 5 GB of free storage and charges $4/month for 20 GB. It is available for PCs, MacBooks, iPhones, iPads, and Android devices and allows organizations to purchase up to 16 TB of storage.

Specialized structure-aware search engines for several areas, including travel, weather and local services, have already been implemented. However, the data available on the web covers a wealth of human knowledge; it is not feasible to define all the possible domains and it is nearly impossible to decide where one domain ends and another begins.

Google has also redefined the laptop with the introduction of the Chromebook, a purely Web-centric device running Chrome-OS. Cloud-based applications, extreme portability, built-in $3G$ connectivity, almost instant-on, and all-day battery life are the main attractions of this device with a keyboard.

Google adheres to a bottom-up, engineer-driven, liberal licensing, and user application development philosophy, while Apple, a recent entry in cloud computing, tightly controls the technology stack, builds its own hardware and requires the applications developed to follow strict rules. Apple products including the iPhone, the iOS, the iTunes Store, Mac OS X, and iCloud offer unparalleled polish and effortless interoperability, while the flexibility of Google results in more cumbersome user interfaces for the broad spectrum of devices running the Android OS.

Google, as well as other cloud service providers, manage vast amounts of data. In a world where users would most likely desire to use multiple cloud services from independent providers the question if the traditional Data Base Management Services (DBMS) are sufficient to ensure interoperability comes to mind. A DBMS efficiently supports data manipulations and query processing, but operates in a single administrative domain and uses a well-defined schema. The interoperability of data management services requires *semantic integration* of services based on different schemas. An answer to the

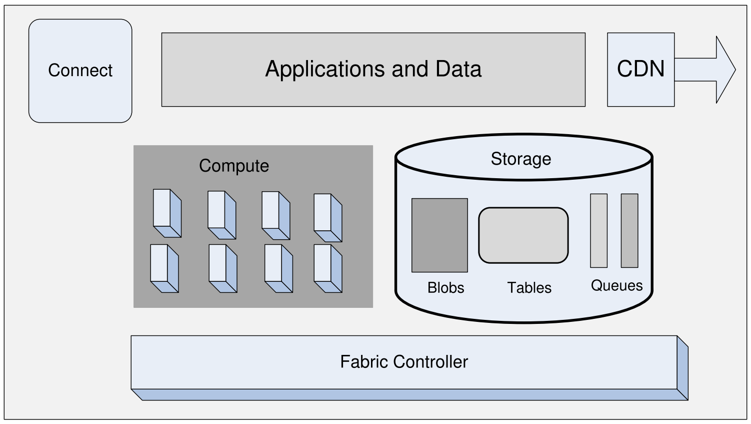


FIGURE 2.7

The components of Windows Azure: Compute – runs cloud applications; Storage – uses blobs, tables, and queues to store data; Fabric Controller – deploys, manages, and monitors applications; CDN – maintains cache copies of data; Connect – allows IP connections between the user systems and applications running on Windows Azure.

limitations of traditional DBMS are the so called *dataspaces* introduced in [178]; dataspaces do not aim at data integration, but at data co-existence.

2.6 MICROSOFT WINDOWS AZURE AND ONLINE SERVICES

Azure and Online Services are PaaS and, respectively, SaaS cloud platforms provided by Microsoft. Azure is an operating system, SQL Azure is a cloud-based version of the SQL Server, and Azure AppFabric (formerly .NET Services) is a collection of services for cloud applications.

Windows Azure has three core components (see Figure 2.7): *Compute* which provides a computation environment, *Storage* for scalable storage, and *Fabric Controller* which deploys, manages, and monitors applications; it interconnects nodes consisting of servers, high-speed connections, and switches. The Content Delivery Network (CDN) maintains cache copies of data to speedup computations. The Connect subsystem supports IP connections between the users and their applications running on Windows Azure. The API interface to Windows Azure is built on REST, HTTP and XML. The platform includes five services: Live Services, SQL Azure, AppFabric, SharePoint, and Dynamics CR. A client library and tools are also provided for developing cloud applications in Visual Studio.

The computations carried out by an application are implemented as one or more *roles*; an application typically runs multiple *instances of a role*. One distinguishes: (i) Web role instances used to create web applications; (ii) Worker role instances used to run Window-based code; and (iii) VM role instances running user-provided Windows Server 2008 R2 images.

Scaling, load balancing, memory management, and reliability are ensured by a *fabric controller*, a distributed application replicated across a group of machines which owns all resources in its environment including computers, switches, and load balancers. The fabric controller is aware of every Windows Azure application and decides where new applications should run. The fabric controller chooses the physical servers to optimize utilization using configuration information uploaded with each Windows Azure application. Configuration information is an XML-based description of how many web role instances, how many Worker role instances, and what other resources the application needs; the fabric controller uses this configuration file to determine how many VMs to create.

Blobs, tables, queue, and drives are used as scalable storage. A blob contains binary data, a container consists of one or more blobs. Blobs can be up to a terabyte and they may have associated metadata, e.g., the information about where a JPEG photograph was taken. Blobs allow a Windows Azure role instance interact with persistent storage as if it were a local NTFS[6] file system. Queues enable web role instances to communicate asynchronously with Worker role instances.

The Microsoft Azure platform currently does not provide or support any distributed parallel computing frameworks, such as *MapReduce, Dryad* or *MPI*, other than the support for implementing basic queue-based job scheduling [208].

2.7 CLOUD STORAGE DIVERSITY AND VENDOR LOCK-IN

The short history of cloud computing shows that cloud services may be unavailable for short, or even for extended periods of time. Such an interruption of service is likely to impact negatively the organization and possibly diminish, or cancel completely, the benefits of utility computing for that organization. The potential for permanent data loss in case of a catastrophic system failure poses an even greater danger.

Last but not least, the single vendor may decide to increase the prices for service and charge more for computing cycles, memory, storage space, and network bandwidth than other cloud service providers. The alternative in this case is switching to another cloud service provider. Unfortunately, this solution could be very costly due to the large volume of data to be transferred from the old to the new provider. Transferring terabytes or possibly petabytes of data over the network takes a fairly long time and incurs substantial charges for the network bandwidth.

A solution to guarding against the problems posed by the vendor lock-up is to replicate the data to multiple cloud service providers. The straightforward replication is very costly and, at the same time, poses technical challenges. The overhead to maintain data consistency could drastically affect the performance of the virtual storage system consisting of multiple full replicas of the organization's data spread over multiple vendors. Another solution could be based on an extension of the design principle of a RAID-5 system used for reliable data storage.

A RAID-5 system uses block-level stripping with distributed parity over a disk array, see Figure 2.8A; the disk controller distributes sequential blocks of data to the physical disks and computes a parity block by bit-wise XOR-ing the data blocks. The parity block is written on a different disk for each file to avoid the bottleneck possible when all parity blocks are written to a dedicated disk, as it is done in case of RAID-4 systems. This technique allows us to recover the data after a single disk loss. For example, if Disk 2 in Figure 2.8 is lost then we still have all the blocks of the third file, $c1$, $c2$, and

[6]NTFS (New Technology File System) is the standard file system of the Microsoft Windows operating system starting with Windows NT 3.1, Windows 2000, and Windows XP.

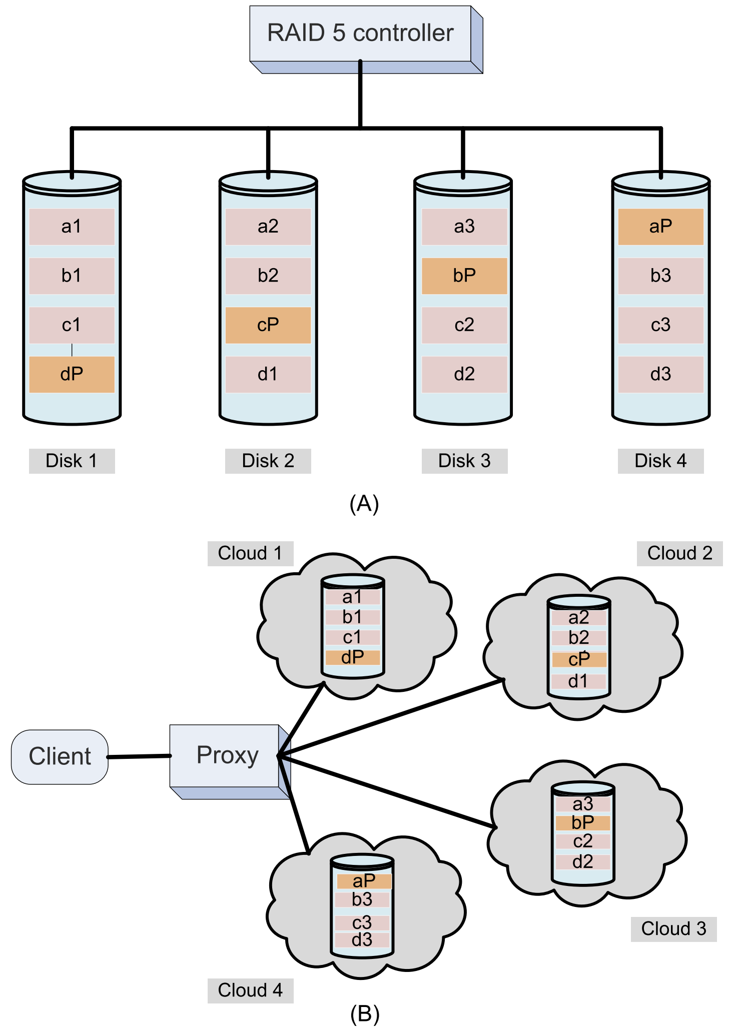


FIGURE 2.8

(A) A $(3,4)$ RAID-5, configuration where individual blocks are stripped over three disks and a parity block is added; the parity block is constructed by XOR-ing the data blocks, e.g., $aP = a1$ XOR $a2$ XOR $a3$. The parity blocks are distributed among the four disks, aP is on disk 4, bP on disk 3, cP on disk 2, and dP on disk 1. (B) A system which strips data across four clouds; the proxy provides transparent access to data.

$c3$ and we can recover the missing blocks for the others as follows:

$$\begin{aligned} a2 &= (a1) \text{ XOR } (aP) \text{ XOR } (a3) \\ b2 &= (b1) \text{ XOR } (bP) \text{ XOR } (b3) \\ d1 &= (dP) \text{ XOR } (d2) \text{ XOR } (d3) \end{aligned} \quad . \tag{2.1}$$

Obviously, we can also detect and correct errors in a single block using the same procedure. The RAID controller allows also parallel access to data (for example, the blocks $a1$, $a2$, and $a3$ can be read and written concurrently) and it can also aggregate multiple write operations to improve performance.

The system in Figure 2.8B strips data across four clusters. The access to data is controlled by a proxy which carries out some of the functions of a RAID controller, as well as authentication and other security related functions. The proxy ensures *before-and-after* atomicity as well as *all-or-nothing* atomicity for data access; the proxy buffers the data, possibly converts the data manipulation commands, optimizes the data access, e.g., aggregates multiple write operations, converts data to formats specific to each cloud, and so on.

This elegant idea immediately raises several questions: How does the response time of such a scheme compare with the one of a single storage system? How much overhead is introduced by the proxy? How could this scheme avoid a single point of failure, the proxy? Are there standards for data access implemented by all vendors?

An experiment to answer some of these questions is reported in [6]; their RACS system uses the same data model and mimics the interface to AWS S3. The S3 system, discussed in Section 2.3, stores the data in *buckets*, each bucket being a flat namespace with *keys* associated with *objects* of arbitrary size but less than 5 GB. The prototype implementation discussed in [6] led the authors to conclude that the costs increases and the performance penalties of the RACS system are relatively minor. The paper also suggests an implementation to avoid the single point of failure by using several proxies. Then the system is able to recover from the failure of a single proxy; clients are connected to several proxies and can access the data stored on multiple clouds.

It remains to be seen if such a solution is feasible in practice for organizations with a very large volume of data, given the limited number of cloud storage providers and the lack of standards for data storage. A basic question is if it makes sense to trade basic tenets of cloud computing, such as simplicity and homogeneous resources controlled by a single administrative authority, for increased reliability and for freedom from vendor lock-in.

This brief discussion hints to the need for standardization and for scalable solutions, two of the many challenges faced by cloud computing in the near future. The pervasive nature of scalability dominates all aspects of cloud management and cloud applications. Solutions performing well on small systems do it no longer when the system scale increases by one or more orders of magnitude. Experiments with small test bed systems produce inconclusive results. The only alternative is to conduct intensive simulations to prove, or disprove, the advantages of a particular algorithm for resource management, or the feasibility of a particular data-intensive application.

We can also conclude that cloud computing poses challenging problems to service providers and to users; the service providers have to develop strategies for resource management subject to quality of service and cost constraints as discussed in Chapter 9. At the same time, the cloud application developers have to be aware of the limitations of the cloud computing model.

2.8 CLOUD COMPUTING INTEROPERABILITY; THE INTERCLOUD

Cloud interoperability could alleviate the concerns that users become hopelessly dependent on a single cloud service provider, the so called vendor lock-in, discussed in Section 2.7. It seems natural to ask the question if an Intercloud, a "cloud of clouds," a federation of clouds that cooperate to provide a better user experience, is technically and economically feasible. The Internet is a network of networks hence, it appears that an Intercloud could be plausible [62–64].

Closer scrutiny shows that the extension of the concept of interoperability from networks to clouds is far from trivial. A network offers one high-level service, the transport of digital information from a source, a host outside a network to a destination, another host, or another network that can deliver the information to its final destination. This transport of information through a network of networks is feasible because before the Internet was born, agreements on basic questions were reached: (a) how to uniquely identify the source and the destination of the information; (b) how to navigate through a maze of networks; and (c) how to actually transport the data between a source and a destination. The three elements on which agreements were reached are, respectively, the IP address, the IP protocol, and transport protocols such as TCP and UDP.

The situation is quite different in cloud computing. First, there are no standards for either storage or processing; second, the clouds we have seen so far are based on different delivery models, SaaS, PaaS, and IaaS. Moreover, the set of services supported by each of these delivery models is not only large, but it is open; new services are offered every few months. For example, in October 2012 Amazon announced new services, the AWS GovCloud (US).

The question if Cloud Service Providers are willing to cooperate to build up an Intercloud is open. Some CSPs may think that they have a competitive advantage due to the uniqueness of the added value of their services. Thus, exposing how they store and process information may adversely affect their business. Moreover, no CSP will be willing to change its internal operation, so a first question is if a Intercloud could be built under these conditions.

Following the concepts borrowed from the Internet, a federation of clouds that does not dictate the internal organization or the structure of a cloud, but only the means to achieve cloud interoperability is feasible. Nevertheless, building such an infrastructure seems to be a formidable task. First, we need a set of standards for interoperability covering items such as: naming, addressing, identity, trust, presence, messaging, multicast, and time. Indeed, we need common standards for identifying all the objects involved, the means to transfer, store, and process information, and we also need a common clock to measure the time between two events.

An Intercloud would then require the development of an *ontology*[7] for cloud computing. Then each cloud service provider would have to create a description of all resources and services using this ontology. Due to the very large number of systems and services the volume of information provided by individual cloud service providers would be so large, that a distributed database, not unlike the Domain Name Service (DNS) would have to be created and maintained. According to [62] this vast amount of information would be stored in Intercloud *root* nodes, analogous with the root nodes of the DNS.

Each cloud would then require an interface, a so-called Intercloud *exchange*, to translate the common language describing all objects and actions included in a request originating from another cloud in terms of its internal objects and actions. To be more precise, a request originating in one cloud would have to be translated from the internal representation in that cloud to a common representation based on the shared ontology and then, at the destination, it should be translated into an internal representation that can be acted upon by the destination cloud. This raises immediately the question of efficiency

[7]An ontology provides the means for knowledge representation within a domain. It consists of a set of domain concepts and the relationships among the concepts.

and performance. This question cannot be fully answered now, as an Intercloud exists only on paper, but there is little doubt that the performance will be greatly affected.

Security is a major concern for cloud users and an Intercloud could only create new threats. The primary concern is that tasks will cross from one administrative domain to another and that sensitive information about the tasks and user could be disclosed during this migration. A seamless migration of tasks in an Intercloud requires a well thought out trust model.

The Public-Key Infrastructure (PKI),[8] an all-or-nothing trust model, is not adequate for an Intercloud where the trust must be nuanced. A nuanced model for handling digital certificates means that one cloud acting on behalf of a user may grant access to another cloud to read data in storage, but not to start new instances.

The solution advocated in [63] for trust management is based on dynamic *trust indexes* that can change in time. The Intercloud roots play the role of Certificate Authority, while the Intercloud exchanges determine the trust indexes between clouds.

Encryption must be used to protect the data in storage and in transit in the Intercloud. The OASIS[9] Key Management Interoperability Protocol (KMIP)[10] is proposed for key management.

In summary, the idea of an Intercloud opens up a wide range of interesting research topics. The practicality of the concepts can only be discussed after the standardization efforts under way at NIST will bear fruits.

2.9 SERVICE-LEVEL AGREEMENTS AND COMPLIANCE-LEVEL AGREEMENTS

A Service Level Agreement (SLA) is a negotiated contract between two parties, the customer and the service provider; the agreement can be legally binding or informal and specifies the services that the customer receives, rather than how the service provider delivers the services. The objectives of the agreement are:

- Identify and define the customer's needs and constraints including the level of resources, security, timing, and quality of service.
- Provide a framework for understanding; a critical aspect of this framework is a clear definition of classes of service and the costs.
- Simplify complex issues; for example, clarify the boundaries between the responsibilities of the clients and those of the provider of service in case of failures.
- Reduce areas of conflict.
- Encourage dialog in the event of disputes.
- Eliminate unrealistic expectations.

[8]PKI is a model to create, distribute, revoke, use, and store digital certificates. It involves several components: (1) The Certificate Authority (CA) binds public keys to user identities in a given domain. (2) The third-party Validation Authority (VA) guarantees the uniqueness of the user identity. (3) The Registration Authority (RA) grantees that the binding of the public key to an individual cannot be challenged, the so called *non-repudiation*.
[9]OASIS stands for Advancing Open Standards for the Information Society.
[10]The KMIP Specification version 1.0 is available at http://docs.oasis-open.org/kmip/spec/v1.0/kmip-spec-1.0.html.

An SLA records a common understanding in several areas: (i) services, (ii) priorities, (iii) responsibilities, (iv) guarantees, and (v) warranties. An agreement usually covers: services to be delivered, performance, tracking and reporting, problem management, legal compliance and resolution of disputes, customer duties and responsibilities, security, handling of confidential information, and termination.

Each area of service in cloud computing should define a "target level of service" or a "minimum level of service" and specify the levels of availability, serviceability, performance, operation, or other attributes of the service, such as billing; penalties may also be specified in the case of non-compliance with the SLA. It is expected that any Service-Oriented Architecture (SOA) will eventually include middleware supporting SLA management; the Framework 7 project supported by the European Union is researching this area, see http://sla-at-soi.eu/.

The common metrics specified by an SLA are service-specific. For example, the metrics used by a *call center* usually are:

1. The abandonment rate – percentage of calls abandoned while waiting to be answered.
2. The average speed to answer – average time before the service desk answers a call.
3. The time service factor – percentage of calls answered within a definite time frame.
4. The first-call resolution – percentage of incoming calls that can be resolved without a callback.
5. The turnaround time – time to complete a certain task.

There are two well-differentiated phases in SLA management: the negotiation of the contract and the monitoring of its fulfillment in real-time. In turn, automated negotiation has three main components:

- The *object of negotiation* which define the attributes and constraints under negotiation.
- The *negotiation protocols* which describe the interaction between negotiating parties.
- The *decision models* responsible for processing proposals and generating counter proposals.

The concept of compliance in cloud computing is discussed in [73] in the context of the user ability to select a provider of service; the selection process is subject to customizable compliance with user requirements such as security, deadlines, and costs. The authors propose an infrastructure called *Compliant Cloud Computing* (C3) consisting of: (i) a language to express user requirements and the Compliance Level Agreements (CLA), and (ii) the middleware for managing CLAs.

The web Service Agreement Specification (WS-Agreement) [31] uses an XML-based language to define a protocol for creating an agreement using a pre-defined template with some customizable aspects; it only supports one-round negotiation without counter proposals. A policy-based framework for automated SLA negotiation for a virtual computing environment is described in [530].

2.10 RESPONSIBILITY SHARING BETWEEN A USER AND THE CSP

After reviewing cloud services provided by Amazon, Google, and Microsoft we are in a better position to understand the differences between SaaS, IaaS, and PaaS. There is no confusion about SaaS, the service provider supplies both the hardware and the application software; the user has direct access to these services through a web interface and has no control on cloud resources. Typical examples are

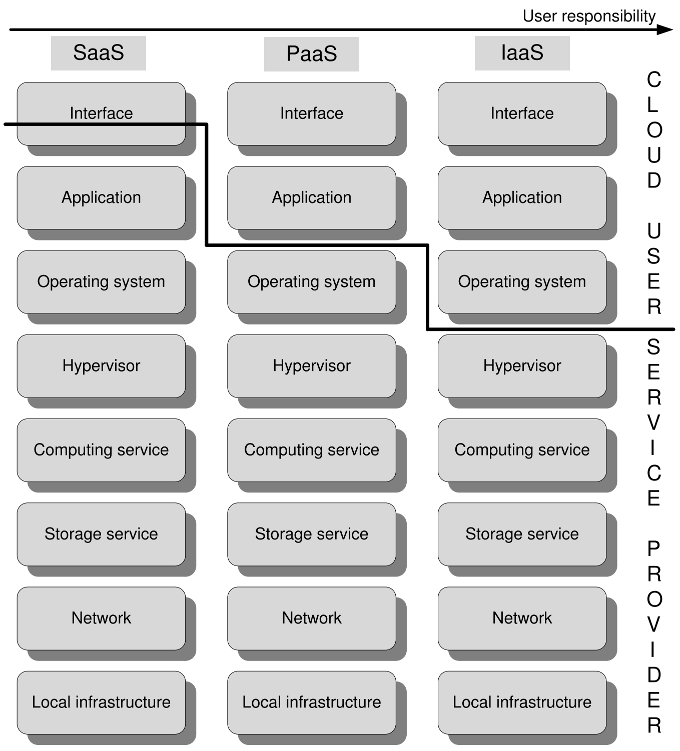


FIGURE 2.9

The limits of responsibility between a cloud user and the cloud service provider.

Google with Gmail, Google docs, Google calendar, Google Groups, and Picasa and Microsoft with the Online Services.

In the case of IaaS, the service provider supplies the hardware (servers, storage, networks), and system software (operating systems, databases); in addition, the provider ensures system attributes such as security, fault-tolerance, and load balancing. The representative of IaaS is AWS.

PaaS provides only a platform including the hardware and system software such as operating systems and databases; the service provider is responsible for system updates, patches, and the software maintenance. PaaS does not allow any user control on the operating system, security features, or the ability to install applications. Typical examples are Google App Engine, Microsoft Azure, and Force.com provided by Salesforce.com.

The level of users control over the system is different in IaaS versus PaaS; IaaS provides total control, PaaS typically provides no control. Consequently, IaaS incurs administration costs similar to a traditional computing infrastructure while these costs are virtually zero for PaaS.

It is critical for a cloud user to carefully read the service level agreement and to understand the limitations of the liability a cloud provider is willing to accept. In many instances the liabilities do not apply to damages caused by a third party or to failures attributed either to customer's hardware and software, or to hardware and software from a third party.

The limits of responsibility between the cloud user and the cloud service provider are different for the three service delivery models as we can see in Figure 2.9. In the case of SaaS the user is partially responsible for the interface; the user responsibility increases in the case of PaaS and includes the interface and the application. In the case of IaaS the user is responsible for all the events occurring in the virtual machine running the application.

For example, if a distributed denial of service attack (DDoS) causes the entire IaaS infrastructure to fail, the cloud service provider is responsible for the consequences of the attack. The user is responsible if the DDoS affects only several instances including the ones running the user application. A recent posting describes the limits of responsibility illustrated in Figure 2.9 and argues that security should be a major concern for IaaS cloud users, see http://www.sans.org/cloud/2012/07/19/can-i-outsource-my-security-to-the-cloud.

2.11 USER EXPERIENCE

There are a few studies of user experience based on a large population of cloud computing users. An empirical study of the experience of a small group of users of the Finish Cloud Computing Consortium is reported in [385]. The main user concerns are: security threats; the dependence on fast Internet connection; forced version updates; data ownership; and user behavior monitoring. All users reported that trust in the cloud services is important, two thirds raised the point of fuzzy boundaries of liability between cloud user and the provider, about half did not fully comprehend the cloud functions and its behavior, and about one third were concerned about security threats.

The security threats perceived by this group of users are: (i) abuse and villainous use of the cloud; (ii) APIs that are not fully secure; (iii) malicious insiders; (iv) account hijacking; (v) data leaks; and (vi) issues related to shared resources. Identity theft and privacy were a major concern for about half of the users questioned; availability, liability and data ownership and copyright was raised by a third of respondents.

The suggested solutions to these problems are: Service Level Agreements and tools to monitor usage should be deployed to prevent the abuse of the cloud; data encryption and security testing should enhance the API security; an independent security layer should be added to prevent threats caused by malicious insiders; strong authentication and authorization should be enforced to prevent account hijacking; data decryption in a secure environment should be implemented to prevent data leakage; and compartmentalization of components and firewalls should be deployed to limit the negative effect of resource sharing.

A broad set of concerns identified by the NIST working group on cloud security includes:

- Potential loss of control/ownership of data.
- Data integration, privacy enforcement, data encryption.
- Data remanence after de-provisioning.
- Multi tenant data isolation.
- Data location requirements within national borders.
- Hypervisor security.

Table 2.6 The reasons driving the decision to use public clouds.

Reason	Percentage who agree
Improved system reliability and availability	50%
Pay only for what you use	50%
Hardware savings	47%
Software license saving	46%
Lower labor costs	44%
Lower maintenance costs	42%
Reduced IT support needs	40%
Ability to take advantage of the latest functionality	40%
Less pressure on internal resources	39%
Solve problems related to updating/upgrading	39%
Rapid deployment	39%
Ability to scale up resources to meet the needs	39%
Ability to focus on core competencies	38%
Take advantage of the improved economics of scale	37%
Reduced infrastructure management needs	37%
Lower energy costs	29%
Reduced space requirements	26%
Create new revenue streams	23%

- Audit data integrity protection.
- Verification of subscriber policies through provider controls.
- Certification/Accreditation requirements for a given cloud service.

A 2010 study conducted by IBM [246] aims to identify barriers for public and private cloud adoption. The study is based on interviews with more than 1 000 individuals responsible for IT decision making around the world. 77% of the respondents cited cost savings as the key argument in favor of public cloud adoption, though only 30% of them believed that public clouds are "very appealing or appealing" for their line of business, versus 64% for private clouds, and 34% for hybrid ones.

The reasons driving the decision to use public clouds and percentage of responders who considered each element as critical are shown in Table 2.6. In view of the high energy costs for operating a data center discussed in Section 9.2 it seems strange that only 29% of the respondents seem to be concerned about lower energy costs.

The top workloads mentioned by the users involved in this study are: data mining and other analytics (83%), application streaming (83%), help desk services (80%), industry specific applications (80%), and development environments (80%).

The study also identified workloads that are not good candidates for migration to a public cloud environment:

- Sensitive data such as employee and health care records.
- Multiple co-dependent services, e.g., online transaction processing.
- Third party software without cloud licensing.

- Workloads requiring auditability and accountability.
- Workloads requiring customization.

Such studies help identify the concerns of potential cloud users and the critical issues for cloud research.

2.12 SOFTWARE LICENSING

Software licensing for cloud computing is an enduring problem without a universally accepted solution at this time. The license management technology is based on the old model of computing centers with licenses given on the basis of named users or as a site license; this licensing technology developed for a centrally managed environment cannot accommodate the distributed service infrastructure of cloud computing or of Grid computing.

Only recently IBM has reached an agreement allowing some of its software products to be used on EC2. Also, MathWorks developed a business model for the use of MATLAB in Grid environments [85]. The SaaS deployment model is gaining acceptance as it allows users to pay only for the services they use.

There is a significant pressure to change the traditional software licensing and find non-hardware based solutions for cloud computing; the increased negotiating power of the users coupled with the increased software piracy has renewed interest in alternative schemes such as those proposed by the SmartLM research project (http://www.smartlm.eu). SmartLM license management requires a complex software infrastructure involving Service Level Agreement, negotiation protocols, authentication, and other management functions.

A commercial product based on the ideas developed by this research project is *elasticLM* which provides license and billing Web-based services [85]. The architecture of the elasticLM license service has several layers: co-allocation, authentication, administration, management, business, and persistency. The authentication layer authenticates communications between the license service and the billing service as well as the individual applications; the persistence layer stores the usage records; the main responsibility of the business layer is to provide the licensing service with the licenses prices; the management coordinates different components of the automated billing service.

When a user requests a license from the license service, the terms of the license usage are negotiated and they are part of a Service Level Agreement document; the negotiation is based on application-specific templates and the license cost becomes part of the SLA. The SLA describes all aspects of resource usage, including the ID of application, duration, number of processors, and guarantees, such as the maximum cost and deadlines. When multiple negotiation steps are necessary, the WS-Agreement Negotiation protocol is used.

To understand the complexity of the issues related to software licensing, we point out some of the difficulties related to authorization. To verify the authorization to use a license, an application must have the certificate of an authority. This certificate must be available locally to the application because the application may be executed in an environment with restricted network access; this opens the possibility for an administrator to hijack the license mechanism by exchanging the local certificate.

2.13 ENERGY USE AND ECOLOGICAL IMPACT OF CLOUD COMPUTING

The discussion of the cloud infrastructure cannot be concluded without an analysis of the energy used for cloud computing and its impact on the environment [278]. Indeed, the energy consumption required by different types of human activities is partially responsible for the greenhouse gas emissions. According to a recent study [408], the greenhouse gas emission due to the data centers is estimated to increase from 116×10^6 tonnes of CO_2 in 2007 to 257 tonnes in 2020 due primarily to increased consumer demand [506].

The energy consumption of large-scale data centers and their costs for energy used for computing and networking and for cooling are significant now and are expected to increase substantially in the future. In 2013, the data centers in the US consumed an estimated 91 billion kWh, the annual output of 34 power each producing 500-megawatts. The consumption is projected to increase to roughly 140 billion kilowatt-hours annually by 2020, the equivalent annual output of 50 power plants and cost $13 billion annually in electricity bills. As a result 100 million metric tons of carbon will be generated each year according to the National Resource Defense Council, see https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

The energy consumption of data centers and the network infrastructure is predicted to reach 10 300 TWh/year[11] in 2030, based on 2010 levels of efficiency [408]. These increases are expected in spite of the extraordinary reduction in energy requirements for computing activities; over the past 30 years the energy efficiency per transistor on a chip has improved by six orders of magnitude.

Communication to and from cloud data centers is also responsible for a significant fraction of energy consumption. The support for network centric content consumes a very large fraction of the network bandwidth; according to the CISCO VNI forecast, consumer traffic was responsible for around 80% of bandwidth use in 2009, and is expected to grow at a faster rate than business traffic. Data intensity for different activities ranges from 20 MB/minute for HDTV streaming, to 10 MB/minute for Standard TV streaming, 1.3 MB/minute for music streaming, 0.96 MB/minute for Internet radio, 0.35 MB/minute for Internet browsing, and 0.0025 MB/minute for Ebook reading [408].

The same study reports that if the energy demand for bandwidth is 4 Watts-hour per Megabyte[12] and if the demand for network bandwidth is 3.2 Gbytes/day/person or 2 570 Exabytes/year for the entire world population, then the energy required for this activity will be 1,175 GW. These estimates do not count very high bandwidth applications that may emerge in the future, such as 3D-TV, personalized immersive entertainment, such as Second Life, or massively multi-player online games.

Now most of the power for large data centers, including cloud computing data centers, comes from power stations burning fossil fuels such as coal and gas. In recent years the contribution of solar, wind, geothermal and other renewable energy sources has steadily increased. Environmentally Opportunistic Computing is a macro-scale computing idea that exploits the physical and temporal mobility of modern computer processes. A prototype called a Green Cloud is described in [527].

A conservative estimation of the electric energy used now by the Information and Communication Technology (ICT) ecosystem is about 1,500 TWh of energy, 10% of the electric energy generated in the entire world. This includes energy used for manufacturing electronic components and computing

[11] One TWh (Tera Watt Hour) is equal to 10^{12} Wh.

[12] In the US, in 2006, the energy consumed to download data from a data center across the Internet was in the range of 9 to 16 Watts hour per Megabyte.

and communication systems, for powering, heating, and cooling IT systems, and for recycling and disposing of obsolete IT equipment. ICT energy consumption equals the total electric energy used for illumination in 1985 and represents the total electric energy generated in Japan and Germany.

Reduction of energy consumption thus, of the carbon footprint of cloud related activities, is increasingly more important for the society. Indeed, more and more applications run on clouds and cloud computing uses more energy than many other human-related activities. Reduction of the carbon footprint can only be achieved through a comprehensive set of technical efforts. The hardware of the cloud infrastructure has to be refreshed periodically and new and more energy efficient technologies have to be adopted; the resource management software has to pay more attention to energy optimization.

2.14 MAJOR CHALLENGES FACED BY CLOUD COMPUTING

Cloud computing inherits some of the challenges of parallel and distributed computing discussed in Chapter 4 and, at the same time, it faces major challenges of its own. The specific challenges differ for the three cloud delivery models, but in all cases the difficulties are created by the very nature of utility computing which is based on resource sharing and resource virtualization and requires a different trust model than the ubiquitous user-centric model we have been accustomed to for a very long time.

The most significant challenge is security; gaining the trust of a large user base is critical for the future of cloud computing. It is unrealistic to expect that a public cloud will provide a suitable environment for all applications. Highly sensitive applications related to the management of the critical infrastructure, healthcare applications, and others will most likely be hosted by private clouds. Many real-time applications will probably still be confined to private clouds. Some application may be best served by a hybrid cloud setup; such applications could keep sensitive data on a private cloud and use a public cloud for some of the processing.

The SaaS model faces similar challenges as other online services required to protect private information such as financial or healthcare services. In this case a user interacts with cloud services through a well-defined interface thus, in principle it is less challenging for the provide of service to close some of the attack channels. Still, such services are vulnerable to denial of service attacks and the users are fearful of malicious insiders. Data in storage is most vulnerable to attacks, so a special attention should be devoted to the protection of storage servers. Data replication necessary to ensure continuity of service in case of storage system failure increases vulnerability. Data encryption may protect data in storage but eventually data must be decrypted for processing and then it is exposed to attacks.

The IaaS is by far the most challenging to defend against attacks; indeed, an IaaS user has considerably more degrees of freedom than allowed by the other two cloud delivery models. An additional source of concern is that the considerable resources of a cloud could serve as the host to initiate attacks against the networking and the computing infrastructure.

Virtualization is a critical design option for this model, but it exposes the system to new sources of attacks. The trusted computing base (TCB) of a virtual environment includes not only the hardware and the hypervisor, but also the management operating system. As we shall see in Section 11.9 the entire state of a VM can be saved to a file to allow migration and recovery, both highly desirable operations; yet, this possibility challenges the strategies to bring the servers belonging to an organization to a desirable and stable state. Indeed, an infected VM can be inactive when the systems are cleaned up and an infected VM can wake up later and infect other systems. This is another example of the deep intertwining of desirable and undesirable effects of virtualization, a defining attribute of cloud computing technologies.

The next major challenge is related to the resource management on a cloud. Any systematic, rather than ad hoc resource management strategy, requires the existence of controllers tasked to implement several classes of policies: admission control, capacity allocation, load balancing, energy optimization, and, last but not least, to provide Quality of Service (QoS) guarantees.

To implement these policies the controllers need accurate information about the global state of the system. Determining the state of a complex system with 10^6 serves or more, distributed over a large geographic area, is not feasible. Indeed, the external load, as well as the state of individual resources changes very rapidly. Thus, controllers must be able to function with incomplete or approximate knowledge of the system state.

It seems reasonable to expect that such a complex system can only function based on self-management principles. But self-management and self-organization raise the bar for the implementation of logging and auditing procedures critical for the security and trust in a provider of cloud computing services. Under self-management it becomes next to impossible to identify the reasons why a certain action that resulted in a security breach was taken.

The last major challenge we want to address is related to interoperability and standardization. Vendor lock-in, the fact that a user is tied to a particular cloud service provider, is a major concern for cloud users, see Section 2.7. Standardization would support interoperability and thus, alleviate some of the fears that a service critical for a large organization may not be available for an extended period of time. But imposing standards at a time when a technology still evolves is not only challenging, but can be counterproductive, as it may stiffen innovation.

From this brief discussion the reader should realize the complexity of the problems posed by cloud computing and understand the wide range of technical and social problems raised by cloud computing. If successful, the effort to migrate the IT activities of many government agencies to public and private clouds will have a lasting effect on cloud computing. Cloud computing can have a major impact on education, but we have seen little effort in this area.

2.15 FURTHER READINGS

A good starting point for understanding the major issues in cloud computing is the 2009 paper "Above the clouds: a Berkeley view of cloud computing" [37]. Content distribution systems are discussed in [511]. The BOINC platform is presented in [32].

Ethical issues in cloud computing are discussed in [485]. A recent book covers topics in the area of distributed systems, including grids, peer-to-peer systems, and clouds [244]. The standardization effort at NIST is described by a wealth of documents [361–369] on the web site http://collaborate.nist.gov.

Information about cloud computing at Amazon, Google, Microsoft, HP, and Oracle is available from the following sites:

- Amazon: http://aws.amazon.com/ec2/
- Google: http://code.google.com/appengine/
- Microsoft: http://www.microsoft.com/windowsazure/
- HP: http://www.hp.com/go/cloud
- Oracle: http://cloud.oracle.com.

Insights on Google clouds are provided by [520] and [525]. [458] and [459] cover comparisons among cloud service providers. A white paper on SLA specification can be found at http://www.itsm.info/SLA*.pdf, a toolkit at http://www.service-level-agreement.net and a web service level agreement (WSLA) at http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.

Energy use and ecological impact are discussed in [7], [225], [408], [497], [506], and [336]. Information about the OpenStack, an open source cloud operating system, is available from the project site http://www.openstack.org/.

The Intercloud is discussed in several papers including [62–64]. Google Docker is presented in [199]. Enterprise migration to IaaS is analyzed in [268]. The sustainability of cloud computing is discussed in [389]. Data portability in cloud computing software testing in the cloud, and the state of the cloud in 2010 are analyzed in [413], [423], and [431], respectively.

[427] presents open source cloud computing tools and [461] discusses software debuggers. Cloud workload migration is the subject of [510]. The cost of AWS spot instances is analyzed in [537] and fault-tolerant middleware for clouds are presented in [549].

2.16 EXERCISES AND PROBLEMS

Problem 1. The list of desirable properties of a large-scale distributed system includes transparency of access, location, concurrency, replication, failure, migration, performance, and scaling. Analyze how each one of these properties applies to AWS.

Problem 2. Compare the three cloud computing delivery models, SaaS, PaaS, and IaaS, from the point of view of the application developers and users. Discuss the security and the reliability of each one of them. Analyze the differences between the PaaS and the IaaS.

Problem 3. Compare the Oracle Cloud offerings (see https://cloud.oracle.com) with the cloud services provided by Amazon, Google, and Microsoft.

Problem 4. Read the IBM report [246] and discuss the workload preferences for private and public clouds and the reasons for the preferences.

Problem 5. Many organizations operate one or more computer clusters and contemplate the migration to private clouds. What are the arguments for and against such an effort?

Problem 6. Evaluate the SLA toolkit at http://www.service-level-agreement.net/. Is the interactive guide useful, what does it miss? Does the SLA template include all clauses that are important in your view, what is missing? Are the examples helpful?

Problem 7. Software licensing is a major problem in cloud computing. Discuss several ideas to prevent an administrator from hijacking the authorization to use a software licence.

Problem 8. Annotation schemes are widely used by popular services such as Flickr photo-sharing service which supports annotation of photos. Sketch the organization of a cloud service used for sharing medical x-ray, tomography, CAT-scan and other medical images and discuss the main challenges for its implementation.

Problem 9. An organization debating whether to install a private cloud or to use a public cloud, e.g., the AWS, for its computational and storage needs, asks your advice. What information will you require to base your recommendation on, and how will you use each one of the following items: (a) the description of the algorithms and the type of the applications the organization will run; (b) the system software used by these applications; (c) the resources needed by each application; (d) the size of the user population; (e) the relative experience of the user population; (f) the costs involved?

Problem 10. A university is debating the question in Problem 9. What will be your advice and why? Should software licensing be an important element of the decision?

SECTION 1

CHAPTER

CONCURRENCY IN THE CLOUD 3

Leslie Lamport began his 2013 Turing Lecture dedicated to Edsger Dijkstra [293] with the observation that concurrency has been known by several names: “I don’t know if concurrency is a science, but it is a field of computer science. What I call concurrency has gone by many names, including parallel computing, concurrent programming, and multiprogramming. I regard distributed computing to be part of the more general topic of concurrency.”

Is there a distinction between concurrency and parallel processing? According to some, concurrency describes the necessity that multiple activities take place at the same time, while parallel processing implies a solution, where there are several processors capable of carrying out the computations required by these activities at the same time, i.e., concurrently. Concurrency emphasizes cooperation and interference among activities, while parallel processing aims to shorten the completion time of the set of activities and it is hindered by cooperation and activity interference.

Execution of multiple activities in parallel can proceed either quasi-independently, or tightly coordinated with an explicit communication pattern. In either case some form of communication is necessary for coordination of concurrent activities. Coordination complicates the description of a complex activity as it has to characterize the work done by individual entities working in concert, as well as the interactions among them.

Communication affects the overall efficiency of concurrent activities and could significantly increase the completion time of a complex task and even hinder the completion of the task. Furthermore, communication requires prior agreement on the communication discipline described by a communication protocol. Measures to ensure that communication problems do not affect the overall orchestration required for the completion of the task are also necessary.

The practical motivations for concurrent execution of computer applications is to overcome the physical limitations of one computer system by distributing the workload to several systems and getting results faster. Concurrency is at the heart of cloud computing, the large workloads generated by many applications run concurrently on multiple instances taking advantage of resources only available on computer clouds.

The chapter starts with an overview of concurrent execution of communicating processes in Section 3.2. Computational models including BSP, a bridging hardware-software model designed to avoid logarithmic losses of efficiency in parallel processing and its version for a multicore computational model are covered in Sections 3.3 and 3.4. Petri Nets, discussed in Section 3.5, are intuitive models able to describe concurrency and conflict.

The concept of process state, critical for understanding concurrency, is covered in Section 3.6. Many functions of a computer cloud require information about process state. For example, controllers for cloud resource management discussed in Chapter 9 require accurate state information. Process coordination is analyzed in Section 3.7 while Section 3.8 presents logical clocks and message delivery rules in an attempt to bridge the gap between the abstractions used to analyze concurrency and the physical systems.

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00004-2

The concept of consistent cuts and distributed snapshots are at the heart of *checkpoint-restart* procedures for long-lasting computations. Checkpoints are taken periodically in anticipation of the need to restart a software process when one or more systems fail; when a failure occurs the computation is restarted from the last checkpoint rather than from the beginning. These concepts are discussed in Sections 3.9 and 3.10. Atomic actions, consensus protocols and load balancing are covered in Sections 3.11, 3.12 and 3.13, respectively. Finally, Section 3.14 presents multithreading and concurrency in Java and FlumeJava.

This chapter reviews theoretical foundations of important algorithms at the heart of system and application software. The concepts introduced in the next sections help us better understand cloud resource management policies and the mechanisms implementing these policies. For a deeper understanding of the many subtile concepts related to concurrency the reader should consult the classical references discussed at the end of the chapter.

3.1 ENDURING CHALLENGES; CONCURRENCY AND CLOUD COMPUTING

Concurrency is a very broad subject and in this chapter we restrict the discussion to topics closely related to cloud computing. We start with a gentle introduction to some of the enduring challenges posed by concurrency and coordination. Coordination starts with the resource allocation to the entities carrying out individual tasks and the distribution of the workload among them. This initial phase is followed by communication during the execution of the tasks, and finally, by the assembly of individual results. Coordination is ubiquitous in our daily life and the lack of coordination has implications on the results. For example, Figure 3.1 shows that lack of coordination and disregard of the rules regarding the management of shared resources lead to traffic deadlock, an unfortunate phenomena we often experience.

Synchronization is another defining aspect of concurrency. The importance of synchronization is best illustrated by the famous dining philosophers problem, see Figure 3.2. Five philosophers sitting at a table alternately think and eat. A philosopher needs the two chopsticks placed left and right of her plate to eat. After finishing eating she must place the chopsticks back on the table to give a chance to her left and right neighbors to eat. The problem is non-trivial, the naive solution when each philosopher picks up the chopstick to the left, and waits for the one to the right to become available, or vice versa, fails because it allows the system to reach a deadlock state, in which no progress is possible. Deadlock would lead to philosopher starvation, a situation that must be avoided.

This problem captures critical aspects of concurrency such as mutual exclusion and resource starvation discussed in this chapter. Edsger Dijkstra proposed the following solution formulated in terms of resources in general:

- Assign a partial order to the resources.
- Impose the rule that resources must be requested in order.
- Impose the rule that no two resources unrelated by order will ever be used by a single unit of work at the same time.

In the dining philosopher problem, the resources are the chopsticks, numbered 1 through 5 and each unit of work, i.e., philosopher, will always pick up the lower-numbered chopstick first, and then the

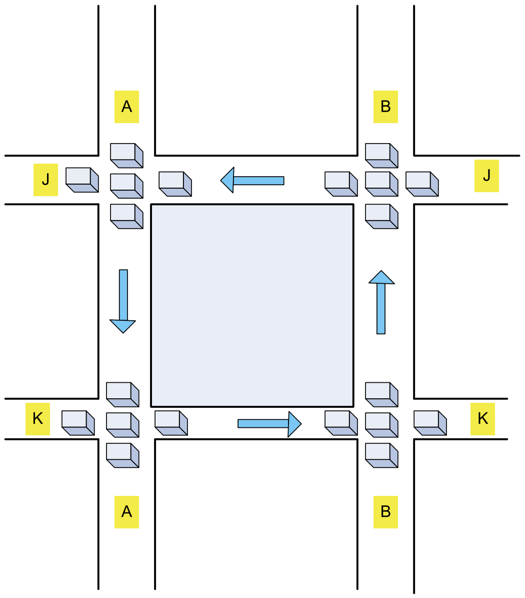


FIGURE 3.1

The consequences of the lack of coordination. The absence of traffic lights or signs in intersections causes a traffic jam. Blocking the intersections, a shared resource for North–South and East–West traffic flows, contributes to the deadlocks.

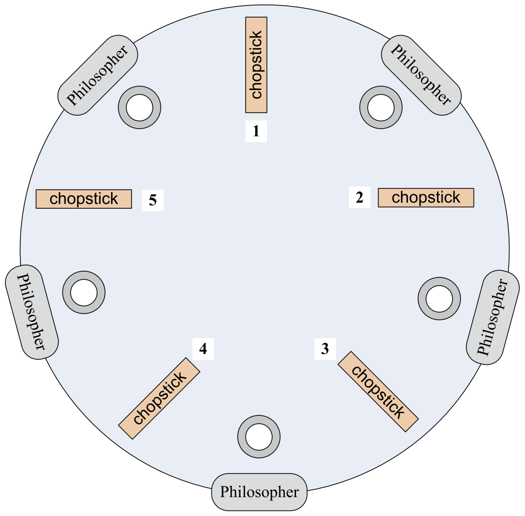


FIGURE 3.2

Dining philosophers problem. To avoid deadlock Dijkstra's solution requires numbering of chopsticks and two additional rules.

higher-numbered one next to her. The order in which each philosopher puts down the chopsticks does not matter.

This solution avoids starvation. If four of the five philosophers pick up their lower-numbered chopstick at the same time, only the highest-numbered chopstick will be left on the table. Therefore, the fifth philosopher will not be able to pick up a chopstick. Only one philosopher will have access to that highest-numbered chopstick, so she will be able to eat using two chopsticks. Reference [306] presents a solution of the dining philosopher problem based on a Petri Net model.

The division of work comes naturally when some activities of a complex task require special competence and can only be assigned to uniquely qualified entities. In other cases, all entities have the same competence and the work should be assigned based on the individual ability to carry out a task more efficiently. Balancing the workload could be difficult in all cases as some activities may be more intense than the others.

Though concurrency reduces the time to completion, it can negatively affect the efficiency of individual entities involved. Sometimes, a complex task consists of multiple stages and transitions from one stage to the next can only occur when all concurrent activities in one stage have finished their assigned work. In this case the entities finishing early have to wait for the others to complete, an effect called *barrier synchronization*.

This discussion shines some light on the numerous challenges inherent to concurrency. Many computational problems are rather complex and concurrency has the potential to greatly affect our ability to compute more efficiently. This motivates our interest in concurrency and its applications to cloud computing.

Parallel and distributed computing exploit concurrency and have been investigated since mid 1960s. Parallel processing refers to concurrent execution on a system with a large number of processors, while distributed computing means concurrent execution on multiple systems, often located at different sites. The communication latency is considerably lower in the first case, while distributed computing could only be efficient for coarse-grained parallel applications when concurrent activities seldom communicate with one another. Metrics such as execution time, speedup, and processor utilization discussed in Chapter 4 characterize how efficiently a parallel or distributed system can process a particular application.

Topics discussed in this chapter such as computational models, checkpointing, atomic actions, consensus algorithms, are relevant to both parallel and distributed computing. Multithreading is more relevant to parallel processing, while load balancing is particularly important to distributed systems.

This distinction is blurred for computer clouds as their infrastructure consists of millions of servers at one data center possibly linked by high speed networks with computers at another data center of the same cloud service provider. Nevertheless, communication latency is a matter of concern in cloud computing as we shall see in Chapters 7, 8, and 9.

3.2 COMMUNICATION AND CONCURRENCY IN COMPUTING

In computer science concurrency is the cooperative execution of multiple processes/threads in parallel. Concurrency can be exploited for minimizing the completion time of a task, while maximizing the efficiency of the computing substrate. Computing substrate is a generic term for the physical systems used during the processing of an application. To analyze the benefits of concurrency we consider the

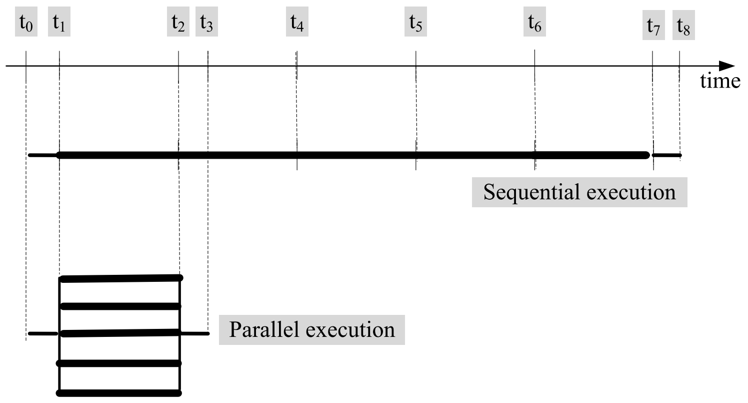


FIGURE 3.3

Sequential versus parallel execution of an application. The sequential execution starts at time t_0 goes through a brief initialization phase till time t_1, then starts the actual image processing. When all images have been processes it enters a brief termination phase at time t_7, and finishes at time t_8. The concurrent execution has its own brief initialization and termination phases, the actual image processing starts at time t_1 and ends at time t_2. The results are available at time $t_3 \ll t_8$. The speedup is close to the maximum speedup.

decomposition of a computation into virtual tasks and relate them to the physical processing elements of the computing substrate.

The larger the cardinality of the set of virtual tasks, the higher the degree of concurrency, thus, of the potential speedup. A parallel algorithm can then be implemented by a parallel program able to run on a system with multiple execution units. A process is a program in execution and requires an address space hosting the code and the data of an application. A thread is light-weight execution unit running in the address space of a process.

The *speedup* of concurrent execution of an application quantifies the effect of concurrent execution and it is defined as the ratio of the completion time of sequential execution of the task versus the concurrent execution completion time. For example, Figure 3.3 illustrates concurrent execution of an application where the workload is partitioned and assigned to five different processors or cores running concurrently. The application is the conversion of 5×10^6 images from one format to another. This is an *embarrassingly parallel* application as the five threads running on five cores, each processing 10^6 images, do not communicate with one another. The speedup S for the example in Figure 3.3 is

$$S = \frac{t_8 - t_0}{t_3 - t_0} \approx 5. \tag{3.1}$$

There are two sides of the concurrency, the algorithmic or logical concurrency discussed in this chapter and the physical concurrency discovered and exploited by the software and the hardware of the computing substrate. For example, a compiler can unroll loops and optimize sequential program execution and a processor core may execute multiple program instructions concurrently, as discussed in Chapter 4.

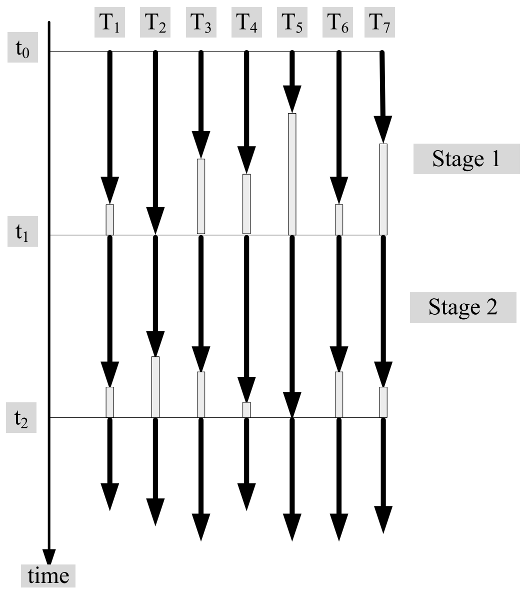


FIGURE 3.4

Barrier synchronization. Seven threads start execution of Stage 1 at time t_0. Threads T_1, T_3, T_4, T_5, T_6, and T_7 finish early and have to wait for thread T_2 before proceeding to Stage 2 at time t_1. Similarly, tasks T_1, T_2, T_3, T_4, T_6, and T_7 have to wait for task T_5, before proceeding to the next stage at time t_2. White bars represent blocked task, waiting to proceed to the next stage.

Concurrency is intrinsically tied to communication, concurrent entities must communicate with one another to accomplish the common task. The corollary of this statement is that communication and computing are deeply intertwined. Explicit communication is built into the blueprint of concurrent activities, we shall call it *algorithmic communication* for lack of a better term. Sometimes, a computation consists of multiple stages when concurrently running threads cannot continue to the next stage until all of them have finished the current one. This leads to inefficiencies as shown in Figure 3.4.

Communication is a more intricate process than the execution of a computational step on a machine that rigorously follows a sequence of instructions from its own repertoire. Besides the sender and the receiver(s), communication involves a third party, a communication channel that may, or may not be reliable. Therefore, the two or more communicating entities have to agree on a communication protocol consisting of multiple messages. *Communication complexity* reflects the amount of communication that the participants of a communication system need to exchange in order to perform certain tasks [533].

Communication speed is considerably slower than computation speed. During the time to send or receive a few bytes a processor could execute billions of instructions. Intensive communication can slow down considerably the group of concurrent threads of an application. Figure 3.5 illustrates the case of such an intensive communication when short bursts of computations alternate with relatively long periods when a thread waits for messages from other threads, the so called *fine-grained* parallelism.

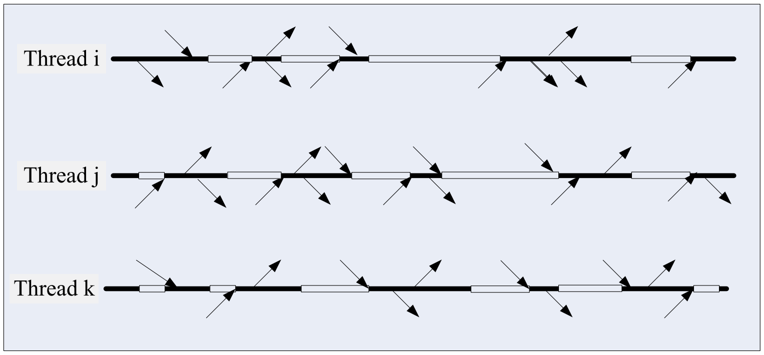


FIGURE 3.5

Fine-grained parallelism. Short bursts of computations of three concurrent threads are interspaces with blocked periods while waiting for messages from other threads. Solid black bars represent running threads while white bars represent blocked threads waiting for messages. Arrows represent messages sent or received by a thread.

The opposite is *coarse-grained* parallelism when little or no communication between the concurrent threads takes place, as seen for the example in Figure 3.3.

The word *message* should be understood in an information theoretical sense rather than the more narrow meaning used in computer networks context. Embarrassingly parallel activities can proceed without message exchanges and enjoy linear or even super-linear speedup, while in all other cases the completion time of communicating activities is increased because the communication bandwidth is significantly lower than the processor bandwidth.

Non-algorithmic communication is required by the organization of a computing substrate consisting of different components that need to act in concert to carry out the required task. For example, in a system consisting of multiple processors and memories, a thread running on one processor may require data stored in the memory of another one. The non-algorithmic communication, unavoidable in a distributed systems, occurs as a side-effect of thread scheduling and data distribution strategies and can dramatically reduce the benefits of concurrency.

Spatial and temporal locality affect the efficiency of a parallel computation. A sequence of instructions or data references enjoy *spatial locality* if the items referenced in a narrow window of time are close in space, e.g., are stored in nearby memory addresses, or nearby sectors on a disk. A sequence of items enjoy *temporal locality* if accesses to the same item are clustered in time.

Non-algorithmic communication complexity may decrease due to locality; it is more likely that a virtual task may find the data it needs in the memory of the physical processor where it runs. Also, the efficiency of the computing substrate may increase because at any given time the scheduler may find sufficient read-to-run virtual tasks to keep the physical processing elements busy. The scale of a system amplifies the negative effects of both algorithmic and non-algorithmic communication. The interaction between the virtual and physical aspects of concurrency gives us a glimpse at the challenges faced by a computational model of concurrent activities.

Today's computing systems implement two computational models, one based on *control flow* and the other one on *data flow*. The ubiquitous von Neumann model for sequential execution implements

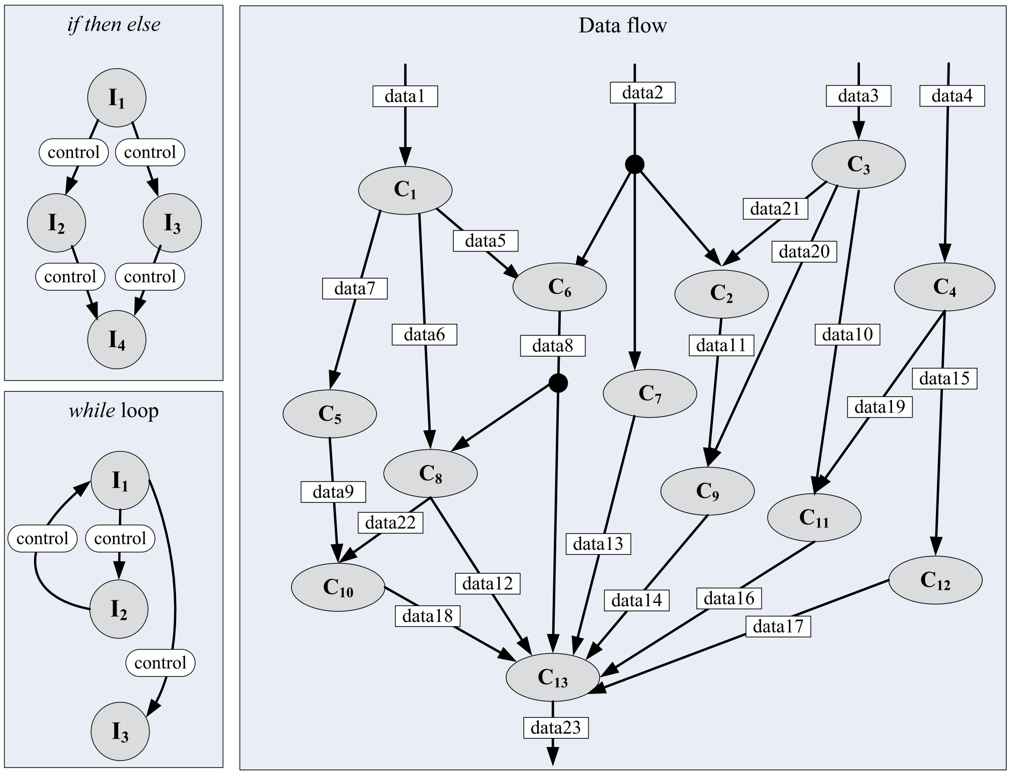


FIGURE 3.6

Control versus data flow models. Left graphs show the flow of control from one instruction I_i, $1 \leq i \leq 4$ to the others for *if then else* and the *while* loop constructs. Either I_2 or I_3 will ever run in the *if then else* construct. The flow of data triggers execution of computations C_i, $1 \leq i \leq 13$ on the right.

the former, at each step the algorithm specifies the step to be executed next. In this model concurrency can only be exploited through the development of a parallel algorithm reflecting the decomposition of a computational task into processes/threads that can run concurrently while exchanging messages to coordinate their progress. In case of control flow the *if then else* construct is shown by the top graph and the *while* loop on the bottom graph in Figure 3.6. Instructions I_2 and I_3 of the *if then else* construct can run concurrently only after instruction I_1 finishes. Instruction I_4 can only be executed when I_2 and I_3 finish execution.

The *data flow* execution model is based on an implicit communication model, where a thread starts execution as soon as its input data become available. The advantages of this execution model are self-evident, it is able to effortlessly extract the maximum amount of parallelism from any computation. The data flow model example in Figure 3.6 shows a maze of computations $C_1, \ldots, C_{13}$ with complex dependencies.

The data flow model allows all computations to run as soon as their input data become available. For example, C_1, C_3 and C_4 start execution concurrently with input $data1$, $data3$, and $data4$, respectively, while C_2 and C_6 wait for completion of C_3 and C_1, respectively. Finally, C_{13} can only start when $data18, data12, data8, data13, data14, data16$ and $data17$ are produced by $C_{10}, C_8, C_6, C_7, C_9, C_{11}$, and C_{12} respectively.

The time required by computations C_i $1 \leq i \leq 13$ to finish execution and deliver data to the ones waiting depends upon the size of the input data which is not known a priori. To capture the dynamics of a computational task the control flow model would require individual computations to send and receive messages, in addition to sending the data.

There are only a few data flow computer systems in today's landscape, but it is not beyond the realm of possibilities to see some added to the cloud computing infrastructure in the next future. Moreover, some of the frameworks for data processing discussed in Chapters 7 and 8 attempt to mimic the data flow execution model to optimize their performance.

Interestingly enough, the Petri Net models discussed in Section 3.5 are powerful enough to express either control flow or data flow. In these bipartite graphs a type of vertices, called places, model system state while the other type of vertices, called transactions, model actions. Tokens flowing out of places trigger transactions and end up in other places indicating a change of system state. Tokens may represent either control or data.

For several decades concurrency was of interest mostly for systems programming and for high-performance computing in science and engineering. The majority of other application developers were content with sequential processing and expected increased computing power due to faster clock rates. Concurrency is now mainstream due to the disruptive effects of the multicore processor technology. Multicore processors are developed in response to the limitation imposed on the clock speed, combined with the insatiable appetite for computing power encapsulated in tiny and energy frugal computing devices. Writing and debugging concurrent software is considerably more challenging than developing sequential code, it requires a different frame of mind and effective development tools.

The next three sections discuss computational models, abstractions needed to gain insight into fundamental aspects of computing and concurrency.

3.3 COMPUTATIONAL MODELS; THE BSP MODEL

Several computational models are used in the analysis of algorithms. Each one of the two broad classes of models, *abstract machines* and *decision trees* specify the set of primitive operations allowed by the model. Turing machines, circuit models, lambda calculus, finite-state machines, cellular automaton, stack machines, accumulator machines, and random access machines are abstract machines used in proofs of computability and for establishing upper bounds on computational complexity of algorithms [443]. Decision tree models are used in proofs of lower bounds on computational complexity.

Computational models. Turing machines are *uniform models of computation*, the same computational device is used for all possible input lengths. Boolean circuits are *non-uniform models of computation*, inputs of different lengths are processed by different circuits. A computational problem $\mathcal{P}$ is associated with the family of Boolean circuits $\mathcal{C} = \{C_1, C_2, \ldots, C_n, \ldots\}$ where each C_n is a Boolean circuit handling inputs of n bits. A family of Boolean circuits $\mathcal{C}_n,\ n \in \mathbb{N}$, is *polynomial-time uniform* if there exists a deterministic Turing machine TM, such that TM runs in polynomial time and $\forall n \in \mathbb{N}$ it outputs a description of $\mathcal{C}_n$ on input 1^n.

A finite-state machine (FSM) consists of a logical circuit $\mathcal{L}$ and a memory $\mathcal{M}$. An execution step with the external input $L^{in} \in \Sigma$ takes the current state $S \in \mathcal{S}$ and uses the logic circuit $\mathcal{L}$ for producing a successor state $S^{new} \in \mathcal{S}$ and an output letter from the same alphabet, Σ, $L^{out} \in \Sigma$. FSMs are used for the implementation of sophisticated processing scheme, such as the ZooKeeper coordination model discussed in Section 7.4, or TCP (Transmission Control Protocol) discussed in Chapter 5 and used by the Internet protocol stack.

The operation of a serial computer using two synchronous interconnected FSMs, a central processing unit (CPU) with a small number of registers and a random-access memory is modeled by a *random-access machine* (RAM). The CPU operates on data stored in its registers. The *parallel random-access machine* (PRAM) is an abstract programming model consisting of a bounded set of RAM processors and a common memory of a potentially unlimited size. Each RAM has its own program and program counter and a unique ID. During a PRAM execution step the RAMs execute synchronously three operations: read from the common memory, perform a local computation, and write to the common memory.

The von Neumann model, based on the von Neumann machine architecture, has been an enduring model of sequential computations. It has continued to allow "a diverse and chaotic software to run efficiently on a diverse and chaotic world of hardware" [492]. The model has endured the rapid pace of technological changes since 1947 when ENIAC performed the first Monte Carlo simulations for the Manhattan Project [216]. One of the brilliant features of the von Neumann model is its clairvoyance, the ability to remain consistent in the face of later concepts such as memory hierarchies, certainly not available in mid 1940s. This model has been the Zeitgeist[1] of computing for more than half a century.

The Bulk-Synchronous Parallel model (BSP). Leslie Valiant developed in the early 1990s a bridging hardware-software model designed to avoid logarithmic losses of efficiency in parallel processing [492]. Valiant argues that parallel and distributed computing should be based on a model emphasizing portability, the algorithms should be parameter-aware and designed to run efficiently on a broad range of systems with a wide variation of parameters. The algorithms have to be written in a language that can be compiled effectively on a computer implementing the BSP model.

BSP is an unambiguous computational model, including parameters quantifying the physical constraints of the computing substrate. It also includes a nonlocal primitive, the barrier synchronization. The BSP model aims to free the programmer from managing memory, communication, and low-level synchronization, provided that the programs are written with sufficient *parallel slackness*. Parallel slackness means hiding communication latency by providing each processor with a large pool of ready-to-run tasks while other tasks are waiting for either a message or for the completion of another operation.

BSP programs are written for v virtual parallel processors running on $p \leq v$ physical processors. This choice permits the compilers for high-level languages to create an executable sharing a virtual address space and to schedule and pipeline computation and communication efficiently. The BSP model includes:

[1]The German word "Zeitgeist" literally translated as "time spirit" is used to identify the dominant set of ideas and concepts of the society in a particular field and at a particular time.

1. Processing elements and memory components.
2. A router involved in message exchanges between pairs of components; the throughput of the router is $\bar{g}$ and s is the startup cost.
3. Synchronization mechanisms acting every L units of time.

The computation involves *supersteps* and *tasks*. A superstep is an execution unit allocated L units of time and consisting of multiple tasks. Each task is a combination of local computations and message exchanges. The computation proceeds as follows:

- At the beginning of a superstep each component is assigned a task.
- At the end of the time allotted to the superstep, after L units of time since its start, a global check determines if the superset has been completed.
  - If so, the next superstep is initiated;
  - Else, another period of L units of time is allocated allowing the superset to continue its execution.

Local operations do not automatically slow down other processors. A processor can proceed without waiting for the completion of processes in the router or in other components when the synchronization is switched off. The model does not make any assumption about communication latency. The value of the periodicity parameter L may be controlled, its lower bound is determined by the hardware, the faster the hardware the lower L, while its upper bound is controlled by the granularity of the parallelism, hence by the software.

The router of the BSP computing engine supports arbitrary *h-relations*, which are supersteps of duration $\bar{g} \times h + s$, when each component sends and receives at most h messages. It is assumed that h is large and $\bar{g} \times h$ is comparable to s. When $g = 2\bar{g}$ and $\bar{g} \times h \geq s$ an h-relation will require at most $g \times h$ units of time.

Hash functions are used by the model to distribute memory accesses to memory units of all BSP components. The mapping of logical or symbolic addresses to physical ones must be efficient and distribute the references as uniformly as possible and this can be done by a pseudo-random mapping. When a superstep requires p random accesses to p components then a component will get with high probability $\log p / \log\log p$ accesses. If a superstep requires $p \log p$ memory accesses, then each component will get not more than about $3 \log p$ accessed and these can be sustained by a router in the optimal bound $\mathcal{O}(\log p)$.

The simulation of a parallel program with $v \geq p \log p$ virtual processors on a BSP computer with p components is discussed next. Each BSP computer component consists of a processor and a memory. Each one of the p components of the BSP computer is assigned v/p virtual processors. The BSP will then simulate one step of a virtual processor in one superstep and each one of the p memory modules will get v/p references if the memory references are evenly spread. The BSP will execute a superstep in an optimal time of $\mathcal{O}(v/p)$.

The multiplication of two $n \times n$ matrices A and B using a standard algorithm is a good example of simulation on a BSP machine with $p \leq n^2$ processors. Each processor is assigned a $n/\sqrt{p} \times n/\sqrt{p}$ submatrix and receives $n/\sqrt{p} \times n/\sqrt{p}$ rows of A and $n/\sqrt{p} \times n/\sqrt{p}$ columns of B. Thus, each processor will carry out $2n^3/p$ additions and multiplications and send $2n^2/\sqrt{p} \leq 2n^3/p$ messages. If each processor sends $2n^2/\sqrt{p}$ messages then the running time is affected only by a constant factor.

Concurrency can be implemented efficiently by replicating data when h is small. In the matrix multiplication example $2n^2/p$ elements of matrices A and B can be distributed to each one of the

p processors, each processor replicates each of its elements $\sqrt{p}$ times and sends them to the $\sqrt{p}$ processors that need the entries. The number of messages sent by each processor is $2n^2\sqrt{p}$. When $g = \mathcal{O}(n\sqrt{p})$ and $L = \mathcal{O}(n^3/p)$ we have the optimal running time of $\mathcal{O}(n^3/p)$. In addition to matrix multiplication, it is shown that several other important algorithms can be implemented efficiently on a BSP computer.

Valiant concludes that the BSP model helps programmability for computations with sufficient slack because the memory and communication management required to implement a virtual shared memory can be achieved with only a constant factor loss in processor utilization [492]. Moreover, the BSP model can be implemented efficiently for different communication technologies and interconnection network topologies, e.g., for a hypercube interconnection network.

3.4 A MODEL FOR MULTICORE COMPUTING

Extracting an optimal multicore processor performance is very challenging. Most of these challenges are intrinsically related to the complexity and diversity of the computational engines as the performance of an application on one system may not translate on high performance on another. Developing parallel algorithms is in itself non-trivial for many applications of interest and the application performance may not scale with the problem size for different computation substrates.

The Multi-BSP [493] is a hierarchical multicore computing model with an arbitrary number of levels. The computing substrate of the model includes multiple levels of cache, as well as the size of the memory. A *depth-d* model is a tree of depth d with caches and memory as the internal nodes of the tree and the processors as the leaves.

A depth-d model requires $4d$ parameters. Level i, $1 \leq i \leq d$, has four parameters (p_i, g_i, L_i, m_i) that quantify the number of subcomponents, the communication bandwidth, the synchronization cost, and the memory/cache size, respectively. The model captures the unavoidable communication cost due to latency L_i and bandwidth g_i as seen in Figure 3.7. An in-depth description of level i parameters shows:

- p_i – the number of $(i-1)$ level components inside an i-th level component. The 1-st level components consist of p_1 raw processors; a computation step on a word in level 1 memory represents one unit of time.
- g_i – the communication bandwidth, the ratio of the number of operations of a processor to the number of words transmitted between the memories of a component at level i and its parent component at level $(i+1)$, in a unit of time. A word represents the data the processor operates on. Level 1 memories can keep up with the processors, thus their data rate, g_0, is one.
- L_i – the cost for barrier synchronization for a level i superstep. The barrier synchronization is between the subcomponents of a component; there is no synchronization across separated branches in the component hierarchy.
- m_i – the number of words of memory inside an i-th level component that is not inside any $(i-1)$ level component.

The number of processors in a level i component is $P_i = p_1 \cdot p_2 \cdot \ldots \cdot p_i$. The number of level i components in a level j component is $Q_{i,j} = p_{i+1} \cdot p_{i+2} \cdot \ldots \cdot p_j$, and the number in the whole system

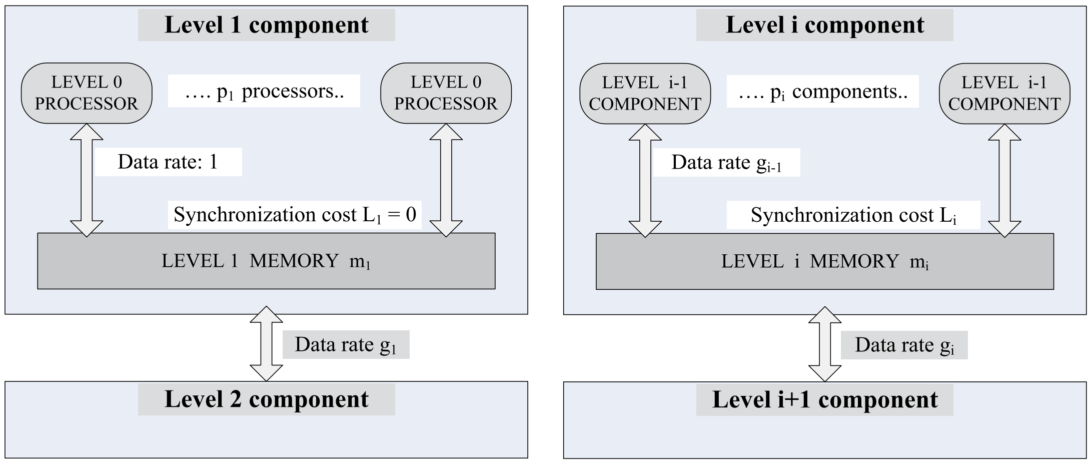


FIGURE 3.7

Multi-BSP model. A *depth-d* model is a tree of depth d with caches and memory as the internal nodes of the tree and the processors as the leaves. p_i – the number of $(i-1)$ level components inside an i-th level component; g_i – the communication bandwidth; L_i – the cost for barrier synchronization for a level i superstep; m_i – the number of words of memory inside an i-th level component.

is $Q_{i,d} = Q_i = p_{i+1} \cdot p_{i+2} \cdot \ldots \cdot p_d$. The total memory at level i component is $M_i = m_i + p_i \cdot m_{i-1} + p_{i-1} \cdot p_i \cdot m_{i-2} + \ldots + (p_2 \cdot \ldots \cdot p_{i-1} \cdot p_i m_1)$. The cost of communication from level 1 to outside level i is $G_i = g_i + g_{i-1} + \cdots + g_1$.

A level i superstep is a construct within a level i component that allows each of p_i level $(i-1)$ components to execute independently until they reach a barrier. Only after reaching the barrier all can exchange information with the m_i memory of the level i component at a communication cost of $(g_i - 1)$. The communication cost is $m_{i-1} g_{i-1}$ where m_i is the number of words communicated between the memory at the i-th level and its level $(i-1)$ subcomponents. The model tolerates constant factors k_{comp}, k_{comm} and k_{synch}, but for each depth d these constants are independent of the (p, g, L, m) parameters.

The question is whether a model with such a large number of parameters could be useful. It is shown that Multi-BSP algorithms for problems such as matrix multiplication, FFT (Fast Fourier Transform) and comparison sorting can be expressed as parameter-free [493]. A *parameter-free* version of an optimal Multi-BSP algorithm with respect to a given algorithm $\mathcal{A}$ means that it is optimal in:

1. Parallel computation steps to within multiplicative constant factors and in total computation steps to within additive lower order terms.
2. Parallel communication costs to within constant multiplicative factors among Multi-BSP algorithms.
3. Synchronization costs to within constant multiplicative factors among Multi-BSP algorithms.

The proofs of optimality of communication and synchronization in [493] are based on previous work on lower bounds on the number of communication steps of distributed algorithms.

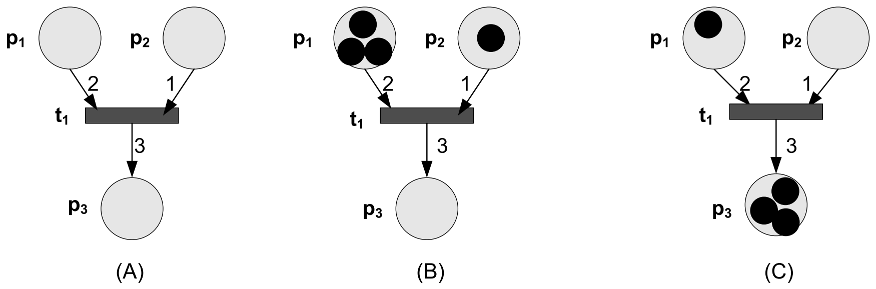


FIGURE 3.8

Petri Nets firing rules. (A) An unmarked net with one transition t_1 with two input places, p_1 and p_2, and one output place, p_3. (B) The marked net, the net with places populated by tokens; the net before firing the enabled transition t_1. (C) The marked net after firing transition t_1, two tokens from place p_1 and one from place p_2 are removed and transported to place p_3.

3.5 MODELING CONCURRENCY WITH PETRI NETS

In 1962 Carl Adam Petri introduced a family of graphs for modeling concurrent systems, the Petri Nets (PNs) [402]. PNs are used to model the dynamic, rather than the static system behavior, e.g., detect synchronization anomalies. PN models have been extended to study the performance of concurrent systems.

PNs are bipartite graphs populated with tokens flowing through the graph. A *bipartite graph* is one with two classes of vertices; arcs always connect a vertex in one class with one or more vertices in the other class. The two classes of PN vertices are *places* and *transitions* thus, the name Place-Transition (P/T) Nets often used for this class of bipartite graphs; arcs connect one place with one or more transitions or a transition with one or more places.

Petri nets model the dynamic behavior of systems. The places of a Petri Net contain tokens; firing of transitions removes tokens from the *input places* of the transition and adds them to its *output places*, see Figure 3.8. Petri Nets can model different activities in a distributed system. A *transition* may model the occurrence of an event, the execution of a computational task, the transmission of a packet, a logic statement, and so on.

The *input places* of a transition model the pre-conditions of an event, the input data for the computational task, the presence of data in an input buffer, the pre-conditions of a logic statement. The *output places* of a transition model the post-conditions associated with an event, the results of the computational task, the presence of data in an output buffer, or the conclusions of a logic statement.

The distribution of tokens in the places of a PN at a given time is called the *marking* of the net and reflects the state of the system being modeled. PNs are very powerful abstractions and can express both concurrency and choice as we can see in Figure 3.9.

Petri nets can model concurrent activities. For example, the net in Figure 3.9A models conflict or choice; only one of the transitions t_1 and t_2 may fire, but not both. Two transitions are said to be *concurrent* if they are causally independent. Concurrent transitions may fire before, after, or in parallel with each other; examples of concurrent transitions are t_1 and t_3 in Figures 3.9B and C.

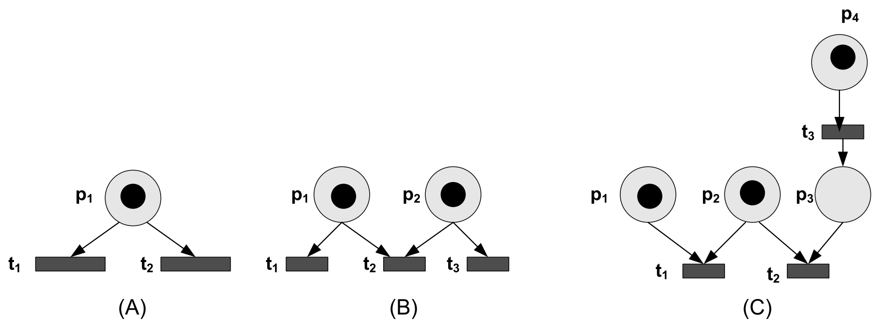


FIGURE 3.9

Petri Nets modeling. (A) Choice; only one of transitions t_1, or t_2 may fire. (B) Symmetric confusion; transitions t_1 and t_3 are concurrent and, at the same time, they are in conflict with t_2. If t_2 fires, then t_1 and/or t_3 is disabled. (C) Asymmetric confusion; transition t_1 is concurrent with t_3 and it is in conflict with t_2 if t_3 fires before t_1.

When choice and concurrency are mixed, we end up with a situation called *confusion*. *Symmetric confusion* means that two or more transitions are concurrent and, at the same time, they are in conflict with another one. For example, transitions t_1 and t_3 in Figure 3.9B, are concurrent and, at the same time, they are in conflict with t_2. If t_2 fires either one or both of them will be disabled. *Asymmetric confusion* occurs when a transition t_1 is concurrent with another transition t_3 and will be in conflict with t_2 if t_3 fires before t_1 as shown in Figure 3.9C.

The concurrent transitions t_2 and t_3 in Figure 3.10A model concurrent execution of two processes. A *marked graph* can model concurrency but not choice; transitions t_2 and t_3 in Figure 3.10B are concurrent, there is no causal relationship between them. Transition t_4 and its input places p_3 and p_4 in Figure 3.10B model synchronization; t_4 can fire only if the conditions associated with p_3 and p_4 are satisfied.

PNs can be used to model *priorities*. The net in Figure 3.10C models a system with two processes modeled by transitions t_1 and t_2; the process modeled by t_2 has a higher priority than the one modeled by t_1. When both processes are ready to run, places p_1 and p_2 hold tokens. When the two processes are ready, transition t_2 will fire first, modeling the activation of the second process. Only after t_2 is activated transition t_1, modeling the activation of the first process, will fire.

Petri Nets are able to model *exclusion*; for example, the net in Figure 3.10D, models a group of n concurrent processes in a shared-memory environment. At any given time only one process may write, but any subset of the n processes may read at the same time, provided that no process writes. Place p_3 models the process allowed to write, p_4 the ones allowed to read, p_2 the ones ready to access the shared memory and p_1 the running tasks. Transition t_2 models the initialization/selection of the process allowed to write and t_1 of the processes allowed to read, whereas t_3 models the completion of a write and t_4 the completion of a read. Indeed p_3 may have at most one token while p_4 may have at most n. If all n processes are ready to access the shared memory, all n tokens in p_2 are consumed when transition t_1 fires. However, place p_4 may contain n tokens obtained by successive firings of transition t_2.

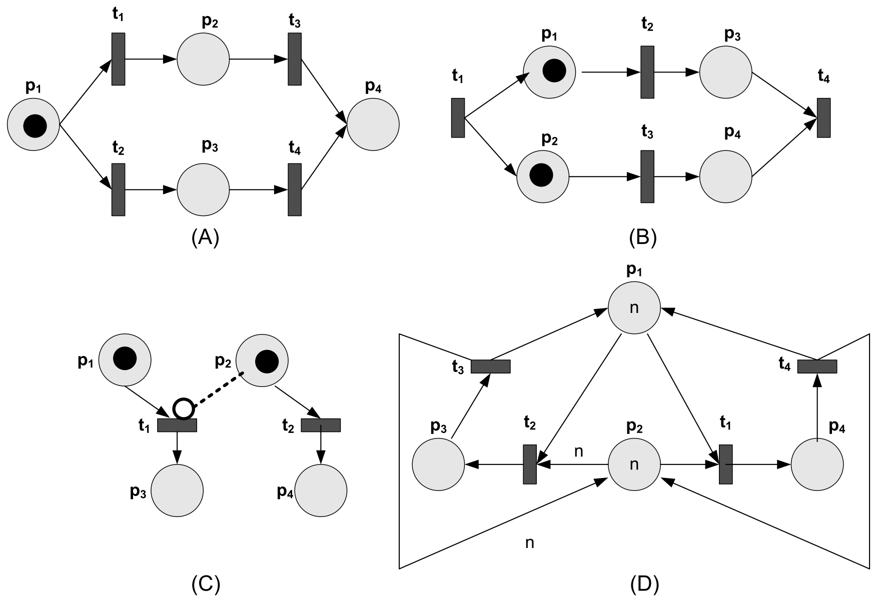


FIGURE 3.10

(A) A state machine; there is the choice of firing t_1, or t_2; only one transition fires at any given time, concurrency is not possible. (B) A *marked graph* can model concurrency but not choice; transitions t_2 and t_3 are concurrent, there is no causal relationship between them. (C) An extended net used to model priorities; the arc from p_2 to t_1 is an inhibitor arc. The process modeled by transition t_1 is activated only after the process modeled by transition t_2 is activated. (D) Modeling exclusion; transitions t_1 and t_2 model writing and, respectively, reading with n processes to a shared memory. At any given time only one process may write, but any subset of the n processes may read at the same time, provided that no process writes.

Formal definitions. After this informal discussion of Petri Nets we switch to a more formal presentation and give several definitions.

Labeled Petri Net: a tuple $N = (p, t, f, l)$ such that:

- $p \subseteq U$ is a finite set of *places*,
- $t \subseteq U$ is a finite set of *transitions*,
- $f \subseteq (p \times t) \cup (t \times p)$ a set of directed arcs, called *flow relations*,
- $l : t \rightarrow L$ a labeling or a weight function

with U a universe of identifiers and L a set of labels. The weight function describes the number of tokens necessary to enable a transition. Labeled PNs describe a static structure; places may contain *tokens* and the distribution of tokens over places defines the state, or the markings of the PN. The dynamic behavior of a PN is described by the structure together with the markings of the net.

Marked Petri Net: a pair (N, s) where $N = (p, t, f, l)$ is a labeled PN and s is a bag[2] over p denoting the markings of the net.

Preset and Postset of Transitions and Places. The preset of transition t_i, denoted as $\bullet t_i$, is the set of input places of t_i and the postset, denoted by $t_i \bullet$, is the set of the output places of t_i. The preset of place p_j, denoted as $\bullet p_j$, is the set of input transitions of p_j and the postset, denoted by $p_j \bullet$, is the set of the output transitions of p_j.

Figure 3.8A shows a PN with three places, p_1, p_2, and p_3, and one transition, t_1. The weights of the arcs from p_1 and p_2 to t_1 are two and one, respectively; the weight of the arc from t_1 to p_3 is three. The preset of transition t_1 in Figure 3.8A consists of two places, $\bullet t_1 = \{p_1, p_2\}$ and its postset consist of only one place, $t_1 \bullet = \{p_3\}$. The preset of place p_4 in Figure 3.10A consists of transitions t_3 and t_4, $\bullet p_4 = \{t_3, t_4\}$ and the postset of p_1 is $p_1 \bullet = \{t_1, t_2\}$.

Ordinary Net. A PN is ordinary if the weights of all arcs are 1. The nets in Figures 3.10A, B, and C are ordinary nets, the weights of all arcs are 1.

Enabled Transition: a transition $t_i \in t$ of the ordinary PN (N, s), with s the initial marking of N, *is enabled* if and only if each of its input places contain a token, $(N, s)[t_i > \Leftrightarrow \bullet t_i \in s$. The notation $(N, s)[t_i >$ means that t_i is enabled. The marking of a PN changes as a result of transition firing; a transition must be enabled in order to fire.

Firing Rule: the firing of the transition t_i of the ordinary net (N, s) means that a token is removed from each of its input places and one token is added to each of its output places; its marking changes $s \mapsto (s - \bullet t_i + t_i \bullet)$. Thus, firing of transition t_i changes a marked net (N, s) into another marked net $(N, s - \bullet t_i + t_i \bullet)$.

Firing Sequence: a nonempty sequence of transitions $\sigma \in t^*$ of the marked net (N, s_0), $N = (p, t, f, l)$ is called a *firing sequence* if and only if there exist markings $s_1, s_2, \ldots, s_n \in \mathcal{B}(p)$ and transitions $t_1, t_2, \ldots, t_n \in t$ such that $\sigma = t_1, t_2, \ldots, t_n$ and for $i \in (0, n)$, $(N, s_i)[t_{i+1} >$ and $s_{i+1} = s_i - \bullet t_i + t_i \bullet$. All firing sequences that can be initiated from marking s_0 are denoted as $\sigma(s_0)$.

Reachability is the problem of finding if marking s_n is reachable from the initial marking s_0, with $s_n \in \sigma(s_0)$. Reachability is a fundamental concern for dynamic systems. The reachability problem is decidable; reachability algorithms require exponential time and space.

Liveness: a marked Petri Net (N, s_0) is said to be *live* if it is possible to fire any transition infinitely often starting from the initial marking, s_0. The absence of deadlock in a system is guaranteed by the liveness of its net model.

Incidence Matrix: given a Petri Net with n transitions and m places, the incidence matrix $F = [f_{i,j}]$ is an integer matrix with $f_{i,j} = w(i, j) - w(j, k)$. Here $w(i, j)$ is the weight of the flow relation (arc) from transition t_i to its output place p_j, and $w(j, k)$ is the weight of the arc from the input place p_j to transition t_k. In this expression $w(i, j)$ represents the number of tokens added to the output place p_j and $w(j, k)$ the number of tokens removed from the input place p_j when transition t_i fires. F^T is the transpose of the incidence matrix.

[2]A bag $\mathcal{B}(\mathcal{A})$ is a multiset of symbols from an alphabet, $\mathcal{A}$; it is a function from $\mathcal{A}$ to the set of natural numbers. For example, $[x^3, y^4, z^5, w^6 \mid P(x, y, z, w)]$ is a bag consisting of three elements x, four elements y, five elements z, and six elements w such that the $P(x, y, z, w)$ holds. P is a predicate on symbols from the alphabet. x is an element of a bag A denoted as $x \in A$ if $x \in \mathcal{A}$ and if $A(x) > 0$.

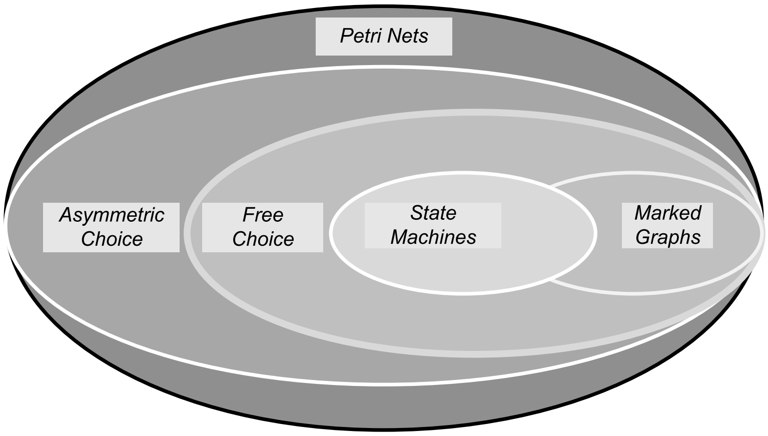


FIGURE 3.11

Classes of Petri Nets.

A marking s_k can be written as a $m \times 1$ column vector and its j-th entry denotes the number of tokens in place j after some transition firing. The necessary and sufficient condition for transition t_k to be enabled at a marking s is that $w(j,k) \leq s(j), \ \forall s_j \in \bullet t_i$, the weight of the arc from every input place of the transition be smaller or equal to the number of tokens in the corresponding input place.

Extended Nets: PNs with inhibitor arcs; an *inhibitor arc* prevents the enabling of a transition. For example, the arc from p_2 to t_1 in the net in Figure 3.10A is an inhibitor arc; the process modeled by transition t_1 can be activated only after the process modeled by transition t_2 is activated.

Modified Transition Enabling Rule for Extended Nets: a transition is not enabled if one of the places in its preset is connected with the transition with an inhibitor arc and if the place holds a token. For example, transition t_1 in the net in Figure 3.10C is not enabled while place p_2 holds a token.

There are several classes of Petri Nets distinguished from one another by their structural properties:

1. State Machines – are used to model finite state machines and cannot model concurrency and synchronization.
2. Marked Graphs – cannot model choice and conflict.
3. Free-choice Nets – cannot model confusion.
4. Extended Free-choice Nets – cannot model confusion but allow inhibitor arcs.
5. Asymmetric Choice Nets – can model asymmetric confusion but not a symmetric one.

This partitioning is based on the number of input and output flow relations from/to a transition or a place and by the manner in which transitions share input places as indicated in Figure 3.11.

State Machine: a Petri Net is a state machine if and only if

$$| \bullet t_i | = 1 \wedge | t_i \bullet | = 1, \ \forall t_i \in t. \tag{3.2}$$

All transitions of a state machine have exactly one incoming and one outgoing arc. This topological constraint limits the expressiveness of a state machine, no concurrency is possible. For example, the transitions t_1, t_2, t_3, and t_4 of state machine in Figure 3.10(a) have only one input and one output arc, the cardinality of their presets and postsets is one. No concurrency is possible; once a choice was made by firing either t_1, or t_2, the evolution of the system is entirely determined. This state machine has four places p_1, p_2, p_3, and p_4 and the marking is a 4-tuple (p_1, p_2, p_3, p_4); the possible markings of this net are $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$, $(0, 0, 0, 1)$, with a token in places p_1, p_2, p_3, or p_4, respectively.

Marked Graph: a Petri Net is a marked graph if and only if

$$| \bullet p_i | = 1 \wedge | p_i \bullet | = 1, \ \forall p_i \in p. \tag{3.3}$$

In a marked graph each place has only one incoming and one outgoing flow relation; thus, marked graphs do no not allow modeling of choice.

Free Choice, Extended Free Choice, and Asymmetric Choice Petri Nets: the marked net, (N, s_0) with $N = (p, t, f, l)$ is a free-choice net if and only if

$$(\bullet t_i) \cap (\bullet t_j) = \emptyset \Rightarrow | \bullet t_i | = | \bullet t_j |, \quad \forall t_i, t_j \in t \tag{3.4}$$

N is an extended free-choice net if $(\bullet t_i) \cap (\bullet t_j) = \emptyset \Rightarrow \bullet t_i = \bullet t_j, \ \forall t_i, t_j \in t.$
N is an asymmetric choice net if and only if $(\bullet t_i) \cap (\bullet t_j) \neq \emptyset \Rightarrow (\bullet t_i \subseteq \bullet t_j)$ or $(\bullet t_i \supseteq \bullet t_j), \ \forall t_i, t_j \in t.$

In an extended free-choice net if two transitions share an input place they must share all places in their presets. In an asymmetric choice net two transitions may share only a subset of their input places.

Several extensions of Petri Nets have been proposed. For example, Colored Petri Nets (CPNs) allow tokens of different colors thus, increase the expressivity of the PNs but do not simplify their analysis. Several extensions of Petri Nets to support performance analysis by associating a random time with each transition have been proposed. In case of Stochastic Petri Nets (SPNs) a random time elapses between the time a transition is enabled and the moment it fires. This random time allows the model to capture the service time associated with the activity modeled by the transition.

Applications of stochastic Petri nets to performance analysis of complex systems is generally limited by the explosion of the state space of the models. Stochastic High-Level Petri Nets (SHLPN) were introduced in 1988 [308]; the SHLPNs allow easy identification of classes of equivalent markings even when the corresponding aggregation of states in the Markov domain is not obvious. This aggregation could reduce the size of the state space by one or more orders of magnitude depending on the system being modeled.

Cloud applications often require a large number of tasks to run concurrently. The interdependencies among these tasks are quite intricate and Petri Nets can be very useful to intuitively present a high level model of the interactions among the tasks. The five classes of systems mentioned earlier, including finite-state machines, as well as control flow and data flow systems can be modeled by Petri Nets. The workflow patterns common to many cloud applications discussed in Section 7.2 translate easily to PN models. Unfortunately, the size of the state space of detailed PN models of complex systems grows very fast and that limits the usefulness of these models.

3.6 PROCESS STATE; GLOBAL STATE OF A PROCESS OR THREAD GROUP

To understand the important properties of distributed systems we use a model, an abstraction based on two critical components, processes/threads and communication channels. A *process* is a program in execution and a *thread* is a light-weight process. A thread of execution is the smallest unit of processing that can be scheduled by an operating system.

A process or a thread is characterized by its *state*. The state is the ensemble of information we need to restart a process or thread after it was suspended. An *event* is a change of state of a process or a thread. The events affecting the state of process p_i are numbered sequentially as $e_i^1, e_i^2, e_i^3, \ldots$ as shown in the space–time diagram in Figure 3.12A. A process p_i is in state σ_i^j immediately after the occurrence of event e_i^j and remains in that state until the occurrence of the next event, e_i^{j+1}.

A process or a thread group is a collection of cooperating processes and threads; to reach a common goal the processes work in concert and communicate with one another. For example, a parallel algorithm to solve a system of partial deferential equations (PDEs) over a domain D may partition the data in several segments and assign each segment to one of the members of the group. The processes or the treads in the group must cooperate with one another and iterate until the common boundary values computed by one process agree with the common boundary values computed by another.

A *communication channel* provides the means for processes/threads to communicate with one another and coordinate their actions by exchanging messages. Without loss of generality we assume that communication among processes is done only by means of *send(m)* and *receive(m)* communication events, where m is a message. We use the term "message" for a structured unit of information, which can be interpreted only in a semantic context by the sender and the receiver. The *state of a communication channel* is defined as follows: given two processes p_i and p_j, the state of the channel, $\xi_{i,j}$, from p_i to p_j consists of messages sent by p_i but not yet received by p_j.

These two abstractions allow us to concentrate on critical properties of distributed systems without the need to discuss the detailed physical properties of the entities involved. The model presented is based on the assumption that a channel is a unidirectional bit pipe of infinite bandwidth and zero latency, but unreliable; messages sent through a channel may be lost, distorted, or the channel may fail, losing its ability to deliver messages. We also assume that the time a process needs to traverse a set of states is of no concern and that processes may fail, or be aborted.

A *protocol* is a finite set of messages exchanged among processes and threads to help them coordinate their actions. Figure 3.12C illustrates the case when communication events are dominant in the local history of processes, p_1, p_2 and p_3. In this case only e_1^5 is a local event; all others are communication events. The particular protocol illustrated in Figure 3.12C requires processes p_2 and p_3 to send messages to the other processes in response to a message from process p_1.

The informal definition of the state of a single process or a thread can be extended to collections of communicating processes/threads. The *global state of a distributed system* consisting of several processes and communication channels is the union of the states of the individual processes and channels [45].

Call h_i^j the history of process p_i up to and including its j-th event, e_i^j, and call σ_i^j the local state of process p_i following event e_i^j. Consider a system consisting of n processes, $p_1, p_2, \ldots, p_i, \ldots, p_n$

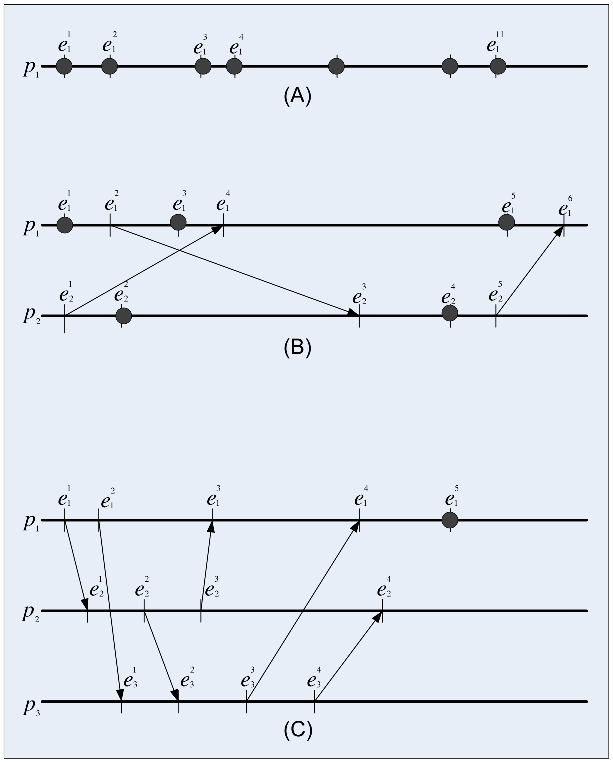


FIGURE 3.12

Space–time diagrams display local and communication events during a process lifetime. Local events are small black circles. Communication events in different processes/threads are connected by lines originating at a *send* event and terminated by an arrow at the *receive* event. (A) All events in case of a single process p_1 are local; the process is in state σ_1 immediately after the occurrence of event e_1^1 and remains in that state until the occurrence of event e_1^2. (B) Two processes/threads p_1 and p_2; event e_1^2 is a communication event, p_1 sends a message to p_2; event e_2^3 is a communication event, process or thread p_2 receives the message sent by p_1. (C) Three processes or threads interact by means of communication events.

with $\sigma_i^{j_i}$ the local state of process p_i; then, the global state of the system is an n-tuple of local states

$$\Sigma^{(j_1,j_2,\ldots,j_n)} = (\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_i^{j_i}, \ldots, \sigma_n^{j_n}). \tag{3.5}$$

The state of the channels does not appear explicitly in this definition of the global state because the state of the channels is encoded as part of the local state of the processes/threads communicating through the channels.

The global states of a distributed computation with n processes form an n-dimensional lattice. The elements of this lattice are global states $\Sigma^{(j_1,j_2,\ldots,j_n)}(\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_n^{j_n})$.

Figure 3.13A shows the lattice of global states of the distributed computation in Figure 3.12B. This is a two-dimensional lattice because we have two processes, p_1 and p_2. The lattice of global states for

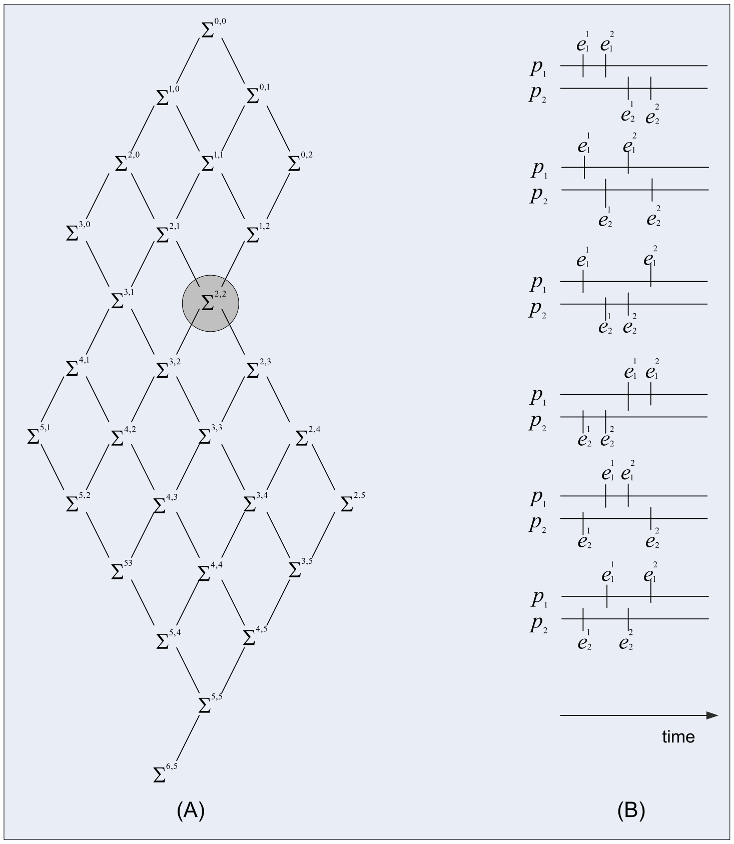


FIGURE 3.13

(A) The lattice of the global states of two processes/threads with the space–time diagrams in Figure 3.12B. (B) The six possible sequences of events leading to the state $\Sigma^{(2,2)}$.

the distributed computation in Figure 3.12C is a three-dimensional lattice, the computation consists of three concurrent processes, p_1, p_2, and p_3.

The initial state of the system in Figure 3.13B is the state before the occurrence of any event and it is denoted by $\Sigma^{(0,0)}$; the only global states reachable from $\Sigma^{(0,0)}$ are $\Sigma^{(1,0)}$, and $\Sigma^{(0,1)}$. The communication events limit the global states the system may reach; in this example the system cannot reach the state $\Sigma^{(4,0)}$ because process p_1 enters state σ_4 only after process p_2 has entered the state σ_1. Figure 3.13B shows the six possible sequences of events to reach the global state $\Sigma^{(2,2)}$:

$$(e_1^1, e_1^2, e_2^1, e_2^2), (e_1^1, e_2^1, e_1^2, e_2^2), (e_1^1, e_2^1, e_2^2, e_1^2), (e_2^1, e_2^2, e_1^1, e_1^2), (e_2^1, e_1^1, e_1^2, e_2^2), (e_2^1, e_1^1, e_2^2, e_1^2). \quad (3.6)$$

An interesting question is: How many paths to reach a global state exist? The more paths exist, the harder it is to identify the events leading to a state when we observe an undesirable behavior of the system. A large number of paths increase the difficulties to debug the system.

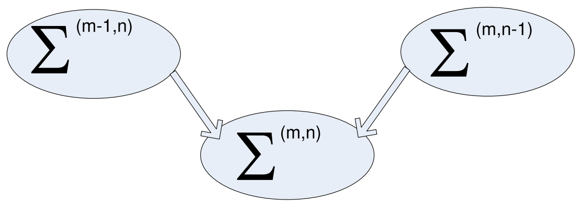


FIGURE 3.14

In the two dimensional case the global state $\Sigma^{(m,n)}$, $\forall (m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$.

We conjecture that in the case of two threads in Figure 3.13A the number of paths from the global state $\Sigma^{(0,0)}$ to $\Sigma^{(m,n)}$ is

$$N_p^{(m,n)} = \frac{(m+n)!}{m!n!}. \tag{3.7}$$

We have already seen that there are six paths leading to state $\Sigma^{(2,2)}$ and, indeed

$$N_p^{(2,2)} = \frac{(2+2)!}{2!2!} = \frac{24}{4} = 6. \tag{3.8}$$

To prove Equation (3.7) we use a method resembling induction; we notice first that the global state $\Sigma^{(1,1)}$ can only be reached from the states $\Sigma^{(1,0)}$ and $\Sigma^{(0,1)}$ and that $N_p^{(1,1)} = (2)!/1!1! = 2$ thus, the formula is true for $m = n = 1$. Then we show that if the formula is true for the $(m-1, n-1)$ case it will also be true for the (m, n) case. If our conjecture is true then

$$N_p^{[(m-1),n]} = \frac{[(m-1)+n)]!}{(m-1)!n!} \tag{3.9}$$

and

$$N_p^{[m,(n-1)]} = \frac{[(m+(n-1)]!}{m!(n-1)!}. \tag{3.10}$$

We observe that the global state $\Sigma^{(m,n)}$, $\forall (m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$, see Figure 3.14, thus

$$N_p^{(m,n)} = N_p^{(m-1,n)} + N_p^{(m,n-1)}. \tag{3.11}$$

It is easy to see that indeed

$$\begin{aligned} \tfrac{[(m-1)+n]!}{(m-1)!n!} + \tfrac{[m+(n-1)]!}{m!(n-1)!} &= (m+n-1)!\left[\tfrac{1}{(m-1)!n!} + \tfrac{1}{m!(n-1)!}\right] \\ &= \tfrac{(m+n)!}{m!n!}. \end{aligned} \tag{3.12}$$

This shows that our conjecture is true thus, Equation (3.7) gives the number of paths to reach the global state $\Sigma^{(m,n)}$ from $\Sigma^{(0,0)}$ when two threads are involved. This expression can be generalized for the case of q threads; using the same strategy it is easy to see that the number of path from the state $\Sigma^{(0,0,\dots,0)}$ to the global state $\Sigma^{(n_1,n_2,\dots,n_q)}$ is

$$N_p^{(n_1,n_2,\dots,n_q)} = \frac{(n_1+n_2+\dots+n_q)!}{n_1!n_2!\dots n_q!} \tag{3.13}$$

Indeed, it is easy to see that

$$N_p^{(n_1,n_2,\dots,n_q)} = N_p^{(n_1-1,n_2,\dots,n_q)} + N_p^{(n_1,n_2-1,\dots,n_q)} + \dots + N_p^{(n_1,n_2,\dots,n_q-1)} \tag{3.14}$$

Equation (3.13) gives us an indication on how difficult it is to debug a system with a large number of concurrent threads.

Many problems in distributed systems are instances of the *global predicate evaluation problem* (GPE) where the goal is to evaluate a Boolean expression whose elements are functions of the global state of the system.

3.7 COMMUNICATION PROTOCOLS AND PROCESS COORDINATION

A major concern in any parallel and distributed system is communication in the presence of channel failures. There are multiple modes for a channel to fail and some lead to messages being lost. In the general case, it is impossible to guarantee that two processes will reach an agreement in case of channel failures, see Figure 3.15.

Statement. *Given two processes p_1 and p_2 connected by a communication channel that can lose a message with probability $\epsilon > 0$, no protocol capable of guaranteeing that two processes will reach agreement exists, regardless of how small the probability ϵ is.*

The proof of this statement is by contradiction. Assume that such a protocol exists and it consists of n messages. Recall that a protocol consists of a *finite* number of messages. Since any message might be lost with probability ϵ the protocol should be able to function when only $n-1$ messages reach their destination, the last one being lost. Induction on the number of messages proves that indeed no such protocol exists; indeed, the same reasoning leads us to conclude that the protocol should function correctly with $n-2$ messages, and so on.

In practice, error detection and error correction codes allow processes to communicate reliably though noisy digital channels. The redundancy of a message is increased by more bits and packaging the message as a codeword; the recipient of the message is then able to decide if the sequence of bits received is a valid codeword and, if the code satisfies some distance properties, then the recipient of the message is able to extract the original message from a bit string in error.

Communication protocols implement not only *error control* mechanisms, but also flow control and congestion control. *Flow control* provides feedback from the receiver, it forces the sender to transmit only the amount of data the receiver is able to buffer and then process. *Congestion control* ensures that the offered load of the network does not exceed the network capacity. In store-and-forward networks individual routers may drop packets when the network is congested and the sender is forced to retransmit. Based on the estimation of the RTT (Round-Trip-Time) the sender can detect congestion and reduce the transmission rate.

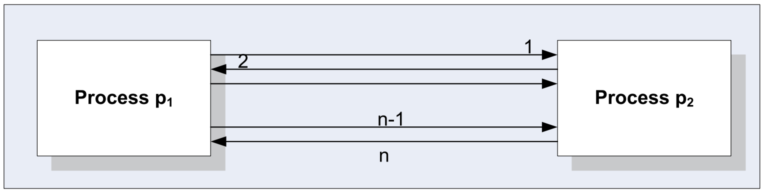


FIGURE 3.15

Process coordination in the presence of errors; each message may be lost with probability ϵ. If a protocol consisting of n messages exists, then the protocol should be able to function properly with $n - 1$ messages reaching their destination, one of them being lost.

The implementation of these mechanisms require the measurement of *time intervals*, the time elapsed between two events; we also need a *global concept of time* shared by all entities that cooperate with one another. For example, a computer chip has an *internal clock* and a predefined set of actions occurs at each clock tick. Each chip has an *interval timer* that helps enhance the system's fault tolerance; when the effects of an action are not sensed after a predefined interval, the action is repeated.

When the entities communicating with each other are networked computers, the precision of the clock synchronization is critical [290]. The event rates are very high, each system goes through state changes at a very fast pace; modern processors run at a 2–4 GHz clock rate. That explains why we need to measure time very accurately; indeed, we have atomic clocks with an accuracy of about 10^{-6} second per year.

An isolated system can be characterized by its *history* expressed as a sequence of events, each event corresponding to a change of the state of the system. Local timers provide relative time measurements. A more accurate description adds to the system's history the time of occurrence of each event as measured by the local timer.

Messages sent by processes may be lost or distorted during transmission. Without additional restrictions regarding message delays and errors, there are no means to ensure a perfect synchronization of local clocks and there are no obvious methods to ensure a global ordering of events occurring in different processes. Determining the global state of a large-scale distributed system is a very challenging problem.

The mechanisms described above are insufficient once we approach the problem of cooperating entities. To coordinate their actions, two entities need a common perception of time. Timers are not enough, clocks provide the only way to measure distributed duration, that is, actions that start in one process and terminate in another.

Global agreement on time is necessary to trigger actions that should occur concurrently, For example, in a real-time control system of a power plant several circuits must be switched on at the same time. Agreement on *the time when events occur* is necessary for distributed recording of events, for example, to determine a precedence relation through a temporal ordering of events. To ensure that a system functions correctly we need to determine that the event causing a change of state occurred before the state change, e.g., the sensor triggering an alarm has to change its value before the emergency procedure to handle the event was activated. Another example of the need for agreement on the time of occurrence of events is in replicated actions. In this case several replicas of a process must log the time of an event in a consistent manner.

Time stamps are often used for event ordering using a global time-base constructed on local virtual clocks [333]. The Δ-protocols [121] achieve total temporal order using a global time base. Assume that local virtual clock readings do not differ by more than π, called *precision* of the global time base. Call g the *granularity of physical clocks*. First, observe that the granularity should not be smaller than the precision; given two events a and b occurring in different processes if $t_b - t_a \leq \pi + g$ we cannot tell which of a or b occurred first [500]. Based on these observations, it follows that the order discrimination of clock-driven protocols cannot be better than twice the clock granularity.

System specification, design, and analysis require a clear understanding of *cause-effect relationships*. During the system specification phase we view the system as a state machine and define the actions that cause transitions from one state to another. During the system analysis phase we need to determine the cause that brought the system to a certain state.

The activity of any process is modeled as a sequence of *events*; hence, the binary relation cause-effect should be expressed in terms of events and should support our intuition that *the cause must precede the effects*. Again, we need to distinguish between local events and communication events. The latter ones affect more than one process and are essential for constructing a global history of an ensemble of processes. Let h_i denote the local history of process p_i and let e_i^k denote the k-th event in this history.

The binary cause-effect relationship between two events has the following properties:

1. Causality of local events can be derived from the process history. Given two events e_i^k and e_i^l local to process p_i

$$\text{if } \ e_i^k, e_i^l \in h_i \ \text{ and } \ k < l \ \text{ then } \ e_i^k \rightarrow e_i^l. \tag{3.15}$$

2. Causality of communication events. Given two processes p_i and p_j and two events e_i^k and e_j^l

$$\text{if } \ e_i^k = send(m) \ \text{ and } \ e_j^l = receive(m) \ \text{ then } \ e_i^k \rightarrow e_j^l. \tag{3.16}$$

3. Transitivity of the causal relationship. Given three processes, p_i, p_j, and p_m and the events e_i^k, e_j^l, and e_m^n

$$\text{if } \ e_i^k \rightarrow e_j^l \ \text{ and } \ e_j^l \rightarrow e_m^n \ \text{ then } e_i^k \rightarrow e_m^n. \tag{3.17}$$

Two events in the global history may be unrelated, neither one is the cause of the other; such events are said to be *concurrent events*.

3.8 COMMUNICATION, LOGICAL CLOCKS, AND MESSAGE DELIVERY RULES

We need to bridge the gap between the physical systems and the abstractions used to describe interacting processes. This section addresses the means to bridge this gap. Communicating processes often run on distant systems whose physical clocks cannot be perfectly synchronized due to communication latency. Global ordering of events in communicating processes running on such systems is not feasible and logical clocks are used instead. Also messages travel through physical channels with different speeds and follow different paths. As a result, the order in which messages are delivered to processes may be different from the order they were sent.

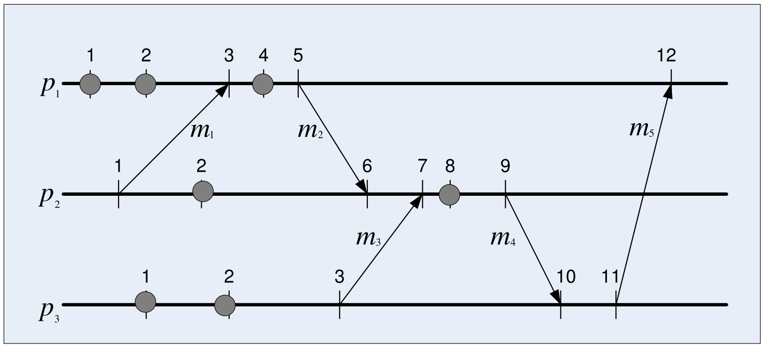


FIGURE 3.16

Three processes and their logical clocks; The usual labeling of events as $e_1^1, e_1^2, e_1^3, \ldots$ is omitted to avoid overloading the figure; only the logical clock values for the local and for the communication events are marked. The correspondence between the events and the logical clock values is obvious: $e_1^1, e_2^1, e_3^1 \rightarrow 1$, $e_1^5 \rightarrow 5$, $e_2^4 \rightarrow 7$, $e_3^4 \rightarrow 10$, $e_1^6 \rightarrow 12$, and so on. Global ordering of all events is not possible; there is no way to establish the ordering of events e_1^1, e_2^1 and e_3^1.

Logical clocks. A *logical clock (LC)* is an abstraction necessary to ensure the clock condition given by Equations (3.24) and (3.25) in the absence of a global clock. Each process p_i maps events to positive integers. Call $LC(e)$ the local variable associated with event e. Each process time stamps each message m sent with the value of the logical clock at the time of sending, $TS(m) = LC(send(m))$. The rules to update the logical clock are specified by the following relationship:

$$LC(e) := \begin{cases} LC + 1 & \text{if } e \text{ is a local event or a } send(m) \text{ event} \\ \max(LC, TS(m) + 1) & \text{if } e = receive(m). \end{cases} \tag{3.18}$$

The concept of logical clocks is illustrated in Figure 3.16 using a modified *space–time diagram* where the events are labeled with the logical clock value. Messages exchanged between processes are shown as lines from the sender to the receiver; the communication events corresponding to sending and receiving messages are marked on these diagrams.

Each process labels local events and send events sequentially until it receives a message marked with a logical clock value larger than the next local logical clock value, as shown in Equation (3.18). It follows that logical clocks do not allow a global ordering of all events. For example, there is no way to establish the ordering of events e_1^1, e_2^1 and e_3^1 in Figure 3.16. Nevertheless, communication events allow different processes to coordinate their logical clocks; for example, process p_2 labels the event e_2^3 as 6 because of message m_2, which carries the information about the logical clock value as 5 at the time message m_2 was sent. Recall that e_i^j is the j-th event in process p_i.

Logical clocks lack an important property, *gap detection*; given two events e and e' and their logical clock values, $LC(e)$ and $LC(e')$, it is impossible to establish if an event e'' exists such that

$$LC(e) < LC(e'') < LC(e'). \tag{3.19}$$

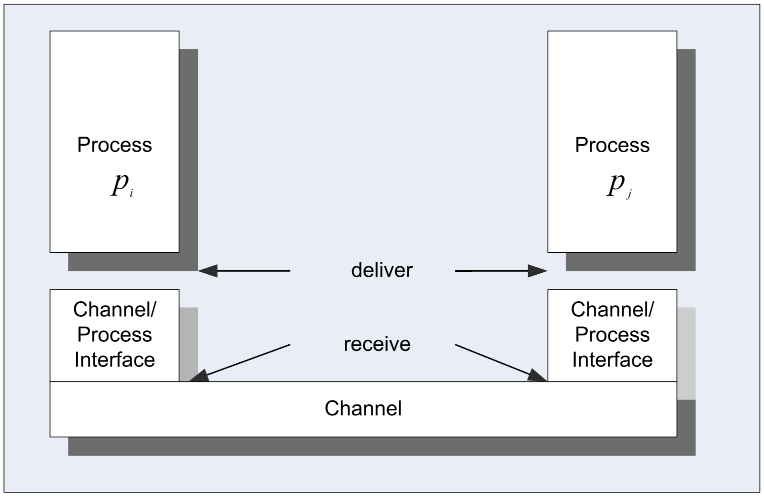


FIGURE 3.17

Message receiving and message delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.

For example, for process p_1 there is an event, e_1^4, between the events e_1^3 and e_1^5 in Figure 3.16; indeed, $LC(e_1^3) = 3$, $LC(e_1^5) = 5$, $LC(e_1^4) = 4$, and $LC(e_1^3) < LC(e_1^4) < LC(e_1^5)$. However, for process p_3, the events e_3^3 and e_3^4 are consecutive though, $LC(e_3^3) = 3$ and $LC(e_3^4) = 10$.

Message delivery rules. The communication channel abstraction makes no assumptions about the order of messages; a real-life network might reorder messages. This fact has profound implications for a distributed application. Consider for example a robot getting instructions to navigate from a monitoring facility with two messages, "turn left" and "turn right", being delivered out of order.

Message receiving and message delivery are two distinct operations; a *delivery rule* is an additional assumption about the channel-process interface. This rule establishes when a message received is actually delivered to the destination process. The receiving of a message m and its delivery are two distinct events in a causal relation with one another, a message can only be delivered after being received, see Figure 3.17

$$receive(m) \rightarrow deliver(m). \tag{3.20}$$

First-In-First-Out (FIFO) delivery implies that messages are delivered in the same order they are sent. For each pair of source-destination processes (p_i, p_j) FIFO delivery requires that the following relation should be satisfied

$$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m'). \tag{3.21}$$

Even if the communication channel does not guarantee FIFO delivery, FIFO delivery can be enforced by attaching a sequence number to each message sent. The sequence numbers are also used to reassemble messages out of individual packets.

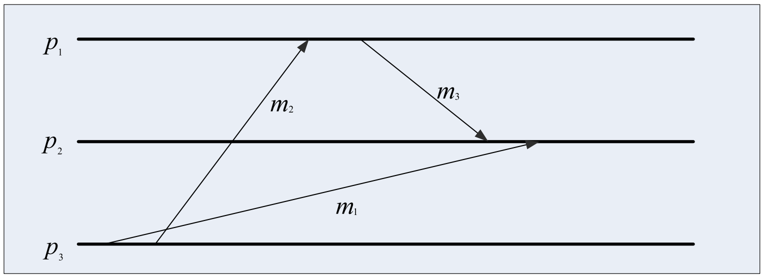


FIGURE 3.18

Violation of causal delivery when more than two processes are involved; message m_1 is delivered to process p_2 after message m_3, though message m_1 was sent before m_3. Indeed, message m_3 was sent by process p_1 after receiving m_2, which in turn was sent by process p_3 after sending message m_1.

Causal message delivery. Causal delivery is an extension of the FIFO delivery to the case when a process receives messages from different sources. Assume a group of three processes, (p_i, p_j, p_k) and two messages m and m'. Causal delivery requires that

$$send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m'). \tag{3.22}$$

When more than two processes are involved in a message exchange, the message delivery may be FIFO, but not causal as shown in Figure 3.18 where we see that

- $deliver(m_3) \rightarrow deliver(m_1)$; according to the local history of process p_2.
- $deliver(m_2) \rightarrow send(m_3)$; according to the local history of process p_1.
- $send(m_1) \rightarrow send(m_2)$; according to the local history of process p_3.
- $send(m_2) \rightarrow deliver(m_2)$.
- $send(m_3) \rightarrow deliver(m_3)$.

The transitivity property and the causality relations above imply that

$$send(m_1) \rightarrow deliver(m_3). \tag{3.23}$$

Call $TS(m)$ the *time stamp* carried by message m. A message received by process p_i is *stable* if no future messages with a time stamp smaller than $TS(m)$ can be received by process p_i. When using logical clocks, a process p_i can construct consistent observations of the system if it implements the following delivery rule: *deliver all stable messages in increasing time stamp order.*

Let us now examine the problem of *consistent message delivery* under several sets of assumptions. First, assume that processes cooperating with each other in a distributed environment have access to a *global real-time clock*, that the message delays are bounded by δ, and that there is no clock drift. Call $RC(e)$ the time of occurrence of event e. A process includes in every message it sends the $RC(e)$, where e is the send message event. The delivery rule in this case is: *at time t deliver all received messages with time stamps up to $(t - \delta)$ in increasing time stamp order.* Indeed, this delivery rule guarantees that under the bounded delay assumption the message delivery is consistent. All messages delivered at time t are in order and no future message with a time stamp lower than any of the messages delivered may arrive.

For any two events, e and e', occurring in different processes, the so called *clock condition* is satisfied if

$$e \rightarrow e' \Rightarrow RC(e) < RC(e'), \ \forall e, e'. \tag{3.24}$$

Oftentimes, we are interested in determining the set of events that caused an event knowing the time stamps associated with all events; in other words, we are interested in deducing the causal precedence relation between events from their time stamps. To do so we need to define the so-called strong clock condition. The *strong clock condition* requires an equivalence between the causal precedence and the ordering of the time stamps

$$\forall e, e', \quad e \ \rightarrow \ e' \ \equiv \ TS(e) \ < \ TS(e'). \tag{3.25}$$

Causal delivery is very important because it allows processes to reason about the entire system using only local information. This is only true in a closed system where all communication channels are known; sometimes the system has *hidden channels* and reasoning based on causal analysis may lead to incorrect conclusions.

3.9 RUNS AND CUTS; CAUSAL HISTORY

Knowledge of the state of several, possibly all, processes in a distributed system is often needed. For example, a supervisory process must be able to detect when a subset of processes is deadlocked; a process might migrate from one location to another or be replicated only after an agreement with others. In all these examples a process needs to evaluate a predicate function of the global state of the system.

We call the process responsible for constructing the global state of the system, the *monitor*; a monitor sends messages requesting information about the local state of every process and gathers the replies to construct the global state. Intuitively, the construction of the global state is equivalent to taking snapshots of individual processes and then combining these snapshots into a global view. Yet, combining snapshots is straightforward if and only if all processes have access to a global clock and the snapshots are taken at the same time; hence, the snapshots are consistent with one another.

A *run* is a total ordering R of all the events in the global history of a distributed computation consistent with the local history of each participant process; a run

$$R = (e_1^{j_1}, e_2^{j_2}, \ldots, e_n^{j_n}) \tag{3.26}$$

implies a sequence of events as well as a sequence of global states.

For example, consider the three processes in Figure 3.19. We can construct a three-dimensional lattice of global states following a procedure similar to the one in Figure 3.13 starting from the initial state $\Sigma^{(000)}$ and proceeding to any reachable state $\Sigma^{(ijk)}$ with i, j, k the events in processes p_1, p_2, p_3, respectively. The run $R_1 = (e_1^1, e_2^1, e_3^1, e_1^2)$ is consistent with both the local history of each process and the global history; this run is valid, the system has traversed the global states

$$\Sigma^{000}, \Sigma^{100}, \Sigma^{110}, \Sigma^{111}, \Sigma^{211} \tag{3.27}$$

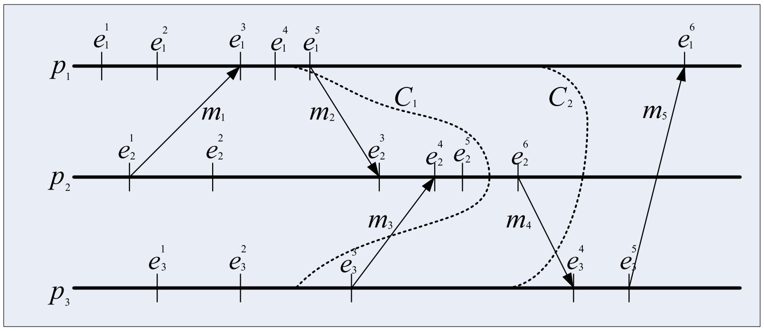


FIGURE 3.19

Inconsistent and consistent cuts: the cut $C_1 = (e_1^4, e_2^5, e_3^2)$ is inconsistent because it includes e_2^4, the event triggered by the arrival of the message m_3 at process p_2, but does not include e_3^3, the event triggered by process p_3 sending m_3 thus, the cut C_1 violates causality. On the other hand, $C_2 = (e_1^5, e_2^6, e_3^3)$ is a consistent cut, there is no causal inconsistency, it includes event e_2^6, the sending of message m_4, without the effect of it, the event e_3^4 receiving the message by process p_3.

On the other hand, the run $R_2 = (e_1^1, e_1^2, e_3^1, e_1^3, e_3^2)$ is invalid because it is inconsistent with the global history. The system cannot ever reach the state Σ^{301}; message m_1 must be sent before it is received, so event e_2^1 must occur in any run before event e_1^3.

A *cut* is a subset of the local history of all processes. If h_i^j denotes the history of process p_i up to and including its j-th event, e_i^j, then a cut C is an n-tuple

$$C = \{h_i^j\} \quad \text{with} \quad i \in \{1, n\} \text{ and } j \in \{1, n_i\}. \tag{3.28}$$

The frontier of the cut is an n-tuple consisting of the last event of every process included in the cut. Figure 3.19 illustrates a *space–time diagram* for a group of three processes, p_1, p_2, p_3 and it shows two cuts, C_1 and C_2. C_1 has the frontier $(4, 5, 2)$, frozen after the fourth event of process p_1, the fifth event of process p_2, and the second event of process p_3. C_2 has the frontier $(5, 6, 3)$.

Cuts provide the necessary intuition to generate global states based on an exchange of messages between a monitor and a group of processes. The cut represents the instance when requests to report individual state are received by the members of the group. Clearly not all cuts are meaningful. For example, the cut C_1 with the frontier $(4, 5, 2)$ in Figure 3.19 violates our intuition regarding causality; it includes e_2^4, the event triggered by the arrival of message m_3 at process p_2 but does not include e_3^3, the event triggered by process p_3 sending m_3. In this snapshot p_3 was frozen after its second event, e_3^2, before it had the chance to send message m_3. Causality is violated and the system cannot ever reach such a state.

Next we introduce the concepts of consistent and inconsistent cuts and runs. A cut closed under the *causal precedence relationship* is called a *consistent cut*. C is a consistent cut if and only if for all events

$$\forall e, e', \quad (e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C. \tag{3.29}$$

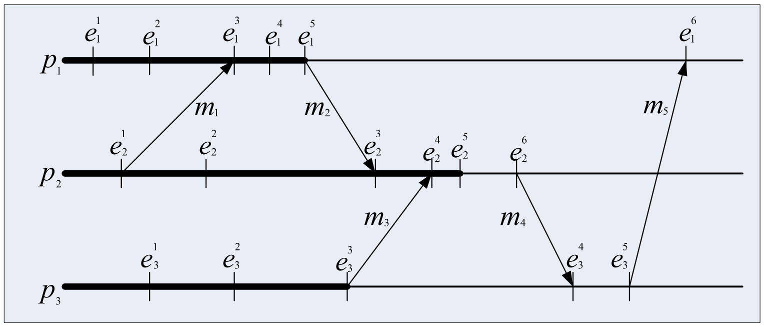

p_1
p_2
p_3
e_1^1 e_1^2 e_1^3 e_1^4 e_1^5 e_1^6
e_2^1 e_2^2 e_2^3 e_2^4 e_2^5 e_2^6
e_3^1 e_3^2 e_3^3 e_3^4 e_3^5
m_1 m_2 m_3 m_4 m_5

FIGURE 3.20

The causal history of event e_2^5, $\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^2, e_3^3\}$, is the smallest consistent cut including e_2^5.

A consistent cut establishes an "instance" of a distributed computation; given a consistent cut we can determine if an event e occurred before the cut.

A run R is said to be *consistent* if the total ordering of events imposed by the run is consistent with the partial order imposed by the causal relation; for all events, $e \rightarrow e'$ implies that e appears before e' in R.

Consider a distributed computation consisting of a group of communicating processes $G = \{p_1, p_2, ..., p_n\}$. The *causal history of event* e, $\gamma(e)$, is the smallest consistent cut of G including event e

$$\gamma(e) = \{e' \in G \mid e' \rightarrow e\} \cup \{e\}. \tag{3.30}$$

The causal history of event e_2^5 in Figure 3.20 is:

$$\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}. \tag{3.31}$$

This is the smallest consistent cut including e_2^5. Indeed, if we omit e_3^3, then the cut $(5, 5, 2)$ would be inconsistent, it would include e_2^4, the communication event for receiving m_3, but not e_3^3, the event caused by sending m_3. If we omit e_1^5, the cut $(4, 5, 3)$ would also be inconsistent, it would include e_2^3 but not e_1^5.

Causal histories can be used as clock values and satisfy the strong clock condition, provided that we equate clock comparison with set inclusion. Indeed,

$$e \rightarrow e' \equiv \gamma(e) \subset \gamma(e'). \tag{3.32}$$

The following algorithm can be used to construct causal histories:

- Each $p_i \in G$ starts with $\theta = \emptyset$.
- Every time p_i receives a message m from p_j it constructs

$$\gamma(e_i) = \gamma(e_j) \cup \gamma(e_k) \tag{3.33}$$

with e_i the *receive* event, e_j the previous local event of p_i, e_k the *send* event of process p_j.

Unfortunately, this concatenation of histories is impractical because the causal histories grow very fast.

Now we present a protocol to construct consistent global states based on the monitoring concepts discusses in this section. We assume a fully connected network. Recall that given two processes p_i and p_j, the state $\xi_{i,j}$ of the channel from p_i to p_j consists of messages sent by p_i but not yet received by p_j. The snapshot protocol of Chandy and Lamport consists of three steps [95]:

1. Process p_0 sends to itself a "take snapshot" message.
2. Let p_f be the process from which p_i receives the"take snapshot" message for the first time. Upon receiving the message, the process p_i records its local state, σ_i, and relays the "take snapshot" along all its outgoing channels without executing any events on behalf of its underlying computation; channel state $\xi_{f,i}$ is set to empty and process p_i starts recording messages received over each of its incoming channels.
3. Let p_s be the process from which p_i receives the "take snapshot" message beyond the first time; process p_i stops recording messages along the incoming channel from p_s and declares channel state $\xi_{s,i}$ as those messages that have been recorded.

Each"take snapshot" message crosses each channel exactly once and every process p_i has made its contribution to the global state; a process records its state the first time it receives a "take snapshot" message and then stops executing the underlying computation for some time. Thus, in a fully connected network with n processes the protocol requires $n \times (n-1)$ messages, one on each channel.

For example, consider a set of six processes, each pair of processes being connected by two unidirectional channels as shown in Figure 3.21. Assume that all channels are empty, $\xi_{i,j} = 0, \ \ i \in \{0, 5\}, \ \ j \in \{0, 5\}$, at the time when process p_0 issues the "take snapshot" message. The actual flow of messages is

- In step 0, p_0 sends to itself the "take snapshot" message.
- In step 1, process p_0 sends five "take snapshot" messages labeled (1) in Figure 3.21.
- In step 2, each of the five processes, p_1, p_2, p_3, p_4, and p_5 sends a "take snapshot" message labeled (2) to every other process.

A "take snapshot" message crosses each channel from process p_i to $p_j, \ i, j \in \{0, 5\}$ exactly once and $6 \times 5 = 30$ messages are exchanged.

3.10 THREADS AND ACTIVITY COORDINATION

While in the early days of computing concurrency was analyzed mostly in the context of the system software, nowadays concurrency is an ubiquitous feature of today's applications. Many applications are data-intensive and resources of one server are insufficient. Such applications require a careful distribution of the workload to multiple instances running concurrently on a large number of servers. This is one of the main attractions of cloud computing. Unquestionably, the need to use effectively the cores of a modern processor has forced many application developers to implement parallel algorithms and use multithreading.

Many concurrent applications are in the embedded systems area. Embedded systems are a class of reactive systems where computations are triggered by external events. Such systems populate the Internet of Things (IoT) and are used by the critical infrastructure. A broad spectrum of such appli-

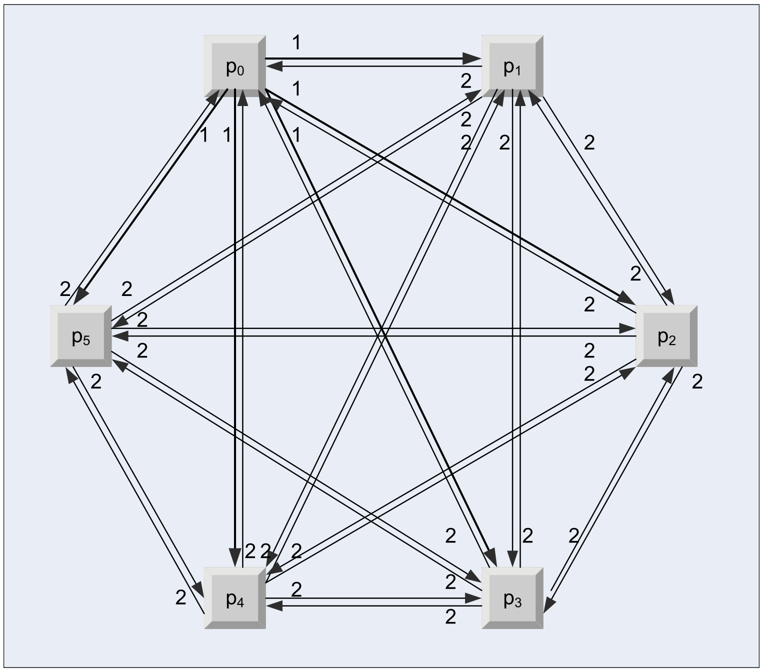


FIGURE 3.21

Six processes executing the snapshot protocol.

cations run multiple threads concurrently to control the ignition of cars, oil processing in a refinery, smart electric meters, heating and cooling systems in homes, or coffee makers. Embedded controllers for reactive real-time applications are implemented as mixed software-hardware systems [407].

Threads under the microscope. Threads are objects created explicitly to execute streams of instructions by an *allocate thread* action. A thread can be in several states as shown in Figure 3.22. Many threads share the core of a processor and the system scheduler is the authority deciding when a thread gets control of the core and enters a *running* state.

The scheduler chooses from the pool of *runnable* threads, the only threads eligible to run. A running thread *yields* the control of the core when it has exhausted the time slot allocated to it, or blocks by executing an *await* action while waiting for the completion of an I/O operation and transition to a *wait* state. The thread could become *runnable* again when the scheduler decides to advance it, e.g., when the I/O operation has finished.

While one may be tempted to think only about application threads, the reality is that to carry out its functions the kernel of an operating system operates a fair number of threads. Figure 3.23 provides a snapshot of the thread population including application and operating system threads. Multithreading at the application level enables the threads to share the resources allocated to the application, while operating system threads act behind the scene supporting a wide range of resource management functions.

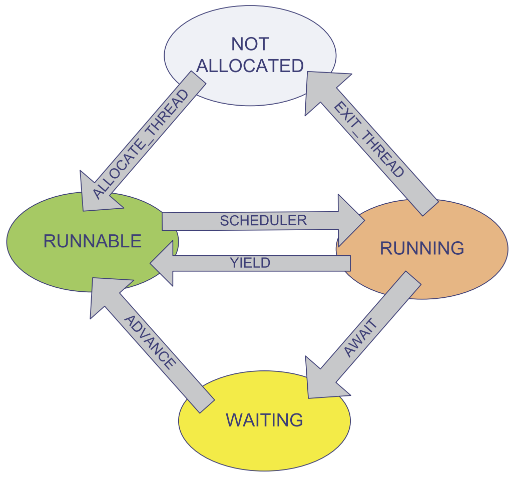


FIGURE 3.22

The state of a thread and the actions triggering a change of state.

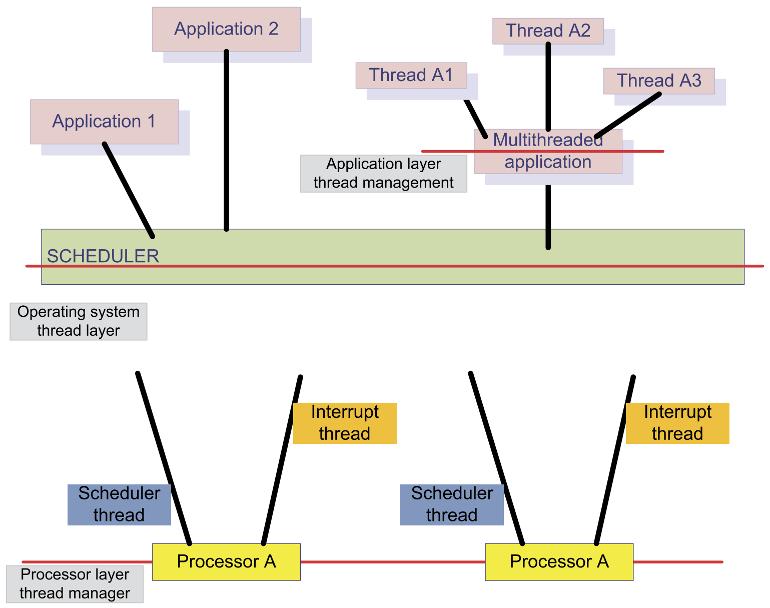


FIGURE 3.23

A snapshot of the thread population. Multiple application threads share a core under the control of a scheduler. Multiple operating system level threads work behind the scene to carry out resource management functions.

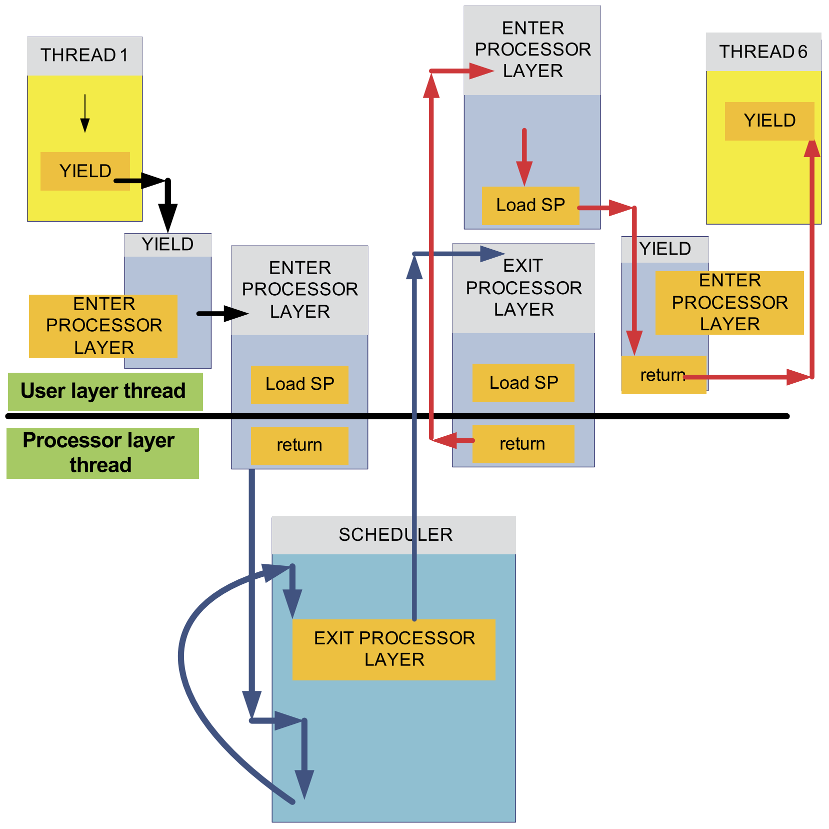


FIGURE 3.24

Application thread scheduling involvers multiple context switches. A context switch saves the current thread state on the system stack. SP is the *stack pointer*.

The operation of the scheduler, while totally transparent to application developers, is fairly complex and multithreading require several context switches as seen in Figure 3.24. A context switch of an application thread involves saving the thread state including registers and the address used when the execution of the suspended thread becomes *runnable* again. Threads are light weight entities, unlike processes when information related to the address space, including pointers to the page tables and the process control block are part of the process state and must be also saved.

Concurrency – the system software side. The kernel of an operating system exploits concurrency for virtualization of system resources such as the processor and the memory. *Virtualization*, covered in depth in Chapter 10, is a system design strategy with a broad range of objectives including:

- Hiding latency and performance enhancement, e.g., schedule a ready-to-run thread when the current thread is waiting for the completion of an I/O operation.

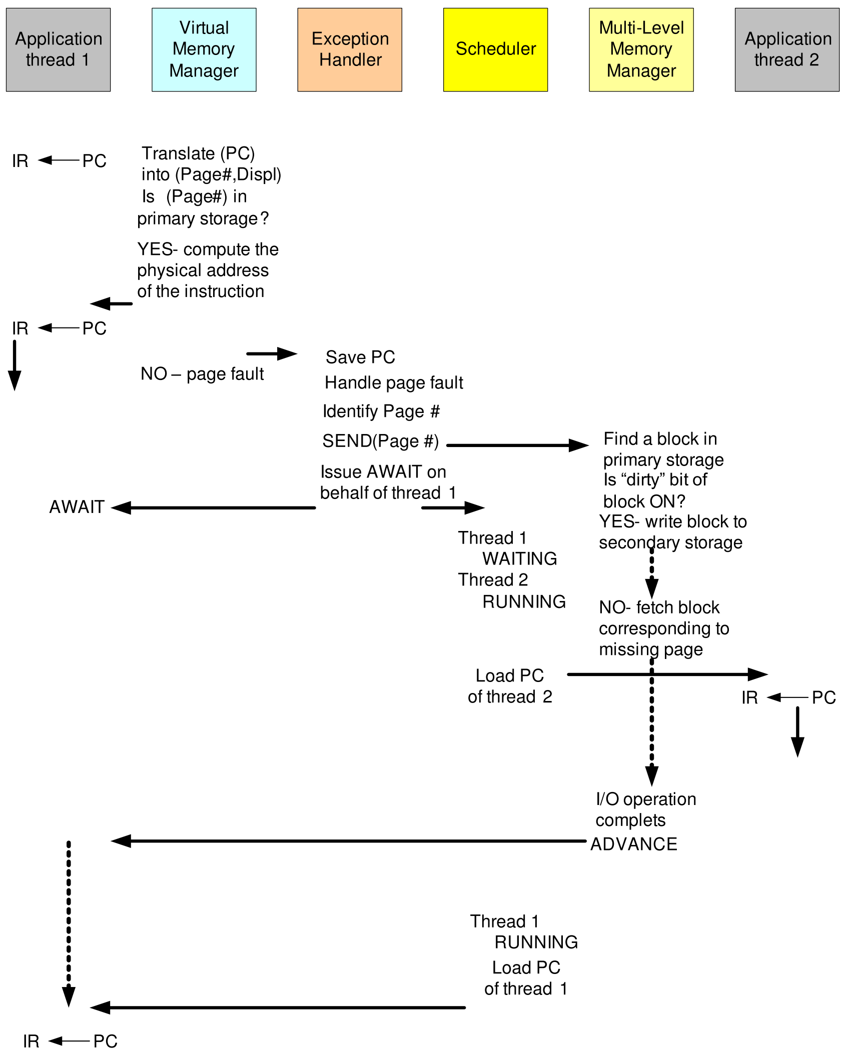

Application thread 1
Virtual Memory Manager
Exception Handler
Scheduler
Multi-Level Memory Manager
Application thread 2
IR ← PC
Translate (PC) into (Page#,Displ) Is (Page#) in primary storage?
YES- compute the physical address of the instruction
IR ← PC
NO – page fault
Save PC
Handle page fault
Identify Page #
SEND(Page #)
Find a block in primary storage Is "dirty" bit of block ON?
YES- write block to secondary storage
Issue AWAIT on behalf of thread 1
AWAIT
Thread 1 WAITING
Thread 2 RUNNING
NO- fetch block corresponding to missing page
Load PC of thread 2
IR ← PC
I/O operation complets
ADVANCE
Thread 1 RUNNING
Load PC of thread 1
IR ← PC

FIGURE 3.25

Context switching when a page fault occurs during the instruction fetch phase. IR is the instruction register containing the current instruction and PC is the program counter pointing to the next instruction to be executed. VMM attempts to translate the virtual address of the next instruction of thread 1 and encounters a page fault. Then thread 1 is suspended waiting for an event when the page is brought in the physical memory from the disk. The Scheduler dispatches thread 2. To handle the fault the Exception Handler invokes the MLMM.

- Avoiding limitations imposed by the physical resources, e.g., allow an application to run in a virtual address space of a standard size, rather than be restricted by the physical memory available on a system.

- Enhancing reliability and performance, as in the case of RAID systems mentioned in Section 2.7.

Sometimes concurrency is used to describe activities that appear to be executed simultaneously, though only one of them may be active at any given time. This is the case of processor virtualization when multiple threads appear to run concurrently on a single core processor. A thread can be suspended due to an external event and a context switch to a different thread takes place. The state of the suspended thread is saved, the state of another thread ready to run is loaded, and then the new thread is activated. The suspended thread will be re-activated at a later point in time.

Dealing with some of the effects of concurrency can be very challenging. Context switching could involve multiple components of a OS kernel including the Virtual Memory Manager (VMM), the Exception Handler (EH), the Scheduler (S), and the Multi-level Memory Manager (MLMM). When a page fault occurs during the fetching of the next instruction, multiple context switches are necessary as shown in Figure 3.25. The thread experiencing the fault is suspended and the scheduler dispatches another thread ready to run while in the mean time the exception handler invokes the MLMM.

If processor/core sharing seems complicated, the operation of a multi core processor running multiple virtual machines for applications running under different operating systems is considerably more complex. In this case resource sharing occurs at the level of an operating system for the threads of one application running under that OS and at hypervisor level for the threads of different virtual machines as shown in Figure 3.26.

Concurrency is often motivated by the desire to enhance the system performance. For example, in a pipelined computer architecture multiple instructions are in different phases of execution at any given time. Once the pipeline is full, a result is produced at every pipeline cycle; an n-stage pipeline could potentially lead to a speedup by a factor of n. There is always a price to pay for increased performance and in this example is design complexity and cost. An n-stage pipeline requires n execution units, one for each stage, as well as a coordination unit. It also requires a careful timing analysis in order to achieve the full speed-up.

This example shows that the management and the coordination of the concurrent activities increase the complexity of a system. The interaction between pipelining and virtual memory further complicates the functions of the kernel; indeed, one of the instructions in the pipeline could be interrupted due to a page fault and the handling of this case requires special precautions, as the state of the processor is difficult to define.

Concurrency – the application software. Concurrency is exploited by application software to speedup a computation and to allow a number of clients to access a service. Parallel applications partition the workload and distribute it to multiple threads running concurrently. Distributed applications, including transaction management systems and applications based on the client-server paradigm discussed in Chapter 4, use extensively concurrency to improve the response time. For example, a web server spawns a new thread when a new request is received thus, multiple server threads run concurrently. A main attraction for hosting Web-based applications is the cloud elasticity, the ability of a service running on a cloud to acquire resources as needed and to pay for these resources as they are consumed.

Communication channels allow concurrent activities to work in concert and coordinate. Communication protocols allow us to transform noisy and unreliable channels into reliable ones which deliver messages in order. As mentioned earlier, concurrent activities communicate with one another via shared memory or via message passing. Multiple instances of a cloud application, a server and the clients of

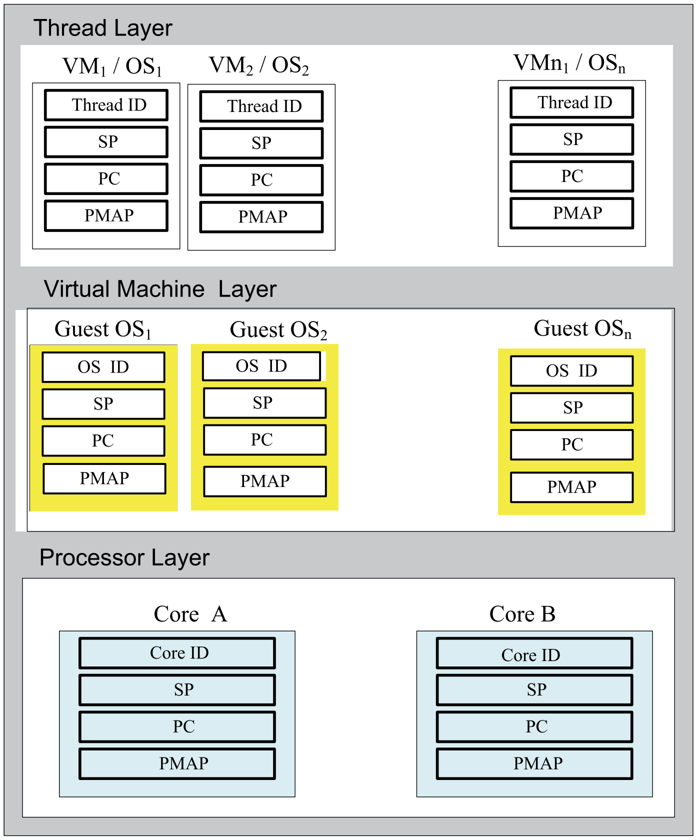


FIGURE 3.26

Thread multiplexing for a multicore server running multiple virtual machines. Multiple system data structures keep track of the contexts of all threads. The information required for context switching includes the ID, the stack pointer (SP), the program counter (PC) and the page table pointer (PMAP).

the service it provides, and many other applications communicate via message passing. The Message Passing Interface (MPI) supports both synchronous and asynchronous communication and it is often used by parallel and distributed applications. Message passing enforces modularity and prevents the communicating activities from *sharing their fate*; a server could fail without affecting the clients which did not use the service during the period the server was unavailable.

The communication patterns in case of a parallel application are more structured, while patterns of communication for concurrent activities of a distributed application are more dynamic and unstructured. Barrier synchronization requires the threads running concurrently to wait until all of them have completed the current task before proceeding to the next. Sometimes, one of the activities, a coordinator, mediates communication among concurrent activities, in other instances individual threads communicate directly with one another.

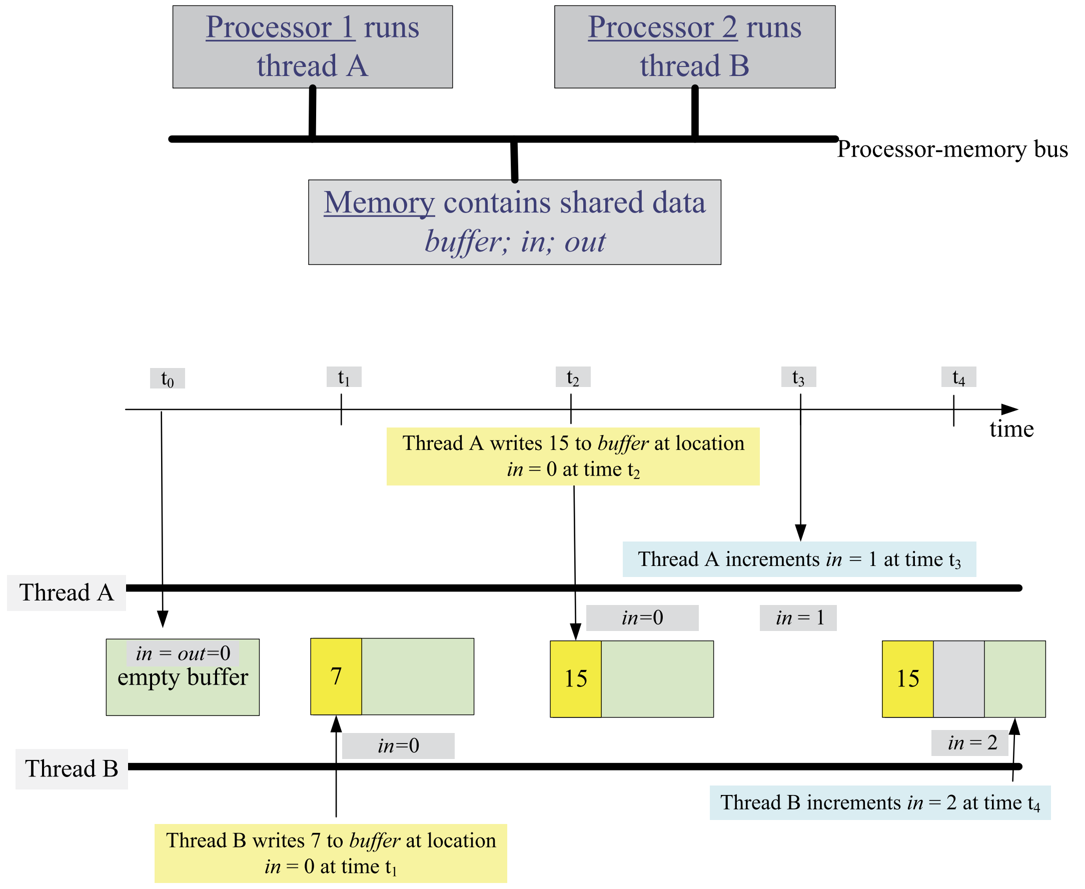

Processor 1 runs thread A
Processor 2 runs thread B
Processor-memory bus
Memory contains shared data
buffer; in; out
t_0
t_1
t_2
t_3
t_4
time
Thread A writes 15 to *buffer* at location *in* = 0 at time t_2
Thread A increments *in* = 1 at time t_3
Thread A
in = *out*=0
empty buffer
7
in=0
15
in=0
in = 1
15
in = 2
Thread B
Thread B increments *in* = 2 at time t_4
Thread B writes 7 to *buffer* at location *in* = 0 at time t_1

FIGURE 3.27

Race condition. Initially, at time t_0, the buffer is empty and *in* = 0. Thread B writes the integer 7 to the buffer at time t_1. Thread B is slow, incrementing the pointer *in* takes time and occurs at time t_4. In the meantime, at time $t_2 < t_4$, a faster thread A writes integer 15 to the buffer, overwrites the content of the first buffer location, and increments the pointer, *in* =1 at time t_3. Finally, at time t_4 thread B increments the pointer *in* = 2.

Coordination of concurrent computations could be quite challenging and involves overhead which ultimately reduces the speed-up of parallel computations. Concurrent execution could be very challenging, e.g., it could lead to *race conditions*, an undesirable effect when the results of concurrent execution depend on the sequence of events. Figure 3.27 illustrates a race condition when two threads communicate using a shared data *buffer*. Both threads can write to the *buffer* location pointed at by *in* and can read from *buffer* location pointed at by *out*. When both threads attempt to write at about the same time, the item written by the second thread overwrites the item written by the first thread.

3.11 CRITICAL SECTIONS, LOCKS, DEADLOCKS, AND ATOMIC ACTIONS

Parallel and distributed applications must take special precautions for handling shared resources. For example, consider a financial application where the shared resource is a user's account. A thread running on behalf of a transaction first accesses the user's account to read the current balance, then updates the balance, and, finally, writes back the new balance. If the thread is interrupted and another thread operating on the same account is allowed to proceed, before the first thread was able to complete the three steps for updating the account, the results of the financial transactions are incorrect.

Another challenge is to deal with a transaction involving the transfer from one account to another. A system crash after the completion of the operation on the first account will lead to an inconsistency, the amount debited from the first account is not credited to the second.

In these cases, as in many other similar situations, a multi-step operation should be allowed to proceed to completion without any interruptions, the operation should be *atomic*. An important observation is that such atomic actions should not expose the state of the system until the action is completed. Hiding the internal state of an atomic action reduces the number of states a system can be in thus, it simplifies the design and the maintenance of the system.

An atomic action is composed of several steps and each one of them may fail. When such a failure occures the system state should be restored to the state prior to the atomic action.

Locks and deadlocks. Concurrency requires a rigorous discipline when threads access shared resources. Concurrent reading of a shared data item is not restricted, while writing a shared data item should be subject to concurrency control. The race conditions discussed in Section 3.10 illustrate the problems created by a hazardous access to a shared resource, the buffer where both threads attempt to write to. The hazards posed by a lack of concurrency control are ubiquitous. Imagine an embedded system at a power plant where multiple events occur concurrently and the event signaling a dangerous malfunction of one subsystem is lost.

In all these cases only one thread should be allowed to modify shared data at any given time and other threads should only be allowed to read or write this data item only after the first one has finished. This process called *serialization* applies to segments of code called *critical sections* that need to be protected by control mechanisms called *locks* permitting access to one and only one thread at a time.

A lock is an object that grants access to a critical section. To enter a critical section a thread must acquire the lock of that section and, after finishing the thread must release the lock, as depicted in Figure 3.28. Only one thread should be successful when multiple threads attempt to acquire the lock at the same time; the other threads must wait until the lock is released.

One may argue that serialization by locking a data structure is against the very nature of concurrency, allowing multiple computations to run at the same, but, without some form of concurrency control it is not possible to guarantee the correctness of results of any computation. Lock-free programming [229] is rather challenging and will not be discussed in this chapter.

A lock should be seen as an antidote to uncontrolled concurrency and should be used sparingly and only to protect a critical section. Like any type of medication, locking has side effects, it does not only increase the execution time, but could lead to deadlocks. Indeed, another potential problem for concurrent execution of multiple processes/threads is the presence of deadlocks. A *deadlock* occurs when processes/threads competing with one another for resources are forced to wait for additional resources held by other processes/threads and none of the processes/threads can finish, see Figure 3.29.

The four Coffman conditions [114], must hold simultaneously for a deadlock to occur:

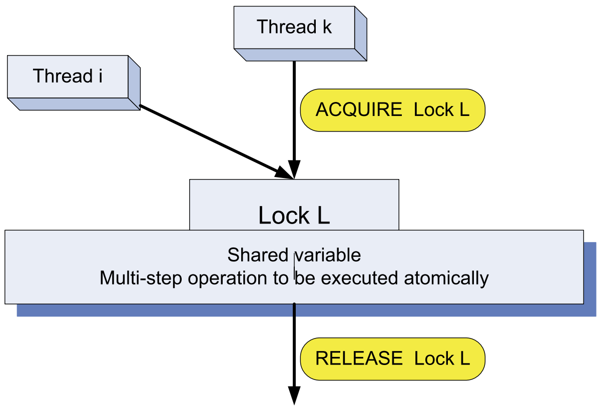


FIGURE 3.28

A lock protects a critical section consisting of multiple operations that have to be executed atomically.

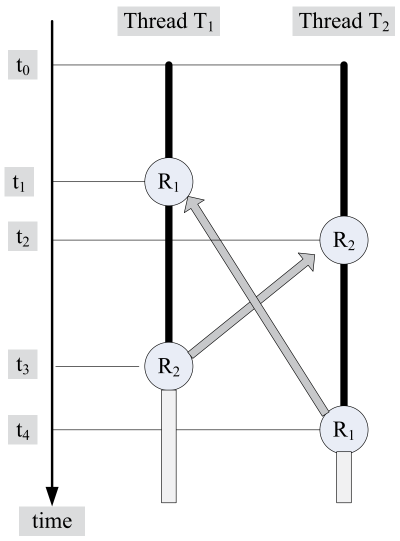


FIGURE 3.29

Thread deadlock. Threads T_1 and T_2 start concurrent execution at time t_0. Both need resources R_1 and R_2 to complete execution. T_1 acquires R_1 at time t_1 and T_2 acquires R_2 at time t_2. At time $t_3 > t_2$ thread T_1 attempts to acquire resource R_2 held by thread T_2 and blocks waiting for it to be released. At time $t_4 > t_3$ thread T_2 attempts to acquire resource R_1 held by thread T_1 and blocks waiting for it to be released. Neither thread can make any progress.

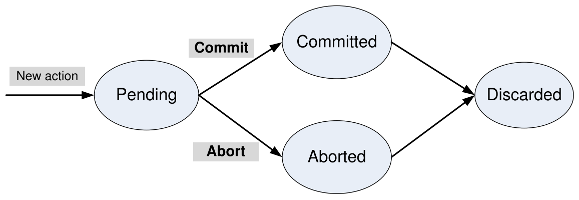


FIGURE 3.30

The states of an *all-or-nothing* action.

1. *Mutual exclusion;* at least one resource must be non-sharable, only one process or one thread may use the resource at any given time.
2. *Hold and wait;* at least one process or one thread must hold one or more resources and wait for others.
3. *No-preemption;* the scheduler or a monitor should not be able to force a process or a thread holding a resource to relinquish it.
4. *Circular wait;* given the set of n processes or threads $\{P_1, P_2, P_3, \dots, P_n\}$, P_1 should wait for a resource held by P_2, P_2 should wait for a resource held by P_3, and so on, P_n should wait for a resource held by P_1.

There are other potential problems related to concurrency. When two or more processes/threads continually change their state in response to changes in the other processes we have a *livelock* condition; the result is that none of the processes can complete its execution. Very often processes/threads running concurrently are assigned priorities and scheduled based on these priorities. *Priority inversion* occurs when a higher priority process/task is indirectly preempted by a lower priority one.

Atomicity. The discussion of the transaction system suggests that an analysis of atomicity should pay special attention to the basic operation of updating the value of an object in storage. Several machine instructions must be executed to modify the contents of a memory location: (i) load the current value in a register; (ii) modify the contents of the register; and (iii) store back the result.

Atomicity cannot be implemented without hardware support; indeed, the instruction set of most processors support the *Test-and-Set* instruction which writes to a memory location and returns the old content of that memory cell as non-interruptible operations. Other architectures support *Compare-and-Swap*, an atomic instruction comparing the contents of a memory location to a given value and, only if the two values are the same, the contents of that memory location is updated atomically.

Two flavors of atomicity can be distinguished: *all-or-nothing* and *before-or-after* atomicity. *All-or-nothing* means that either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted. In our previous examples a transaction is either carried out successfully, or the record targeted by the transaction is returned to its original state. The states of an *all-or-nothing* action are shown in Figure 3.30.

To guarantee the all-or-nothing property of an action we have to distinguish preparatory actions, which can be undone, from irreversible ones, such as the alteration of the only copy of an object. Such preparatory actions are: allocation of a resource, fetching a page from secondary storage, allocation of memory on the stack, and so on. One of the golden rules of data management is never to change the

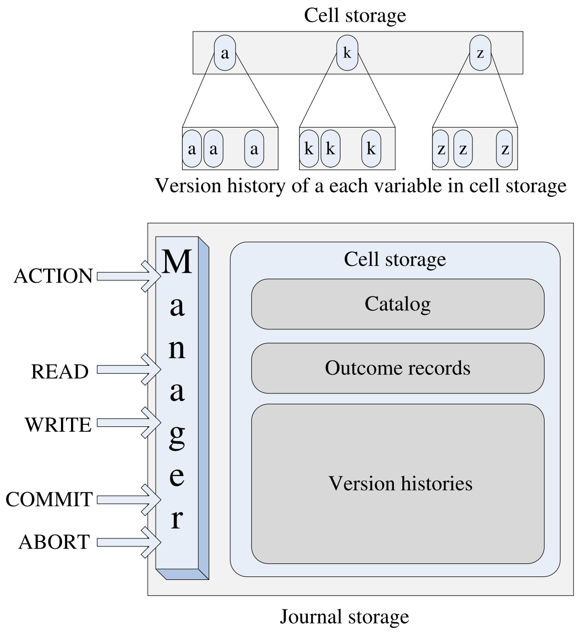


FIGURE 3.31

Storage models. Cell storage does not support all-or-nothing actions. When we maintain the version histories it is possible to restore the original content but we need to encapsulate the data access and provide mechanisms to implement the two phases of an atomic all-or-nothing action. The journal storage does precisely that.

only copy. Maintaining the history of changes of an object and a log of all activities allow us to deal with system failures and to ensure consistency.

An all-or-nothing action consists of a *pre-commit* and a *post-commit* phase; during the former it should be possible to backup from it without leaving any trace, while the later phase should be able to run to completion. The transition from the first to the second phase is called a *commit point*. During the *pre-commit* phase all steps necessary to prepare the post-commit phase, such as check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space, must be carried out; during this phase no results should be exposed and no actions that are irreversible should be carried out. Shared resources allocated during the pre-commit cannot be released until after the commit point. The commit step should be the last step of an all-or-nothing action.

A discussion of storage models illustrates the effort required to support all-or-nothing atomicity, see Figure 3.31. The common storage model implemented by hardware is the so-called *cell storage*, a collection of cells each capable to hold an object, e.g., the primary memory of a computer where each cell is addressable. Cell storage does not support all-or-nothing actions, once the contents of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell.

To be able to restore a previous value we have to maintain a *version history* for each variable in the cell storage. The storage model that supports all-or-nothing actions is called *journal storage*. Now the cell storage is no longer accessible to the action but the access is mitigated by a *storage manager*.

In addition to the basic primitives to *Read* an existing value and to *Write* a new value in cell storage, the storage manager uniquely identifies an action that changes the value in cell storage and when the action is aborted it is able to retrieve the version of the variable before the action and restore it. When the action is committed then the new value should be written to the cell.

Figure 3.31 shows that for a journal storage, in addition to the version histories of all variables affected by the action, we have to implement a catalog of variables and also to maintain a record identifying each new action. A new action first invokes the *Action* primitive; at that time an outcome record uniquely identifying the action is created. Then, every time the action accesses a variable, the version history is modified and, finally, the action either invokes a *Commit* or an *Abort* primitive. In the journal storage model the action is atomic and follows the state transition diagram in Figure 3.30.

Before-or-after atomicity means that, from the point of view of an external observer, the effect of multiple actions is as if these actions have occurred one after another, in some order; a stronger condition is to impose a sequential order among transitions. In our example the transaction acting on two accounts should either debit the first account and then credit the second one, or leave both accounts unchanged. The order is important, as the first account cannot be left with a negative balance.

Atomicity is a critical concept for our efforts to build reliable systems from unreliable components and, at the same time, to support as much parallelism as possible for better performance. Atomicity allows us to deal with unforeseen events and to support coordination of concurrent activities. The unforeseen event could be a system crash, a request to share a control structure, the need to suspend an activity, and so on; in all these cases we have to save the state of the process or of the entire system in order to be able to restart it at a later time.

As atomicity is required in many contexts, it is desirable to have a systematic approach rather than an ad hoc one. A systematic approach to atomicity must address several delicate questions:

- How to guarantee that only one atomic action has access to a shared resource at any given time.
- How to return to the original state of the system when an atomic action fails to complete.
- How to ensure that the order of several atomic actions leads to consistent results.

Answers to these questions increase the complexity of the system and often generate additional problems. For example, access to shared resources can be protected by locks, but when there are multiple shared resources protected by locks concurrent activities may deadlock. A *lock* is a construct which enforces sequential access to a shared resource; such actions are packaged in the *critical sections* of the code. If the lock is not set, a thread first locks the access, then enters the critical section and finally unlocks it; a thread wishing to enter the critical section finds the lock set and waits for the lock to be reset. A lock can be implemented using the hardware instructions supporting atomicity.

Semaphores and monitors are more elaborate structures ensuring serial access. Semaphores force processes to queue up when the lock is set and are released from this queue and allowed to enter the critical section one by one. Monitors provide special procedures to access the shared data, see Figure 3.32. *The mechanisms for the process coordination we described require the cooperation of all activities,* the same way traffic lights prevent accidents only as long as the drivers follow the rules.

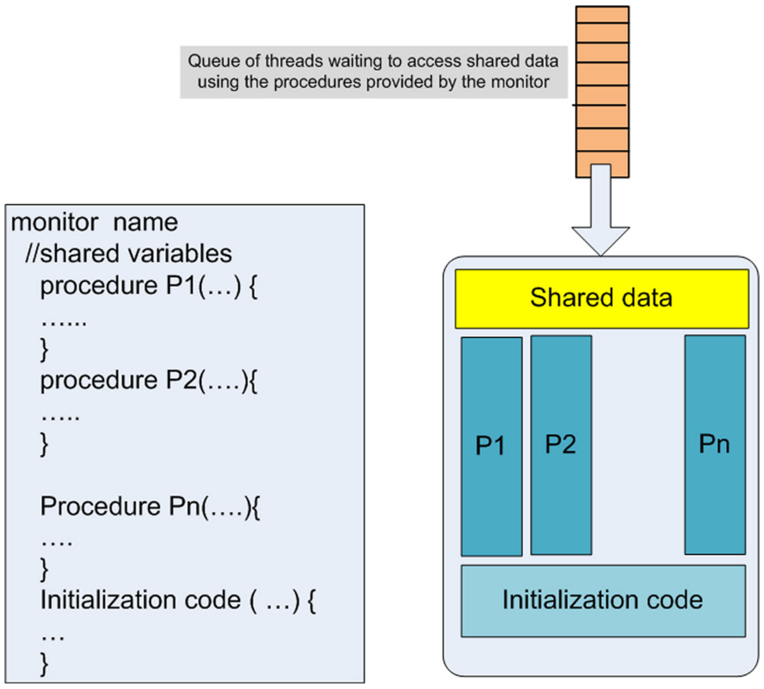


FIGURE 3.32

A monitor provides special procedures to access the data in a critical section.

3.12 CONSENSUS PROTOCOLS

Consensus is a pervasive problem in many areas of human endeavor; consensus is the process of agreeing to one of several alternates proposed by a number of agents. We restrict our discussion to the case of a distributed system when the agents are a set of processes expected to reach consensus on a single proposed value.

No fault-tolerant consensus protocol can guarantee progress [174], but protocols which guarantee freedom from inconsistencies (safety) have been developed. A family of protocols to reach consensus based on a finite state machine approach is called *Paxos*.[3]

A fair number of contributions to the family of Paxos protocols are discussed in the literature. Leslie Lamport has proposed several versions of the protocol including Disk Paxos, Cheap Paxos, Fast Paxos, Vertical Paxos, Stoppable Paxos, Byzantizing Paxos by Refinement, Generalized Consensus and

[3]Paxos is a small Greek island in the Ionian Sea; a fictional consensus procedure is attributed to an ancient Paxos legislative body. The island had a part-time parliament as its inhabitants were more interested in other activities than in civic work; "the problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing" according to Leslie Lamport [291] (for additional papers see http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html).

Paxos, and Leaderless Byzantine Paxos. Lamport has also published a paper on the fictional part-time parliament in Paxos [291] and a layman's dissection of the protocol [292].

The *consensus service* consists of a set of n processes. *Clients* send requests to processes and propose a value and wait for a response; the goal is to get the set of processes to reach consensus on a single proposed value. The *basic Paxos* protocol is based on several assumptions about the processors and the network:

- The processes run on processors and communicate through a network; the processors and the network may experience failures, but not Byzantine failures. A Byzantine failure is a fault presenting different symptoms to different observers. In a distributed system a Byzantine failure could be an *omission failure*, e.g., a crash failure, failure to receive a request or to send a response; it could also be a *commission failure*, e.g., process a request incorrectly, corrupt the local state, and/or send an incorrect or inconsistent response to a request.
- The processors: (i) operate at arbitrary speeds; (ii) have stable storage and may rejoin the protocol after a failure; (iii) can send messages to any other processor.
- The network: (i) may lose, reorder, or duplicate messages; (ii) messages are sent asynchronously and may take arbitrary long time to reach the destination.

The *basic Paxos* considers several types of entities: (a) *client,* an agent that issues a request and waits for a response; (b) *proposer,* an agent with the mission to advocate a request from a client, convince the acceptors to agree on the value proposed by a client, and to act as a coordinator to move the protocol forward in case of conflicts; (c) *acceptor*, an agent acting as the fault-tolerant "memory" of the protocol; (d) *learner,* an agent acting as the replication factor of the protocol and taking action once a request has been agreed upon; and finally (e) the *leader,* a distinguished proposer.

A *quorum* is a subset of all acceptors. A proposal has a proposal number pn and contains a value v. Several types of requests flow through the system such as *prepare* and *accept*.

In a typical deployment of the algorithm an entity plays three roles, as proposer, acceptor, and learner. Then the flow of messages can be described as follows [292]: "clients send messages to a leader; during normal operations the leader receives the client's command, assigns it a new command number i, and then begins the i-th instance of the consensus algorithm by sending messages to a set of acceptor processes." By merging the roles, the protocol "collapses" into an efficient client-master-replica style protocol.

A proposal consists of a pair, a unique proposal number and a proposed value, (pn, v); multiple proposals may propose the same value v. A value is chosen if a simple majority of acceptors have accepted it. We need to guarantee that at most one value can be chosen, otherwise there is no consensus. The two phases of the algorithm are:

Phase I.

1. *Proposal preparation:* a proposer (the leader) sends a proposal $(pn = k, v)$. The proposer chooses a proposal number $pn = k$ and sends a *prepare message* to a majority of acceptors requesting:

   - that a proposal with $pn < k$ should not be accepted;
   - the $pn < k$ of the highest number proposal already accepted by each acceptor.

2. *Proposal promise:* An acceptor must remember the proposal number of the highest proposal number it has ever accepted as well as the highest proposal number it has ever responded to. The acceptor can accept a proposal with $pn = k$ if and only if it has not responded to a prepare request with $pn > k$; if it has already replied to a prepare request for a proposal with $pn > k$ then it should not reply. Lost messages are treated as an acceptor that chooses not to respond.

Phase II.

1. *Accept request:* if the majority of acceptors respond, then the proposer chooses the value v of the proposal as follows:

   - the value v of the highest proposal number selected from all the responses;
   - an arbitrary value if no proposal was issued by any of the proposers.

   The proposer sends an *accept request* message to a quorum of acceptors including $(pn = k, v)$
2. *Accept:* If an acceptor receives an *accept message* for a proposal with the proposal number $pn = k$ it must accept it if and only if it has not already promised to consider proposals with a $pn > k$. If it accepts the proposal it should register the value v and send an *accept* message to the proposer and to every learner; if it does not accept the proposal it should ignore the request.

The following properties of the algorithm are important to show its correctness: (1) a proposal number is unique; (2) any two sets of acceptors have at least one acceptor in common; and (3) the value sent out in Phase 2 of the algorithm is the value of the highest numbered proposal of all the responses in Phase 1.

Figure 3.33 illustrates the flow of messages for the consensus protocol. A detailed analysis of the message flows for different failure scenarios and of the properties of the protocol can be found in [292]. We only mention that the protocol defines three safety properties: (1) non-triviality – the only values that can be learned are proposed values; (2) consistency – at most one value can be learned; and (3) liveness – if a value v has been proposed, eventually every learner will learn some value, provided that sufficient processors remain non-faulty. Figure 3.34 shows the message exchange when there are three actors involved.

A distributed coordination system discussed in Section 7.4, the ZooKeeper, borrows several ideas from the Paxos algorithm:

- A leader proposes values to the followers;
- Leaders wait for acknowledgments from a quorum of followers before considering a proposal committed (learned);
- Proposals include epoch numbers, which are similar to ballot numbers in Paxos.

In Section 6.6 we discuss Chubby, a locking service based on the Paxos algorithm.

3.13 LOAD BALANCING

A general formulation of the load balancing problem is to evenly distribute $\mathcal{N}$ objects to $\mathcal{P}$ places. Another formulation of the load balancing is in the context of placing m balls into $n < m$ bins, chosen independently and uniformly at random. The question in this case is to find the maximum number of

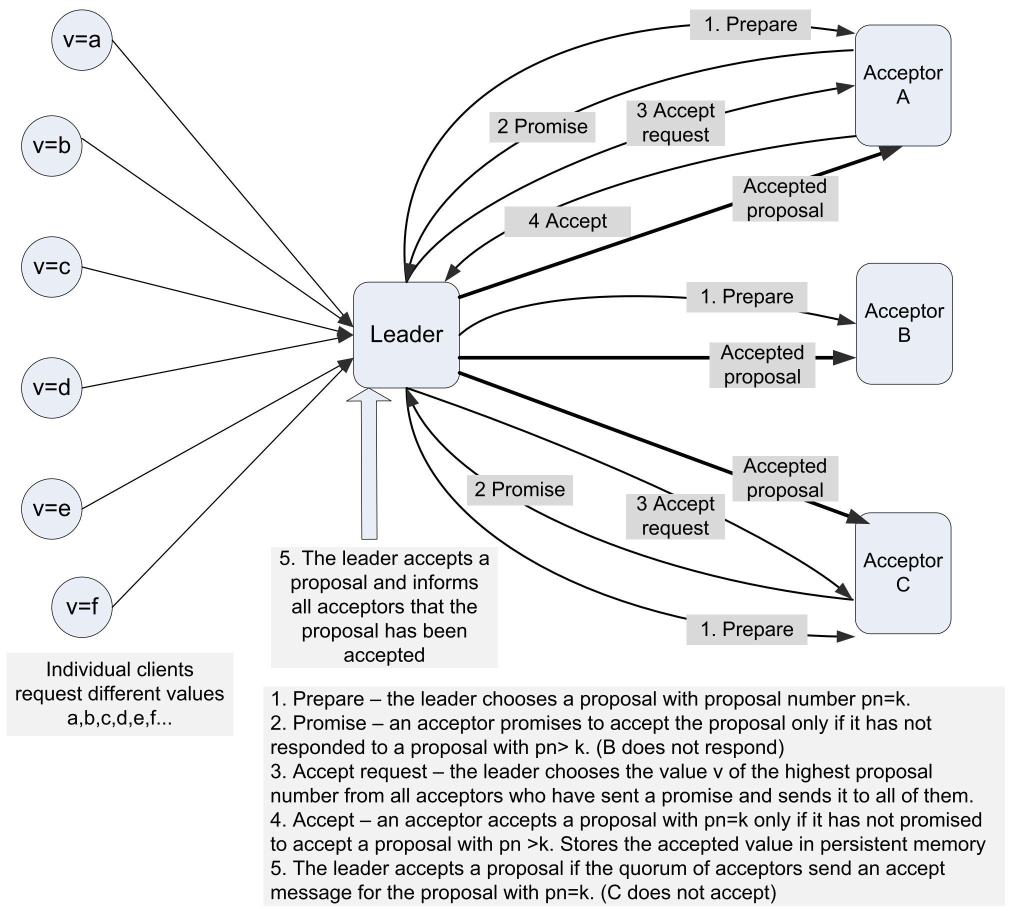

v=a
v=b
v=c
v=d
v=e
v=f
Individual clients request different values a,b,c,d,e,f...
Leader
1. Prepare
2 Promise
3 Accept request
4 Accept
Accepted proposal
Acceptor A
Acceptor B
Acceptor C
5. The leader accepts a proposal and informs all acceptors that the proposal has been accepted
1. Prepare – the leader chooses a proposal with proposal number pn=k.
2. Promise – an acceptor promises to accept the proposal only if it has not responded to a proposal with pn> k. (B does not respond)
3. Accept request – the leader chooses the value v of the highest proposal number from all acceptors who have sent a promise and sends it to all of them.
4. Accept – an acceptor accepts a proposal with pn=k only if it has not promised to accept a proposal with pn >k. Stores the accepted value in persistent memory
5. The leader accepts a proposal if the quorum of acceptors send an accept message for the proposal with pn=k. (C does not accept)

FIGURE 3.33

The flow of messages for the Paxos consensus algorithm. Individual clients propose different values to the leader who initiates the algorithm. Acceptor A accepts the value in message with proposal number pn=k; acceptor B does not respond with a promise while acceptor C responds with a promise, but ultimately does not accept the proposal.

balls in any bin. Load balancing can also be formulated in the context of hashing as the problem of placing m items sequentially in $n < m$ buckets and the question is to determine the maximum time to find an item.

The load balancing problem in a distributed system is formulated as follows: given a set $\mathcal{T}$ of tasks distribute them to a set of $\mathcal{P}$ processors which compute at the same rate, such that only one task can run at any given time on one processor; there is no preemption and each task runs to completion. Knowing the execution time of each task the question is how to distribute them to minimize the completion time. Unfortunately, load balancing, as well as scheduling problems, are NP-complete [183].

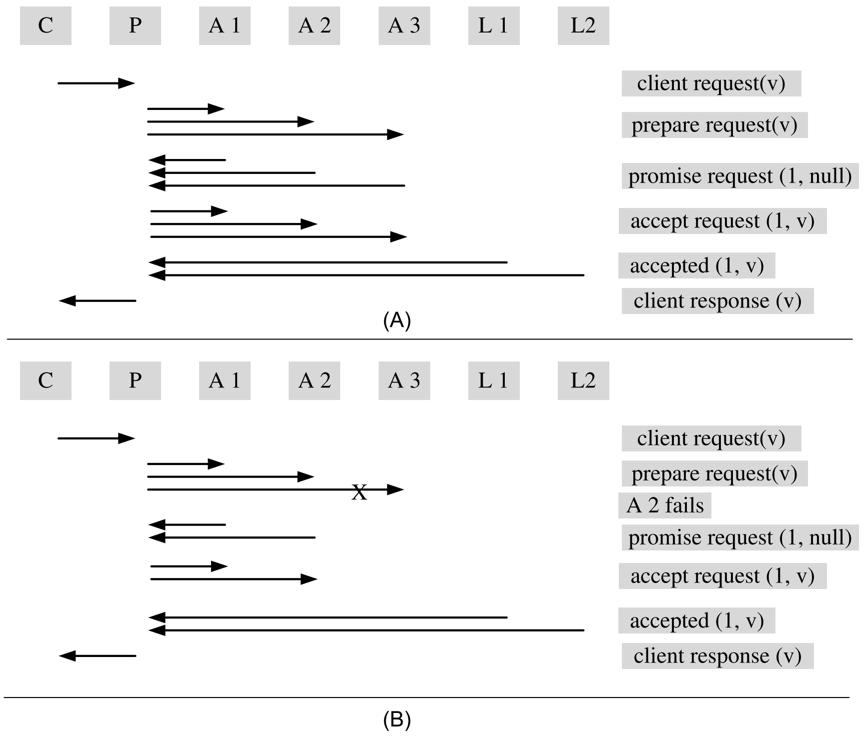


FIGURE 3.34

The basic Paxos with three actors: proposer (P), three acceptors (A1, A2, A3), and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are (A) Successful first round when there are no failures. (B) Successful first round of Paxos when an acceptor fails.

The importance of load balancing is undeniable and practical solutions to overcome the algorithmic complexity are widely used. For example, randomization suggests to distribute the tasks to processors chosen independently and uniformly at random. This random distribution strategy should lead to an almost equal load of processors, provided that there are enough tasks and that the distribution of the task execution times is rather narrow. Several other heuristics are used in practice.

The balls-and-bins model. The load balancing problem is often discussed using the balls-and-bins model. In this model we define the *load* of a bin as the number of balls in the bin. The question asked is what is $\max(\mathcal{L}_i),\ 1 \le i \le n$, the maximum load in any bin, once all the n balls have chosen a bin independently and uniformly at random. The answer is that with *high probability*, namely with a probability $p \ge 1 - \mathcal{O}(1/n)$ [197]

$$\max(\mathcal{L}_i) \approx \frac{\log n}{\log \log n}. \tag{3.34}$$

This result applies also to task scheduling in a distributed system. Interestingly enough this solution does not involve communication among the tasks, the processors, or among the tasks and the processors.

A rather surprising result proven in [44] is that a better load balance is achieved when the balls are placed sequentially and for each ball we choose two bins independently and uniformly at random, then place the ball into the less full bin. In this case the maximal load, $\max(\mathcal{L}_i^2),\ 1 \leq i \leq n$ is

$$\max(\mathcal{L}_i^2) \leq \frac{\log\log n}{\log 2} + \mathcal{O}(1) \ \text{ with high probability.} \tag{3.35}$$

This result, discussed in depth in [348,350] and called *the power of two choices* shows that having two or more choices leads to an exponential improvement of the load balance. The following discussion is based on the *layered induction approach* introduced by Azar [44], where the number of bins that contain at least j balls conditioned on the number of bins that contain at least $(j-1)$ balls is inductively bound.

If the choice for each ball is extended from 2 to d bins then the result is further improved. The GREEDY algorithm in [44] considers an (m, n, d) problem, n initially empty bins, m balls to be placed sequentially in the bins, and d choices made independently and uniformly at random with replacement. Each ball is placed in the least loaded of the d bins and ties are broken arbitrarily. Then the maximum load, the maximum number of balls in a bin, has an upper bound

$$\max(\mathcal{L}_i^d) \leq \frac{\log\log n}{\log d} + \mathcal{O}(1) \ \text{ with high probability.} \tag{3.36}$$

An intuitive justification of this results is discussed next. Call β_k the number of bins with *at least* k balls stacked on top of one another in the order they have been placed in the bin. The *hight* of the top ball in a bin with k balls is k.

We wish to determine β_{k+1}, a high probability upper bound of the cardinality of bins loaded with at least $k+1$ balls. A bin will contain at least $k+1$ balls if in the previous round it had at least k balls. Recall that there are at least β_k such bins thus, the probability of choosing a bin with k or more balls from the set of n bins is β_k/n. But there are $d > 2$ choices, therefore the probability that a ball lands in a bin already containing k or more balls drops at each step at least quadratically and is

$$p_{n,k,d} = \left(\frac{\beta_k}{n}\right)^d . \tag{3.37}$$

The number of balls with hight at least $(k+1)$ is dominated by a Bernoulli random variable with the probability of success equals to $p_{n,k,d}$. This implies that

$$\beta_{k+1} \leq c \times \left[n \times \left(\frac{\beta_k}{n}\right)^d \right] \tag{3.38}$$

with c a constant. It follows that after $j = \mathcal{O}(\log\log n)$ steps the fraction β_k/n drops below $1/n$ thus, $\beta_j < 1$.

The sequential ball placement required in the algorithms discussed in this section and the decision to choose one of two bins, or one of d bins, deserves further scrutiny. It implies either a centralized system where an agent makes the choice, or some form of communication between the balls allowing them to reach an agreement about the placement strategy. But communication is expensive, for example during the time of a short message exchange a modern processor could execute several billion floating point operations.

A trade-off between load balance and communication is inevitable, we can reduce the maximum load only through coordination thus, the price to pay is increased communication. In a distributed system a server is not aware of the tasks it has not communicated with, while tasks are unaware of the actions of other tasks and may only know the load of the servers. Global coordination among tasks is prohibitively expensive.

Parallelization of the randomized load balance. One of the questions examined in [348,350] is how to parallelize the randomized load balance. Once again, an extended balls-and-bins model is used and the goal is to minimize the maximum load and the communication complexity. Each one of the m balls begins by choosing d out of the n bins as prospective destination.

The choices are made independently and uniformly at random with replacement of balls and the final decision of the destination bin requires r rounds of communication, with two stages per round. At each stage communication is done in parallel using short messages including an ID or an index. In the first stage each ball sends messages to all prospective bins and in the second stage each bin sends messages to all the balls the bin has received a message from. During the final round the balls commit to a bin.

The strategies analyzed are symmetric, all balls-and-bins use the same algorithm and all possible destinations are chosen independently and uniformly at random. An algorithm is *asynchronous* if an entity, a ball or a bin, does not have to wait for a round to complete; it only has to wait for messages addressed to it, rather than waiting for messages addressed to another entity. A round is *synchronous* if a barrier synchronization is required between some pairs of rounds.

A load lower bound for a broad class of algorithms like the one in [44] with r-rounds of communication derived in [350] is

$$\Omega\left(\sqrt[r]{\frac{\log n}{\log\log n}}\right) \tag{3.39}$$

with at least constant probability. Therefore, no algorithm can achieve a maximum load $\mathcal{O}(\log\log n)$ with high probability in a constant number of communication rounds.

A random graph $G(v, e)$ is used to represent the model and to derive this result. Each bin is a vertex v in this graph and each ball is an undirected edge, e. When $d = 2$ the vertices of the two edges of a ball correspond to the two prospective bins. There are no self-loops in this graph where S denotes the set of edges. Multiple edges correspond to two balls that have chosen the same pair of bins. Selection of a bin by a ball transforms the undirected edge representing the ball into a directed edge oriented toward the vertex, or bin, the ball chooses as its destination. The goal is to minimize the maximum in-degree over all vertices of the graph, in other words to avoid conflicts.

$\mathcal{N}(e)$, the *neighborhood* of an edge $e \in S$ is the set of all edges incident to an endpoint of e and

$$\mathcal{N}(S) = \cup_{e \in S}\mathcal{N}(e). \tag{3.40}$$

Similarly $\mathcal{N}(v)$ is the *neighborhood* of a vertex v. $\mathcal{N}_l(e)$, the *l-neighborhood* of an edge $e \in S$ is defined inductively as

$$\mathcal{N}_1(e) = \mathcal{N}(e), \quad \mathcal{N}_l(e) = \mathcal{N}(\mathcal{N}_{l-1}(e)). \tag{3.41}$$

$\mathcal{N}_{l,x}(e)$, the *(l, x) – neighborhood* of an edge $e(x, y)$ is defined inductively as

$$\mathcal{N}_{1,x}(e) = \mathcal{N}(x) - \{e\} \;\; \mathcal{N}_{l,x}(e) = \mathcal{N}(\mathcal{N}_{l-1,x}(e)) - \{e\}. \tag{3.42}$$

In an r round protocol the balls make their choice in the final round, therefore each ball knows everything only about the balls in its $(r-1)$ neighborhood.

A ball $e = (x; y)$ learns from each bin about its *l-neighborhood* consisting of two subgraphs corresponding to $\mathcal{N}_{l,x}(e)$ and $\mathcal{N}_{l,y}(e)$. When the two subgraphs of the ball's *l-neighborhood* are isomorphic rooted trees, with the roots x and y, we say that the ball has a symmetric *l-neighborhood* or, that the ball is *confused*, and then the ball chooses the destination bin using a fair coin flip.

A tree of depth r where the root has degree T and each internal node has $T-1$ children is called a (T, r)-*rooted, balanced tree*. A (T, r) tree in graph G is *isolated* if it is a connected component of G with no edges of multiplicity greater than one. A random graph with n vertices and n edges contains an isolated $(T, 2)$ with

$$T = \left(\sqrt{2} - \mathcal{O}(1)\right)\sqrt{\frac{\log n}{\log\log n}} \tag{3.43}$$

with constant probability as shown in [350]. A corollary of this statement is that any non-adaptive, symmetric load distribution strategy for the balls-and-bins problem with n balls and n bins and with $d = 2$ and $r = 2$, has a final load of at least

$$\left(\frac{\sqrt{2}}{2} - \mathcal{O}(1)\right)\sqrt{\frac{\log n}{\log\log n}} \tag{3.44}$$

with at least constant probability. Indeed, half of the confused balls (edges) in an isolated $(T, 2)$ tree adjacent to the root will orient themselves towards the root as we have assumed that the balls flip an unbiased coin to choose the bin. This result can be extended for a range of r and d.

The balls-and-bins model has applications to hashing. The hashing implementation discussed in [275] uses a single hash function to map keys to entries in a table and in case of a collision, i.e., when two or more keys map to the same table entry, all the colliding keys are stored in a linked list called a *chain*. The table entries are heads of chains and the longest search time occurs for the longest chain. The length of the longest chain is $\mathcal{O}(\frac{\log n}{\log\log n})$ with a high probability when n keys are inserted into a table with n entries and each key is mapped to an entry of the table independently and uniformly, a process known as *perfect random hashing*.

The search time can be substantially reduced by using two hash functions and placing an item in the shorter of the two chains [263]. To search for an element, we have to search through the chains linked to the two entries given by both hash functions. If the n keys are sequentially inserted into the table with n entries, the length of the longest chain thus, the maximum time to find an item, is $\mathcal{O}(\log\log n)$ with high probability.

The two-choice paradigm can be applied effectively to routing virtual circuits in interconnection networks with low congestion. The paradigm is also used to optimize the emulation of shared-memory multiprocessor (SMM) systems on a distributed-memory multiprocessor systems (DMM). The emulation algorithm should minimize the time needed by the DMM to emulate one step of the SMM.

The layered induction approach is also used in dynamic scenarios, e.g., when a new ball is inserted in the system [349]. Another technique to analyze load balancing based on the balls-and-bins model is the *witness tree*. To compute a bound for the probability of a "heavily-loaded" system event we have to identify a *witness tree* of events and then estimate the probability that the witness tree occurs. This probability can be bounded by enumerating all possible witness trees and summing their individual probabilities of occurrence.

3.14 MULTITHREADING AND CONCURRENCY IN JAVA; FLUMEJAVA

Java is a general-purpose computer programming language designed with portability in mind at Sun Microsystems.[4] Java applications are typically compiled to bytecode and can run on a Java Virtual Machine (JVM) regardless of the computer architecture. Java is a class-based, object-oriented language with support for concurrency. It is one of the most popular programming language and it is widely used for a wide range of applications running on mobile devices and computer clouds.

Java Threads. Java supports processes and threads. Recall that a process has a self-contained execution environment, has its own private address space and run-time resources. A thread is a lightweight entity within a process. A Java application starts with one thread, the *main thread* which can create additional threads.

Memory consistency errors occur when different threads have inconsistent views of the same data. Synchronized methods and synchronized statements are the two idioms for synchronization. Serialization of *critical sections* is protected by specifying the *synchronized* attribute in the definition of a class or method. This guarantees that only one thread can execute the critical section and each thread entering the section sees the modification done. Synchronized statements must specify the object that provides the intrinsic lock.

The current versions of Java, support atomic operations of several datatypes with methods such as *getAndDecrement(), getAndIncrement()* and *getAndSet()*. An effective way to control data sharing among threads is to share only immutable data among threads. A class is made *immutable* by marking all its fields as *final* and declaring the class as *final*.

A *Thread* in the *java.lang.Thread* class executes an object of type *java.lang.Runnable*. The *java.util.concurrent* package provides better support for concurrency than the *Thread* class. This package reduces the overhead for thread creation and prevents too many threads overloading the CPU and depleting the available storage. A *thread pool* is a collection of *Runnable* objects and contains a queue of tasks waiting to get executed.

[4]The design of Java was initiated in 1991 by James Gosling, Mike Sheridan, and Patrick Naughton with a C/C++-style syntax. Five principles guided its design: (1) simple, object-oriented, and familiar; (2) architecture-neutral and portable; (3) robust and secure; (4) interpreted, threaded, and dynamic; and (5) high performance. Java 1.0 was released in 1995. Java 8 is the only version currently supported for free by Oracle, a company that acquired Sun Microsystems in 2010.

Threads can communicate with one another via *interrupts*. A thread sends an interrupt by invoking an *interrupt* on the *Thread* object to the thread to be interrupted. The thread to be interrupted is expected to support its own interruption. *Thread.sleep* causes the current thread to suspend execution for a specified period.

The executor framework works with Runnable objects which cannot return results to the caller. The alternative is to use *java.util.concurrent.Callable*. A *Callable* object returns an object of type *java.util.concurrent.Future*. The *Future* object can be used to check the status of a *Callable* object and to retrieve the result from it. Yet, the *Future* interface has limitations for the asynchronous execution and the *CompletableFuture* extends the functionality of the *Future* interface for asynchronous execution.

Non-blocking algorithms based on low-level atomic hardware primitives such as compare-and-swap (CAS) are supported by Java 5.0 and later versions. The fork-join framework introduced in Java 7 supports the distribution of work to several workers and then waiting for their completion. The *join* method allows one thread to wait for completion of another.

FlumeJava. A Java library used to develop, test, and run efficient data parallel pipelines is described in [92]. FlumeJava is used to develop data parallel applications such as MapReduce discussed in Section 7.5.

At the heart of the system is the concept of *parallel collection* which abstracts the details of data representation. Data in a parallel collection can be an in-memory data structure, one or more files, BigTable discussed in Section 6.9, or a MySQL database. Data-parallel computations are implemented by composition of several operations for parallel collections.

In turn, parallel operations are implemented using *deferred evaluation*. The invocation of a parallel operation records the operation and its arguments in an internal graph structure representing the execution plan. Once completed, the execution plan is optimized.

The most important classes of the FlumeJava library are the $Pcollection < T >$ used to specify a immutable bag of elements of type T and the $PTable < K, V >$ representing an immutable multi-map with keys of type K and values of type V. The internal state of a $PCollection$ object is either *deferred* or *materialized*, i.e. not yet computed or computed, respectively. The $PObject < T >$ class is a container for a single Java object of type T and can be either deferred or materialized.

$parallelDo()$ supports element-wise computation over an input $PCollection < T >$ to produce a new output $PCollection < S >$. This primitive takes as the main argument a $DoFn < T, S >$, a function-like object defining how to map each value in the input into zero or more values in the output. In the following example from [92] $collectionOf(strings())$ specifies that the $parallelDo()$ operation should produce an unordered $PCollection$ whose $String$ elements should be encoded using UTF-8[5]

[5]UTF-8 is a character encoding standard defined by Unicode capable of encoding all possible characters. The encoding is variable-length and uses 8-bit code units.

```
Pcollection<String> words =
     lines.parallelDo(new DoFn<String, String> ( ) {
         void  process (String line, EmitFn<String> emitFn {
              for (String word : splitIntoWords(line) ) {
                    emitFn.emit(word);
               }
           }
       }, collectionOf(strings( ) ));
```

Other primitive operations are $groupByKey()$, $combineValues()$ and $flatten()$.

- $groupByKey()$ converts a multi-map of type $PTable < K, V >$. Multiple key/value pairs may share the same key into a uni-map of type $PTable < K, Collection < V >>$ where each key maps to an unordered, plain Java Collection of all the values with that key.
- $combineValues()$ takes an input $PTable < K, Collection < V >>$ and an associative combining function on Vs, and returns a $PTable < K, V >$ where each input collection of values has been combined into a single output value.
- $flatten()$ takes a list of $PCollection < T >$s and returns a single $PCollection < T >$ that contains all the elements of the input $PCollections$.

Pipelined operations are implemented by concatenation of functions. For example, if the output of function f is applied as input of function g in a $ParallelDo$ operation then two $ParallelDo$ compute f and $f \otimes g$. The optimizer is only concerned with the structure of the execution plan and not with the optimization of user-defined functions.

FlumeJava traverses the operations in the plan of a batch application in forward topological order, and executes each operation in turn. Independent operations are executed simultaneously. FlumeJava exploits not only the task parallelism but also the data parallelism within operations.

3.15 HISTORY NOTES AND FURTHER READINGS

In 1965 Edsger Dijkstra posed the problem of synchronizing N processes, each with a section of code called its *critical section*, so that two properties are satisfied: *mutual exclusion* – no two critical sections are executed concurrently; and *livelock freedom* – if some process is waiting to execute its critical section, then some process will eventually execute its critical section [147]. Lamport comments: “Dijkstra was aware from the beginning of how subtle concurrent algorithms are and how easy it is to get them wrong. He wrote a careful proof of his algorithm. The computational model implicit in his reasoning is that an execution is represented as a sequence of states, where a state consists of an assignment of values to the algorithm’s variables plus other necessary information such as the control state of each process (what code it will execute next).”

Producer-consumer synchronization was the second fundamental concurrent programming problem identified by Dijkstra. An equivalent formulation of the problem is: given a bounded FIFO (first-in-first-out), the producer stores data into an N-element buffer and the consumer retrieves the data. The algorithm uses three variables: N – the buffer size, in – the infinite sequence of unread input values, and out – the sequence of values output so far. In his discussion of the producer-consumer synchronization algorithm Lamport notes that "The most important class of properties one proves about an algorithm are invariance properties. A state predicate is an invariant iff it is true in every state of every execution."

Lamport notes that "Petri nets are a model of concurrent computation especially well-suited for expressing the need for arbitration. Although simple and elegant, Petri nets are not expressive enough to formally describe most interesting concurrent algorithms." He also mentioned that the first scientific examination of fault tolerance was Dijkstra's 1974 seminal paper on self-stabilization [148], a work ahead of its time. Arguably, the most influential study of concurrency models was Milner's Calculus of Communicating Systems (CCS) [343], [344]. A number of formalisms based on the standard model were introduced for describing and reasoning about concurrent algorithms, including Amir Pnueli's temporal logic introduced in 1977 [404].

Further readings. A fair number of textbooks discuss theoretical as well as practical aspects of concurrency. For example, [517] is dedicated to concurrency in transactional processing systems and [401] analyzes concurrency and consistency. The text [297] covers concurrent programming in Java while [521] presents multithreading in C++.

The von Neumann architecture was introduced in [81]. The BSP and Multi-BSP models were introduced by Valiant in [492] and [493], respectively. Models of computations are discussed in [443].

Petri Nets were introduced by Carl Adam Petri in [402]. An in-depth discussion on concurrency theory and system modeling with PNs can be found in [403]. The discussion of distributed systems leads to the observation that the analysis of communicating processes requires a more formal framework. Tony Hoare realized that a language based on execution traces is insufficient to abstract the behavior of communicating processes and developed *communicating sequential processes* (CSP) [238].

Milner initiated an axiomatic theory called the Calculus of Communicating System (CCS), [344]. Process algebra is the study of concurrent communicating processes within an algebraic framework. The process behavior is modeled as a set of equational axioms and a set of operators. This approach has its own limitations, the real-time behavior of the processes, the true concurrency still escapes this axiomatization.

Seminal papers in distributed systems are authored by Mani Chandy and Leslie Lamport [95], by Leslie Lamport [290], [291], [292], Tony Hoare [238], and Robin Milner [344]. The collection of contributions with the title "Distributed systems", edited by Sape Mullender includes some of these papers.

A survey of techniques and results related to the power of two random choices is presented in [349]. Seminal results on this subject are due to Azar [44], Karp [263], [197], Mitzenmacher [348,350] and others.

3.16 EXERCISES AND PROBLEMS

Problem 1. Non-linear algorithms do not obey the rules of scaled speed-up. For example, it was shown that when the concurrency of an $\mathcal{O}(N^3)$ algorithm doubles, the problem size increases only by slightly more than 25%. Read [456] and explain this result.

Problem 2. Given a system of four concurrent threads t_1, t_2, t_3, and t_4 we take a snapshot of the consistent state of the system after 3, 2, 4, and 3 events in each thread, respectively; all but the second event in each thread are local events. The only communication event in thread t_1 is to send a message to t_4 and the only communication event in thread t_3 is to send a message to t_2. Draw a space–time diagram showing the consistent cut; mark individual events on thread t_i as e_i^j.

How many messages are exchanged to obtain the snapshot in this case? The snapshot protocol allows the application developers to create a checkpoint. An examination of the checkpoint data shows that an error has occurred and it is decided to trace the execution. How many potential execution paths must be examined to debug the system?

Problem 3. The run-time of a data-intensive application could be days, or possibly weeks, even on a powerful supercomputer. Checkpoints are taken for a long-running computation periodically and when a crash occurs the computation is restarted from the latest checkpoint. This strategy is also useful for program and model debugging; when one observes wrong partial results the computation can be restarted from a checkpoint where the partial results seem to be right. Express η, the *slowdown due to checkpointing*, for a computation when checkpoints are taken after a run lasting τ units of time and each checkpoint requires κ units of time. Discuss optimal choices for τ and κ. The checkpoint data can be stored locally, on the secondary storage of each processor, or on a dedicated storage server accessible via a high-speed network. Which solution is optimal and why?

Problem 4. What is in your opinion the critical step in the development of a systematic approach to all-or-nothing atomicity? What does a systematic approach means? What are the advantages of a systematic versus an ad hoc approach to atomicity? The support for atomicity affects the complexity of a system. Explain how the support for atomicity requires new functions/mechanisms and how these new functions increase the system complexity. At the same time, atomicity could simplify the description of a system; discuss how it accomplishes this.

The support for atomicity is critical for system features which lead to increased performance and functionality such as: virtual memory, processor virtualization, system calls, and user-provided exception handlers. Analyze how atomicity is used in each case.

Problem 5. The Petri Net in Figure 3.10D, models a group of n concurrent processes in a shared-memory environment. At any given time only one process may write, but any subset of the n processes may read at the same time, provided that no process writes. Identify the firing sequences, the markings of the net, the postsets of all transition and the presets of all places. Can you construct a state machine to model the same process?

Problem 6*. Consider a computation consisting of n stages with a barrier synchronization among the N threads at the end of each stage. Assuming that you know the distribution of the random execution time of each thread for each stage show how one could use order statistics [129] to estimate the completion time of the computation.

Problem 7. Consider the data flow graph of a computation C in Figure 3.6. Call t_1, t_2, t_3, and t_4 the time when the data inputs $data1, data2, data3$, and $data4$ become available. Call $T_i,\ 1 \le i \le 13$ the time required by computations $C_i,\ 1 \le i \le 13$, respectively, to complete. Express T the total time required by C to complete function of t_i and T_i.

Problem 8. Discuss the factors affecting parallel slackness including characteristics of the parallel computation such as fine versus coarse grain, and the characteristics of the workload and of the computing substrate.

Problem 9*. In Section 3.13 we discussed the *power of two choices* for the balls and bins problem with n bins. Instead of placing each ball in one random bin, we choose two random bins for each ball, place it in the one that currently has fewest balls, and proceed in this manner sequentially for each ball. Prove Equation (3.35).

Hint – the idea of the proof: Call B_i the number of bins with more than i balls at the end. We wish to find an upper bound, β_i for B_i. The probability that a ball is placed in bin q with at least $i+1$ balls in it is

$$Pr(N_q \geq i+1) \leq \left(\frac{\beta_1}{n}\right)^2. \tag{3.45}$$

Indeed, both choices of placing this ball must be in bins with at least i balls. The distribution of bins B_{i+1} is dominated by the binomial distribution $Bin\left(n, \left(\frac{\beta_1}{n}\right)^2\right)$. The mean of this distribution is $\left(\frac{\beta_1}{n}\right)^2$. According to Chernoff bound

$$\beta_{i+1} = c\left(\frac{\beta_1}{n}\right)^2 \tag{3.46}$$

with some constant c. Therefore $\frac{\beta_1}{n}$ decreases quadratically and the following holds

$$i \approx \frac{\ln \ln n}{\ln 2} \Rightarrow \beta_1 < 1. \tag{3.47}$$

It follows that the maximum number of balls in a bin is $\frac{\ln \ln n}{\ln 2}$ with high probability.

Problem 10. What is the difference between wait for graph and resource allocation graph?

CHAPTER

PARALLEL AND DISTRIBUTED SYSTEMS 4

Parallel processing has mesmerized the computational science and engineering community since the early days of the computing era resulting in fascination with high-performance computer systems and, ultimately, with supercomputers. It was hard to expose the parallelism in many scientific applications, but the harder the problem, the more satisfying it was to develop parallel algorithms, implement them, wait for the next generation of processors running at a higher clock rate and enjoy impressive speedup. The enterprise computing world seemed more skeptical and less involved in parallel processing.

Almost half a century after the dawn of the computing era, an eternity in the age of silicon, the disruptive multicore technology forced the community to realize the need to understand and exploit concurrency. There is no point now to wait for faster clock rates, we should better design algorithms and applications able to use all cores of a modern processor.

Things changed again when cloud computing showed that there are new applications that can effortlessly exploit parallelism and, in the process, generate huge revenues. A new era in parallel and distributed systems began, the era of Big Data hiding nuggets of useful information and requiring massive amounts of computing resources. In this era "coarse" is good and "fine" is not good, at least as far as the granularity of parallelism is concerned. The new challenge is to obtain the results faster by effectively harnessing the power of millions of multicore processors.

Cloud computing is intimately tied to parallel and distributed processing. Cloud applications are based on the *client–server* paradigm. A relatively simple software, a *thin-client*, is often running on the user's mobile device with limited resources, while the computationally-intensive tasks are carried out on the cloud. Many cloud applications use a number of instances running concurrently. Transaction processing systems including web-based services represent a large class of applications hosted by computing clouds. Such applications run multiple instances of the service and require reliable and in-order delivery of messages.

Early on scientists and engineers understood that parallel processing requires specialized hardware and system software. It was also clear that the interconnection fabric was critical for the performance of parallel processing systems. Building high-performance computing systems proved to be a major challenge.

The list of companies aiming to support parallel processing and ending as casualties of this effort is long and includes names such as: Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCube, Sequent, Tandem, Thinking Machines, and possibly others, now forgotten. The difficulties of developing new programming models and the effort to design programming environments for parallel applications added to the challenges faced by all these companies.

Computer clouds are large-scale distributed systems, collections of autonomous and heterogeneous systems. Cloud organization is based on a large number of ideas and the experience accumulated since the first electronic computer was used to solve computationally challenging problems. In this chapter

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00005-4
Copyright © 2018 Elsevier Inc. All rights reserved.

we overview concepts in parallel and distributed systems important for understanding basic challenges in the design and use of computer clouds. Data-level and thread-level parallelism, parallel computer architectures, SIMD architectures, and GPUs are discussed in Sections 4.1, 4.2, 4.3, and 4.4, respectively. Application speedup and Amdahl's Law, including its formulation for multicore processors, are analyzed in Sections 4.5 and 4.6.

Organization principles for distributed systems such as modularity, layering, and virtualization presented in Sections 4.7, 4.8, 4.10 are applied to the design of peer-to-peer and large-scale systems discussed in Sections 4.11 and 4.12, respectively. Finally, Section 4.13 presents composability bounds and scalability, a prelude for the discussion of cloud self-organization in Section 13.4.

4.1 DATA, THREAD-LEVEL AND TASK-LEVEL PARALLELISM

As demonstrated by nature, the ability to work as a group and carry out many tasks in parallel represents a very efficient way to reach a common goal. Thus, we should not be surprised that the thought that individual computer systems should work in parallel for solving complex applications was formulated early on, at the dawn of the computer age.

Parallel processing allows us to solve large problems by splitting them into smaller ones and solving them concurrently. Parallel processing was considered for many years the holy grail for solving data-intensive problems encountered in many areas of science, engineering, and enterprise computing.

Parallel processing required major advances in several areas including, algorithms, programming languages and environments, performance monitoring, computer architecture, interconnection networks, and, last but not least, solid state technologies. In many instances discovering parallelism is quite challenging and the development of parallel algorithms requires a considerable effort.

Fine-gained versus coarse-grained parallelism. We distinguish *fine-grained* from *coarse-grained* parallelism, a topic discussed in Section 3.2. In the former case only relatively small blocks of code can be executed in parallel, without the need to communicate or synchronize with other threads or processes, whereas in the latter case large blocks of code can be executed concurrently.

Numerical computations involving linear algebra operations exhibit fine-grained parallelism. For example, many numerical analysis problems, such as solving large systems of linear equations, or solving systems of Partial Differential Equations (PDEs) require algorithms based on domain decomposition methods. The speedup of applications displaying fine-grained parallelism is considerably lower that those of coarse-grained applications. Indeed, even on systems with a fast interconnect the processor speed is orders of magnitude higher than the communication speed.

Concurrent processes or threads communicate using shared-memory or message-passing. Shared-memory is used by multicore processors where each core has private L1 instruction and data caches, as well as an L2 cache, while all cores share the L3 cache. Shared-memory is not scalable thus, seldom used in supercomputers and large clusters. Message-passing is used in large-scale distributed systems. The discussion in this chapter is restricted to this communication paradigm.

Shared-memory is extensively used by the system software. The system stack is an example of shared-memory used to save the state of a process or thread at the time of a context switch. The kernel of an operating system uses control structures such as processor and core tables for multiprocessor and multicore system management, process and thread tables for process and thread management, page tables for virtual memory management, and so on. Multiple application threads running on a multicore processor often communicate via the shared-memory of the system. Debugging a message-passing application is considerably easier than debugging a shared-memory one.

Data-level parallelism. This is an extreme form of coarse-grained parallelism. It is based on partitioning data into large chunks or blocks or segments and running concurrently either multiple programs or copies of the same program, each on a different data block. In the later case the paradigm is called *Same Program Multiple Data* (SPMD). There are the so called *embarrassingly parallel* problems where little or no effort is needed to extract parallelism and to run a number of concurrent tasks with little or no communication among them.

Assume that we wish to search for the occurrence of an object in a set of n images, or of a string of characters in n records. Such a search can be conducted in parallel. In all these instances the time required to carry out the computational task using N servers is reduced by a factor of N, the speedup is almost linear in the number of servers used. This type of data-parallel applications are at the heart of enterprise computing on computer clouds and will be discussed in depth in Chapter 7. The MapReduce programming model will be presented in Section 7.5 followed by the discussion of Hadoop and Yarn in Section 7.7.

Decomposition of a large problem into a set of smaller problems that can be solved concurrently is sometimes trivial and can be implemented in hardware, a topic discussed in Section 4.2. For example, assume that we wish to manipulate the image of a three-dimensional object represented as a *3D* lattice of $n \times n \times n$ points. To rotate the image we apply the same transformation to each one of the n^3 points. Such a transformation can be done by a *geometric engine*, a processor designed to carry out the transformation of a subset of n^3 points concurrently. Graphics Processing Units (GPUs), discussed in Section 4.4, initially designed as graphics engines are now widely used for data-intensive applications.

Thread-level and task-level parallelism. The term thread-level parallelism is overloaded. In the computer architecture literature it is used for data-parallel execution using a GPU. In this case a *thread* is a subset of vector elements processed by one of the lanes of a multithreaded processor, see Section 4.4. *Hyper-threading* is used to describe multiple execution threads possibly running concurrently, but on a single core, see Section 4.2. Threads are also used in a multicore processor to run concurrently multiple processes. Database applications are memory-intensive and I/O-intensive and multiple *threads* are used to hide the latency of memory and I/O access.

The concept of task-level parallelism is also overloaded. In the context of scheduling, a job consists of multiple tasks scheduled either independently or co-scheduled when they need to communicate with one another. Tasks are often fine-grained execution units each one given control of resources for relatively short time to guarantee a low latency response time.

Cloud computing is also very appealing for a class of applications attempting to identify the optimal parameters of a model that best fit experimental data. Such applications involve computationally-intensive tasks. Multiple instances running concurrently test the fitness of different sets of parameters. The results are then compared to determine the optimal set of model parameters.

There are also numerical simulations of complex systems which require an optimal design of a physical system. Multiple design alternatives are compared and the optimal one is selected according to several optimization criteria. Consider for example the design of a circuit using Field Programmable Gate Arrays (FPGAs). An FPGA is an integrated circuit designed to be configured by the customer using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

As multiple choices for the placement of components and for interconnecting them exist, the designer could run concurrently N versions of the design choices and choose the one with the best performance, e.g., minimum power consumption. Alternative optimization objectives could be to reduce cross-talk among the wires or to minimize the overall noise. Each alternative configuration requires hours, or maybe days of computing hence, running them concurrently reduces the design time considerably.

4.2 PARALLEL ARCHITECTURES

There is a vast literature on parallel architectures. In this section we review basic concepts and ideas that play now, and will continue to play in the future, an important role in the evolution of computer clouds.

Control flow versus data flow processor architecture. The dominant processor architecture is the *control flow* architecture pioneered by John von Neumann [81]. The implementation of the processor control flow is straightforward, the *program counter* determines the next instruction to be loaded into the *instruction register* and then executed. The execution is strictly sequential, until a branch is encountered.

But there is an alternative, the *data flow* architecture when operations are carried out at the time when their input becomes available. Though only a few general-purpose data-flow systems are available today,[1] this alternative computer architecture is widely used by network routers, digital signal processors, and other special-purpose systems. The lack of locality, the inefficient use of cache, and ineffective pipelining are most likely some of the reasons why data flow general-purpose processors are not as popular as control flow processors.

Data flow is emulated by von Neumann processors. Indeed, Tomasulo's algorithm for dynamic instruction scheduling [228], developed at IBM in 1967, uses reservation stations to hold instructions waiting for their input to become available and the register renaming for out-of-order instruction execution. It should not be surprising that some of the systems discussed in Chapters 7, 8, and 9 apply the data flow model for task scheduling on large clusters. The power of this model for supporting optimal parallel execution is unquestionable. We should probably expect soon the addition of general-purpose data flow systems to the cloud infrastructure.

Bit-level and instruction-level parallelism. Parallelism at different levels can be exploited by a von Neumann processor. These levels are:

1. *Bit-level parallelism.* A computer word is a fixed-sized piece of data handled as a unit by the instruction set or the hardware of the processor. The number of bits in a word has increased gradually from 4-bit processors to 8-bit, 16-bit, and 32-bit processors. This has reduced the number of instructions required to process larger size operands and allowed a dramatic performance improvement. During this evolutionary process the number of address bits have also increased from 32 bits to 64 bits in 2004 allowing instructions to reference a larger address space, from 2^{32}, about 4 GB, to 2^{64} or 17 179 869 184 GB.

[1]The motto of one company producing general-purpose data flow systems, Maxeler Technologies *www.maxeler.com* is "Passionate for performance."

Table 4.1 The basic pipeline of a superscalar processor. Two instructions are executed per clock cycle; instructions $i, i+2, i+4, i+6$ and $i+8$ are executed by unit 1 and instructions $i+1, i+3, i+5, i+7$ and $i+9$ are executed by unit 2. The pipeline has five stages: instruction fetch (IF), instruction decode (ID), instruction execution (EX), memory access (MEM), and write back (WB). Once the pipeline is full two instructions finish execution every clock cycle.

	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$	IF	ID	EX	MEM	WB				
$i+2$		IF	ID	EX	MEM	WB			
$i+3$		IF	ID	EX	MEM	WB			
$i+4$			IF	ID	EX	MEM	WB		
$i+5$			IF	ID	EX	MEM	WB		
$i+6$				IF	ID	EX	MEM	WB	
$i+7$				IF	ID	EX	MEM	WB	
$i+8$					IF	ID	EX	MEM	WB
$i+9$					IF	ID	EX	MEM	WB

2. *Instruction-level parallelism* (ILP). Computers have used multi-stage processing pipelines to speedup execution for sometime. Once an n-stage pipeline is full, an instruction is completed at every clock cycle unless the pipeline is stalled.

Instruction-Level Parallelism (ILP). A closer look at ILP gives us some insight into the architectural sophistication of modern processors. Pipelining, multiple-issue, dynamic instructions scheduling, branch prediction, speculative execution, and multithreading are some of the architectural features designed to maximize the IPC (Instructions Per clock Cycle), or equivalently, to minimize its inverse, the CPI (Cycles Per Instruction).

Pipelining means splitting of an instruction into a sequence of steps that can be executed concurrently by different circuitry on the chip. A basic pipeline of a RISC (Reduced Instruction Set Computing) architecture consists of five stages.[2] A *superscalar processor* executes more than one instruction per clock cycle see Table 4.1 [228]. A Complex Instruction Set Computer (CISC) architecture could have a much large number of pipelines stages, e.g., an Intel Pentium 4 processor has a 35-stage pipeline.

There are several types of *hazards*, instances when unchecked pipelining would produce incorrect results. Data, structural, and control hazards have to be handled carefully. *Data hazards* occur when the instructions in the pipeline are dependent upon one another. For example, a Read after Write (RAW) hazard occurs when an instruction operates with data in register that is being modified by a previous instruction. A Write after Read (WAR) hazard occurs when an instruction modifies data in a register being used by a previous instruction; finally, a Write after Write (WAW) hazard occurs when two instructions in a sequence attempt to modify the data in the same register and the sequential execution order is violated.

[2]The number of pipeline stages in different RISC processors varies. For example, ARM7 and earlier implementations of ARM processors have a three stage pipeline, fetch, decode, and execute. Higher performance designs, such as the ARM9, have deeper pipelines: Cortex-A8 pipeline has thirteen stages.

Structural hazards occur when the circuits implementing different hardware functions are needed by two or more instructions at the same time. For example, a single memory unit is accessed during the instruction fetch stage where the instruction is retrieved from memory and it is also accessed during the memory stage where data is written to the memory. Structural hazards can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline. *Control hazards* are due to conditional branches. On many instruction pipeline microarchitectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline, normally during the fetch stage.

The architecture should *preserve exception behavior*, any change in instruction order must not change the order in which exceptions are raised, to ensure program correctness. Another necessary condition for correctness is to *preserve instruction flow*, the flow of data between instructions that produce results and consume them.

A *pipeline stall* is the delay in the execution of an instruction in an instruction pipeline in order to resolve a hazard. Such stalls could drastically affect the performance. *Pipeline scheduling* separates dependent instruction from the source instruction by the pipeline latency of the source instruction. Its effect is to reduce the number of stalls.

Dynamic instruction scheduling reduces the number of pipeline stalls, but adds to circuit complexity. *Register renaming* is sometimes supported by *reservation stations.* A reservation station fetches and buffers an operand as soon as it becomes available. A pending instruction designates the reservation station it will send its output to. A reservation station stores the following information: (1) the instruction; (2) buffered operand values (when available); and (3) the ID of the reservation station number providing the operand values.

Tomasulo's algorithm uses register renaming to correctly perform out-of-order execution. Reservation station registers hold either a real value or a placeholder value. If a real value is unavailable to a destination register during the issue stage, a placeholder value is initially used. The placeholder value is a tag indicating which reservation station will produce the real value. When the unit finishes and broadcasts the result on the CDB (Common Data Bus), the placeholder will be replaced with the real value [228].

Flynn's computer architecture taxonomy. In 1966 Michael Flynn proposed a taxonomy of computer architectures based on the number of *concurrent instructions* and the number of *data streams*:

- SISD (Single Instruction Single Data);
- SIMD (Single Instruction, Multiple Data);
- MIMD (Multiple Instructions, Multiple Data);
- MISD (Multiple Instructions Single Data) is a fourth possible architecture, but it is very rarely used, mostly for fault tolerance.

SISD architecture. SISD processors, have been around since the ENIAC, the system built at the University of Pennsylvania's Moore School of Electrical Engineering between 1943 and 1946 by J. Presper Eckert and John Mauchly. Individual cores of a modern multicore processor are SISD and support the execution of a single thread or process at any given time. A *superscalar* processor executes more than one instruction per clock cycle. A single core superscalar is still a SISD processor.

SIMD architecture. The architecture supports vector processing. When a SIMD instruction is issued, the operations on individual vector components are carried out concurrently. For example, to add two vectors $(a_0, a_1 \ldots\ldots a_{63})$ and $(b_0, b_1 \ldots\ldots b_{63})$, all 64 pairs of vector elements are added concurrently and all the sums $(a_i + b_i), 0 \leq i \leq 63$ are available at the same time.

The first use of SIMD instructions was in vector supercomputers such as the CDC Star-100 and the Texas Instruments ASC in early 1970s. Vector processing was especially popularized by Cray in the 1970s and 1980s, by attached vector processors such as those produced by the FPS (Floating Point Systems), and by supercomputers such as the Thinking Machines CM-1 and CM-2.

Sun Microsystems introduced SIMD integer instructions in its "VIS" instruction set extensions in 1995 for UltraSPARC I microprocessor. The first widely-deployed SIMD instruction set for gaming was Intel's MMX extensions to the *x86* architecture. IBM and Motorola then added AltiVec to the POWER architecture. There have been several extensions to the SIMD instruction sets for both architectures as we shall see in Section 4.3.

MIMD architecture. A MIMD architecture refers to a system with several processors that function asynchronously and independently; at any time, different processors may be executing different instructions on different data. Several processors can share a common memory and we distinguish several types of multiprocessor systems: UMA, NUMA, and COMA, uniform, non-uniform, and cache only memory access, respectively.

A MIMD system could have a distributed memory. Processors and memory modules communicate with one another using an interconnection network, such as a hypercube, a *2D* torus, a *3D* torus, an omega network, or another network topology, see Section 5.6. Today, most supercomputers are MIMD machines and some use GPUs instead of traditional processors. Multicore processors with multiple processing units are now ubiquitous.

As more powerful processors were needed, the concept of *hyper-threading* was developed and in 2002 Intel introduced Xeon and later Pentium 4 processors. Hyper-threading takes advantage of unused processor resources and presents itself to the operating system as a two core processor.

Multicore processors support true MIMD execution. Each core has its own register file, ALU and floating-point execution units. As mentioned earlier, each core has its private L1 instruction and data caches as well as L2 cache; all cores of a processor share the L3 cache.

From supercomputers to distributed systems. Modern supercomputers derive their power from architecture and parallelism rather than faster processors running at higher clock rates. The supercomputers of today consist of a very large number of processors and cores communicating through very fast custom interconnects.

In mid 2012 the most powerful supercomputer was an IBM Sequoia-BlueGene/Q Linux-based system powered by Power BQC 16-core processors running at 1.6 GHz. The system, installed at Lawrence Livermore National Laboratory and called Jaguar, has a total of 1 572 864 cores and 1 572.864 TB of memory, achieves a sustainable speed of 16.32 PFlops, and consumes 7.89 MW of power. Later in 2012 a Cray XK7 system, Titan, installed at the Oak Ridge National Laboratory (ORNL) was coronated as the most powerful supercomputer in the world. The system had 560 640 processors, including 261 632 Nvidia K20x accelerator cores; it achieved a speed of 17.59 PFlops on the Linpack benchmark.

In 2016 the most powerful supercomputer was Sunwai TaihuLight at the National Supercomputer Center in Wixi, China with 10 649 600 cores with the peak bandwidth of 125.436 PFlops. The system needed 15.371 MW of power. Its Linpack performance is 93.0146 PFlops and has 1 310.720 TB of memory. Several most powerful systems listed in [486] are powered by Nvidia 2050 GPU. Some of the top 10 supercomputers use the InfiniBand interconnect discussed in Section 5.8.

The next natural step was triggered by advances in communication networks when low-latency and high-bandwidth Wide Area Networks (WANs) allowed individual systems, many of them multiprocessors, to be geographically separated. Large-scale distributed systems were first used for scientific and engineering applications and took advantage of the advancements in system software, programming models, tools, and algorithms developed for parallel processing.

4.3 SIMD ARCHITECTURES; VECTOR PROCESSING AND MULTIMEDIA EXTENSIONS

SIMD architectures have significant advantages over the other systems described by Flynn's classification scheme. Some of these advantages are:

1. Exploit a significant level of data-parallelism. Enterprise applications in data mining and multimedia applications, as well as the applications in computational science and engineering using linear algebra benefit the most.
2. Allow mobile device to exploit parallelism for media-oriented image and sound processing using SIMD extensions of traditional Instruction Set Architecture (ISA).
3. Are more energy efficient than MIMD architecture. Only one instruction is fetched for multiple data operations, rather than fetching one instruction per operation.
4. Have a higher potential speedup than MIMD architectures. SIMD potential speedup could be twice as large as that of MIMD.
5. Allows developers to continue thinking sequentially.

Three flavors of the SIMD architecture are encountered in modern processor design: (a) Vector architecture; (b) SIMD extensions for mobile systems and multimedia applications; and (c) Graphics Processing Units (GPUs).

Vector architectures. Vector computers operate using vector registers holding as many as 64 or 128 vector elements. Vector functional units carry out arithmetic and logic operations using data from vector registers as input and disperse the results back to memory. The vector load-store units are pipelined, hide memory latency, and leverage memory bandwidth. The memory system spreads access to multiple *memory banks* which can be addressed independently.

Chaining allows vector operations to start as soon as individual elements of vector source operands become available and operate on *convoys*, sets of vector instructions that can potentially be executed together. Multiple *lanes* process several vector elements per clock cycle. Each lane contains a subset of the vector register file and one execution pipeline from each functional unit.

Vector length registers support handling of vectors whose length is not a multiple of the length of the physical vector registers, e.g., a vector of length 100 when the vector register can only contain 64 vector elements. *Vector mask registers* disable/select vector elements and are used by conditional statements. Non-adjacent vector elements of a multidimensional array can be loaded into a vector register, by specifying the *stride*, the distance between elements to be gathered in one register. Scatter-gather operations support processing of sparse vectors. A *gather* operation takes an *index vector* and fetches the vector elements at the addresses given by adding a base address to the offsets given by the index vector; as a result a dense vector is loaded in a vector register. A *scatter* operation does the inverse, it scatters the elements of a vector register to addresses given by the index vector and the base address.

SIMD extensions for multimedia applications. The name of this class of SIMD architectures reflects the basic architectural philosophy – augmenting an existing instruction set of a scalar processor with a set of vector instructions. SIMD extensions have obvious advantages over vector architecture:

1. Low cost to add circuitry to an existing ALU.
2. Little extra state is added thus, the extensions have little impact on context-switching.
3. Do not pose additional complications to the virtual memory management for cross-page access and page-fault handling.
4. Need little extra memory bandwidth.

Multimedia applications often run on mobile devices and operate on narrower data types than the native word size. For example, graphics applications use 3×8 bits for colors and one 8-bit for transparency, audio applications use 8, 16, or 24-bit samples. To accommodate narrower data types carry chains have to be disconnected. For example a 256-bit adder can be partitioned to perform simultaneously 32, 16, 8 or 4 additions on 8, 16, 32, or 64 bit, respectively. The instructions *opcode* now encode the data type and neither sophisticated addressing modes supported by vector architectures such as stride-base addressing or scatter-gather, nor mask registers are supported.

Intel extended its $x86 - 64$ instruction set architecture. In 1996 Intel introduced MMX (Multi-Media Extensions) which supports eight 8-bit, or four 16-bit integer operations. MMX was followed by multiple generations of streaming SIMD extensions (SSE) in 1999 and ending with SSE4 in 2007. The SSEs operate on eight 8-bit integers, four 32-bit or two 64-bit either integer or floating-point operations.

AVX (Advanced Vector Extensions) introduced by Intel in 2010 operates on four 64-bit either integer or floating-point operations. Several members of the AVX family of Intel processors are: Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and its follower, the Baby Lake announced in August 2016. AMD offers several family of processors with multimedia extensions including the Steamroller.

Floating-point performance models for SIMD architecture. The gap between the processor and the memory speed, though bridged by different level of caches, is still a major factor affecting the performance of many applications. Applications displaying low spatial and temporal locality are particularly affected by gap. The effects of this gap are also most noticeable for SIMD architectures and floating-point operations. The concept of *arithmetic intensity*, defined as the number of floating-point operations per byte of data read, is used to characterize application scalability and to quantify the performance of SIMD systems.

The arithmetic intensity of applications involving dense matrices is high and this means that dense matrix operations scale with problem size, while sparse matrix applications have a low arithmetic intensity, therefore do not scale well with the problem size. Applications involving spectral methods and FFT (Fast Fourier Transform) have an average arithmetic intensity.

The *roofline* model captures the fact that the performance of an application is limited by its arithmetic intensity and by the memory bandwidth. A graph depicting the floating-point performance function of the arithmetic intensity is shown in Figure 4.1. The memory bandwidth limits the performance at low arithmetic intensity and this effect is captured by the sloped line of the graph. As the arithmetic intensity increases, the floating-point performance of the processor is the limiting factor captured as the straight line of the graph.

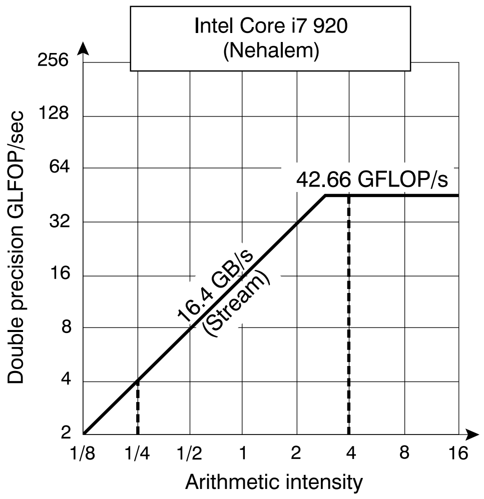


FIGURE 4.1

The roofline performance model for Intel i7 920. When the arithmetic intensity is lower than about 3 the memory bandwidth of 16.4 GB/sec is the bottleneck. The processor delivers 42.66 Gflops and this limits the performance of applications with arithmetic intensity larger than about 3.

4.4 GRAPHICS PROCESSING UNITS

The desire to support real-time graphics with vectors of two, three, or four dimensions led to the development of Graphics Processing Units (GPUs). GPUs are very efficient at manipulating computer graphics. GPUs produced by Intel, NVIDIA, and AMD/ATI are also used in embedded systems, mobile phones, personal computers, workstations, and game consoles. GPU processing is based on a heterogeneous execution model with a CPU acting as the *host* connected with a GPU called the *device*.

The highly parallel structures of GPUs are based on SIMD execution and support parallel processing of large data blocks. A GPU has multiple *multithreaded SIMD* processors. The current-generation of GPUs, e.g., Fermi from NVIDIA, have 7 to 15 multithreaded SIMD processors. Compared with vector processors, each multithreaded SIMD processor has several wide and shallow SISD *lanes*. For example, an NVIDIA GPU has 32 768 registers divided among the 16 physical SIMD lanes; each lane has 2 048 registers.

A typical processing execution includes the following steps:

1. CPU copies the input data from the main memory to the GPU memory.
2. CPU instructs the GPU to start processing using the executable in the GPU memory.
3. GPU uses multiple cores to execute the parallel code.
4. When done the GPU copies the result back to the main memory.

The GPU programming model is Single-Instruction-Multiple-Thread (SIMT). GPUs are often programmed in CUDA, a C-like programming language developed by NVIDIA, the pioneer of GPU-accelerated computing. All forms of GPU parallelism are unified as CUDA threads. In the SIMT model a *thread* is associated with each data element. Thousands of CUDA threads could run concurrently.

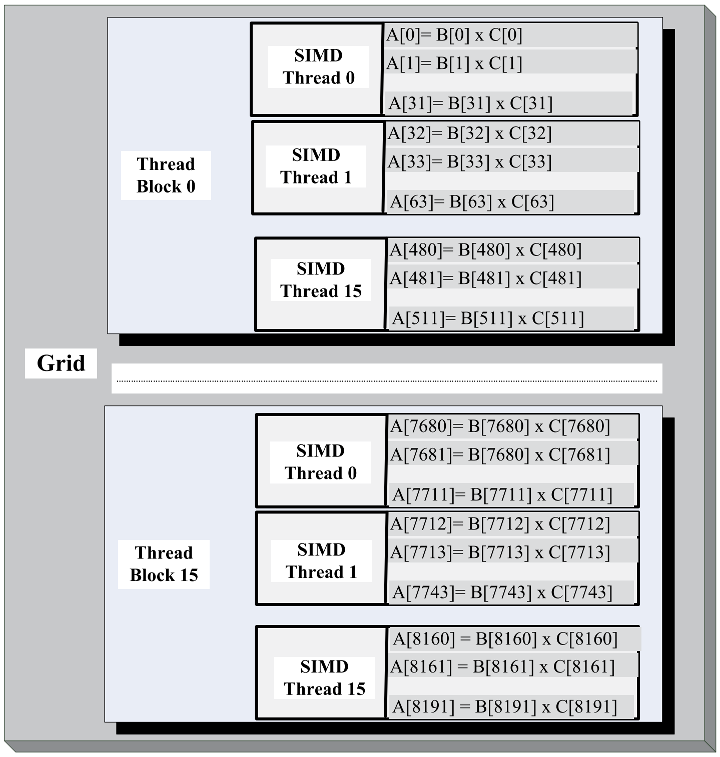


FIGURE 4.2

Grid, blocks, and threads. The grid has 8192 components of vector A. There are 16 blocks with 512 vector components each. Each bloc has 6 threads and each thread operates on 32 components of vector A.

Threads are organized in *blocks*, groups of 512 vector elements, and multiple blocks form a grid. Figure 4.2 illustrates the grid representing a vector A with 8192 components; there are 16 blocks, each block has 16 SIMD threads; each thread operates on 32 elements of the vector.

A two-level scheduling mechanisms assigns threads to the multiple *lanes* of a multithreaded SIMD processor. A *thread block scheduler* assigns thread blocks to multithreaded SIMD processors and then a *tread scheduler* assigns threads to SIMD *lanes*. Figure 4.3 illustrates scheduling for the grid shown in Figure 4.2. The thread blocks must be able to run independently.

The NVIDIA GPU memory has the following organization:

- Each SIMD lane has an off-chip private memory for stack frame, spilling registers, private variables, and GPU data in L1 and L2 caches.
- Each multithread SIMD processor has on-chip local memory shared by all its lanes, thus, by the threads within the block scheduled on the processor.
- GPU memory. The host can read from and write to this off-chip memory.

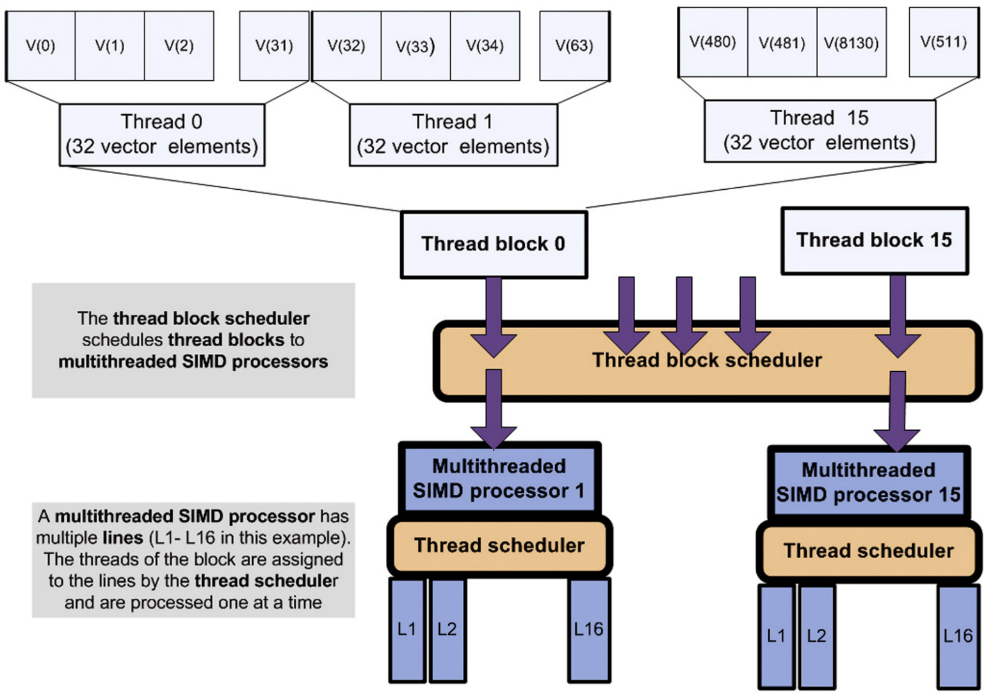

V(0)
V(1)
V(2)
V(31)
V(32)
V(33)
V(34)
V(63)
V(480)
V(481)
V(8130)
V(511)
Thread 0 (32 vector elements)
Thread 1 (32 vector elements)
Thread 15 (32 vector elements)
Thread block 0
Thread block 15
The thread block scheduler schedules thread blocks to multithreaded SIMD processors
Thread block scheduler
Multithreaded SIMD processor 1
Multithreaded SIMD processor 15
A multithreaded SIMD processor has multiple lines (L1- L16 in this example). The threads of the block are assigned to the lines by the thread scheduler and are processed one at a time
Thread scheduler
Thread scheduler
L1
L2
L16
L1
L2
L16

FIGURE 4.3

The execution for the grid in Figure 4.2. The *thread block scheduler* assigns thread blocks to multithreaded SIMD processors. A *thread scheduler* running on each multithreaded SIMD processor assigns threads to the SIMD *lanes* of the processors.

The Fermi architecture is used in the GeForce 400 and 500 NVIDIA GPU processor series. The host interface of Fermi connects the GPU to the CPU via a PCI-Express v2 bus with peak transfer rate of 8 GB/s. The DRAM supports up to 6 GB of GDDR5 DRAM memory thanks to 64-bit addressing capability and has a 192 GB/sec bandwidth. The clock runs at 1.5 GHz and the peak performance is 1.5 Flops. It should not be surprising that cloud service providers now offer GPU instances as we have seen in Section 2.4.

4.5 SPEEDUP, AMDHAL'S LAW, AND SCALED SPEEDUP

Parallel hardware and software systems allow us to solve problems demanding more resources than those provided by a single system and, at the same time, to reduce the time required to obtain a solution. The speedup measures the effectiveness of parallelization; in the general case the *speedup* of the parallel computation is defined as

$$S(N) = \frac{T(1)}{T(N)}, \tag{4.1}$$

with $T(1)$ the execution time of the sequential computation and $T(N)$ the execution time when N parallel computations are carried out.

Amdahl's Law. Gene Myron Amdahl[3] was a theoretical physicist turned computer architect best known for Amdahl's Law. In a seminal paper published in 1967 [26] Amdhal argued that the fraction of a computation which is not paralellizable is significant enough to favor single processor systems. He reasoned that large-scale computing capabilities can be achieved by enhancing the performance of single processors, rather than building multiprocessor systems.

Though this thesis was disproved, Amdahl's Law is a fundamental result used to predict the theoretical maximum speedup for a program using multiple processors. This law states that *the portion of the computation which cannot be parallelized determines the overall speedup.* If α is the fraction of running time a sequential program spends on non-paralellizable segments of the computation then, the maximum speedup achievable S is

$$S = \frac{1}{\alpha}. \tag{4.2}$$

To prove this result call σ the sequential time and π the parallel time and start from the definitions of $T(1)$, $T(N)$, and α:

$$T(1) = \sigma + \pi, \qquad T(N) = \sigma + \frac{\pi}{N}, \qquad \text{and} \qquad \alpha = \frac{\sigma}{\pi + \sigma}. \tag{4.3}$$

Then

$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \pi/N} = \frac{1 + \pi/\sigma}{1 + (\pi/\sigma) \times (1/N)}. \tag{4.4}$$

But

$$\pi/\sigma = \frac{1-\alpha}{\alpha} \tag{4.5}$$

Thus, for large N

$$S = \frac{1 + (1-\alpha)/\alpha}{1 + (1-\alpha)/(N\alpha)} = \frac{1}{\alpha + (1-\alpha)/N} \approx \frac{1}{\alpha} \tag{4.6}$$

An alternative formulation of Amdahl's Law is that if a fraction f of a computation is enhanced by a speedup S then the overall speedup is

$$S_{overall}(f, S) = \frac{f}{(1-f) + \frac{f}{S}} \qquad \text{or} \qquad S_{overall}(f, S) = \frac{1}{\frac{1}{f} + \frac{1}{S} - 1} \tag{4.7}$$

Scaled speedup. Amdahl's Law applies to a *fixed problem size*; in this case the amount of work assigned to each one of the parallel processes decreases when the number of processes increases and this affects the efficiency of the parallel execution.

[3] Gene Amdhal contributed significantly to the development of several IBM systems including System/360 and then started his own company, Amdahl Corporation; his company produced high performance systems in the 1970s.

When the problem size is allowed to change, Gustafson's Law gives the *scaled speedup* with N parallel processes as

$$S(N) = N - \alpha(N-1). \tag{4.8}$$

As before, we call σ the sequential time; now π is the *fixed parallel time per process*; α is given by Equation (4.3). The sequential execution time, $T(1)$, and the parallel execution time with N parallel processes, $T(N)$, are

$$T(1) = \sigma + N\pi \quad \text{and} \quad T(N) = \sigma + \pi. \tag{4.9}$$

Then the scaled speedup is

$$S(N) = \frac{T(1)}{T(N)} = \frac{\sigma + N\pi}{\sigma + \pi} = \frac{\sigma}{\sigma + \pi} + \frac{N\pi}{\sigma + \pi} = \alpha + N(1-\alpha) = N - \alpha(N-1). \tag{4.10}$$

Amdahl's Law expressed by Equation (4.2) and the *scaled speedup* given by Equation (4.8) assume that all processes are assigned the same amount of work. The scaled speedup assumes that the amount of work assigned to each process is the same regardless of the problem size. Then, to maintain the same execution time the number of parallel processes must increase with the problem size. The scaled speedup captures the essence of efficiency, namely, that the limitations of the sequential part of a code can be balanced by increasing the problem size.

4.6 MULTICORE PROCESSOR SPEEDUP

We now live in the age of multicore processors brought about by the limitations imposed on solid state devices by the laws of physics. Increased power dissipation due to faster clock rates makes the heat removal more challenging and this implies that in the future we could only expect a modest increase of the clock rate.

Moore's Law stating that the number of transistors on a chip doubles approximately every 1.5 years will still hold for a number of years. Multicore processors use the billions of transistors on a chip to deliver significantly higher computing power and process more data every second. Yet, the ability to get data in and out of the chip is limited by the number of pins. Increasing the number of cores on a chip faces its own physical limitations.

There are alternative designs of multicore processors and the next question is to investigate chip configurations most useful for applications exhibiting a limited parallelism. The cores can be identical or different from one another, there could be a few powerful cores or a larger number of less powerful cores. Theoretically, the cores could be configured automatically or be immutable.

More cores will lead to high speedup of highly parallel applications, a powerful core will favor highly sequential applications. A chameleonic system will adapt to the actual level of parallelism though, if feasible, changing the core configuration will incur some overhead and will challenge application developers. Even considering the re-configuration overhead, the speedup of automatic core configuration will be superior to either symmetric or asymmetric core design.

The design space of multicore processors should be driven by cost-performance considerations. A design will be cost-effective if the speedup achieved will exceed the *cost up* defined as the multicore

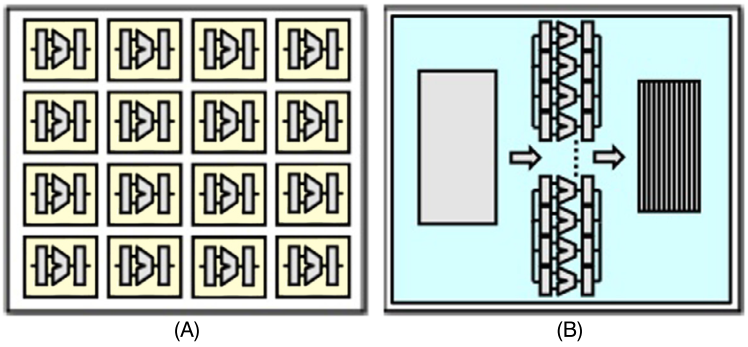


FIGURE 4.4

16-BCE chip. Symmetric core processor with two different configurations: (A) sixteen 1-BCE cores; (B) one 16-BCE core.

processor cost divided by the single-core processor cost. The cost of a multicore processor depends on the number of cores and the complexity, ergo, the power of individual cores.

The *Basic Core Equivalent* (BCE) concept was introduced to quantify the resources of individual cores. A *symmetric core* processor may have n BCEs with r resources each. Alternatively, the $n \times r$ resources may be distributed unevenly in an *asymmetric core* processor.

A quantitative analysis of the design choices based on an extension of Amdahl's Law to multicore processors is presented in [235]. We expect this analysis to confirm the obvious, that is: *the larger the parallelizable fraction f of an application, the larger the speedup.* This analysis is based on a number of simplifying assumptions:

1. A number of factors such as the chip area, the power dissipation, or combinations of these two with other factors limit the number of cores to n BCE. These limitations consider only on-chip resources. Off-chip resources such as shared caches, memory controllers, or interconnection networks are assumed to be approximately identical for the alternative designs.
2. The performance of a single BCE core is the unity. When the chip is limited to n BCEs, all cores are identical, and the performance of each core is r then the total number of cores on a chip is $\lceil n/r \rceil$. Figure 4.4 shows a symmetric 16-BCE chip with two configuration: one with sixteen 1-BCE cores and the other with one 16-BCE core.
3. The sequential performance of r BCEs is denoted as $perf(r)$. When $perf(r) > r$ the speedup of both sequential and parallel execution increases, therefore the designers should increase as much as feasible the core resources and implicitly the individual core performance. On the other hand, when $perf(r) < r$ increasing individual core performance increases the performance for sequential execution but lowers that of the parallel execution. Therefore, the following analysis is focused on the case when $perf(r) < r$. A good model is $perf(r) = \sqrt{r}$ when the performance doubles, triples and quadruples for 4, 9, 16 cores, respectively.

The first case analyzed assumes a symmetric multicore chip. There are n/r cores on the chip; for example, when $n = 32$ the chip could have 16 cores of 2 BCE each, 8 cores of 4 BCE each, and so on. Call f the parallelizable fraction of a computation. One core will run the $(1 - f)$ sequential component of the computation and n/r cores will run the parallel component of the computation, f,

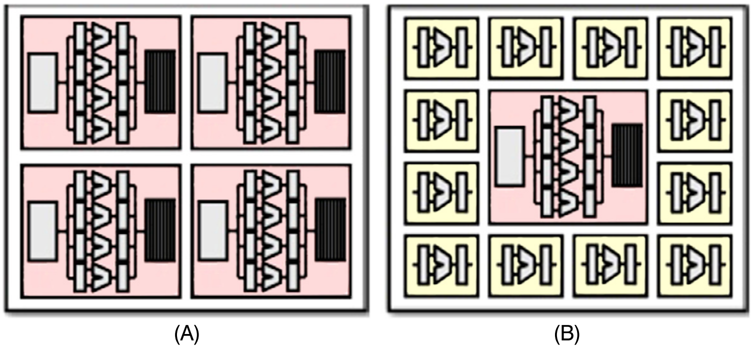


FIGURE 4.5

16-BCE chip. (A) Symmetric core processor with four 4-BCE cores; (B) Asymmetric core processor with one 4-BCE core and twelve 1-BCE cores.

with a performance $perf(r)$. According to Equation (4.7) the speedup will be

$$Speedup_{symcore}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \cdot r}{n \cdot perf(r)}}. \tag{4.11}$$

In an asymmetric multicore processor more powerful cores will coexist with less powerful ones. Figure 4.5 illustrates the differences between symmetric and asymmetric cores; the asymmetric core processor has one 4-BCE core and twelve 1-BCE cores. The sequential performance will benefit from the more powerful core running four times faster, while the parallel performance is $perf(r)$ from the 4-BCE core and 1 each from the remaining $(n - r)$, in our case twelve 1-BCE cores. The speedup with one powerful core and $(n - r)$ 1-BCE cores is then

$$Speedup_{asymcore}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}. \tag{4.12}$$

A dynamic multicore chip could configure its n BCE depending on the fraction f of a particular application. If the sequential component of the application, $(1 - f)$, is large then configure the chip as one r-BCE core while in parallel mode use all base cores in parallel. In this case

$$Speedup_{dyncore}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}}. \tag{4.13}$$

What non-obvious conclusions can be drawn from this analysis? First, the speedup of asymmetric multicore processors is always larger and, in some cases, could be significantly larger than the speedup of symmetric core processors. For example, the largest speedup when $f = 0.975$ and $n = 1\,024$ is achieved for a configuration with one 345-BCE core and 679 1-BCE cores. Second, increasing the power of individual cores is beneficial even for symmetric core processors. For example, the maximum speedup occurs for seven 1-BCE cores when $f = 0.975$ and $n = 256$.

Not to be forgotten should be that task scheduling on asymmetric and dynamic multicore processors will be fairly difficult. There are also other factors affecting the performance ignored by the simple model discussed in [235].

4.7 DISTRIBUTED SYSTEMS; SYSTEM MODULARITY

A distributed system is a collection of autonomous and heterogeneous systems able to communicate effectively with each other. The question is: How could such a collection be organized to cooperate and compute efficiently? In spite of intensive research efforts spanning many years, an optimal organization of a large-scale system has eluded us.

The inability to conceive an optimal general-purpose system reflects the realization that a system cannot be looked upon in isolation, it should be analyzed in the context of the environment it is expected to operate in. The more complex and diverse this environment, the less likely it is to display an asymptotically optimal performance for all, or virtually all applications. The organization and the system management may be optimal for a class of applications, yet lead to a sub-optimal performance for others.

Distributed systems have been around for several decades. For example, distributed file systems and network file systems have been used for user convenience and for improving reliability and functionality of file systems for many years, see Section 6.3. Modern operating systems allow a user to *mount* a remote file system and access it the same way a local file system is accessed, but with a performance penalty due to larger communication costs. The *Remote Procedure Call* (RPC) supports inter-process communication and allows a procedure on a system to invoke another procedure running in a different address space, possibly on a remote system.

A *distributed system* is a collection of computers, connected through a network and a distribution software called *middleware*, which enables computers to coordinate their activities and to share the resources of the system; the users perceive the system as a single, integrated computing facility. The middleware should support a set of desirable properties of the distributed system:

- Access transparency; local and remote information objects should be accessed using identical operations.
- Location transparency; information objects should be accessed without knowledge of their location.
- Concurrency transparency; processes running concurrently should share information objects without interference among them.
- Replication transparency; multiple instances of information objects should be used to increase reliability without the knowledge of users or applications.
- Failure transparency; the faults should be concealed.
- Migration transparency; the information objects in the system should be moved without affecting the operation performed on them.
- Performance transparency; the system should be reconfigured based on the load and quality of service requirements.
- Scaling transparency; the system and the applications should scale without a change in the system structure and without affecting the applications.

A distributed system has several characteristics: its components are autonomous, scheduling and other resource management and security policies are implemented by each system. There are multiple points of control and multiple points of failure in a distributed system and some sources may not be accessible at all times. Distributed systems can be scaled by adding additional resources and can be designed to maintain availability even at low levels of hardware / software / network reliability. Availability is a characteristic of a system aiming to ensure an agreed level of operational performance for an extended period of time.

Modularity. Modularity is a basic concept in the design of man-made systems; a system is made out of components, or modules, with well-defined functions. Modularity supports the separation of concerns, encourages specialization, improves maintainability, reduces costs, and decreases the development time of a system.

Modularity has been used extensively since the industrial revolution for building every imaginable product, from weaving looms to steam engines, from watches to automobiles, from electronic devices to airplanes. Individual modules are often made of sub-assemblies. Modularity can reduce cost for the manufacturer and for the consumers. The same module may be used by a manufacturer in multiple products; to repair a defective product a consumer only replaces the module causing the malfunction rather than the entire product. Modularity encourages specialization, as individual modules can be developed by experts with deep understanding of a particular field. It also supports innovation, it allows a module to be replaced with a better one, without affecting the rest of the system.

It is no surprise that the hardware, as well as the software systems are composed of modules interacting with one another through well-defined interfaces. The software development for distributed systems is more challenging than for sequential systems and these challenges are amplified by the scale of the system and the diversity of applications.

Modularity, layering, and hierarchy are some of the means to cope with the complexity of distributed application software. Software modularity, the separation of a function into independent, interchangeable modules requires well-defined interfaces specifying the elements provided and supplied to a module [392]. A modular software design is driven by several principles outlined in [140]:

1. Information hiding – the user of a module does not need to know anything about the internal mechanism of the module to make effective use of it.
2. Invariant behavior – the functional behavior of a module must be independent of the site or context from which it is invoked.
3. Data generality – the interface to a module must be capable of passing any data object an application may require.
4. Secure arguments – the interface to a module must not allow side-effects on arguments supplied to the interface.
5. Recursive construction – a program constructed from modules must be usable as a component in building larger programs or modules.
6. System resource management – resource management for program modules must be performed by the computer system and not by individual program modules.

Some of these principles are implicitly supported by the enforced modularity. The system should prevent modules to make private resource allocation decisions and should support a global address space.

Chapters 7 and 8 discuss applications and system software for large-scale distributed systems, the computer clouds. The modularity concept is dissected and its applications are reviewed in the next section.

4.8 SOFT MODULARITY VERSUS ENFORCED MODULARITY

The progress made in system design is notable not in the least due to a number of principles guiding the design of parallel and distributed systems. One of these principles is specialization; this means that a number of functions are identified and an adequate number of system components are configured to provide these functions. For example, data storage is an intrinsic function and storage servers are a ubiquitous presence in most systems. This brings us to the modularity concept.

Modularity allows us to build a complex software system from a set of components built and tested independently. A requirement for modularity is to clearly define the interfaces between modules and enable the modules to work together. The steps involved in the transfer of the flow of control between the caller and the callee are:

1. The caller saves its state including the registers, the arguments, and the return address on the stack.
2. The callee loads the arguments from the stack, carries out the calculations and then transfers control back to the caller.
3. The caller adjusts the stack, restores its registers, and continues its processing.

Soft modularity. We distinguish *soft modularity* from *enforced modularity*. The former implies dividing a program into modules which call each other and communicate using shared-memory or follow the procedure call convention.

Soft modularity hides the details of the implementation of a module and has many advantages: once the interfaces of the modules are defined, the modules can be developed independently; a module can be replaced with a more elaborate, or with a more efficient one, as long as its interfaces with the other modules are not changed. The modules can be written using different programming languages and can be tested independently.

Soft modularity presents a number of challenges. It increases the difficulty of debugging; for example, a call to a module with an infinite loop will never return. There could be naming conflicts and wrong context specifications. The caller and the callee are in the same address space and may misuse the stack, e.g., the callee may use registers that the caller has not saved on the stack, and so on.

Strongly-typed languages may enforce soft modularity by ensuring type safety at compile time or at run time, it may reject operations or function class which disregard the data types, or it may not allow class instances to have their class altered. Soft modularity may be affected by errors in the run-time system, errors in the compiler, or by the fact that different modules are written in different programming languages.

Enforced modularity. The ubiquitous client–server paradigm is based on enforced modularity; this means that the modules are forced to *interact only by sending and receiving messages.* This paradigm leads to a more robust design, the clients and the servers are independent modules and may fail separately.

Moreover, the servers are stateless, they do not have to maintain state information. A server may fail and then come back up without the clients being affected, or even noticing the failure of the server. The system is more robust as it does not allow errors to propagate. Enforced modularity makes an attack less likely because it is difficult for an intruder to guess the format of the messages or the sequence numbers of segments, when messages are transported by TCP.

Last but not least, resources can be managed more efficiently. For example, a server typically consists of an ensemble of systems, a *front-end* system which dispatches the requests to multiple *back-end* systems which process the requests. Such an architecture exploits the elasticity of a computer cloud infrastructure, the larger the request rate, the larger the number of back-end systems activated.

The client–server paradigm. This paradigm allows systems with different processor architecture, e.g., 32-bit or 64-bit, with different operating systems, e.g., multiple versions of operating systems, such as Linux, Mac OS, or Microsoft Windows, libraries and other system software, to cooperate. The client–server paradigm increases flexibility and choice; the same service could be available from multiple providers, a server may use services provided by other servers, a client may use multiple servers, and so on.

Heterogeneity of systems based on the client–server paradigm is less of a blessing, the problems it creates outweigh its appeal. Heterogeneity adds to the complexity of the interactions between a client and a server as it may require conversion from one data format to another, e.g., from little-endian to big-endian or vice-versa, or conversion to a canonical data representation. There is also uncertainty in terms of response time as some servers may be more performant than others or may have a lower workload.

A major difference between the basic models of grid and cloud computing is that the former does not impose any restrictions regarding heterogeneity of the computing platforms, whereas homogeneity used to be a basic tenet of computer clouds infrastructure. Originally, a computer cloud was a collection of homogeneous systems, systems with the same architecture and running under the same or very similar system software. We have already seen in Section 2.4 that nowadays computer clouds exhibit some level of heterogeneity.

The clients and the servers communicate through a network that can be congested. Transferring large volumes of data through the network can be time-consuming; this is a major concern for data-intensive applications in cloud computing. Communication through the network adds additional delay to the response time. Security becomes a major concern as the traffic between a client and a server can be intercepted.

Remote Procedure Call (RPC). RPCs were introduced in the early 1970s by Bruce Nelson and used for the first time at PARC (Palo Alto Research Park). PARC is credited with many innovative ideas in distributed systems including the development of the Ethernet, the GUI interfaces, bitmap displays, and the Alto system.

RPC is often used for the implementation of client–server systems interactions. For example, the Network File System (NFS) introduced in 1984 was based on Sun's RPC. Many programming languages support RPCs. For example, Java Remote Method Invocation (Java RMI) provides a functionality similar to the one of UNIX RPC methods; XML-RPC uses XML to encode HTML-based calls. The RPC standard is described in RFC 1831.

To use an RPC, a process may use special services *PORTMAP* or *RPCBIND* available at port 111 to register and for service lookup. RPC messages must be well-structured; they identify the RPC and are addressed to an RPC demon listening at an RPC port. *XDP* is a machine independent representation standard for RPC.

RPCs reduce the *fate sharing* between caller and the callee. RPCs take longer than local calls due to communication delays. Several RPC semantics are used to overcome potential communication problems:

- *At least once:* a message is resent several times and an answer is expected. The server may end up executing a request more than once, but an answer may never be received. This semantics is suitable for operation free of side-effects.
- *At most once:* a message is acted upon at most once. The sender sets up a timeout for receiving the response. When the timeout expires an error code is delivered to the caller. This semantics requires the sender to keep a history of the time-stamps of all messages as messages may arrive out-of-order. This semantics is suitable for operations which have side effects.
- *Exactly once:* it implements the *at most once* semantics and requests an acknowledgment from the server.

Applications of the client–server paradigm. The large spectrum of applications attests to the role played by the client–server paradigm in the modern computing landscape. Examples of popular applications of the client–server paradigm are numerous and include: the World Wide Web, the Domain Name System (DNS), the X-windows, electronic mail, see Figure 4.6A, event services, see Figure 4.6B, and so on.

The World Wide Web illustrates the power of the client–server paradigm and its effects on the society. As of June 2011 there were close to 350 million web sites, in 2017 there are around one billion web sites. The web allows users to access *resources* such as text, images, digital music, and any imaginable type of information previously stored in a digital format. A *web page* is created using a description language called HTML (Hypertext Description Language). The information in each web page is encoded and formatted according to some standard, e.g., GIF, JPEG for images, MPEG for videos, MP3 or MP4 for audio, and so on.

The web is based upon a "pull" paradigm; the resources are stored at the server's site and the client pulls them from the server. Some web pages are created "on the fly" others are fetched from the disk. The client, called a *web browser* and the server communicate using an application-level protocol called HTTP (HyperText Transfer Protocol) built on top of the TCP transport protocol.

The web server also called an *HTTP server* listens at a well known port, port 80, for connections from clients. Figure 4.7 shows the sequence of events when a client browser sends an HTTP request to a server to retrieve some information and the server constructs the page on the fly and then the browser sends another HTTP request for an image stored on the disk. First a TCP connection between the client and the server is established using a process called a *three-way handshake.* The client provides an arbitrary initial sequence number in a special segment with the *SYN* control bit on; then the server acknowledges the segment and adds its own arbitrarily chosen initial sequence number; finally, the client sends its own acknowledgment *ACK* as well as the HTTP request and the connection is established. The time elapsed from the initial request till the server's acknowledgment reaches the client is called the RTT (Round-Trip Time).

The *response time*, defined as the time from the instance the first bit of the request is sent until the last bit of the response is received, consists of several components: the RTT, the *server residence time*, the time it takes the server to construct the response, and the data transmission time. RTT depends on the network latency, the time it takes a packet to cross the network from the sender to the receiver. The data transmission time is determined by the network bandwidth. In turn, the server residence time depends on the server load.

Often, the client and the server do not communicate directly, but through a proxy server as shown in Figure 4.8. Proxy servers could provide multiple functions; for example, they may filter client requests

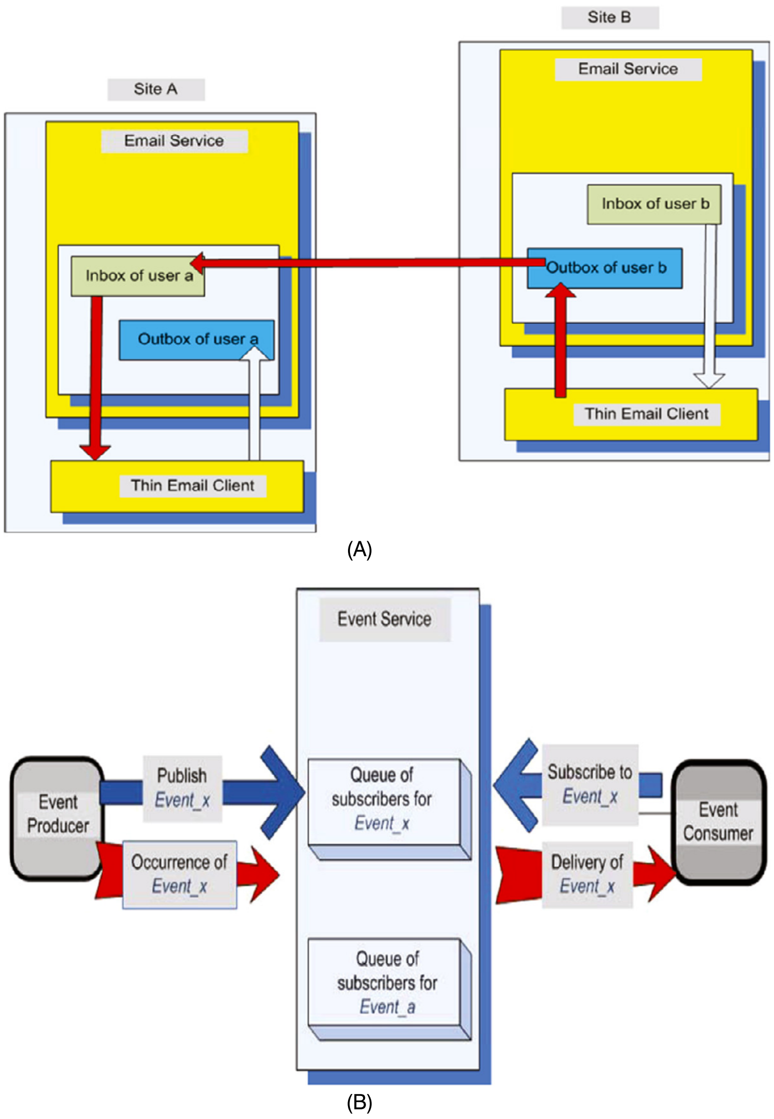


FIGURE 4.6

(A) Email service; the sender and the receiver communicate asynchronously using inboxes and outboxes. Mail daemons run at each site. (B) An event service supports coordination in a distributed system environment. The service is based on the publish-subscribe paradigm; an event producer publishes events and an event consumer subscribes to events. The server maintains queues for each event and delivers notifications to clients when an event occurs.

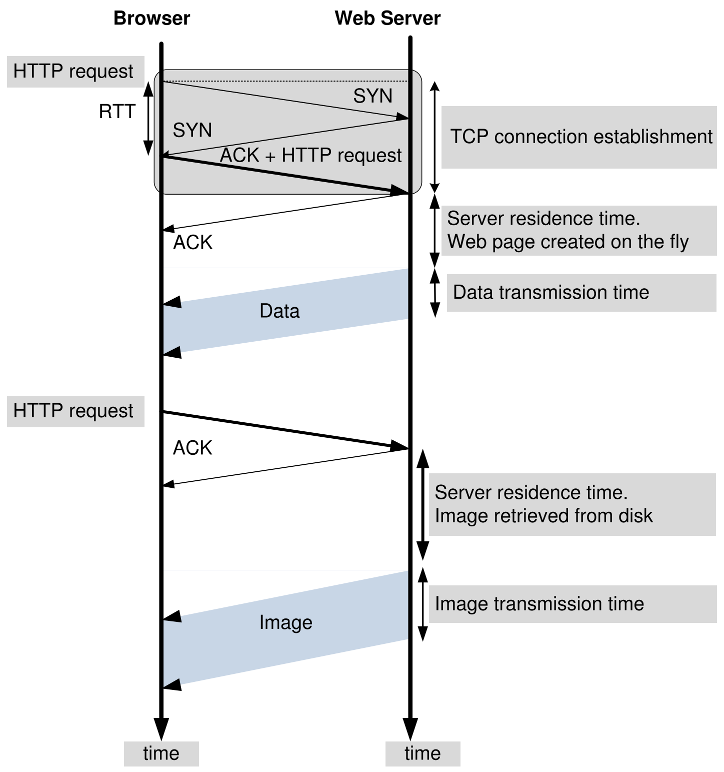


FIGURE 4.7

Client–server communication, the World Wide Web. The *three-way handshake* involves the first three messages exchanged between the client browser and the server. Once the TCP connection is established the HTTP server takes its time to construct the page to respond to the first request; to satisfy the second request the HTTP server must retrieve an image from the disk. The *response time* includes the RTT, the server residence time, and the data transmission time.

and decide whether or not to forward the request based on some filtering rules. A proxy server may redirect a request to a server in close proximity of the client or to a less loaded server. A proxy can also act as a cache and provide a local copy of a resource, rather than forward the request to the server.

Another type of client–server communication is *HTTP-tunneling* used most often as a means of communication from network locations with restricted connectivity. Tunneling means encapsulation of a network protocol, in our case HTTP acts as a wrapper for the communication channel between the client and the server, see Figure 4.8.

4.9 LAYERING AND HIERARCHY

Layering and hierarchy have been present in social systems since ancient times. For example, the Spartan Constitution, called Politeia, describes a Dorian society based on rigidly layered social system and

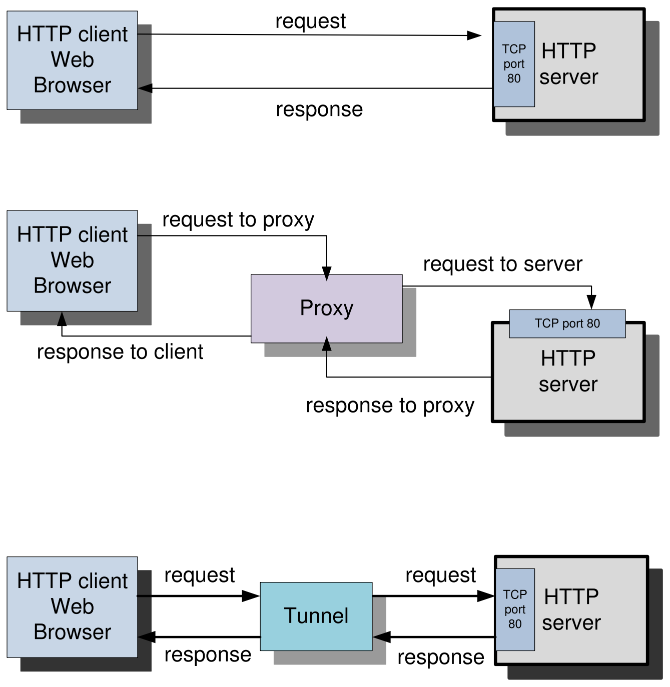


FIGURE 4.8

A client can communicate directly with the server, it can communicate through a proxy, or it may use tunneling to cross the network.

a strong military. Nowadays, in a modern society, we are surrounded by organizations structured hierarchically. We have to recognize that layering and hierarchical organization have their own problems, could negatively affect the society, impose a rigid structure and affect social interactions, increase the overhead of activities, and prevent the system from acting promptly when such actions are necessary.

Layering demands modularity as each layer fulfills a well-defined function. The communication patterns in case of layering are more restrictive, a layer is expected to communicate only with the adjacent layers. This restriction, the limitation of communication patterns, clearly reduces the complexity of the system and makes it easier to understand its behavior.

There is no surprise that modularity, layering, and hierarchy are critical for computer and communication systems. Since the early days of computing large programs have been split into modules, each with a well-defined functionality. Modules with related functionalities have then been grouped together into numerical, graphics, statistical, and many other types of libraries.

Layering helps us dealing with complicated problems when we have to separate concerns that prevent us from making optimal design decisions. To do so we define layers that address each concern and design clear interfaces between the layers.

Probably, the best example is layering of communication protocols. Early on it was recognized the need of accommodating a variety of physical communication channels that carry electromagnetic, optical, or acoustic signals thus, the need for a *physical* layer. The next concern is how to transport bits, not signals between two systems directly connected to one another by a communication channel thus, the need for a *data link* layer.

Communication requires networks with multiple intermediate nodes. When bits have to traverse a chain of intermediate nodes from a source to the destination the concern is how to forward the bits from one intermediate node to the next, so the *network* layer was introduced. Then, it was recognized that the source and the recipient of information are in fact outside the network and the sender only wants the data to reach destination unaltered. Therefore, the *transport* layer was deemed necessary. Finally, the data sent and received has a meaning only in the context of an application thus, the need for the *application* layer.

Strictly enforced layering can prevent optimizations. For example, cross-layer communication in networking was proposed to allow wireless applications to take advantage of information available at the Media Access Control (MAC) sub-layer of the data link layer. This example shows that layering gives us insight as to where to place the basic mechanisms for error control, flow control, and congestion control of the network protocol stack.

An interesting question is if a layered cloud architecture could be designed that has practical implications for the future development of computing clouds. One could argue that it may be too early for such an endeavor, that we need time to fully understand how to better organize a cloud infrastructure and we need to gather data to support the advantages of one approach over another.

On the other hand, there are other systems where it is difficult to envision a layered organization because of the complexity of the interaction between the individual modules. Consider for example an operating system which has a set of well-defined functional components:

- The processor management subsystem, responsible for processor virtualization, scheduling, interrupt handling, and execution of privileged operations and system calls.
- The virtual memory management subsystem, responsible for translating virtual addresses to physical addresses.
- The multi-level memory management subsystem, responsible for transferring storage blocks between different memory levels, most commonly between primary and secondary storage.
- The I/O subsystem, responsible for transferring data between the primary memory and the I/O devices.
- The networking subsystem responsible for network communication.

The processor management interacts with all the other subsystems and there are also multiple interactions between the other subsystems; therefore, it seems unlikely that a layered organization would be feasible in this case.

4.10 VIRTUALIZATION; LAYERING AND VIRTUALIZATION

Virtualization abstracts the underlying physical resources of a computer or communication system and simplifies their use, isolates users from one another, and supports replication which, in turn, increases the elasticity of the system. Virtualization has been used successfully since the late 1950s; a virtual memory based on paging was first implemented on the Atlas computer at University of Manchester in the United Kingdom, in 1959.

Virtualization simulates the interface to a physical object by any one of four means [434]:

1. Multiplexing: create multiple virtual objects from one instance of a physical object. For example, a processor is multiplexed among a number of processes or threads.
2. Aggregation: create one virtual object from multiple physical objects. For example, a number of physical disks are aggregated into a RAID disk.
3. Emulation: construct a virtual object from a different type of a physical object. For example, a physical disk emulates a Random Access Memory.
4. Multiplexing and emulation. Examples: virtual memory with paging multiplexes real memory and disk and a virtual address emulates a real address; the TCP protocol emulates a reliable bit pipe and multiplexes a physical communication channel and a processor.

Virtualization is a critical aspect of cloud computing, equally important for the providers and the consumers of cloud services, and plays an important role for:

- System security, as it allows isolation of services running on the same hardware.
- Performance and reliability, as it allows applications to migrate from one platform to another.
- The development and management of services offered by a provider.
- Performance isolation.

In a cloud computing environment a hypervisor or virtual-machine monitor runs on the physical hardware and exports hardware-level abstractions to one or more guest operating systems. A guest OS interacts with the virtual hardware in the same manner it would interact with the physical hardware, but under the watchful eye of the hypervisor which traps all privileged operations and mediates the interactions of the guest OS with the hardware. For example, a hypervisor would control I/O operations to two virtual disks implemented as two different set of tracks on a physical disk. New services can be added without the need to modify an operating system.

User convenience is a necessary condition for the success of the utility computing paradigm; one of the multiple facets of user convenience is the ability to run remotely using the system software and libraries required by the application. User convenience is a major advantage of a VM architecture versus a traditional operating system. For example, an AWS user could submit an Amazon Machine Image (AMI) containing the applications, libraries, data, and the associated configuration settings; the user could choose the operating system for the application, then start, terminate, and monitor as many instances of the AMI as needed, using the web service APIs and the performance monitoring and management tools provided by the AWS.

There are side effects of virtualization, notably the *performance penalty* and the *hardware costs*. All privileged operations of a VM must be trapped and validated by the hypervisor which, ultimately, controls the system behavior. The increased overhead introduced by the hypervisor has a negative impact on the performance.

The cost of a system running multiple VMs is higher than the cost of a system running a traditional OS. In the former case the physical hardware is shared among a set of guest operating systems and it is typically configured with faster and/or multicore processors, more memory, larger disks, and additional network interfaces as compared to a system running a traditional operating system.

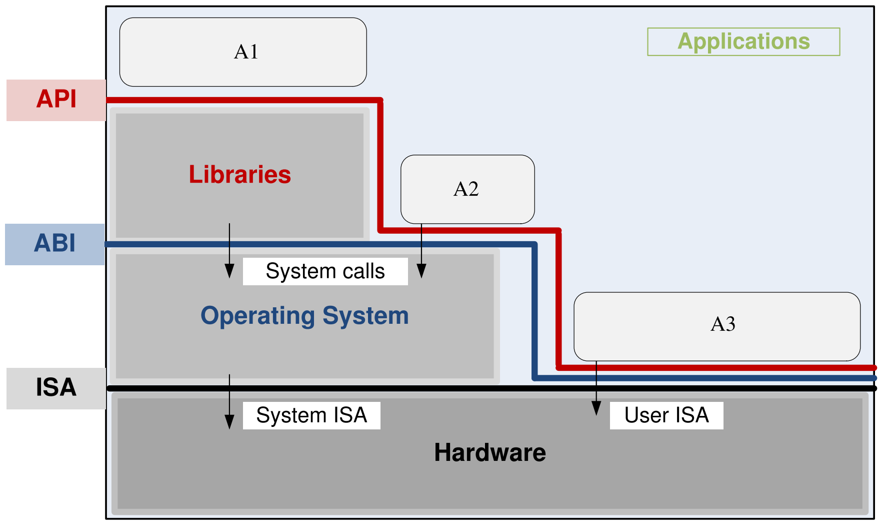


FIGURE 4.9

Layering and interfaces between layers in a computer system. The software components including applications, libraries, and operating system interact with the hardware via several interfaces: the Application Program Interface (API), the Application Binary Interface (ABI), and the Instruction Set Architecture (ISA). An application uses library functions (A1), makes system calls (A2), and executes machine instructions (A3).

Layers and interfaces between layers. A common approach to managing system complexity is to identify a set of *layers* with well-defined *interfaces* among them. The interfaces separate different levels of abstraction. Layering minimizes the interactions among the subsystems and simplifies the description of the subsystems. Each subsystem is abstracted through its interfaces with the other subsystems thus, we are able to design, implement, and modify the individual subsystems independently.

The ISA (Instruction Set Architecture) defines the set of instructions of a processor; for example, the Intel architecture is represented by the *x86*-32 and *x86*-64 instruction sets for systems supporting 32-bit addressing and 64-bit addressing, respectively. The hardware supports two execution modes, a *privileged*, or *kernel* mode, and a *user* mode.

The instruction set consists of two sets of instructions, *privileged* instructions that can only be executed in kernel mode and the *non-privileged* instructions that can be executed in user mode. There are also *sensitive instructions* that can be executed in kernel and in user mode, but behave differently, see Section 10.3.

Computer systems are fairly complex and their operation is best understood when we consider a model similar with the one in Figure 4.9 which shows the interfaces between the software components and the hardware [455]. The hardware consists of one or more multicore processors, a system interconnect (e.g., one or more busses) a memory translation unit, the main memory, and I/O devices, including one or more networking interfaces.

Applications written mostly in High Level Languages (HLL) often call library modules and are compiled into *object code*. Privileged operations, such as I/O requests, cannot be executed in user mode; instead, application and library modules issue *system calls* and the operating system determines if the privileged operations required by the application do not violate system security or integrity and, if so, executes them on behalf of the user. The binaries resulting from the translation of HLL programs are targeted to a specific hardware architecture.

The first interface, at the boundary of the hardware and the software, is the *Instruction Set Architecture* (ISA). The next interface is the *Application Binary Interface* (ABI) which allows the ensemble consisting of the application and the library modules to access the hardware. ABI does not include privileged system instructions, instead it invokes system calls.

Finally, the *Application Program Interface* (API) defines the set of instructions the hardware was designed to execute and gives the application access to the ISA. API includes HLL library calls which often invoke system calls. Recall that a *process* is the abstraction for the code of an application at execution time; a *thread* is a light-weight process. ABI is the projection of the computer system seen by the process and API is the projection of the same system from the perspective of the HLL program.

The binaries created by a compiler for a specific ISA and a specific operating systems are not portable. Such code cannot run on a computer with a different ISA, or on the computer with the same ISA, but a different OS. It is possible though to compile an HLL program for a VM environment, as shown in Figure 4.10. In this case portable code is produced and distributed and then converted by binary translators to the ISA of the host system. A *dynamic binary translation* converts blocks of guest instructions from the portable code to the host instructions and leads to a significant performance improvement, as such blocks are cached and reused.

4.11 PEER-TO-PEER SYSTEMS

Distributed systems discussed in this chapter allow access to resources in a tightly controlled environment. System administrators enforce security rules and control the allocation of physical, rather than virtual resources. In all models of network-centric computing prior to utility computing a user maintained direct control of the software and the data residing on remote systems.

This user-centric model, in place since early 1960s, was challenged in 1990s by the peer-to-peer (P2P) model. P2P systems share some ideas with computer clouds. The new distributed computing model promoted the idea of low-cost access to storage and CPU cycles provided by participant systems. In this case, the resources are located in different administrative domains. The P2P systems are self-organizing and decentralized, while the servers in a cloud are in a single administrative domain and have a central management.

P2P systems exploit the network infrastructure to provide access to distributed computing resources. Decentralized applications developed in the 1980s such as SMTP (Simple Mail Transfer Protocol), a protocol for Email distribution, and NNTP (Network News Transfer Protocol), an application protocol for dissemination of news articles, are early examples of P2P systems.

Systems developed in late 1990s such as the music-sharing system Napster gave participants access to storage distributed over the network. The first volunteer-based scientific computing, SETI@home, used free cycles of participating systems to carry out compute-intensive tasks.

The P2P model represents a significant departure from the client–server model, the cornerstone of distributed applications for several decades. P2P systems have several desirable properties [426]:

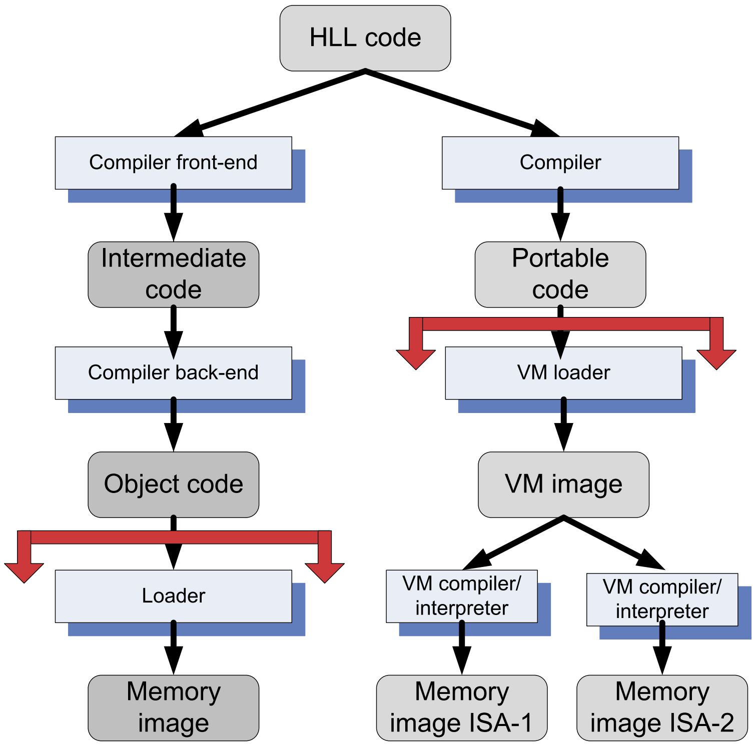


FIGURE 4.10

High Level Language (HLL) code can be translated for a specific architecture and operating system. The HLL code can also be compiled into portable code and then the portable code can be translated for systems with different ISAs. The shared/distributed resulting code is the object code in the first case and the portable code in the second case.

- Require a minimally dedicated infrastructure, as resources are contributed by the participating systems.
- Are highly decentralized.
- Are scalable, the individual nodes are not required to be aware of the global state.
- Are resilient to faults and attacks, as few of their elements are critical for the delivery of service and the abundance of resources can support a high degree of replication.
- Individual nodes do not require excessive network bandwidth as servers used in case of the client–server model do.
- The systems are shielded from censorship due to the dynamic and often unstructured system architecture.

Undesirable properties of peer-to-peer systems are also notable. Decentralization raises the question if P2P systems can be managed effectively and provide the security required by various applications.

The fact that they are shielded from censorship makes them a fertile ground for illegal activities including distribution of copyrighted content.

The new paradigm was embraced by applications other than file sharing. Since 1999 new P2P applications such as the ubiquitous Skype, a voice over IP telephony service,[4] data streaming applications such as Cool Streaming [546] and BBC's online video service, content distribution networks such as CoDeeN [511], and volunteer computing applications based on the Berkeley Open Infrastructure for Networking Computing (BOINC) platform [32], have proved their appeal to users.

Skype reported in 2008 that 276 million registered users have used more than 100 billion minutes for voice and video calls. The site www.boinc.berkeley.edu reports that at the end of June 2012 volunteer computing involved more than 275 000 individuals and more than 430 000 computers supplying a monthly average of almost 6.3×10^9 MFlops. It is also reported that the peer-to-peer traffic accounts for a very large fraction of the Internet traffic, with estimates ranging from 40% to more than 70%.

Many groups from industry and academia rushed to develop and test new ideas taking advantage of the fact that P2P applications do not require a dedicated infrastructure. Applications such as Chord [466] and Credence [509] address issues critical for the effective operation of decentralized systems.

Chord is a distributed lookup protocol to identify the node where a particular data item is stored. The routing tables are distributed and, while other algorithms for locating an object require the nodes to be aware of most of the nodes of the network, Chord maps a key related to an object to a node of the network using routing information about a few nodes only.

Credence is an object reputation and ranking scheme for large-scale P2P file sharing systems. Reputation is of paramount importance for systems which often include many unreliable and malicious nodes. In the decentralized algorithm used by *Credence* each client uses local information to evaluate the reputation of other nodes and shares its own assessment with its neighbors. The credibility of a node depends only on the votes it casts.

Each node computes the reputation of another node based solely on the degree of matching with its own votes and relies on like-minded peers. Overcite [470] is a P2P application to aggregate documents based on a three-tier design. The web front-ends accept queries and display the results while servers crawl through the web to generate indexes and to perform keyword searches; the web back-ends store documents, metadata, and coordination state on the participating systems.

The rapid acceptance of the new paradigm triggered the development of a new communication protocol allowing hosts at the network periphery to cope with the limited network bandwidth available to them. BitTorrent is a peer-to-peer file sharing protocol enabling a node to download/upload large files from/to several hosts simultaneously.

P2P systems differ in their architecture. Some do not have any centralized infrastructure, while others have a dedicated controller, but this controller is not involved in resource-intensive operations. For example, Skype has a central site to maintain the user accounts; the users sign in and pay for specific activities at this site. The controller for a BOINC platform maintains membership and is involved in task distribution to participating systems. The nodes with abundant resources in systems without any centralized infrastructure often act as *supernodes* and maintain information useful to increasing the system efficiency, e.g., indexes of the available content.

[4]Skype allows close to 700 million registered users from many countries around the globe to communicate using a proprietary voice-over-IP protocol. The system developed in 2003 by Niklas Zennström and Julius Friis was acquired by Microsoft in 2011 and nowadays is a hybrid P2P and client–server system.

Regardless of the architecture, P2P systems are built around an *overlay network*, a virtual network superimposed over the real network. Each node maintains a table of *overlay links* connecting it with other nodes of this virtual network, each node being identified by its IP addresses. Two types of overlay networks, *unstructured* and *structured*, are used by P2P systems. Random walks starting from a few bootstrap nodes are usually used by systems desiring to join an unstructured overlay.

Each node of a structured overlay has a unique key which determines its position in the structure; the keys are selected to guarantee a uniform distribution in a very large name space. Structured overlay networks use *key-based routing* (KBR); given a starting node v_0 and a key k, the function $KBR(v_0, k)$ returns the path in the graph from v_0 to the vertex with key k. Epidemic algorithms are often used by unstructured overlays to disseminate network topology.

4.12 LARGE-SCALE SYSTEMS

The developments in computer architecture, storage technology, networking, and software during the last several decades of the twentieth century coupled with the need to access and process information led to several large-scale distributed system developments:

- The web and the semantic web expected to support composition of services (not necessarily computational services) available on the web. The web is dominated by unstructured or semi-structured data, while the semantic web advocates inclusion of semantic content in web pages.
- The Grid, initiated in early 1990s by National Laboratories and universities primarily for applications in science and engineering.

The need to share data from high energy physics experiments motivated Sir Tim Berners-Lee, who worked at CERN at Geneva in late 1980s, to put together the two major components of the World Wide Web: HTML (Hypertext Markup Language) for data description and HTTP (Hypertext Transfer Protocol) for data transfer. The web opened a new era in data sharing and ultimately led to the concept of network-centric content.

The *semantic Web* is an effort to enable lay people to find, share, and combine information available on the web more easily. The name was coined by Berners-Lee to describe "a web of data that can be processed directly and indirectly by machines." It is a framework for data sharing among applications based on the Resource Description Framework (RDF). In this vision, the information can be readily interpreted by machines, so machines can perform more of the tedious work involved in finding, combining, and acting upon information on the web.

The semantic web is "largely unrealized" according to Berners-Lee. Several technologies are necessary to provide a formal description of concepts, terms, and relationships within a given knowledge domain; they include the Resource Description Framework (RDF), a variety of data interchange formats, and notations such as RDF Schema (RDFS) and the Web Ontology Language (OWL).

Gradually, the need to make computing more affordable and to liberate the users from the concerns regarding system and software maintenance reinforced the idea of concentrating computing resources in data centers. Initially, these centers were specialized, each running a limited palette of software systems, as well as applications developed by the users of these systems. In the early 1980s major research organizations, such as the National Laboratories and large companies, had powerful computing centers supporting large user populations scattered throughout wide geographic areas. Then the idea to link such centers in an infrastructure resembling the power grid was born; the model known as network-centric computing was taking shape.

A *computing grid* is a distributed system consisting of a large number of loosely coupled, heterogeneous, and geographically dispersed systems in different administrative domains. The term *computing grid* is a metaphor for accessing computer power with similar ease as we access power provided by the electric grid. Software libraries known as *middleware* were furiously developed since early 1990s to facilitate access to grid services.

The vision of the grid movement was to give a user the illusion of a very large virtual supercomputer. The autonomy of the individual systems and the fact that these systems were connected by wide-area networks with latency higher than the latency of the interconnection network of a supercomputer posed serious challenges to this vision. Nevertheless, several "Grand Challenge" problems, such as protein folding, financial modeling, earthquake simulation, and climate/weather modeling, run successfully on specialized grids. The Enabling Grids for Escience project is arguably the largest computing grid; along with the LHC Computing Grid (LCG), the Escience project aims to support the experiments using the Large Hadron Collider (LHC) at CERN which generates several gigabytes of data per second, or 10 PB (petabytes) per year.

In retrospect, two basic assumptions about the infrastructure prevented the grid movement from having the impact its supporters were hoping for. The first is the heterogeneity of the individual systems interconnected by the grid. The second is that systems in different administrative domain are expected to cooperate seamlessly. Indeed, the heterogeneity of the hardware and of the system software poses significant challenges for application development and for application mobility.

At the same time, critical areas of system management including scheduling, optimization of resource allocation, load balancing, and fault-tolerance are extremely difficult in a heterogeneous system. The fact that resources are in different administrative domains further complicates many, already difficult, problems related to security and resource management. While very popular in the science and the engineering communities, the grid movement did not address the major concerns of enterprise computing community and did not make a noticeable impact on the IT industry.

Cloud computing is a technology largely viewed as the next big step in the development and deployment of an increasing number of distributed applications. The companies promoting cloud computing seem to have learned the most important lessons from the grid movement. Computer clouds are typically homogeneous. An entire cloud shares the same security, resource management, cost and other policies, and last but not least, it targets enterprise computing. These are some of the reasons why several agencies of the US Government including the Health and Human Services, the Center for Disease Control (CDC), NASA, Navy's Next Generation Enterprise Network (NGEN), and Defense Information Systems Agency (DISA) have launched cloud computing initiatives and conduct actual system developments intended to improve the efficiency and effectiveness of their information processing needs.

4.13 COMPOSABILITY BOUNDS AND SCALABILITY

Nature creates complex systems from simple components. For example, a vast variety of proteins are linear chains assembled from twenty one amino acids, the building blocks of proteins. Twenty amino acids are naturally incorporated into polypeptides and are encoded by the genetic code.

Imitating nature, man-made systems are assembled from sub-assemblies; in turn, a sub-assembly is made from several modules, each module could consist of sub-modules, and so on. Composability has natural bounds imposed by the laws of physics as we have seen when discussing heat dissipation of solid state devices. As the number of components increases, the complexity of a system also increases.

The limits of composability are reached as new physical phenomena affect the system whenever the physical size of the individual components changes. A recent paper with the suggestive title “When every atom counts” [345] discusses the fact that even the most modern solid state fabrication facilities cannot produce chips with consistent properties. The percentage of defective or substandard chips has been constantly increasing as the components have become smaller and smaller.

The lack of consistency in the manufacturing process of solid state devices is attributed to the increasingly smaller size of the physical components of a chip. This problem is identified by the International Technology Roadmap for Semiconductors as “a red brick,” a problem without a clear solution, a wall that could prevent the further progress. Chip consistency is no longer feasible because the transistors and the “wires” on a chip are so small that random differences in the placement of an atom can have a devastating effect, e.g., it can increase the power consumption by an order of magnitude and slowdown the chip by as much as 30%.

As the features become smaller and smaller the range of the *threshold voltage,* the voltage needed to turn a transistor on and off, has been widening and many transistors have this threshold voltage at or near zero thus, they cannot operate as switches. While the range for the 28 nm technology was approximately between $+0.01$ and $+0.4$ V, the range for the 20 nm technology is between -0.05 and $+0.45$ V and the range becomes even wider, from -0.18 to $+0.55$ for the 14 nm technology.

There are physical bounds for the composition of analog systems: noise accumulation, heat dissipation, cross-talk, the interference of signals on multiple communication channels, and several other factors limit the number of components of an analog system. Digital systems have more distant bounds, but composability is still limited by physical laws.

There are virtually no bounds on composition of digital computing and communication systems controlled by software. The Internet is a network of networks and a prime example of composability with distant bounds. Computer clouds are another example; a cloud is composed of a very large number of servers and interconnects, each server is made up of multiple processors, and each processor has multiple cores. Software is the ingredient which pushes the composability bounds and liberates computer and communication system from the limits imposed by physical laws.

In the physical world the laws valid at one scale break down at a different scale, e.g., the laws of classical mechanics are replaced at atomic and subatomic scale by quantum mechanics. Thus, we should not be surprised that scale really matters in the design of computing and communication systems. Indeed, architectures, algorithms, and policies that work well for systems with a small number of components very seldom scale up.

For example, many computer clusters have a front-end which acts as the nerve center of the system, manages communication with the outside world, monitors the entire system, and supports system administration and software maintenance. A computer cloud has multiple such nerve centers and new algorithms to support collaboration among these centers must be developed. Scheduling algorithms that work well within the confines of a single system cannot be extended to collections of autonomous systems when each system manages local resources; in this case, as in the previous example, entities must collaborate with one another and this requires communication and consensus.

Another manifestation of this phenomenon is in the vulnerabilities of large-scale distributed systems. The implementation of Google's Bigtable revealed that many distributed protocols designed to protect against network partitions and fail-stop are unable to cope with failures due to scale [96]. Memory and network corruption, extended and asymmetric network partitions, systems that fail to respond, and large clock skews occur with increased frequency in a large-scale system and they interact with one another in a manner that greatly affects the overall system availability.

Scaling has other dimensions than just the number of components; the space plays an important role, the communication latency is small when the component systems are clustered together within a small area and allows us to implement efficient algorithms for global decision making, e.g., consensus algorithms. When, for the reasons discussed in Section 1.6, the data centers of a cloud provider are distributed over a large geographic area, transactional database systems are of little use for most online transaction oriented systems and a new type of data store has to be introduced in the computational ecosystem.

Societal scaling means that a service is used by a very large segment of population and/or is a critical element of the infrastructure. There is no better example to illustrate how societal scaling affects the system complexity than communication supported by the Internet. The infrastructure supporting the service must be highly available. A consequence of redundancy and of the measures to maintain consistency is increased system complexity.

At the same time, the popularity of the service demands simple and intuitive means to access the infrastructure. Again, the system complexity increases due to the need to hide the intricate mechanisms from a lay person with little understanding of the technology. The vulnerability of wireless systems has increased due to the desire to design wireless devices that: (a) operate efficiently in terms of power consumption; (b) present the user with a simple interface and few choices; and (c) satisfy a host of other popular functions. This is happening at the time when not many smart phone and tablet users understand the security risks of wireless communication.

4.14 HISTORY NOTES AND FURTHER READINGS

Two theoretical developments in 1930s were critical for the development of modern computers; the first was the publication of Alan Turing's 1936 paper [489]. The paper provided a definition of a universal computer, called a Turing machine, which executes a program stored on tape; the paper also proved that there were problems such as the halting problem, that could not be solved by any sequential process. The second major development was the publication in 1937 of Claude Shannon's master's thesis at MIT "A Symbolic Analysis of Relay and Switching Circuits" in which he showed that any Boolean logic expression can be implemented using logic gates.

The first Turing complete[5] computing device was Z3, an electro-mechanical device built by Konrad Zuse in Germany in May 1941; Z3 used a binary floating-point representation of numbers and was program-controlled by a film-stock. The first programmable electronic computer ENIAC, built at the Moore School of Electrical Engineering at the University of Pennsylvania by a team led by John

[5] A Turing complete computer is equivalent to a universal Turing machine except for memory limitations.

Presper Eckert and John Mauchly, became operational in July 1946 [337]; ENIAC, unlike Z3, used a decimal number system and was program-controlled by patch cables and switches.

John von Neumann, the famous mathematician and theoretical physicist, contributed fundamental ideas for modern computers [81,504,505]. He was one of the most brilliant minds of the twentieth century, with an uncanny ability to transform fuzzy ideas and garbled thoughts to crystal clear and scientifically sound concepts. John von Neumann drew the insight for the stored-program computer from Alan Turing's work[6] and from his visit at University of Pennsylvania.

John von Neumann thought that ENIAC was an engineering marvel, but was less impressed with the awkward manner to "program" it by manually connecting cables and setting switches. He introduced the so-called "von Neumann architecture" in a report published in 1940s. To this day he is faulted by some because he failed to mention in this report the sources of his insight.

John von Neumann led the development at the Institute of Advanced Studies at Princeton of MANIAC, an acronym for "mathematical and numerical integrator and computer." MANIAC was closer to the modern computers than any of its predecessors. This computer was used for sophisticated calculations required by the development of the hydrogen bomb nicknamed "Ivy Mike" secretly detonated on November 1, 1952, over an island that no longer exists in the South Pacific. In a recent book [158] the historian of science George Dyson writes: "The history of digital computing can be divided into an Old Testament whose prophets, led by Leibnitz, supplied the logic, and a New Testament whose prophets led by von Neumann built the machines. Alan Turing arrived between them."

In 1951 Sir Maurice Vincent Wilkes developed the concept of microprogramming first implemented in the EDSAC 2 computer. Wilkes is also credited with the idea of symbolic labels, macros and subroutine libraries.

Third-generation computers were built during the 1964–1971 period; they made extensive use of integrated circuits (ICs) and ran under the control of an operating systems. MULTICS (Multiplexed Information and Computing Service) was an early time-sharing operating system for the GE 645 mainframe, developed jointly by MIT, GE, and Bell Labs. Multics (often called Multix) had numerous novel features and implemented a fair number of interesting concepts such as: a hierarchical file system, access control lists for file information sharing, dynamic linking, and on-line reconfiguration.

In his address "A Career in Computer System Architecture" MIT Professor Jack Dennis writes: "In 1960 Professor John McCarthy, now at Stanford University and known for his contributions to artificial intelligence, led the *Long Range Computer Study Group* (LRCSG) which proposed objectives for MIT's future computer systems. I had the privilege of participating in the work of the LRCSG, which led to Project MAC and the Multics computer and operating system, under the organizational leadership of Prof. Robert Fano and the technical guidance of Prof. Fernando Corbató. At this time Prof. Fano had a vision of the Computer Utility – the concept of the computer system as a repository for the knowledge of a community – data and procedures in a form that could be readily shared – a repository that could be built upon to create ever more powerful procedures, services, and active knowledge from those already in place. Prof. Corbató's goal was to provide the kind of central computer installation and operating system that could make this vision a reality. With funding from DARPA, the Defense

[6]Alan Turing came to Institute of Advanced Studies at Princeton in 1936 and got his Ph.D. there in 1938; von Neumann offered him a position at the Institute but, as the dark clouds signaling the approaching war were gathering over Europe, Turing decided to go back to England.

Advanced Research Projects Agency, the result was Multics...... in the 1970s I found it easy to get government funding. The agencies were willing to fund pretty wild ideas, and I was supported to do research on *data flow* architecture, first by NSF and later by the DOE" (http://csg.csail.mit.edu/Users/dennis/essay.htm).

The development of the UNIX system was a consequence of the withdrawal of Bell Labs from the Multics project in 1968. UNIX was developed in 1969 for a DEC PDP minicomputer by a group led by Kenneth Thompson and Dennis Ritchie [422]. According to [421] "the most important job of *UNIX* is to provide a file-system." The same reference discusses another concept introduced by the system: "For most users, communication with UNIX is carried on with the aid of a program called the Shell. The Shell is a command line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs."

The first microprocessor, the Intel 4004, announced in 1971, performed binary-coded decimal (BCD) arithmetic using 4-bit words; it was followed in 1971 by Intel 8080, the first 8-bit microprocessor, and by its competitor, Motorola 6800 released in 1974. The first 16-bit multi-chip microprocessor, IMP-16, was announced in 1973 by National Semiconductor. The 32-bit microprocessors appeared in 1979; widely used Motorola MC68000, had 32-bit registers and supported 24-bit addressing. Intel's 80286 was introduced in 1982. The 64-bit processor era was inaugurated by AMD64, an architecture called x86-64, backward compatible with Intel x86 architecture. Dual-core processors appeared in 2005; multicore processors are ubiquitous in today's servers, PCs, tablets, and even smart phones.

The development of distributed systems was only possible after major advances in communication technology. The Advanced Research Projects Agency (ARPA) created in 1958 funded research at multiple universities and businesses sites and the ARPANET project to connect them all into a network was initiated.

The Internet is a global network based on the Internet Protocol Suite (TCP/IP); its origins can be traced back to 1965 when Ivan Sutherland, the Head of the Information Processing Technology Office (IPTO) at ARPA, encouraged Lawrence Roberts who had worked previously at MIT's Lincoln laboratories to become the Chief Scientist at IPTO and to initiate a networking project based on packet switching rather than circuit switching.

Leonard Kleinrock from UCLA developed the theoretical foundations for packet-switched networks in early 1960s and for hierarchical routing in packet-switching networks in early 1970s. Kleinrock published the first paper in 1961 and the first book on packet-switching theory in 1964.

In August 1968 DARPA released a request for quotation (RFQ) for the development of packet-switches called Interface Message Processors (IMPs). A group from Bolt Beranek and Newman (BBN) won the contract. Several researchers including Robert Kahn from BBN, Lawrence Roberts from DARPA, Howard Frank from Network Analysis Corporation, and Leonard Kleinrock from UCLA and their teams played a major role in the overall ARPANET architectural design. The idea of open-architecture networking was first introduced by Kahn in 1972 and his collaboration with Vint Cerf from Stanford led to the design of TCP/IP. Three groups, one at Stanford, one at BBN, and one at UCLA won the DARPA contract to implement the TCP/IP.

In 1969 BBN installed the first IMP at UCLA. The first two interconnected nodes of the ARPANET were the Network Measurement Center at the UCLA's School of Engineering and Applied Science and the SRI International in Menlo Park, California. Two more nodes were added at UC Santa Barbara and University of Utah. By the end of 1971 there were 15 sites interconnected by ARPANET.

Ethernet technology, developed by Bob Metcalfe at Xerox PARC in 1973 and other local area network technologies, such as token passing rings, allowed the personal computers and the workstations to be connected to the Internet in the 1980s. As the number of Internet hosts increased, it was no longer feasible to have a single table of all hosts and their addresses. The Domain Name System (DNS) was invented by Paul Mockapetris of USC/ISI. The DNS permitted a scalable distributed mechanism for resolving hierarchical host names into an Internet address.

UC Berkeley with support from DARPA rewrote the TCP/IP code developed at BBN and incorporated it into the Unix BSD system. In 1985 Dennis Jennings started the NSFNET program at NSF to support the general research and academic communities.

The first distributed computing programs were a pair of worm programs called Creeper and Reaper. In 1970s Creeper was using the idle CPU cycles of processors in the ARPANET to copy itself onto the next system and then delete itself from the previous one. Then it was modified to remain on all previous computers. Reaper deleted all copies of the Creeper.

Further readings. Several texts are highly recommended. "Computer Architecture: A Quantitative Approach" [228] by John Hennessy and David Patterson is the authoritative reference for computer architecture. The text "Principles of Computer Systems Design" co-authored by Jerome Saltzer and Frans Kaashoek [434] covers basic concepts in computer system design. "Computer networks: a top-down approach featuring the Internet" by James Kurose and Keith Ross is a good introduction to networking.

Amdahl's paper [26] is a classic and [421] and [422] are the references for UNIX. [235] covers multicore processors.

The development of the MULTICS system [117,118] had a lasting impact on the design and implementation of computer systems. Load balancing in distributed system is analyzed in [159].

A comprehensive survey of peer-to-peer systems was published in 2010 [426]. *Chord* [466] and *Credence* [509] are important references in the area of peer-to-peer systems.

4.15 EXERCISES AND PROBLEMS

Problem 1. Do you believe that the homogeneity of a large-scale distributed systems is an advantage? Discuss the reasons for your answer. What aspects of hardware homogeneity are the most relevant in your view and why? What aspects of software homogeneity do you believe are the most relevant and why?

Problem 2. Peer-to-peer systems and clouds share a few goals, but not the means to accomplish them. Compare the two classes of systems in terms of architecture, resource management, scope, and security.

Problem 3. Explain briefly how the *publish-subscribe* paradigm works and discuss its application to services such as bulletin boards, mailing lists, and so on. Outline the design of an event service based on this paradigm, see Figure 4.6B. Can you identify a cloud service that emulates an event service?

Problem 4. Tuple-spaces can be thought of as an implementation of a distributed shared-memory. Tuple-spaces have been developed for many programming languages including Java, Lisp, Python, Prolog, Smalltalk, and Tcl (Tool Command Language). Explain briefly how tuple spaces work. How secure and scalable are the tuple spaces you are familiar with, e.g., JavaSpaces?

Problem 5. Arithmetic intensity is defined as the number of floating-point operations divided by the number of bytes in the main memory accessed for running a program.

1. Give examples of computations with low, medium, and high arithmetic intensity.
2. Derive a formula relating the attainable performance to the peak performance of a processor, the peak memory bandwidth, and the arithmetic intensity.
3. How is this formula related to the roofline model?

Problem 6. Dynamic instruction scheduling is discussed in Section 3.5 of [228].

1. What is the role of the reservation stations used by Tomasulo's approach for dynamic instruction scheduling?
2. Draw a diagram of a system with two reservation station.
3. What are the instruction execution steps for dynamic instruction scheduling?

Problem 7. There are several approaches to hardware multithreading, fine-grained, coarse-grained, and simultaneous (SMT). What are benefits and disadvantages of each one of them and if and where are they used.

Problem 8. Amazon realized the benefits of EC2 instances with GPU co-processors for data-intensive applications and in 2016 introduced the P2 instances.

1. Is it beneficial to have the GPUs attached as co-processors, or is it a disadvantage?
2. Is the thread scheduling inside a GPU done by hardware or controlled by the application?
3. Is it beneficial to add more CUDA threads to an application?

Problem 9. Reference [56] analyzes the power consumption of computing, storage, and networking infrastructure in a cloud data center. Find the percentage of the total power consumed by the CPUs, the DRAM, the disks, the networking, and the other consumers and then suggest the most effective means to reduce the power consumption.

SECTION 2

CHAPTER

CLOUD ACCESS AND CLOUD INTERCONNECTION NETWORKS

5

The decades-long evolution of microprocessor and storage technologies, computer architecture and software systems, parallel algorithms and distributed control strategies, paved the way to cloud computing. The interconnectivity supported by a continually evolving Internet made cloud computing feasible. A cloud is built around a high-performance interconnect, the servers of a cloud infrastructure communicate through high-bandwidth and low-latency networks. Unquestionably, communication is at the heart of cloud computing.

At the scale of a single system optimal performance requires a balance between the bandwidth, the number of operations per unit of time, of the three major subsystems, CPU, memory, and I/O. The reality is that processor bandwidth is orders of magnitude higher than the I/O bandwidth and much higher than the memory bandwidth. This imbalance is further amplified by a well-known empirical law, the Moore's Law whose corollary is that the processor performance doubles every 18 months or so, much faster than memory and I/O.

The effects of this imbalance are inevitably amplified in a large-scale system where the interconnect allows a very large number of processors to work together under the control of the orchestration software. The designers of a cloud computing infrastructure are acutely aware that the communication bandwidth goes down and the communication latency increases the farther from the CPU data travels. The limits of cloud interconnection networks latency and bandwidth are tested by the demands of a cloud infrastructure with millions of servers.

Cloud workloads fall into four broad categories based on their dominant resource needs: CPU-intensive, memory-intensive, I/O-intensive, and storage-intensive. While the first two benefit from, but do not require, high-performing networking, the last two do. Networking performance directly impacts the performance of I/O- and storage-intensive workloads.

A recent report [454] forecasts that by 2018, more than 10% of hyperconverged integrated systems[1] deployments will suffer from avoidable network-induced performance problems, up from less than 1% of today's systems. It is also forecasted that 60% of providers will start offering integrated networking services, along with compute and storage services.

The costs of the networking infrastructure continue to raise at a time when the costs of the other components of the cloud infrastructure continue to decrease. Moreover, many cloud applications including analytics and applications in science and engineering are data-intensive and network-intensive and require more expensive networks with higher bandwidth and lower latency. The networking equipment represents about 8%, the servers account for 57%, and the power represents 31% [232] of the monthly costs of a CSP.

[1]Hyperconvergence is a software-centric architecture that tightly integrates compute, storage, networking, virtualization, and possibly other technologies in a commodity hardware box supported by a single vendor.

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00007-8

This chapter is focused on communication and the discussion starts with an overview of the network used to access the cloud, the Internet, a packet-switched network of networks, and the World Wide Web in Sections 5.1, 5.2, and 5.3, followed by a discussion of Named Data Networks and Software Defined Networks in Sections 5.4 and 5.5, respectively. Then the focus is changed to the communication fabric used inside the cloud infrastructure and to the analysis of interconnection networks architecture and algorithms.

After an overview of the interconnection networks in Section 5.6 the chapter covers multistage networks, InfiniBand and Myrinet, and Storage Area Networks in Sections 5.7, 5.8, and 5.9, respectively. A scalable data center architecture and network resource management are the topics of Sections 5.10 and 5.11. Then the limelight changes again, this time to content delivery networks and vehicular networks in Sections 5.12 and 5.13. Further readings, historical notes, and a set of exercises and problems conclude the chapter.

5.1 PACKET-SWITCHED NETWORKS AND THE INTERNET

The Internet, a packet-switched network, provides access to computer clouds. A packet-switched network transports data units called *packets* through a maze of *switches* where packets are queued and routed towards their destination. Packets are subject to random delays, loss, and may arrive at their final destination out of order.

A few basic concepts are defined next. A *datagram* is a transfer unit in a packet-switched network. In addition to its *payload* a datagram has a header containing control information necessary for its transport through the network. A *network architecture* describes the protocol stack used for communication. A *protocol* is a set of rules on how to communicate, it specifies the actions taken by the sender and the receiver of a data unit. A network *host* identifies a system located at the network edge capable to initiate and to receive communication, be it a computer, a mobile device such as a phone, or a sensor.

Network architecture and protocols. A packet-switched network has a *network core* consisting of routers and control systems interconnected by very high bandwidth communication channels and a *network edge* where the end-user systems reside.

A packet-switched network is a complex system consisting of a very large number of autonomous components subject to complex and, sometimes, contradictory requirements. Basic strategies for implementing a complex system are *layering* and *modularization*. Layering means decomposing a complex function into elements interacting through well-defined channels, a layer can only communicate with its adjacent layers.

The protocol stack of the Internet, based on the TCP/IP network architecture is shown in Figure 5.1. At the sending host data flows down the protocol stack from the application layer to the transport layer, then to the network layer, and to the data link layer. The physical layer pushes the streams of bits through a physical communication link encoded either as electrical, optical, or electromagnetic signals. The corresponding data units for the five layer architecture are: messages, segments, packets, frames, and encoded bits, respectively.

The *transport layer* is responsible for end-to-end communication, from an application running on the sending host to its peer, running on the destination host using either TCP or UDP protocols. The network layer decides where the packet should be sent, either to another router, or to a destination

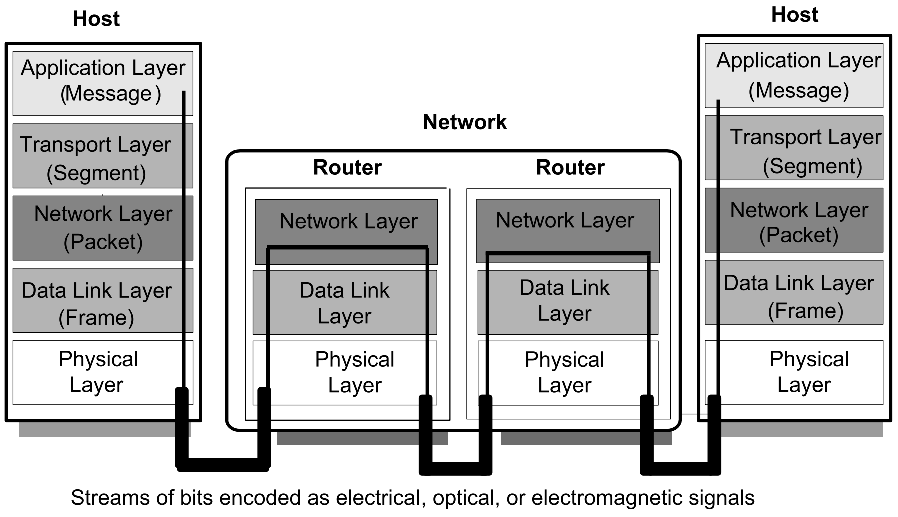


FIGURE 5.1

The Internet protocol stack. Applications running on hosts at the edge of the network communicate using application layer protocols. The transport layer deals with end-to-end delivery. The network layer is responsible for routing a packet through the network. The data link layer ensures reliable communication between adjacent nodes of the network, and the physical layer transports streams of bits encoded as electrical, optical, or electromagnetic signals (the thick lines represent such bit pipes).

host connected to a local area network connected to the router. IP, the *network layer* protocol, guides packets through the packet-switched network from the point of entry to the place where a packet exits the network. The *data link layer* encapsulates the packet for the communication link to the next hop. Once a packet reaches a router the bits are passed to the data link and then to the network layer.

A protocol on one system communicates with its *peer* on another system. For example, the transport protocol on the sender, host A, communicates with the transport protocol on the receiver, host B. On the sending side, A, the transport protocol encapsulates the data from the application layer and adds control information as headers that can only be understood by its peer, the transport layer on host B. When the peer receives the data unit, it carries out a decapsulation, retrieves the control information, removes the headers, then passes the payload to the next layer up, the application layer on host B.

The payload for the data link layer at the sending site includes the network header and the payload at the network layer. In turn, the network layer payload includes transport layer header and its payload consisting of the application layer header and application data.

The Internet. The Internet is a network of networks, a collection of separate, autonomous, and distinct networks. All networks adhere to a common framework and use: (i) globally unique IP addresses; (ii) the IP (Internet Protocol) routing protocol; and (iii) the Border Gateway Routing (BGP) protocol. BGP is a path vector reachability protocol making core routing decisions. BGP maintains a table of IP networks designating network reachability among autonomous systems. BGP makes routing decisions based on path, network policies, and/or rule sets.

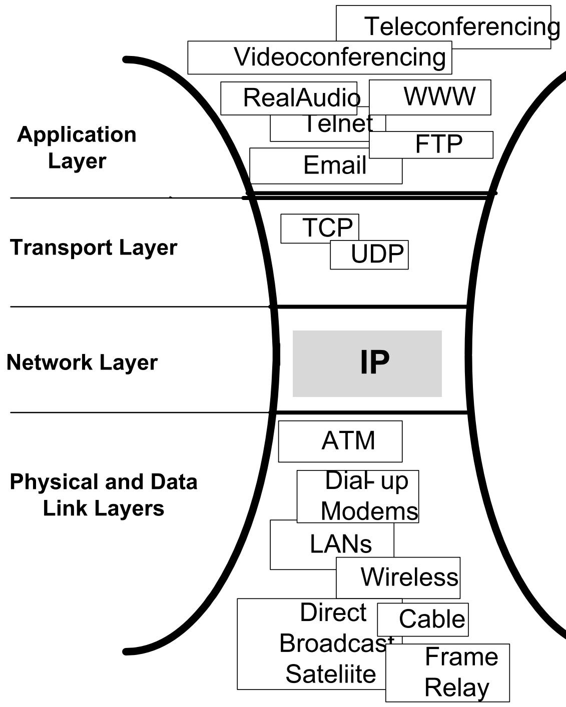


FIGURE 5.2

The hourglass network architecture of the Internet. Regardless of the application, the transport protocol, and the physical network, all packets are routed from the source to the destination using the IP protocol and the IP address of the destination.

An *IP address* is a string of integers uniquely identifying every host connected to the Internet. An IP address allows the network to identify first the destination network and then the host in that network where a datagram should be delivered. A host may have multiple IP addresses and it may be connected to more than one network. A host could be a supercomputer, a workstation, a laptop, a mobile phone, a network printer, or any other physical device with a network interface.

The Internet is based on a hourglass network architecture, see Figure 5.2. *The hourglass architecture is partially responsible for the explosive growth of the Internet*, it allowed the lower layers of the architecture to evolve independently from the upper layers. The communication technology drives the dramatic change of the lower layers of the Internet architecture including the increase of the communication channels bandwidth, the widespread use of wireless networks, and of satellite communication. The software and the applications are the engines of progress for the upper layers of the architecture.

The hourglass model reflects the *end-to-end* architectural design principle. The model captures the fact that all packets transported through the Internet use IP to reach their destination. IP only provides *best effort delivery*. Best effort delivery means that any router along the path from the source to the destination may drop a packet when it is overloaded.

Another important architectural design principle of the Internet is the *separation between the forwarding and the routing planes*. The *forwarding plane* decides what to do with packets arriving on an inbound interface of a router. The plane uses a table to lookup the destination address of an incoming packet, then it retrieves the information to determine the path from the receiving element to the proper outgoing interface(s) through the internal forwarding fabric of the router. The *routing plane* is responsible for building the routing table in each router. The separation of the two planes allowed the forwarding plane to function, while the routing evolved.

In addition to the IP or logical address, each network interface, the hardware connecting a host with a network, has a unique *physical* or *MAC address*. While the MAC address is permanently assigned to a network interface of the device, the IP address may be dynamically assigned. The IP address of a mobile device changes depending on the device location and the network it is connected to.

The Dynamic Host Configuration Protocol (DHCP) is an automatic configuration protocol. DHCP assigns an IP address to a client system. A DHCP server has three methods of allocating IP addresses:

1. Dynamic allocation – a network administrator assigns a range of IP addresses to DHCP. During network initialization each client computer on the LAN is configured to request an IP address from the DHCP server. The request-and-grant process uses a lease concept with a controllable time period, allowing the DHCP server to reclaim (and then reallocate) IP addresses that are not renewed.
2. Automatic allocation – the DHCP server permanently assigns a free IP address to a client, from the range defined by the administrator.
3. Static allocation – the DHCP server allocates an IP address based on a manually filled in table with (MAC address – IP address) pairs. Only a client with a MAC address listed in this table is allocated an IP address.

Once a packet reaches the destination host it is delivered to the proper transport protocol daemon which, in turn, delivers it to the application which listens to an abstraction of the end-point of a logical communication channel called a *port*, Figure 5.3. The processes or threads running an application use an abstraction called *socket* to send and receive data through the network. A socket manages one queue of incoming messages and another one for outgoing messages.

Internet transport protocols. The Internet uses two transport protocols, a connectionless datagram protocol, UDP (User Datagram Protocol) and a connection-oriented protocol, TCP (Transport Control Protocol). The header of a datagram contains information sufficient for routing through the network from the source to the destination. The arrival time and order of delivery of datagrams are not guaranteed.

To ensure efficient communication, the UDP transport protocol assumes that error checking and error correction are either not necessary or performed by the application. Datagrams may arrive out of order, duplicated, or may not arrive at all. Applications using UDP include: the DNS (Domain Name System), VoIP, TFTP (Trivial File Transfer Protocol), streaming media applications such as IPTV, and online games.

TCP provides reliable, ordered delivery of a stream of bytes from an application on one system to its peer on the destination system. An application sends/receives data units called *segments* to/from a specific port, an abstraction of and end-point of a logical communication link. TCP is the transport protocol used by the World Wide Web, email, file transfer, remote administration, and many other important applications.

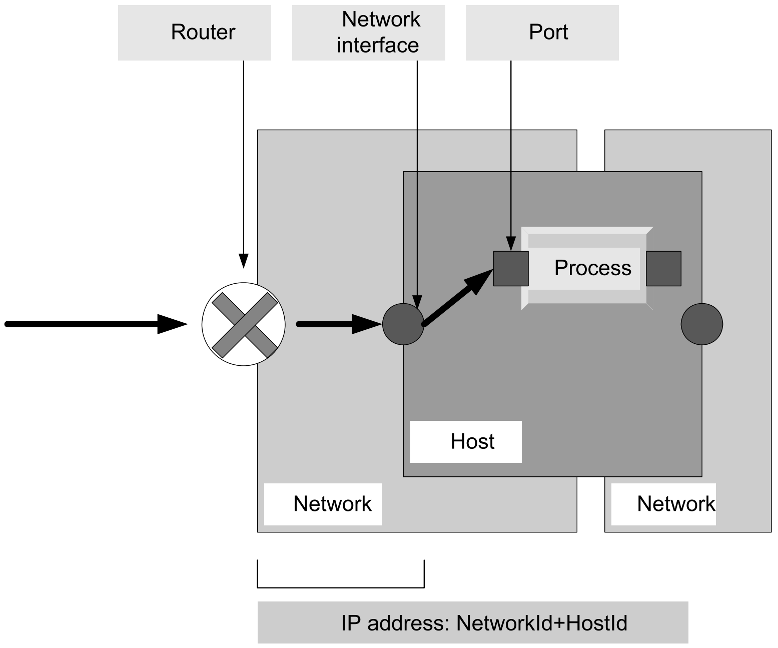


FIGURE 5.3

Packet delivery to processes and threads; a packet is first routed by the IP protocol to the destination network and then to the host specified by the IP address. Applications listen to *ports*, abstractions of the end point of a communication channel.

TCP uses an end-to-end *flow control mechanism* based on a sliding-window, a range of packets the sender can send before receiving an acknowledgment from the receiver. This mechanisms allows the receiver to control the rate of segments sent and process them reliably.

A network has a finite capacity to transport data and when its load is approaching this capacity, we witness undesirable effects, the routers start dropping packets, the delays and the jitter increase. An obvious analogy is a highway where the time to travel from point A to point B increases dramatically in case of congestion. A solution for traffic management is to introduce traffic lights limiting the rate at which new traffic is allowed to enter the highway and this is precisely what the TCP emulates.

TCP uses several mechanisms for *congestion control*, see Section 5.3. These mechanisms control the rate of the data entering the network, keeping the data flow below a rate that would lead to a network collapse and enforcing a fair allocation among flows. Acknowledgments coupled with timers are used to infer network conditions between the sender and receiver.

TCP congestion control policies are based on four algorithms, *slow-start, congestion avoidance, fast retransmit*, and *fast recovery*. These algorithms use local information, such as the RTO (retransmission timeout) based on the estimated RTT (round-trip time) between the sender and receiver, as well as the variance in this round trip time to implement the congestion control policies. UDP is a connectionless protocol thus, there are no means to control the UDP traffic.

The review of basic networking concepts in this section shows why process-to-process communication incurs a significant overhead. While raw speed of fiber optic channels can reach Tbps,[2] the actual transmission rate for end-to-end communication over a wide area network can only be of the order of tens of Mbps and the latency is of the order of milliseconds. This has important consequences for the development of computer clouds. The term "speed" is used informally to describe the maximum data transmission rate, or the capacity of a communication channel; this capacity is determined by the physical bandwidth of the channel and this explains why the term channel "bandwidth" is also used to measure the channel capacity, or the maximum data rate.

5.2 THE TRANSFORMATION OF THE INTERNET

The Internet is continually evolving under the pressure of its own success and the need to accommodate new applications and a larger number of users. Initially conceived as a data network, a network designed to transport data files, the Internet has morphed into today's network supporting data-streaming and applications with real-time constraints such as the Lambda service offered by the AWS. The discussion in this section is restricted to the aspects of the Internet evolution relevant to cloud computing.

Tier 1, 2, and 3 networks. To understand the architectural consequences of Internet evolution we discuss first the relation between two networks. *Peering* means that two networks exchange traffic between each other's customers freely. *Transit* requires a network to pay another one for accessing the Internet. The term *customer* means that a network is receiving money to allow Internet access.

Based on these relations the networks are commonly classified as Tier 1, 2, and 3. A *Tier 1 network* can reach every other network on the Internet without purchasing IP transit or paying settlements; examples of Tire 1 networks are Verizon, ATT, NTT, Deutsche Telecom, see Figure 5.4.

A *Tier 2 network* is an Internet service provider who engages in the practice of peering with other networks, but who still purchases IP transit to reach some portion of the Internet; Tier 2 providers are the most common providers on the Internet. A *Tier 3 network* purchases transit rights from other networks (typically Tier 2 networks) to reach the Internet. A *point-of-presence (POP)* is an access point from one place to the rest of the Internet.

An *Internet exchange point* (IXP) is a physical infrastructure allowing *Internet Service Providers* (ISPs) to exchange Internet traffic. IXPs interconnect networks directly, via the exchange, rather than through one or more third party networks. The advantages of the direct interconnection are numerous, but the primary reasons to implement an IXP are cost, latency, and bandwidth. Traffic passing through an exchange is typically not billed by any party, whereas traffic to an ISP's upstream provider is.

IXPs reduce the portion of an ISP's traffic which must be delivered via their upstream transit providers, thereby reducing the average per-bit delivery cost of their service. Furthermore, the increased number of paths found through the IXP improves routing efficiency and fault-tolerance. A typical IXP consists of one or more network switches, to which each of the participating ISPs connects.

New technologies such as web applications, cloud computing, and content-delivery networks are reshaping the definition of a network as we can see in Figure 5.5 [287]. The World Wide Web, gam-

[2]NTT (Nippon Telegraph and Telephone) achieved a speed of 69.1 Tbps in 2010 using wavelength division multiplexing of 432 wavelengths with a capacity of 171 Gbps over a 240 km-long optical fiber.

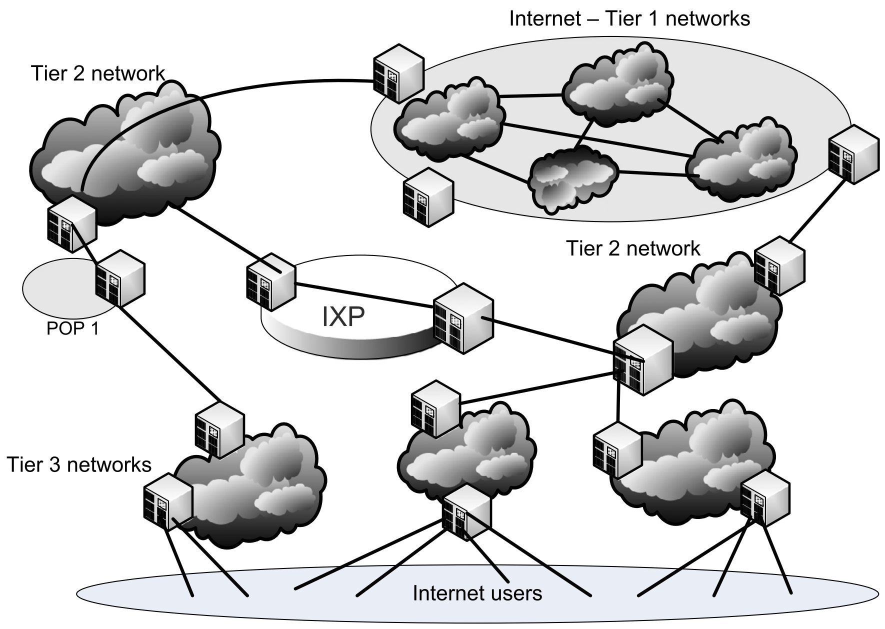


FIGURE 5.4

The relation of Internet networks based on the transit and paying settlements. There are three classes of networks, Tier 1, 2, and 3; an IXP is a physical infrastructure allowing ISPs to exchange Internet traffic.

ing, and entertainment are merging and more computer applications are moving to the cloud. Data streaming consumes an increasingly larger fraction of the available bandwidth as high definition TV sets become less expensive and content providers such as Netflix and Hulu offer customers services that require a significant increase of the network bandwidth.

Does the network infrastructure adequately respond to the current demand for bandwidth? The Internet infrastructure in the US is falling behind in terms of network bandwidth, see Figure 5.6. A natural question to ask is: Where is the actual bottleneck limiting the bandwidth available to a typical Internet broadband user? The answer is: the "last mile," the link connecting the home to the ISP network. Recognizing that the broadband access infrastructure ensures continual growth of the economy and allows people to work from any site, Google has initiated the Google Fiber Project which aims to provide a one Gbps access speed to individual households through FTTH.[3]

Migration to IPv6. The Internet Protocol, Version 4 (IPv4), provides an addressing capability of 2^{32}, or approximately 4.3 billion addresses, a number that proved to be insufficient. Indeed, the Internet Assigned Numbers Authority (IANA) assigned the last batch of 5 address blocks to the Regional In-

[3]The fiber-to-the-home (FTTH) is a broadband network architecture that uses optical fiber to replace the copper-based local loop used for the last mile network access to home.

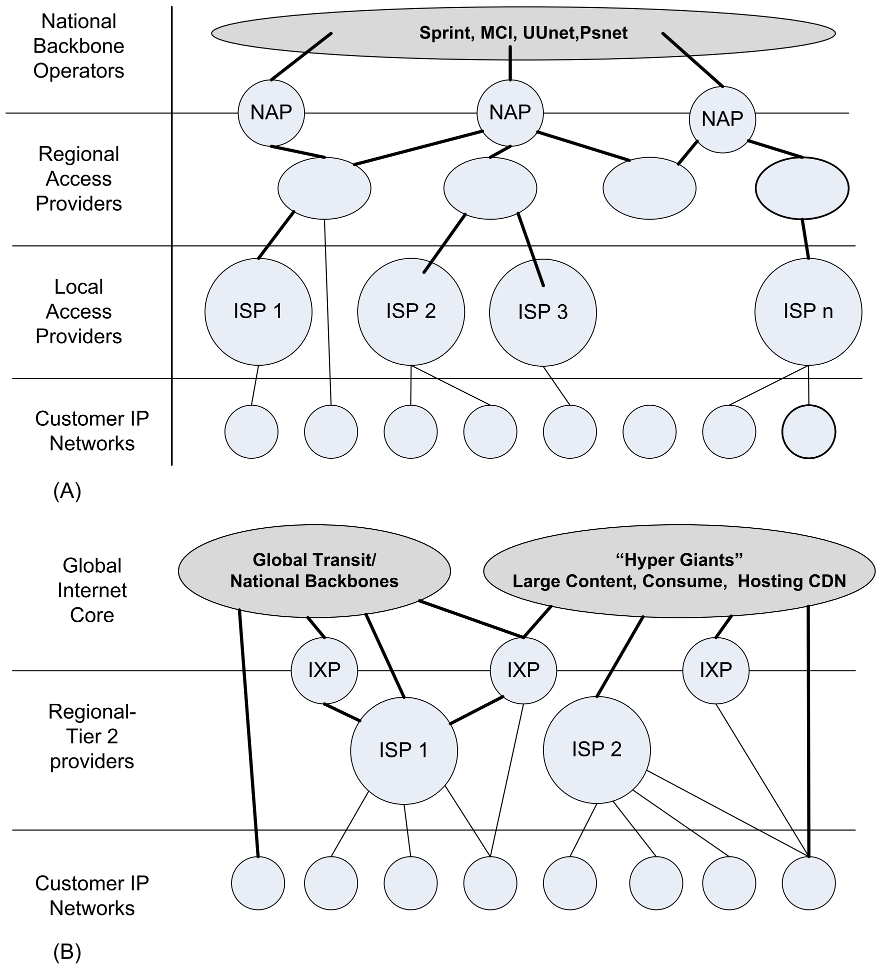

National Backbone Operators
Sprint, MCI, UUnet,Psnet
NAP
NAP
NAP
Regional Access Providers
Local Access Providers
ISP 1
ISP 2
ISP 3
ISP n
Customer IP Networks
(A)
Global Internet Core
Global Transit/ National Backbones
"Hyper Giants" Large Content, Consume, Hosting CDN
IXP
IXP
IXP
Regional-Tier 2 providers
ISP 1
ISP 2
Customer IP Networks
(B)

FIGURE 5.5

The transformation of the Internet; the traffic carried by Tier 3 networks increased from 5.8% in 2007 to 9.4% in 2009; Goggle applications accounted for 5.2% of the traffic in 2009 [287].

ternet Registries in February 2011, officially depleting the global pool of completely fresh blocks of addresses; each of the address blocks represents approximately 16.7 million possible addresses.

The Internet Protocol, Version 6 (IPv6), provides an addressing capability of 2^{128}, or 3.4×10^{38} addresses. There are other major differences between IPv4 and IPv6:

- *Multicasting.* IPv6 does not implement traditional IP broadcast, i.e. the transmission of a packet to all hosts on the attached link using a special broadcast address and, therefore, does not define broadcast addresses. IPv6 supports new multicast solutions, including embedding rendezvous point

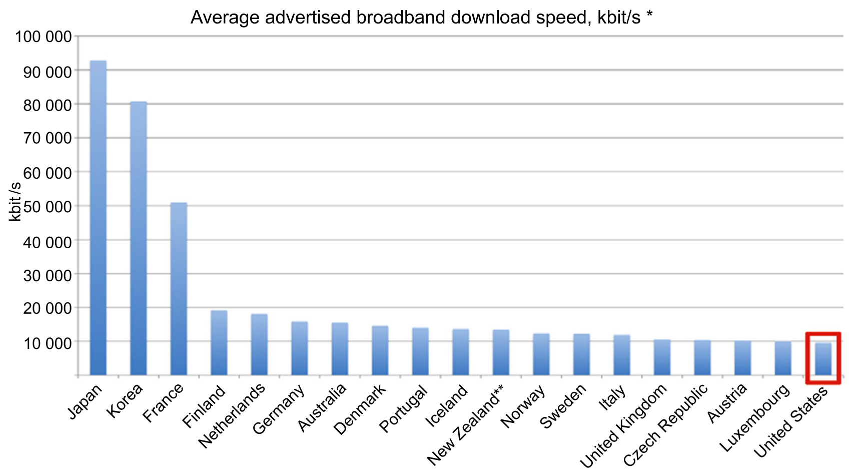


FIGURE 5.6

The broadband access, the average download speed advertised by several countries.

addresses in an IPv6 multicast group address. This solution simplifies the deployment of inter-domain solutions.

- *Stateless address autoconfiguration (SLAAC).* IPv6 hosts can configure themselves automatically when connected to a routed IPv6 network using the Internet Control Message Protocol version 6 (ICMPv6) router discovery messages. When first connected to a network, a host sends a link-local router solicitation multicast request for its configuration parameters. If suitably configured, routers respond to such a request with a router advertisement packet that contains network-layer configuration parameters.
- *Mandatory support for network security.* Internet Network Security (IPsec) is an integral part of the base protocol suite in IPv6 while it is optional for IPv4. IPsec is a protocol suite operating at the IP layer. Each IP packet is authenticated and encrypted. Other security protocols, e.g., the Secure Sockets Layer (SSL), the Transport Layer Security (TLS) and the Secure Shell (SSH) operate at the upper layers of the TCP/IP suite. IPsec uses several protocols: (1) Authentication Header (AH) supports connectionless integrity, data origin authentication for IP datagrams, and protection against replay attacks; (2) Encapsulating Security Payload (ESP) supports confidentiality, data-origin authentication, connectionless integrity, an anti-replay service, and limited traffic-flow confidentiality; (3) Security Association (SA) provides the parameters necessary to operate the AH and/or ESP operations.

Unfortunately, migration to IPv6 is a very challenging and costly proposition [115]. A simple analogy allows us to explain the difficulties related to migration to IPv6. The telephone numbers in North America consist of 10 decimal digits. This scheme supports up to 10 billion phones but, in practice, we have fewer available numbers. Indeed, some phone numbers are wasted because we use area codes based on geographic proximity and, on the other hand not all available numbers in a given area are allocated.

Table 5.1 Web statistics collected from a sample of several billion pages detected during Google's crawl and indexing pipeline.

Metric	Value
Number of sample pages analyzed	4.2×10^9
Average number of resources per page	44
Average number of GETs per page	44.5
Average number of unique host names encountered per page	7
Average size transferred over the network per page, including HTTP headers	320 KB
Average number of unique images per page.	29
Average size of the images per page	206 KB
Average number of external scripts per page	7
Number of sample SSL (HTTPS) pages analyzed	17×10^6

To overcome the limited number of phone numbers in this scheme, large organizations use private phone extensions that are typically 3 to 5 digits long; thus, a single public phone number can translate to 1000 phones for an organization using a 3 digit extension. Analogously, Network Address Translation (NAT) allow a single public IP address to support hundreds or even thousands of private IP address. In the past NAT did not work well with applications such as VoIP (Voice over IP) and VPN (Virtual Private Network). Nowadays Skype and STUN VoIP applications work well with NAT. Now NAT-T and SSLVPN support VPN NAT.

If the telephone companies decide to promote a new system based on 40 decimal digit phone numbers we will need new telephones. At the same time we will need new phone books, much thicker as each phone number is 40 characters instead of 10, each individual needs a new personal address book, and virtually all the communication and switching equipment and software need to be updated. Similarly, the IPv6 migration involves upgrading all applications, hosts, routers, and DNS infrastructure; also, moving to IPv6 requires backward compatibility, any organization migrating to IPv6 should maintain a complete IPv4 infrastructure.

5.3 WEB ACCESS AND THE TCP CONGESTION CONTROL WINDOW

The web supports access to content stored on a cloud. Virtually all cloud computing infrastructures allow users to interact with their computations on the cloud using web-based systems. It should be clear that the metrics related to web access are important for designing and tuning the networks. The site http://code.google.com/speed/articles/web-metrics.html provides statistics about metrics such as the size and the number of resources and Table 5.1 summarizes these metrics. The statistics are collected from a sample of several billion pages detected during Google's crawl and indexing pipeline.

Such statistics are useful to tuning the transport protocols to deliver optimal performance in terms of latency and throughput. Metrics, such as the average size of a page, the number of GET operations, are useful to explain the results of performance measurements carried out on existing systems and to propose changes to optimize the performance as discussed next. HTTP, the application protocol for web browsers uses TCP and takes advantage of its congestion control mechanisms.

TCP flow control and congestion control. To achieve high channel utilization, avoid congestion, and, at the same time, ensure a fair sharing of the network bandwidth TCP uses two mechanisms, flow control and congestion control. A *flow control* mechanism throttles the sender, feedback from the receiver forces the sender to transmit only the amount of data the receiver is able to buffer and then process. TCP uses a sliding window flow control protocol. If W is the window size, then the data rate S of the sender is:

$$S = \frac{W \times MSS}{RTT} \text{ bps} = \frac{W}{RTT} \text{ packets/second} \tag{5.1}$$

where MSS and RTT denote the Maximum Segment Size and the Round Trip Time, respectively, assuming that MMS is 1 packet. If S is too small the transmission rate is smaller than the channel capacity, while a large S leads to congestion. The channel capacity depends on the network load as the physical channels along the path of a flow are shared with many other Internet flows.

The actual window size W is affected by two factors: (a) the ability of the receiver to accept new data and (b) the sender's estimation of the available network capacity. The receiver specifies the amount of additional data it is willing to accept in the *receive window* field of every frame. The receiver's window shifts when the receiver receives and acknowledges a new segment of data. When a receiver advertises a window size of zero, the sender stops sending data and starts the persist timer. This timer is used to avoid the deadlock when a subsequent window size update from the receiver is lost.

When the persist timer expires, the sender sends a small packet and the receiver responds by sending another acknowledgment containing the new window size. In addition to the flow control provided by the receiver, the sender attempts to infer the available network capacity and to avoid overloading the network. The source uses the losses and the delay to determine the level of congestion. If $awnd$ denotes the receiver window and $cwnd$ the congestion window set by the sender, the actual window should be:

$$W = \min(cwnd, awnd). \tag{5.2}$$

Several algorithms are used to calculate $cwnd$ including Tahoe and Reno, developed by Van Jacobson in 1988 and 1990. Tahoe was based on slow start (SS), congestion avoidance (CA), and fast retransmit (FR). The sender probes the network for spare capacity and detects congestion based on loss. The slow start means that the sender starts with a window of two times MSS, $init\_cwnd = 1$. For every packet acknowledged, the congestion window increases by 1 MSS so that the congestion window effectively doubles for every RTT.

When the congestion window exceeds the threshold, $cwnd \geq ssthresh$, the algorithm enters the CA state. In CA state, on each successful acknowledgment $cwnd \leftarrow cwnd + 1/cwnd$ and on each RTT $cwnd \leftarrow cwnd + 1$. The fast retransmit is motivated by the fact that the timeout is too long thus, a sender retransmits immediately after 3 duplicate acknowledgments without waiting for a timeout. Two adjustments are made in this case:

$$\mathit{flightsize} = \min(awnd, cwnd) \quad \text{and} \quad \mathit{ssthresh} \leftarrow \max(\mathit{flightsize}/2, 2) \tag{5.3}$$

and the system enters in the slow start state, $cwnd = 1$. The pseudocode describing the Tahoe algorithm is:

```
for every ACK {
        if (W < ssthresh) then W++        (SS)
        else    W += 1/W                  (CA)
    }
    for every loss {
        ssthresh = W/2
            W   = 1
    }
```

The pattern of usage of the Internet has changed. Measurements reported by different sources [156] show that in 2009 the average bandwidth of an Internet connection was 1.7 Mbps. More than 50% of the traffic required more than 2 Mbps and could be considered broadband, while only about 5% of the flows required less that 256 Kbps and could be considered narrowband. Recall that the average web page size is in the range of 384 KB.

While the majority of the Internet traffic is due to long-lived, bulk data transfer, e.g., video and audio streaming, the majority of transactions are short-lived, e.g., web requests. So a major challenge is to ensure some fairness for short-lived transactions.

To overcome the limitations of the slow start, application strategies have been developed to reduce the time to download data over the Internet. For example, two browsers, Firefox 3 and Google Chrome open up to six TCP connections per domain to increase the parallelism and to boost start-up performance when downloading a web page. The Internet Explorer 8 opens 180 connections. Clearly, these strategies circumvent the mechanisms for congestion control and incur a considerable overhead. It is argued that a better solution is to increase the initial congestion window of TCP and the arguments presented in [156] are:

- The TCP latency is dominated by the number of RTT's during the slow start phase. Increasing the $init\_cwnd$ parameter allows the data transfer to be completed with fewer RTT's.
- Given that the average page size is 384 KB, a single TCP connection requires multiple RTT's to download a single page.
- It ensures fairness between short-lived transactions which are a majority of Internet transfers and the long-lived transactions which transfer very large amounts of data.
- It allows faster recovery after losses through Fast Retransmission.

It can be shown that the latency of a transfer completing during the slow start without losses is given by the expression

$$\left\lceil \log_{\gamma}\left(\frac{L(\gamma-1)}{init\_cwnd}+1\right)\right\rceil \times RTT + \frac{L}{C} \tag{5.4}$$

with L the transfer size, C the bottleneck-link rate, and γ a constant equal to 1.5 or 2 depending if the acknowledgments are delayed or not; $L/init\_cwnd \geq 1$. In the experiments reported in [156] the TCP latency was reduced from about 490 msec when $init\_cwnd = 3$ to about 466 msec for $init\_cwnd = 16$.

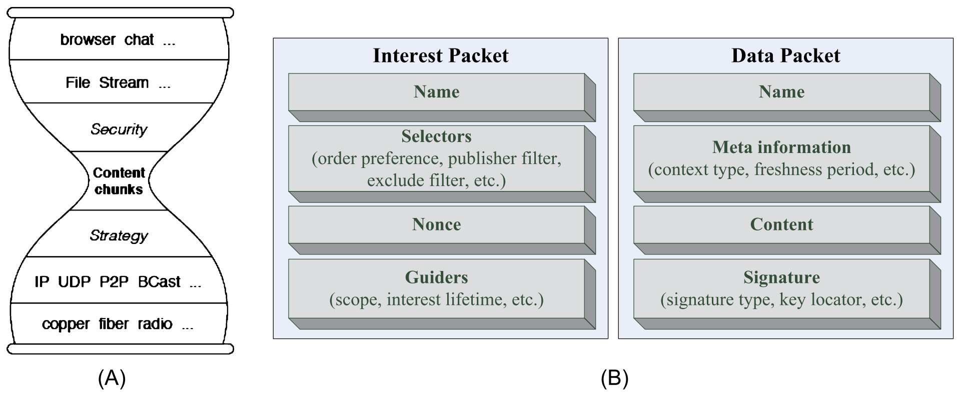

browser chat ...
File Stream ...
Security
Content chunks
Strategy
IP UDP P2P BCast ...
copper fiber radio ...
(A)
Interest Packet
Name
Selectors (order preference, publisher filter, exclude filter, etc.)
Nonce
Guiders (scope, interest lifetime, etc.)
Data Packet
Name
Meta information (context type, freshness period, etc.)
Content
Signature (signature type, key locator, etc.)
(B)

FIGURE 5.7

(A) The NDN hourglass architecture parallels the one of the Internet, it separates the lower layers of the protocol stack from the upper ones thus, naming of the data can evolve independently from networking. (B) The semantics of the NDN networking service is to fetch a data chunk identified by name, while the Internet semantics is to deliver a packet to a given network address through an end-to-end channel identified by the source and the destination IP addresses. Two packet types, Interest and Data, see http://named-data.net/doc/ndn-tlv/ are used for NDN routing and forwarding.

5.4 NAMED DATA NETWORKS

The Internet is a network of *communication networks* where the communication units, the packets, only name the communication end points. Today's Internet is mostly used as a *distribution network* by applications in areas such as digital media, electronic commerce, Big Data analytics, and so on.

In a distribution network communication is *content-centric*, named objects are requested by an agent and, once a site holding the object is located, the network transports it to the site that requested the object. The end user in a distribution network is oblivious to the location of the data and it is only interested in the content. The data in today's communication networks is tied to a particular host and this makes data replication and migration difficult.

The idea of a content-centric network has been around for some time. In 1999 the TRIAD project at Stanford[4] [205,487] proposed to use the name of an object to route towards a replica of it. In this proposal the Internet Relay Protocol performs name-to-address conversion using the routing information maintained by relay nodes. The Name-Based Routing Protocol performs a function similar to the BGP protocol, it supports a mechanism for updating the routing information in relay nodes.

In 2006 the Data Oriented Network Architecture (DONA) project at U.C. Berkeley [150] extended TRIAD incorporating security and persistence as primitives of the new network architecture. DONA architecture exposes two primitives: FIND – allows a client to request a particular piece of data by its name and REGISTER – enables content providers to indicate their intent to serve a particular data

[4]TRIAD stands for Translating Relaying Internet Architecture Integrating Active Directories.

object. In 2006 Van Jacobson from UCLA argued that Named Data Networks (NDNs) should be the architecture of the future Internet.

In 2012 the Internet Research Task Force (IRTF) established an information-centric networking (ICN) research working group for investigating architecture designs for NDN. A 2014 survey of ICN research and a succinct presentation of NDNs can be found in [532] and [548], respectively. The important features of the NDN architecture addressed by the current research efforts include namespaces, trust models, in-network storage, data synchronization, and last but not least, rendezvous, discovery, and bootstrapping.

The hourglass model can be extended, the packets can name objects rather than communication endpoints. The NDNs hourglass model separates the lower layers of the protocol stack from the upper ones thus, data naming can evolve independently from networking, see Figure 5.7A.

The NDN packet delivery is driven by the data consumers, the communication is initiated by an agent generating an *Interest* packet containing the name of the data, see Figure 5.7B. Once the Interest packet reaches a network host which has a copy of the data item, a *Data* packet containing the name, the data contents, and the signature is generated. The signature consists of the producer's key. The *Data* packet follows the route traced by the *Interest* packet and it is delivered to the data consumer agent. An NDN router maintains the information necessary to forward the packet:

1. *Content Store* – local cache for *Data* packets previously crossing the router. When an Interest packet arrives, a search of the content store determines if a matching data exists and if so the data is forwarded on the same router interface the Interest packet was received. If not, the router uses a data structure, the Forwarding Information Base, to forward the packet. More recently, NDN supports more persistent and larger-volume in-network storage, called Repositories providing services similar to that of the Content Delivery Networks.
2. *Forwarding Information Base* – the entries of this data structure are populated by a name-prefix based procedure. The Forwarding Strategy retrieves the longest prefix matched entry from forwarding information base for an Interest packet.
3. *Pending Interest Table* (PIT) – stores all the Interest packets the router has forwarded but have not been satisfied yet. A PIT entry records the data name carried in the Internet, together with its incoming and outgoing router interface(s). When a Data packet arrives, the router finds the matching PIT entry and forwards the data to all downstream interfaces listed in that PIT entry, then removes the PIT entry, and caches the Data packet in the Content Store.

Names are essential in NDN, though namespace management is not part of the NDN architecture. The scope and contexts of NDN names are different. Globally accessible data must have globally unique names, while local names require only local routing and must only be locally unique. There is no namespace exhaustion in NDN, while migration to IPv6 has become a necessity due to the IP address limitations of IPv4.

There are other important differences between NDN and TCP/IP. Each *Data* packet is cryptographically signed thus the system supports data-centric security, while TCP/IP security is left to the communication endpoints. An NDN router announces name prefixes for the data it is willing to serve, while an IP router announces IP prefixes.

NDN is a universal overlay,[5] it can run over any datagram network and, conversely, any datagram network, including IP, can run over NDN. Classical algorithms such as Open Shortest Path First (OSPF) route data chunks using component-wise longest prefix match of the name in an Interest packet with the FIB entries of a router.

Wide area applications can operate over IP tunnels and islands of NDN nodes could be interconnected by tunneling over non-NDN clouds. Scalable forwarding along with robust and effective trust management are critical challenges for the future of NDN networks. Namespace design is at the heart of NDNs as it involves application data, communication, storage models, routing, and security.

NDN could be useful for cloud data centers and in particular for those supporting the IaaS cloud delivery model. Data is replicated and multiple instances could access data concurrently from their nearest storage servers. This would be useful for some MapReduce applications but it would require major changes in the frameworks supporting this paradigm. On the other hand, many applications cache data in the server memory and they would only marginally benefit from the NDN support.

5.5 SOFTWARE DEFINED NETWORKS

Software-defined Networks (SDN) extend basic principles of resource virtualization to networking to allow programmatic control of communication networks. SDNs introduce an abstraction layer separating network configuration from the physical communication resources. More precisely, a network operating system running inside a control layer, sandwiched between the application layer and the infrastructure layer, allows applications to re-configure dynamically the communication substrate to adapt to their security, scalability, and manageability needs.

Though enthusiastically embraced by networking vendors, the SDN technology is largely conceptual and there is little agreement either on the architecture, the APIs, or the overlay networks among vendors. A 2012 white paper of the Open Network Foundation (ONF) (https://www.opennetworking.org/sdn-resources) promoted OpenFlow-based SDNs. According to ONF a new network architecture is necessary for several reasons including changing traffic patterns due to several factors such as the rise of cloud services and the need for more bandwidth demanded by Big Data applications. The ONF architecture includes several components:

- Applications – generate network requirements to controllers.
- Controllers – translate application requirements to SDN datapaths and provide the applications with an abstract view of the network.
- Datapaths – logical network devices which expose control to the physical substrate; consist of agents and forwarding engines.
- Control-to-Data-Plane Interfaces – the interfaces between SDN controllers and SDN datapaths.
- Northbound Interfaces – interfaces between applications and controllers.
- Interface Drivers and Agents – driver-agent pairs.
- Management and Administration.

[5] An overlay network is a network built on top of another physical network, its nodes are connected by virtual links, each of which corresponds to a path, possibly through many physical links, in the underlying network.

OpenFlow is an API for programming data plane switches. The data path and the control paths of an OpenFlow switch consist of a flow table including an action associated with each flow entry and a controller which programs the flow entry. The controller configures and manages the switch and receives events from the switch.

5.6 INTERCONNECTION NETWORKS FOR COMPUTER CLOUDS

Interconnection networks for multiprocessor systems, supercomputers, and cloud computing are discussed in the next sections. Computing and communication are deeply intertwined as we have seen in Chapters 3 and 4 and interconnection networks are critical for the performance of computer clouds and supercomputers.

Several concepts important for understanding interconnection networks are introduced next. A network consists of nodes and links or communication channels. The *degree* of a node is the number of links the node is connected to. The *nodes* of a interconnection network could be processors, memory units, or servers. The *network interface* of a node is the hardware connecting it to the network.

Switches and *communication channels* are the elements of the interconnection fabric. Switches receive data packets, look inside each packet to identify the destination IP addresses, then use the routing tables to forward the packet to the next hop towards its final destination. An *n-way switch* has n ports that can be connected to n communication links. An interconnection network can be *non-blocking* if it is possible to connect any permutation of sources and destinations at any time. An interconnection network is *blocking* if this requirement is not satisfied.

While processor and memory technology have followed Moore's Law, interconnection networks have evolved at a slower pace and have become a major factor in determining the overall performance and cost of the system. For example, from 1997 to 2010 the speed of the ubiquitous Ethernet network has increased from 1 to 100 Gbps. This increase is slightly slower than the Moore's Law for traffic [354] which predicted, 1 Tbps Ethernet by 2013.

Interconnection networks are distinguished by their topology, routing, and flow control. *Network topology* is determined by the way nodes are interconnected, routing decides how a message gets from its source to destination, and the flow control negotiates how the buffer space is allocated. There are two basic types of network topologies:

- Static networks where there are direct connections between servers;
- Switched networks where switches are used to interconnect the servers.

The topology of an interconnection network determines the *network diameter*, the average distance between all pairs of nodes, the *bisection width*, the minimum number of links cut to partition the network into two halves, the bisection bandwidth, as well as the cost and the power consumption [271]. When a network is partitioned into two networks of the same size the bisection bandwidth measures the communication bandwidth between the two.

The *full bisection bandwidth* allows one half of the network nodes to communicate simultaneously with the other half of the nodes. Assume that half of the nodes inject data into the network at a rate B Mbps. When the bisection bandwidth is B then the network has full bisection bandwidth. The switching fabric must have sufficient bi-directional bandwidth for cloud computing.

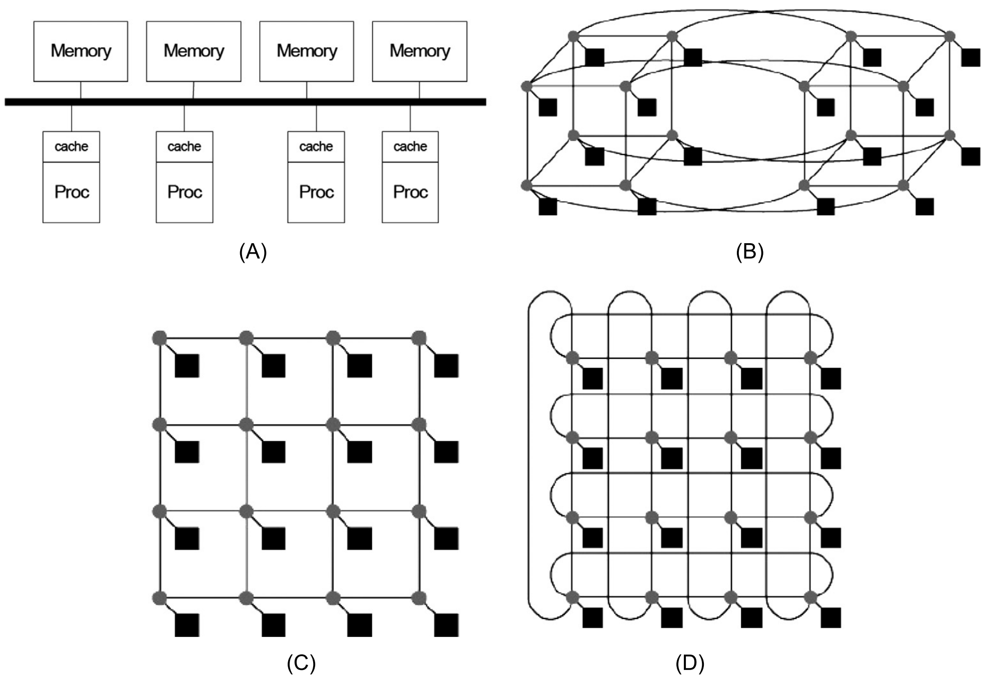


FIGURE 5.8

Static interconnection networks. (A) Bus; (B) Hypercube; (C) 2D-mesh: (D) 2D-torus.

Some of the most popular topologies with a static interconnect are:

- Bus, a simple and cost-effective network, see Figure 5.8A. It does not scale, but it is easy to implement cache coherence through snooping for distributed memory systems. A bus is often used in shared memory multiprocessor systems.
- Hypercube of order n, see Figure 5.8B. A hypercube has a good bisection bandwidth, the number of nodes is $N = 2^n$, the degree is $n = \log N$, and the average distance between nodes is $\mathcal{O}(N)$ hops. Example of use: SGI Origin 2000.
- 2-D mesh, see Figure 5.8C. An $n \times n$ 2D-mesh has many paths to connect nodes, has a cost of $\mathcal{O}(n)$, and the average latency is $\mathcal{O}(\sqrt{n})$. A mesh is not symmetric on edges thus, its performance is sensitive to the placement of communicating nodes on edges versus the middle. Example of use: Intel Paragon supercomputer of the 1990s.
- Torus, avoids the asymmetry of the mesh, but has a higher cost in terms of the number of components. A torus is good for applications using nearest-neighbor communication, see Figure 5.8D. It is prevalent for proprietary interconnects. Example of use: 6-D Mesh/torus of Fujitsu K supercomputer.

Switched networks have multiple layers of switches connecting the nodes as shown in Figure 5.9:

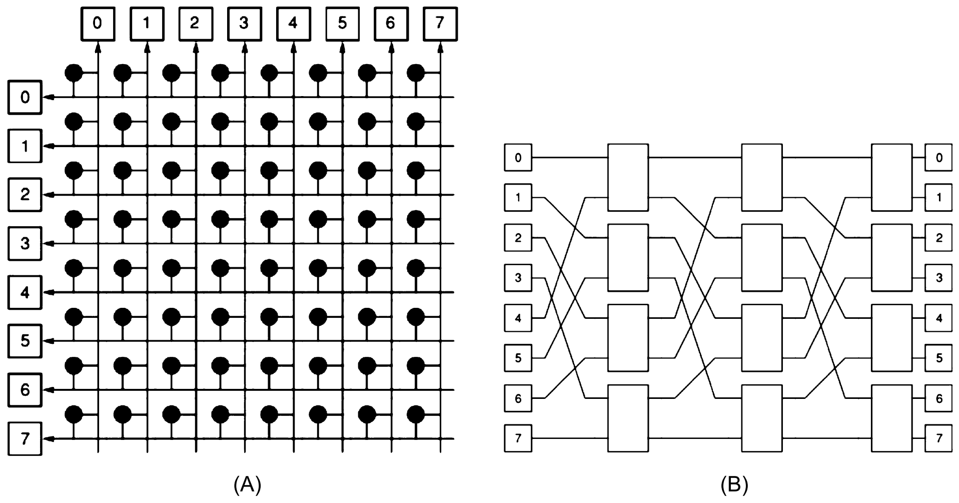


FIGURE 5.9

Switched networks. (A) An 8 × 8 crossbar switch. 16 nodes are interconnected by 49 switches represented by the dark circles; (B) An 8 × 8 Omega switch. 16 nodes are interconnected by 12 switches represented by white rectangles.

- Crossbar switch – has N^2 crosspoint switches, see Figure 5.9A
- Omega (Butterfly, Benes, Banyan, etc.) have $(N \log N)/2$ switches, see Figure 5.9B. The cost is $\mathcal{O}(N \log N)$ and the latency is $\mathcal{O}(\log N)$. Omega networks are low diameter networks.

Cloud interconnection networks. The cloud infrastructure consists of one or more Warehouse Scale Computers (WSCs) discussed in Section 8.2. A WSC has a hierarchical organization with a large number of servers interconnected by high-speed networks.

The networking infrastructure of a cloud must satisfy several requirements including scalability, cost, and performance. The network should allow low-latency, high speed communication, and, at the same time, provide *location transparent communication* between servers. In other words, each server should be able to communicate with every other server with similar speed and latency. This requirement ensures that *applications need not be location aware* and, at the same time, it reduces the complexity of the system management.

Typically, the networking infrastructure is organized hierarchically. The servers of a WSC are packed into racks and interconnected by a rack router. Then rack routers are connected to cluster routers which in turn are interconnected by a local communication fabric. Finally, inter data center networks connect multiple WSCs [277]. Clearly, in a hierarchical organization true location transparency is not feasible. Cost considerations ultimately decide the actual organization and performance of the communication fabric.

Oversubscription is a particularly useful measure of the fitness of an interconnection network for a large scale cluster. Oversubscription is defined as the ratio of the worst-case achievable aggregate bandwidth among the servers to the total bisection bandwidth of the interconnect. An oversubscription

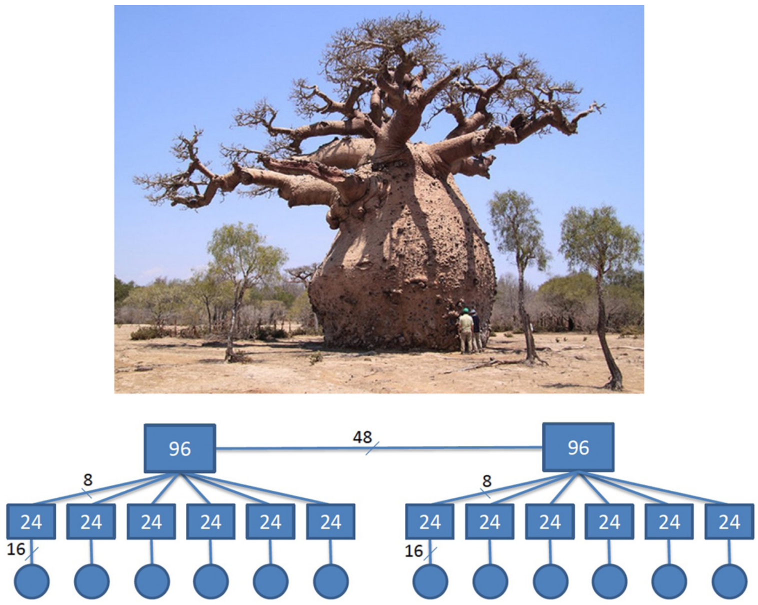


FIGURE 5.10

Fat-trees. (Top) A fat-tree in nature. (Bottom) A 192 node fat-tree interconnection network with two 96-way and twelve 24-way switches in a computer cloud.

of one to one indicates that any host may communicate with an arbitrary hosts at the full bandwidth of the interconnect. An oversubscription value of 4 to 1 means that only 25% of the bandwidth available to servers can be attained for some communication patterns. Typical oversubscription figures are in the 2.5 to 1 and 8 to 1 range.

The cost of the routers and the number of cables interconnecting the routers are major components of the overall cost of an interconnection network. Wire density has scaled up at a slower rate than processor speed and wire delay has remained constant over time thus, better performance and lower costs can only be achieved with innovative router architecture. This motivates the need to take a closer look at the actual design of routers.

Fat-trees. A special instance of the Clos topology discussed in Section 5.7, fat-trees, are optimal interconnects for large-scale clusters and, by extension, for WSCs. When using a fat-tree interconnect servers are placed at the leafs of the tree, while switches populate the root and the internal nodes of the tree. Fat-trees have additional links to increase the bandwidth near the root of the tree. Some set of paths in a fat-tree will saturate all bandwidth available to the end hosts for arbitrary communication patterns. A fat-tree communication architecture can be built with cheap commodity parts as all switching elements of a fat-tree are identical.

Figure 5.10 shows a 192 node fat-tree built with two 96-way switches and twelve 24-way switches. The two 96-way switches at the root are connected via 48 links. Each 24-way switch has 6 $\times$ 8 up-

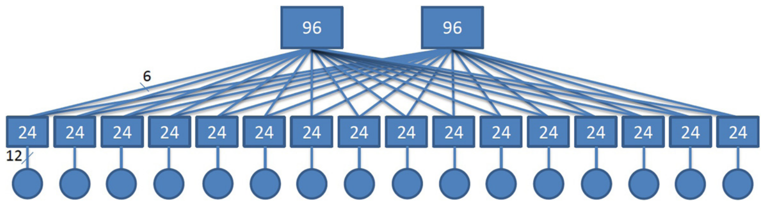


FIGURE 5.11

A 192 node fat-tree interconnect with two 96-way and twelve 24-way switches.

Table 5.2 A side-by-side comparison of performance and cost figures of several interconnection network topologies for 64 nodes.

Property	2D mesh	2D torus	Hypercube	Fat-tree	Fully connected
BW in # of links	8	16	32	32	1024
Max/Avg hop count	14/7	8/4	6/3	11/9	1/1
I/O ports per switch	5	5	7	4	64
Number of switches	64	64	64	192	64
Total number of links	176	192	256	384	2080

link connections to the root and 6 × 16 down connections to 16 servers. Another 192 node fat-tree interconnect with two 96-way and twelve 24-way switches is shown in Figure 5.11.

Table 5.2 from [228] summarizes the performance/cost for a 2D-mesh, a 2D-torus, a Hypercube of order 7, a fat-tree, and a fully connected network. Two dimensions of interconnection network performance, the bisection bandwidth and the average and maximum number of hops, along with three elements affecting the cost, the number of ports per switch, the number of switches, and the total number of links are shown. The fat-tree has the largest bisection bandwidth with the smallest number of I/O ports per switch while the fully connected interconnect has a prohibitively large number of links.

5.7 MULTISTAGE INTERCONNECTION NETWORKS

There are *low-radix* and *high-radix* routers. The former have a small number of ports, while the latter have a large number of ports. The number of intermediate routers in a high-radix network is greatly reduced. Such networks enjoy lower latency and reduced power consumption.

Every five years during the past two decades the pin bandwidth of the chips used for switching has increased by approximately an order of magnitude as a result of the increase in the signaling rate and in the number of signals. High-radix chips divide the bandwidth into a larger number of narrow ports while low-radix chips divide the bandwidth into a smaller number of wide ports.

Clos networks. Clos networks were invented in early 1950s by Charles Clos from Bell Labs [112]. A Clos network is a multistage non-blocking network with an odd number of stages, see Figure 5.12A.

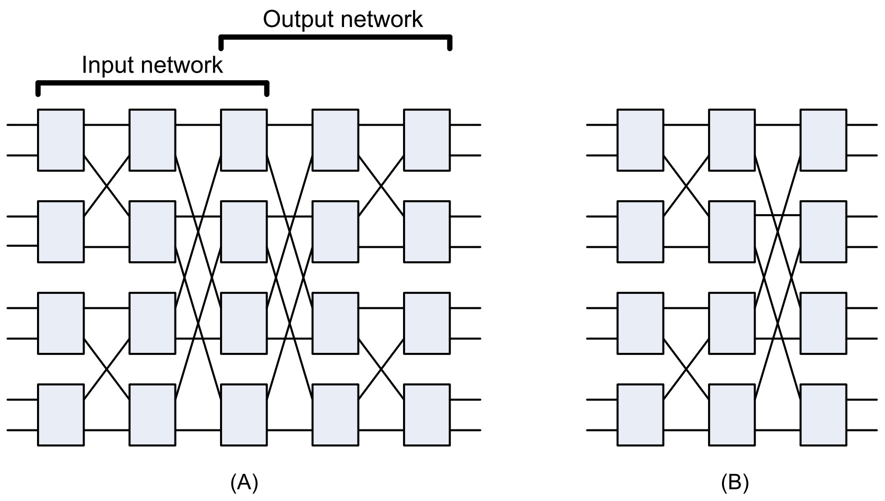


FIGURE 5.12

(A) A 5-stage Clos network with radix-2 routers and unidirectional channels; the network is equivalent to two back-to-back butterfly networks. (B) The corresponding folded-Clos network with bidirectional channels; the input and the output networks share switch modules.

The network consists of two butterfly networks where the last stage of the input is fused with the first stage of the output.

The name butterfly comes from the pattern of inverted triangles created by the interconnections, which look like butterfly wings. A butterfly network transfers the data using the most efficient route, but it is blocking, it cannot handle a conflict between two packets attempting to reach the same port at the same time. In a Clos network all packets overshoot their destination and then hop back to it. Most of the time the overshoot is not necessary and increases the latency, a packet takes twice as many hops as it really needs.

In a *folded Clos* topology the input and the output networks share switch modules. Such networks are sometimes called *fat-tree*; many commercial high-performance interconnects such as Myrinet, InfiniBand, and Quadrics implement a fat-tree topology. Some folded Clos networks use low-radix routers, e.g., the Cray XD1 uses radix-24 routers. The latency and the cost of the network can be lowered using high-radix routers.

The *Black Widow* topology extends the folded Clos topology and has a lower cost and latency; it adds side links and this permits a statical partitioning of the global bandwidth among peer subtrees [447]. The Black Widow topology is used in Cray computers.

Flattened butterfly networks. The flattened butterfly topology [270] is similar to the generalized hypercube that was proposed in the early 1980s, but the wiring complexity is reduced and this topology is able to exploit high-radix routers. When constructing a flattened butterfly we start with a conventional butterfly and combine the switches in each row into a single, higher-radix one. Each router is linked to more processors and this halves the number of router-to-router connections.

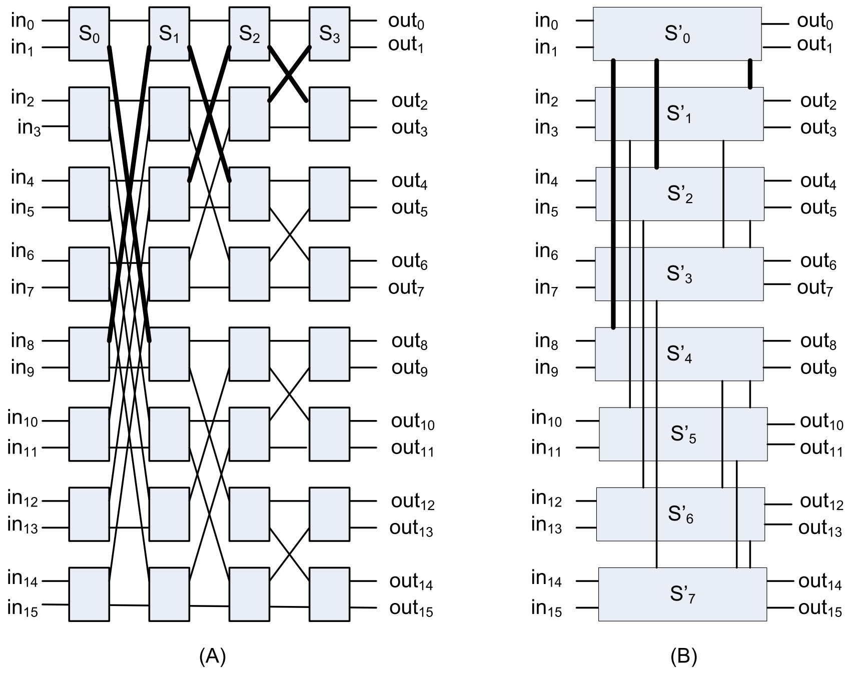


FIGURE 5.13

(A) A 2-ary 4-fly butterfly with unidirectional links. (B) The corresponding 2-ary 4-flat flattened butterfly is obtained by combining the four switches S_0, S_1, S_2 and S_3 in the first row of the traditional butterfly into a single switch S_0' and by adding additional connections between switches [270].

The latency is reduced as data from one processor can reach another processor with fewer hops, though the physical path may be longer. For example, in Figure 5.13A we see a 2-ary 4-fly butterfly; we combine the four switches S_0, S_1, S_2 and S_3 in the first row into a single switch S_0'. The flattened butterfly adaptively senses congestion and overshoots only when it needs to. On adversarial traffic pattern, the flattened butterfly has a similar performance as the folded Clos, but provides over an order of magnitude increase in performance compared to the conventional butterfly.

The network cost for computer clusters can be reduced by a factor of two when high-radix routers (radix-64 or higher) and the flattened butterfly topology are used according to [271]. The flattened butterfly does not reduce the number of local cables, e.g., backplane wires from the processors to routers, but it reduces the number of global cables. The cost of the cables represents as much as 80% of the total network cost, e.g., for a 4K system the cost savings of the flattened buttery exceed 50%.

5.8 INFINIBAND AND MYRINET

InfiniBand is an interconnection network used by high-performance computing systems, as well as computer clouds. As of 2014 InfiniBand was the most commonly used interconnect in supercomput-

Table 5.3 The evolution of the speed of several high-speed interconnects including SDR, DDR, QDR, FDR, and EDR data rates of InfiniBand.

Network	Year	Speed
Gigabit Ethernet (GE)	1995	1 Gbps
10-GE	2001	10 Gbps
40-GE	2010	40 Gbps
Myrinet	1993	1 Gbps
Fiber Channel	1994	1 Gbps
InfiniBand (IB)	2001	2 Gbps (1X SDR)
	2003	8 Gbps (4X SDR)
	2005	16 Gbps (4X DDR) & 24 Gbps (12X SDR)
	2007	32 Gbps (4X QDR)
	2011	56 Gbps (4X FDR)
	2012	100 Gbps (4X EDR)

ers. The architecture of InfiniBand is based on a switched fabric rather than a shared communication channel. In a shared channel architecture all devices share the same bandwidth; the higher the number of devices connected to the channel, the less bandwidth is available to each one of them.

InfiniBand has a very high throughput and very low latency and it is backed by top companies in the industry, including Dell, HP, IBM, Intel, and Microsoft. Intel manufactures InfiniBand host bus adapters and network switches. From Section 5.7 we know that a switched fabric is fault-tolerant and scalable. Every link of the fabric has exactly one device connected at each end of the link thus, the worst case is the same as the typical case. It follows that the performance of InfiniBand can be much greater than that of a shared communication channel such as the Ethernet.

InfiniBand architecture implements the “Bandwidth-out-of-the-box” concept and delivers bandwidth traditionally trapped inside a server across the interconnect fabric. InfiniBand supports several signaling rates and the energy consumption depends on the throughput. Links can be bonded together for additional throughput as shown in Table 5.3. The architectural specifications of InfiniBand define multiple operational data rates: single data rate (SDR), double data rate (DDR), quad data rate (QDR), fourteen data rate (FDR), and enhanced data rated (EDR).

The signaling rates are: 2.5 Gbps in each direction per connection for an SDR connection; 5 Gbps for DDR; 10 Gbps for QDR; 14.0625 Gbps for FDR; 25.78125 Gbps for EDR per lane. The SDR, DDR and QDR link encoding is 8 B/10 B, every 10 bits sent carry 8 bits of data. Thus single, double, and quad data rates carry 2, 4, or 8 Gbit/s useful data, respectively, as the effective data transmission rate is four-fifths the raw rate.

InfiniBand allows links to be configured for a specified speed and width; the reactivation time of the link can vary from several nanoseconds to several microseconds. Exadata and Exalogic systems from Oracle implement the InfiniBand QDR with 40 Gbit/s (32 Gbit/s effective) using Sun Switches. The InifiniBand fabric is used to connect compute nodes, compute nodes with storage servers, and Exadata and Exalogic systems.

In addition to high throughput and low latency InfiniBand supports quality of service guarantees and failover – the capability to switch to a redundant or standby system. It offers point-to-point bidirectional serial links intended for the connection of processors with high-speed peripherals, such as disks, as well as multicast operations.

Note also that the deployment of InfiniBand pushes the limitations of the buses used in personal computers (PCs) and other personal devices as well. Introduced as a standard PC architecture in early 90's the PCI bus has evolved from 32 bit/33 MHz to 64 bit/66 MHz while PCH-X has doubled the clock rate to 133 MHz. Buses have severe electrical, mechanical, and power issues. The number of pins of a parallel bus is quite large; the number of pins necessary for each connection is quite large, e.g., a 64 bit PCI bus requires 90 pins.

Myrinet is an interconnect for massively parallel systems developed at Caltech and later at a company called Myricom. Its main features are [69]:

1. Robustness ensured by communication channels with flow control, packet framing, and error control.
2. Self-initializing, low-latency, cut-through switches.
3. Host interfaces that can map the network, select routes, and translate from network addresses to routes, as well as handle packet traffic.
4. Streamlined host software that allows direct communication between user processes and the network.

The design of Myrinet benefits from the experience gathered by a project to construct a high-speed local-area network at USC/ISI. A Myrinet is composed of point-to-point, full-duplex links connecting hosts and switches, an architecture similar to the one of the ATOMIC project at USC. Multiple-port switches may be connected to other switches and to the single-port host interfaces in any topology. Transmission is synchronous at the sending end, while the receiver circuits are asynchronous. The receiver injects control symbols in the reverse channel of the link for flow control. Myrinet switches use blocking-cut-through routing similar to the one in Intel Paragon and Cray T3D.

Myrinet supports high data rates; a Myrinet link consists of a full-duplex pair of 640 Mbps channels and has regular topology with elementary routing circuits in each node. The network is scalable, its aggregate capacity grows with the number of nodes. Simple algorithmic routing avoids deadlocks and allows multiple packets to flow concurrently through the network.

There is a significant difference between *store and forward* and *cut-through or wormhole* networks. In the former an entire packet is buffered and its checksum is verified in each node along the path from the source to the destination. In wormhole networks the packet is forwarded to its next hop as soon as the header is received and decoded. This decreases the latency, but a packet can still experience blocking if the outgoing channel expected to carry it to the next node is in use. In this case the packet has to wait until the channel becomes free.

A comparison of Myrinet and Ethernet performance as a communication substrate for MPI libraries is presented in [321]. MPI library implementations for Ethernet have a higher message latency and lower message bandwidth because they use the OS network protocol stack. The NAS benchmarks,[6] MG messaging over Myrinet only achieves a 5% higher performance then TCP messaging over Ethernet.

[6]NASA Parallel Benchmarks, NAS, are used to evaluate the performance of parallel supercomputers. The original benchmark included five kernels: IS – Integer Sort, random memory access, EP – Embarrassingly Parallel, CG – Conjugate Gradient, MG – Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive, FT – discrete 3D Fast Fourier Transform, all-to-all communication.

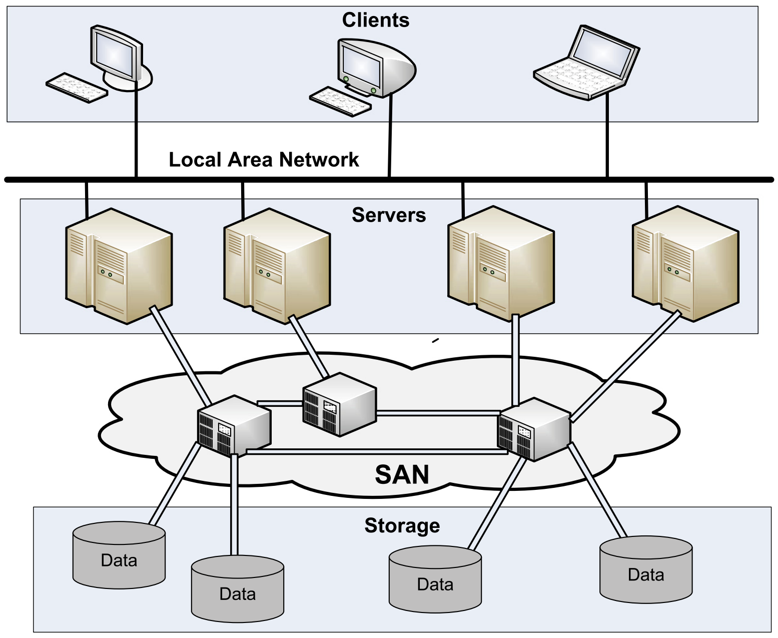


FIGURE 5.14

A storage area network interconnects servers to servers, servers to storage devices, and storage devices to storage devices. Typically it uses fiber optics and the FC protocol.

5.9 STORAGE AREA NETWORKS AND THE FIBRE CHANNEL

A storage area network (SAN) is a specialized, high-speed network for data block transfers. SANs interconnect servers to servers, servers to storage devices, and storage devices to storage devices, see Figure 5.14. A SAN consists of a communication infrastructure and a management layer. The Fibre Channel (FC) is the dominant architecture of SANs.

FC it is a layered protocol with several layers depicted in Figure 5.15:

A. The three lower layer protocols: FC-0, the physical interface; FC-1, the transmission protocol responsible for encoding/decoding; and FC-2, the signaling protocol responsible for framing and flow control. FC-0 uses laser diodes as the optical source and manages the point-to-point fiber connections; when the fiber connection is broken, the ports send a series of pulses until the physical connection is re-established and the necessary handshake procedures are followed.

FC-1 controls the serial transmission and integrates data with clock information. It ensures encoding to the maximum length of the code, maintains DC-balance, and provides word alignment. FC-2 provides the transport methods for data transmitted in 4-byte ordered sets containing data and control characters. It handles the topologies, based on the presence or absence of a fabric, the communication models, the classes of service provided by the fabric and the nodes, sequence and exchange identifiers, and segmentation and reassembly.

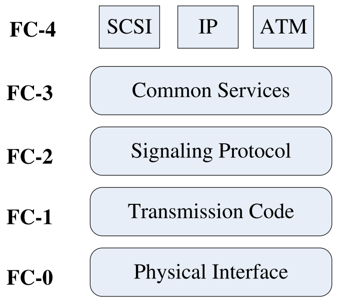


FIGURE 5.15

FC protocol layers. The FC-4 supports communication with the Small Computer System Interface (SCSI), the IP, and the Asynchronous Transfer Mode (ATM) network interfaces.

B. Two upper layer protocols: FC-3, the common services layer; and FC-4, the protocol mapping layer. FC-3 supports multiple ports on a single-node or fabric using:

- Hunt groups – sets of associated ports assigned an alias identifier that allows any frame containing that alias to be routed to any available port within the set.
- Striping to multiply bandwidth, using multiple ports in parallel to transmit a single information unit across multiple links.
- Multicast and broadcast to deliver a single transmission to multiple destination ports or to all ports.

To accommodate different application needs, FC supports several classes of service:

Class 1: rarely used blocking connection-oriented service; acknowledgments ensure that the frames are received in the same order in which they are sent, and reserve full bandwidth for the connection between the two devices.

Class 2: acknowledgments ensure that the frames are received; allows the fabric to multiplex several messages on a frame-by-frame basis; as frames can take different routes it does not guarantee in-order delivery, it relies on upper layer protocols to take care of frame sequence.

Class 3: datagram connection; no acknowledgments.

Class 4: connection-oriented service. Virtual circuits (VCs) established between ports, guarantee in-order delivery and acknowledgment of delivered frames. The fabric is responsible for multiplexing frames of different VCs. Supports Guaranteed Quality of Service (QoS), including bandwidth and latency. This layer is intended for multimedia applications.

Class 5: isochronous service for applications requiring immediate delivery, without buffering.

Class 6: supports dedicated connections for a reliable multicast.

Class 7: similar with Class 2, but used for the control and management of the fabric; a connection-less service with notification of non-delivery.

While every device connected to a LAN has a unique MAC address, each FC device has a unique ID called the WWN (World Wide Name), a 64 bit address. Every port in the switched fabric has its

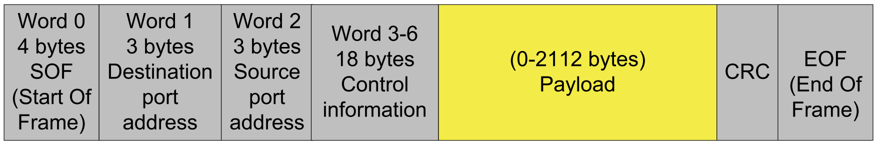


FIGURE 5.16

The format of an FC frame; the payload can be at most 2112 bytes, larger data units are carried by multiple frames.

own unique 24-bit address consisting of: the domain (bits 23–16), the area (bits 15–08), and the port physical address (bits 07–00).

The switch of a switched fabric environment assigns dynamically and maintains the port addresses. When a device with a WWN address logs into a given port, the switch maintains the correlation between that port address and the WWN address of the device using the Name Server. The Name Server is a component of the fabric operating system, which runs inside the switch.

The format of an FC frame is shown in Figure 5.16. Zoning permits finer segmentation of the switched fabric; only the members of the same zone can communicate within that zone. It can be used to separate different environments, e.g., a Microsoft Windows NT from a UNIX environment.

Several other protocols are used for SANs. Fibre Channel over IP (FCIP) allows transmission of Fibre Channel information through the IP network using tunneling. Tunneling is a technique for network protocols to encapsulate a different payload protocol; it allows a network protocol to carry a payload over an incompatible delivery-network, or to provide a secure path through an untrusted network.

Tunneling allows a protocol normally blocked by a firewall to cross it wrapped inside a protocol that the firewall does not block. For example, an HTTP tunnel can be used for communication from network locations with restricted connectivity, e.g., behind NATs, firewalls, or proxy servers. Restricted connectivity is a commonly-used method to lock down a network to secure it against internal and external threats.

Internet Fibre Channel Protocol (iFCP) is a gateway-to-gateway protocol that supports communication among FC storage devices in a SAN, or on the Internet using TCP/IP; iFCP replaces the lower-layer Fibre Channel transport with TCP/IP and Gigabit Ethernet. With iFCP, Fibre Channel devices connect to an iFCP gateway or switch and each Fibre Channel session is terminated at the local gateway and converted to a TCP/IP session via iFCP.

5.10 SCALABLE DATA CENTER COMMUNICATION ARCHITECTURES

The question addressed now is: How to organize the communication infrastructure of large cloud data centers to get the best performance at the lowest cost? Several architectural styles for *data center networks* (DCNs) attempting to provide an answer to this question face major challenges:

- The aggregate cluster bandwidth scales poorly with the cluster size.
- The bandwidth needed by many cloud applications comes at a high price and the cost of the interconnect increases dramatically with the cluster size.

Table 5.4 A side-by-side comparison of performance and cost of hierarchical and fat-tree networks over a span of six years [17].

	Hierarchical Network			Fat-tree		
Year	**10 GigE**	**Servers**	**Cost/GigE**	**10 GigE**	**Servers**	**Cost/GigE**
2001	28-port	4480	\$25.3 K	28-port	5488	\$4.5K
2004	32-port	7680	\$4.4 K	48-port	27,688	\$1.6K
2006	64-port	10,240	\$2.1 K	48-port	27,688	\$1.2K
2008	128-port	20,480	\$1.8 K	48-port	27,688	\$0.3K

- The communication bandwidth of DCNs may become oversubscribed by a significant factor depending on the communication patterns.

We only mention two DCN architectural styles, the *three-tier* and the *fat-tree* DCNs. The former has a multiple-rooted tree topology with three layers, core, aggregate, and access. The servers are directly connected to switches at the leafs of the tree at the *access layer*.

Enterprise switches at the root of the tree form the *core layer* and connect together the switches at the *aggregate layer* and also connect the data center to the Internet. The uplinks of the *aggregate layer* switches connect them to the core layer and their download links connect to the access layer. The three-tier DCN architecture is not suitable for computer clouds, it is not particularly scalable, the bisection bandwidth is not optimal, and the switches at the core layer are expensive and power-hungry.

As noted in Section 5.6 the fat-tree topology is optimal for computer clouds, the bandwidth is not severely affected for messages crossing multiple switches and the interconnection network can be built with commodity rather than enterprise switches. An implementation of the fat-tree topology proposed in [17] is discussed in this section. Several principles guide the design of this network:

- The network should scale to a very large number of nodes.
- The fat-tree should have multiple core switches.
- The network should support multi-path routing. The equal-cost multi-path (ECMP) routing algorithm [241] which performs static load splitting among flows should be used.
- The building blocks of the network should be switches with optimal cost/performance ratios.

Table 5.4 shows the performance and the cost of hierarchical and fat-tree interconnection networks expressed in GigE[7] evolved in the span of six years. The cost per GigE of both types of networks decreased by one order of magnitude, yet this cost/performance indicator for fat-tree built with commodity switches is almost an order of magnitude lower than that of the hierarchic networks. The choice of multi-rooted fat-tree topology and multi-path routing is justified because in 2008 the largest cluster that could be supported with a single rooted core 128-port router with 1:1 oversubscription would have been limited to 1280 nodes.

[7]The IEEE 802.3-2008 standard defines GigE as a technology for transmitting Ethernet frames. 1 GigE corresponds to a rate of one gigabit per second, i.e., 10^9 bits per second.

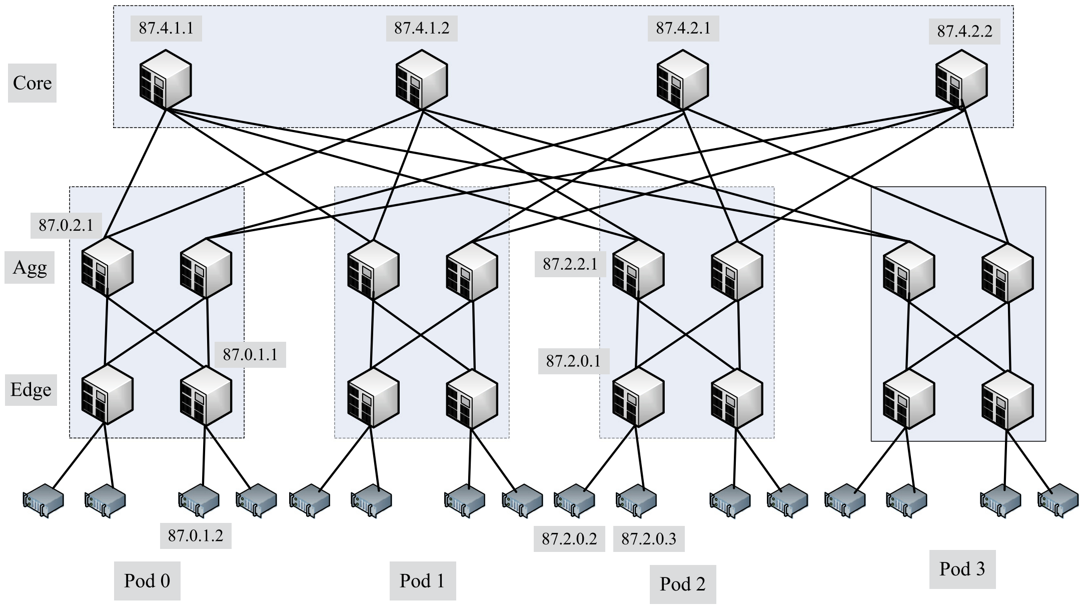


FIGURE 5.17

A fat-tree network with $k = 4$-port switches. Four core 4-way switches are at the root, there are four pods, and two switches at the aggregation layer and two at the edge layer of each pod. Each switch at the edge of a pod is connected to two servers.

A WSC interconnect can be organized as a fat-tree with k-port switches and $k = 48$, but the same fat-tree organization can be supported for any k. The network consists of k *pods*[8] each pod has two layers and $k/2$ switches at each layer. Each switch at the lower layer is connected directly to $k/2$ servers. The other $k/2$ ports are connected to $k/2$ of the k ports in the aggregation layer. The total number of switches is $k(k+1)$ and the total number of servers connected to the system is k^2. There are $(k/2)^2$ paths connecting every pair of servers.

A WSC with 16 384 servers can be built with 128-port switches and one with 262 144 servers will requires 512-port switches. Figure 5.17 shows a fat-tree interconnection network for $k = 4$. The core, the aggregation, and the edge layers are populated with 4-port switches. Each core switch is connected with one of the switches at the aggregation layer of each pod. The network has four pods, there are four switches at each pod, two at aggregation layer and two at the edge. Four servers are connected to each pod.

The IP addresses of switches have the form $87.pod.switch.1$, and the switches are numbered left to right, and bottom to top. The core switches have addresses of the form $10.k.j.i$ where j and i denote the coordinates of the switch in the $(k/2)^2$ core switch grid starting from top-left. For example, the four switches of pod 2 have IP addresses 87.2.0.1, 87.2.1.1, 87.2.2.1, and 87.2.3.1.

[8] A pod is a repeatable design pattern to maximize the modularity, scalability, and manageability of data center networks.

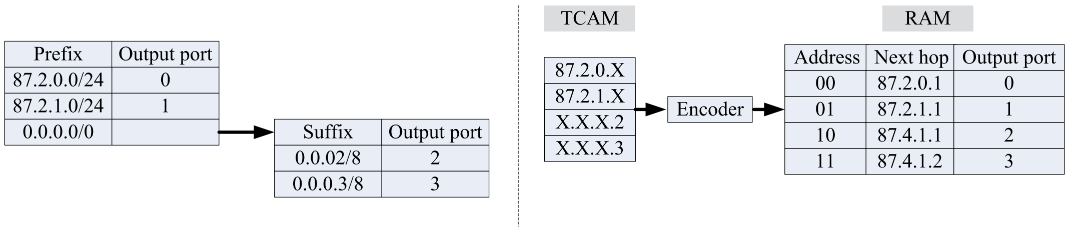


FIGURE 5.18

(Left) The two level routing tables for switch 87.2.2.1. Two incoming packets for IP addresses 10.2.1.2 and 10.3.0.3 are forwarded on ports 1 and 3, respectively. (Right) The RAM implementation of a two-level TCAM routing table.

Servers have IP addresses of the form $87.pod.switch.serverID$ where $serverID$ is the server position in the subnet of the edge router starting from left to right. For example, the IP addresses of the two servers connected to the switch with IP address 87.2.0.1 are 87.2.0.2 and 87.2.0.3.

We can see that there are multiple paths between any pairs of servers. For example, packets sent by the server with IP address 87.0.1.2 to server with IP address 87.2.0.3 can follow the following routes:

$$\begin{aligned} &87.0.1.1 \mapsto 87.0.2.1 \mapsto 87.4.1.1 \mapsto 87.2.2.1 \mapsto 87.2.0.1 \\ &87.0.1.1 \mapsto 87.0.2.1 \mapsto 87.4.1.2 \mapsto 87.2.2.1 \mapsto 87.2.0.1 \\ &87.0.1.1 \mapsto 87.0.1.1 \mapsto 87.4.2.1 \mapsto 87.2.2.1 \mapsto 87.2.0.1 \\ &87.0.1.1 \mapsto 87.0.1.1 \mapsto 87.4.2.2 \mapsto 87.2.2.2 \mapsto 87.2.0.1 \end{aligned} \tag{5.5}$$

Packet routing supports two-level prefix lookup and it is implemented using two-level routing tables. This strategy may increase the lookup latency but the prefix search can be done in parallel and could compensate for the latency increase. The main table entries are of the form (*prefix*, *output port*) and could have an additional pointer to a secondary table or could be terminating if none of its entries point to a secondary table. A secondary table consists of (*suffix*, *output port*) entries and may be pointed to by more than one first level entry.

Figure 5.18 (Left) shows the two level routing tables for switch 87.2.2.1 and routing for two incoming packets for servers with the IP addresses 87.2.1.2 and 87.3.0.3; the incoming packets are forwarded on ports 1 and 3, respectively. Lookup engines use a ternary version of content-addressable memory (CAM), called TCAM. Figure 5.18 (Right) shows that TCAM stores address prefixes and suffixes, which index a RAM that stores the IP address of the next hop and the output port.

The prefix entries are stored with numerically smaller addresses first and the right-handed (suffix) entries in larger addresses. The output of the CAM is encoded so that the entry with the numerically smallest matching address is the output. When the destination IP address of a packet matches both a left-handed and a right-handed entry, then the left-handed entry is chosen.

The k switches in a pod have terminating prefixes to the subnets in that pod. When two servers in the same pod but on a different subnets communicate, all upper-level switches of the pod will have a terminating prefix pointing to the destination subnet's switch.

For all outgoing pod traffic, the pod switches have a default /0 prefix with a secondary table matching the least-significant byte of the destination IP address, the server ID. Traffic diffusion occurs only in the first half of a packet's journey. Once a packet reaches a core switch, there is exactly one link to its destination pod, and that switch will include a terminating $/16$ prefix for the pod of that packet $(87.pod.0.0/16, port)$. Once a packet reaches its destination pod, the receiving upper-level pod switch will also include a $(10.pod.switch.0/24, port)$ prefix to direct the packet to its destination subnet switch, where it is finally switched to its destination server.

Each pod switch assigns terminating prefixes for subnets in the same pod and add a /0 prefix with a secondary table matching the $serverID$s for inter-pod traffic. The routing tables for the upper pod switches are generated with the pseudocode of Algorithm 5.1. For the lower pod switches, the /24 subnet prefix in line 3 is omitted since that subnet's own traffic is switched, and intra- and inter-pod traffic should be evenly split among the upper switches.

Algorithm 5.1 Generates aggregation switch routing tables.

```
1  foreach  $pod\ x \in [0, k-1]$  do
2      foreach  $switch\ z \in [k/2, k-1]$  do
3          foreach  $subnet\ i \in [0, k/2-1]$  do
4              $addPrefix(10.x.z.1, 10.x.i.0/24, i)$;
5          end
6          $addPrefix(10.x.z.1, 0.0.0.0/0, 0)$;
7          foreach  $hostID\ i \in [2, (k/2)+1]$  do
8              $addSuffix(10.x.z.1, 0.0.0.i/8, (i-2+z) \mod (k/2)+(k/2)$;
9          end
10     end
11 end
```

Core switches contains only terminating /16 prefixes pointing to their destination pods, as shown in Algorithm 5.2.

Algorithm 5.2 Generates routing tables for core switches.

```
foreach  $j \in [0, k-1]$  do
    foreach  $i \in [1, k/2]$  do
        foreach  $destination\ pod \in [0, k/2-1]$  do
            $addPrefix(10.k.j.i, 10.x.0.0/16, x)$;
        end
    end
end
```

The maximum numbers of first-level prefixes and second-level suffixes are k and $k/2$, respectively.

Flow classification with dynamic port-reassignment in pod switches overcomes cases of local congestion when two flows compete for the same output port, even though another port that has the same cost to the destination is underused.

Power consumption and heat dissipation are major concerns for the cloud data centers. Switches at the higher tiers of the interconnect in data centers consume several kW and the entire interconnection infrastructure consumes hundreds to thousands of kW.

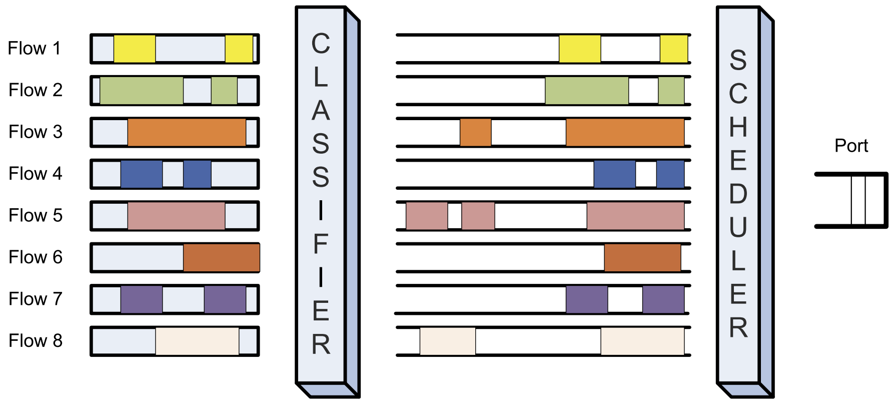


FIGURE 5.19

Fair Queuing – packets are first classified into flows by the system and then assigned to a queue dedicated to the flow; queues are serviced one packet at a time in round-robin order and empty queues are skipped.

5.11 NETWORK RESOURCE MANAGEMENT ALGORITHMS

A critical aspect of resource management in cloud computing is to guarantee the communication bandwidth required by an application as specified by an SLA. The solutions to this problem are based on the strategies used for some time on Internet to support the data streaming QoS requirements.

Cloud interconnects consist of communication links of limited bandwidth and switches of limited capacity. When the load exceeds its capacity, a switch starts dropping packets because it has limited input buffers for the switching fabric and for the outgoing links, as well as limited CPU cycles. A switch must handle multiple flows, pairs of source-destination end-points of the traffic, thus, a scheduling algorithm has to manage several quantities at the same time: the *bandwidth*, the amount of data each flow is allowed to transport, the *timing* when the packets of individual flows are transmitted, and the *buffer space* allocated to each flow.

Communication and computing require scheduling therefore, it should be no surprise that the first algorithm we discuss can be used for scheduling packets transmission, as well as threads. A first strategy to avoid network congestion is to use a FCFS scheduling algorithm. The advantage of FCFS algorithm is a simple management of the three quantities: bandwidth, timing, and buffer space. Nevertheless, the FCFS algorithm does not guarantee fairness; greedy flow sources can transmit at a higher rate and benefit from a larger share of the bandwidth.

Fair Queuing (FQ). The algorithm ensures that a high-data-rate flow cannot use more than its fair share of the link capacity. Packets are first classified into flows by the system and then assigned to a queue dedicated to the flow. Packet queues are serviced one packet at a time in round-robin (RR) order, Figure 5.19. FQ's objective is *max–min* fairness. This means that first it maximizes the minimum data rate of any of the data flows and then it maximizes the second minimum data rate. Starvation of expensive flows is avoided but the throughput is low.

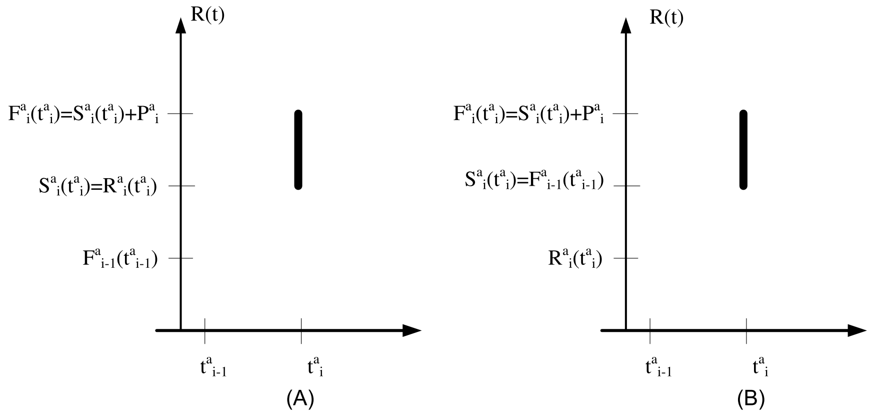

R(t)
$F^a_i(t^a_i)=S^a_i(t^a_i)+P^a_i$
$S^a_i(t^a_i)=R^a_i(t^a_i)$
$F^a_{i-1}(t^a_{i-1})$
t^a_{i-1}
t^a_i
(A)
R(t)
$F^a_i(t^a_i)=S^a_i(t^a_i)+P^a_i$
$S^a_i(t^a_i)=F^a_{i-1}(t^a_{i-1})$
$R^a_i(t^a_i)$
t^a_{i-1}
t^a_i
(B)

FIGURE 5.20

Transmission of a packet i of flow a arriving at time t_i^a of size P_i^a bits. The transmission starts at time $S_i^a = \max[F_{i-1}^a, R(t_i^a)]$ and ends at time $F_i^a = S_i^a + P_i^a$ with $R(t)$ the number of rounds of the algorithm. (A) The case $F_{i-1}^a < R(t_i^a)$. (B) The case $F_{i-1}^a \geq R(t_i^a)$.

The FQ algorithm guarantees the fairness of buffer space management, but does not guarantee fairness of bandwidth allocation; indeed, a flow transporting large packets will benefit from a larger bandwidth [355].

The FQ algorithm in [139] proposes a solution to this problem. First, it introduces a *Bit-by-bit Round-robin (BR)* strategy; as the name implies, in this rather impractical scheme a single bit from each queue is transmitted and the queues are visited in a round-robin fashion. Let $R(t)$ be the number of rounds of the BR algorithm up to time t and $N_{active}(t)$ the number of active flows through the switch. Call t_i^a the time when the packet i of flow a, of size P_i^a bits arrives and call S_i^a and F_i^a the values of $R(t)$ when the first and the last bit, respectively, of the packet i of flow a are transmitted. Then,

$$F_i^a = S_i^a + P_i^a \quad \text{and} \quad S_i^a = \max[F_{i-1}^a, R(t_i^a)]. \tag{5.6}$$

The quantities $R(t)$, S_i^a and F_i^a in Figure 5.20 depend only on the arrival time of the packets, t_i^a, and not on their transmission time, provided that a flow a is active as long as

$$R(t) \leq F_i^a \quad \text{when} \quad i = \max(j | t_i^a \leq t). \tag{5.7}$$

The authors of [139] use for packet-by-packet transmission time the following non-preemptive scheduling rule which emulates the BR strategy: *the next packet to be transmitted is the one with the smallest* F_i^a. A preemptive version of the algorithm requires that the transmission of the current packet be interrupted as soon as one with a shorter finishing time, F_i^a, arrives.

A fair allocation of the bandwidth does not have an effect on the timing of the transmission. A possible strategy is to allow less delay for the flows using less than their fair share of the bandwidth. The same paper [139] proposes the introduction of a quantity called the *bid*, B_i^a, and scheduling the packet

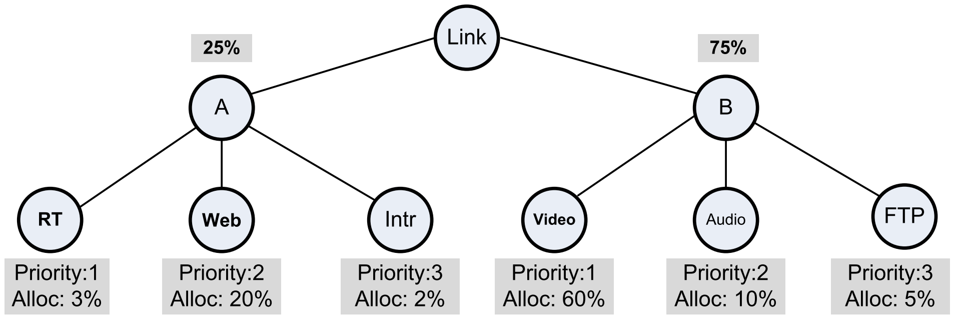


FIGURE 5.21

CBQ link sharing for two groups A, of short-lived traffic, and B, of long-lived traffic, allocated 25% and 75% of the link capacity, respectively. There are six classes of traffic with priorities 1, 2, and 3. The RT (real-time) and the video streaming have priority 1 and are allocated 3% and 60%, respectively, of the link capacity. Web transactions and audio streaming have priority 2 and are allocated 20% and 10%, respectively of the link capacity. Intr (interactive applications) and FTP (file transfer protocols) have priority 3 and are allocated 2% and 5%, respectively, of the link capacity.

transmission based on its value. The bid is defined as

$$B_i^a = P_i^a + \max[F_{i-1}^a, (R(t_i^a) - \delta)], \tag{5.8}$$

with δ a non-negative parameter. The properties of the FQ algorithm, as well as the implementation of a non-preemptive version of the algorithm are analyzed in [139].

The Stochastic Fairness Queuing (SFQ) algorithm is a simpler and less accurate implementation of the FQ algorithms and requires less calculations. SFQ ensures that each flow has the opportunity to transmit an equal amount of data and takes into account data packet sizes [338].

Class-Based Queuing (CBQ). This algorithm is a widely used strategy for link sharing proposed by Sally Floyd and Van Jacobson in 1995 [176]. The objective of CBQ is to support flexible link sharing for applications which require bandwidth guarantees such as VoIP, video-streaming, and audio-streaming. At the same time, CBQ supports some balance between short-lived network flows, such as web searches, and long-lived ones, such as video-streaming or file transfers.

CBQ aggregates the connections and constructs a hierarchy of classes with different priorities and throughput allocations. To accomplish link sharing, CBQ uses several functional units: (i) a *classifier* which uses the information in the packet header to assign arriving packets to classes; (ii) an *estimator* of the short-term bandwidth for the class; (iii) a *selector*, or scheduler, to identify the highest priority class to send next and, if multiple classes have the same priority, to schedule them on a round-robin base; and (iv) a *delayer* to compute the next time when a class that has exceeded its link allocation is allowed to send.

The classes are organized in a tree-like hierarchy; for example, in Figure 5.21 we see two types of traffic, group A corresponding to short-lived traffic and group B corresponding to long-lived traffic. The leaves of the tree are considered Level 1 and in this example include six classes of traffic: real-time,

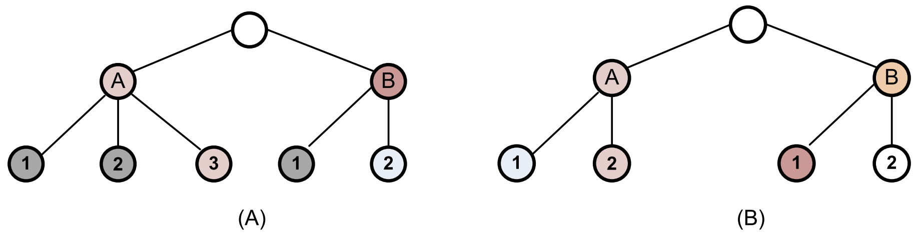


FIGURE 5.22

There are two groups A and B and three types of traffic, e.g., web, real-time, and interactive, denoted as $1, 2$, and 3. (A) Group A and class $A.3$ traffic are underlimit and unsatisfied; classes $A.1$, $A.2$ and $B.1$ are overlimit, unsatisfied and with persistent backlog and have to be regulated; type $A.3$ is underlimit and unsatisfied; group B is overlimit. (B) Group A is underlimit and unsatisfied; Group B is overlimit and needs to be regulated; class $A.1$ traffic is underlimit; class $A.2$ is overlimit and with persistent backlog; class $B.1$ traffic is overlimit and with persistent backlog and needs to be regulated.

web, interactive, video streaming, audio streaming, and file transfer. At Level 2 there are the two classes of traffic, A and B. The root, at Level 3 is the link itself.

The link sharing policy aims to ensure that if sufficient demand exists then, after some time intervals, each interior or leaf class receives its allocated bandwidth. The distribution of the "excess" bandwidth follows a set of guidelines, but does not support mechanisms for congestion avoidance.

A class is *overlimit* if over a certain recent period it has used more than its bandwidth allocation (in bytes per second), *underlimit* if it has used less, and *atlimit* if it has used exactly its allocation. A leaf class is *satisfied* if it is underlimit and has a persistent backlog and it is *unsatisfied* otherwise; a non-leaf class is unsatisfied if it is underlimit and has some descendent class with a persistent backlog. A precise definition of the term "persistent backlog" is part of a local policy. A class does not need to be *regulated* if it is underlimit or if there are no unsatisfied classes; the class should be regulated if it is overlimit and if some other class is unsatisfied and this regulation should continue until the class is no longer overlimit or until there are no unsatisfied classes, see Figure 5.22 for two examples.

The Linux kernel implements a link sharing algorithm called *Hierarchical Token Buckets* (HTB) inspired by CBQ. In CBQ every class has an *assured rate* (AR); in addition to the AR every class in HTB has also a *ceil rate* (CR), see Figure 5.23. The main advantage of HTB over CBQ is that it allows *borrowing*. If a class C needs a rate above its AR it tries to borrow from its parent; then the parent examines its children and, if there are classes running at a rate lower that their AR, the parent can borrow from them and reallocate it to class C.

5.12 CONTENT DELIVERY NETWORKS

Computer clouds support not only network-centric computing but also network-centric content. For example, we shall see in Chapter 6 that Internet video was expected to generate in 2013 over 18 Exabytes of data per month. Video traffic will account for 79% of the global Internet traffic by 2020. The vast amount of data stored on the cloud has to be delivered efficiently to a large user population.

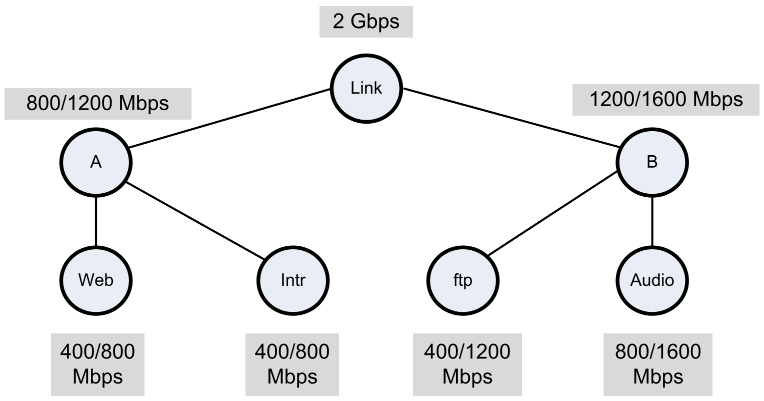


FIGURE 5.23

HTB packet scheduling uses for every node a ceil rate in addition to the allowed rate.

Content Delivery Networks (CDNs) offer fast and reliable content delivery and reduce communication bandwidth by caching and replication. A CDN receives the content from an *Origin* server, then replicates it to its *Edge* cache servers; the content is delivered to an end-user from the "closest" Edge server.

CDNs are designed to support scalability, to increase reliability and performance, and to provide better security. The volume of transactions and data transported by the Internet increases dramatically every year; additional resources are necessary to accommodate the additional load placed on the communication and storage systems and to improve the end-user experience. CDNs place additional resources provisioned to absorb the traffic caused by *flash crowds*[9] and, in general, to provide capacity on demand.

The additional resources are placed strategically throughout the Internet to ensure scalability. The resources provided by a CDN are replicated and when one of the replicas fails, the content is available from another one; the replicas are "close" to the consumers of the content and this placement reduces the start-up time and the communication bandwidth. A CDN uses two types of servers; the *origin* server updated by the content provider and *replica* servers which cache the content and serve as authoritative reference for client requests. Security is a critical aspect of the services provided by a CDN; the replicated content should be protected from the increased risk of cyber fraud and unauthorized access.

A CDN can deliver static content and/or live or on-demand streaming media. *Static content* refers to media that can be maintained using traditional caching technologies because changes are infrequent; examples of static content are: HTML pages, images, documents, software patches, and audio and/or video files. *Live media* refers to live events when the content is delivered in real time from the encoder to the media server. On-demand delivery of audio and/or video streams, movie files and music clips

[9]The term flash crowds refers to an event which disrupts the life of a very significant segment of the population, such as an earthquake in a very populated area, and causes the Internet traffic to increase dramatically.

provided to the end-users is content-encoded and then stored on media servers. Virtually all CDN providers support static content delivery, while live or on-demand streaming media is considerably more challenging.

CDN providers and protocols. The first CDN was setup by *Akamai*, a company evolved from an MIT project to optimize network traffic. Since its inception Akamai has placed some 20 000 servers in 1000 networks in 71 countries; in 2009 it controlled some 85% of the market [394].

Akamai mirrors the contents of clients on multiple systems placed strategically through the Internet. Though the domain name (but not sub-domain) is the same, the IP address of the resource requested by a user points to an Akamai server rather than the customer's server. Then the Akamai server is automatically picked depending on the type of content and the network location of the end user.

There are several other active commercial CDNs including EdgeStream providing video streaming and Limelight Networks providing distributed on-demand and live delivery of video, music, and games. There are several academic CDNs: Coral is a freely-available network designed to mirror web content, hosted on PlanetLab; Globule is an open-source collaborative CDN developed at the Vrije Universiteit in Amsterdam.

The communication infrastructure among different CDN components uses a fair number of protocols including: Network Element Control Protocol (NECP), Web Cache Coordination Protocol (WCCP), SOCKS, Cache Array Routing Protocol (CARP), Internet Cache Protocol (ICP), Hypertext Caching Protocol (HTCP), and Cache Digest described succinctly in [394]. For example, caches exchange ICP queries and replays to locate the best sites to retrieve an object; HTCP is used to discover HTTP caches, to cache data, to manage sets of HTTP caches and monitor cache activity.

CDN organization, design decisions, and performance. There are two strategies for CDN organization; in the so-called *overlay*, the network core does not play an active role in the content delivery. On the other hand, the *network* approach requires the routers and the switches to use dedicated software to identify specific application types and to forward user's requests based on predefined policies.

The first strategy is based exclusively on content replication on multiple caches and redirection based on proximity to the end-user. In the second approach the network core elements redirect content requests to local caches or redirect data center's incoming traffic to servers optimized for specific content type access. Some CDNs including Akamai use both strategies.

Important design and policy decisions for a CDN are:

1. Placement of the edge servers.
2. Content selection and delivery.
3. Content management.
4. Request routing policies.

The placement problem is often solved with suboptimal heuristics using as input the workload patterns and the network topology. The simplest, but a costly approach for content selection and delivery, is the *full-site* replication suitable for static content; the edge servers replicate the entire content of the origin server. On the other hand, the *partial-site* selection and delivery retrieves the base HTML page from the origin server and the objects referenced by this page from the edge caches. The objects can be replicated based on their popularity, or on some heuristics.

The content management depends on the caching techniques, the cache maintenance and the cache update policies. CDNs use several strategies to manage the consistency of content at replicas: periodic updates, updates triggered by the content change, and on-demand updates.

The request-routing in a CDN directs users to the closest edge server that can best serve the request; metrics, such as network proximity, client perceived latency, distance, and replica server load are taken into account when routing a request. Round-robin is a non-adaptive request-routing which aims to balance the load; it assumes that all edge servers have similar characteristics and can deliver the content.

Adaptive algorithms perform considerably better, but are more complex and require some knowledge of the current state of the system. The algorithm used by *Akamai* takes into consideration metrics such as: the load of the edge server, the bandwidth currently available to a replica server, the reliability of the connection of the client to the edge servers.

CDN routing can exploit an organization where several edge servers are connected to a service node aware of the load and the information about each edge server connected to it and attempts to implement a *global load balancing policy*. An alternative is *DNS-based routing* when a domain name has multiple IP addresses associated to it and the service provider's DNS server returns the IP addresses of the edge servers holding the replica of the requested object; then the client's DNS server chooses one of them.

Another alternative is the *HTTP redirection*; in this case a web server includes in the HTTP header of a response to a client the address of another edge server. Finally, *IP anycasting* requires that the same IP address is assigned to several hosts and the routing table of a router contains the address of the host closest to it.

The critical metrics of CDN performance are:

1. Cache hit ratio – the ratio of the number of cached objects versus total number of objects requested.
2. Reserved bandwidth for the origin server.
3. Latency – it is based on the perceived response time by the end users.
4. Edge server utilization.
5. Reliability – based on packet loss measurements.

CDNs will face considerable challenges in the future due to increased appeal of data streaming and to the proliferation of mobile devices such as of smart phones and tablets. On-demand video streaming requires enormous bandwidth and storage space, as well as powerful servers; CDNs for mobile networks must be able to dynamically reconfigure the system in response to spatial and temporal demand variations.

Content-Centric Networks. Content-Centric Networks (CCNs) are related to information-centric networking architectures such as Named Data Networks (NDNs) discussed in Section 5.4 where content is named and transferred throughout the network. In such networks the request for a named content is routed to the producer or to any entity that can deliver the expected content object.

CCNs content may be served from the cache of any router. Content is signed by its producer and the consumer is able to verify the signature before actually using the content. CCNs supports opportunistically caching. According to http://chris-wood.github.io/2015/06/16/CCN-vs-CDN.html CCNs offer a number of advantages. The popularity of the content need not be predicted beforehand, while providing the benefits not offered by IP-based CDNs including:

1. Active and intelligent forwarding strategies for routers.
2. Publisher mobility is easily supported via CCN routing protocols.
3. Congestion control can be enforced within the network.
4. The existing (and problematic) IP stack can be completely replaced with a new set of layers.
5. Existing APIs can be completely reworked to focus on content, not on addresses.
6. Content security is not tied to the channel, but to the content itself.

Table 5.5 VANETs applications, contents, local interest, local validity, and lifetime.

Application	Contents	Local interest	Local validity	Lifetime
Safety warnings	Dangerous Road	All	100 m	10 s
Safety warnings	Accident	All	500 m	30 s
Safety warnings	Work zone	All	1 km	Construction
Public service	Emergency vehicle	All	500 m	10 min
Public service	Highway information	All	5 km	All day
Driving	Road congestion	All	5 km	30 min
Driving	Navigation Map	Subscribers	5 km	30 min

The concept of *content service network* (CSN) was introduced in [318]. CSNs are overlay networks built around CDNs to provide an infrastructure service for processing and transcoding.

5.13 VEHICULAR AD HOC NETWORKS

A vehicular ad hoc network (VANET) consists of groups of moving or stationary vehicles connected by a wireless network. Until recently the main use of VANETs was to provide safety and comfort to drivers in vehicular environments. This view is changing, vehicular ad hoc networks are seen now as an infrastructure for an intelligent transportation system with increasing number of autonomous vehicles, and for any activity requiring Internet connectivity in a smart city. Also, VANETs allow on-board computers of mostly stationary vehicles, e.g., vehicles at an airport parking, to serve as resources of a mobile computer cloud with minimum help from the Internet infrastructure.

The contents produced and consumed by vehicles has *local relevance* in terms of time, space, and agents involved, the producer and the consumer. Vehicle-generated information has *local validity*, a limited spatial scope, an *explicit lifetime*, a limited temporal scope, and *local interest*, it is relevant to agents in a limited area around the vehicle. For example, the information that a car is approaching a congested area of a highway is relevant only for that particular segment of the road, at a particular time, and for vehicles nearby. The attributes of vehicular contents are summarized in Table 5.5 from [299].

One of the distinguishing characteristics of VANETs is the *content-centric distribution*, the content is important, the source is not. This is in marked contrast to the Internet where an agent demands information from a specific source. For example, traffic information floods a specific area and vehicle retrieves it without concern for the source of it, while an Internet request for highway traffic information is directed to a specific site. Vehicle applications collect sensor data and vehicles collaborate sharing sensory data. Sensory data is collected by vehicle-installed cameras, by on-board instruments. For example, *CarSpeak* allows a vehicle to access sensors on neighboring vehicles in the same manner in which it can access its own [284]. *Waze* is a community-based traffic and navigation application allowing drivers real-time traffic and road information.

VANET communication protocols are similar to the ones used by wired networks, each host has an IP address. Assigning IP addresses to moving vehicles is far from trivial and often requires a Dynamic Host Configuration Protocol (DHCP) server, a heresy for ad hoc networks that operate without any infrastructure, using self-organization protocols. Vehicles frequently join and leave the network and content of interest cannot be consistently bound to a unique IP address. A router typically relays and then deletes content.

5.14 FURTHER READINGS

The "Brief history of the Internet" [288] was written by the Internet pioneers Barry Leiner, Vinton Cerf, David Clark, Robert Kahn, Leonard Kleinrock, Daniel Lynch, Jon Postel, Larry Roberts, and Stephen Wolff.

The widely used text of Kurose and Ross [283] is an excellent introduction to basic networking concepts. The book by Bertsekas and Gallagher [65] gives insights into the performance evaluation of computer networks. The classic texts on queuing theory of Kleinrock [274] are a required reading for those interested in network analysis.

A survey of information-centric networking research is given in [532] while [548] is a succinct presentation of NDNs. Several publications related to software-defined networks are available on the site of the Open Network Foundation, https://www.opennetworking.org/sdn-resources.

The Moore's Law for traffic is discussed in [354]. The class-based queuing algorithm was introduced by Floyd and Van Jacobson in [176]. The *Black Widow* topology for system interconnects is analyzed in [447]. An extensive treatment of Storage Area Networks can be found in [481].

Alternative organizations of networks have been discussed in the literature. Scale-free networks and their applications are described by Barabási and Albert in [14–16,51]. The small-worlds networks were introduced by Watts and Strogatz in [515]. Epidemic algorithms for the dissemination of topological information are presented in [210,255,256]. Erdös–Rény random graphs are analyzed in [71,165]. Energy efficient protocols for cooperative networks are discussed in [163].

The future of fiber networks is covered in [289]. Vehicle ad hoc networks and their applications to vehicular cloud computing are discussed in [191,299,518]. An analysis of peer-to-peer networks is reported in [192]. Network management for private clouds, p2p networks, and virtual networks are presented in [351], [352], and [359], respectively.

5.15 EXERCISES AND PROBLEMS

Problem 1. Four ground rules for an open-architecture principle are cited in the "Brief history of the Internet." Read the paper and analyze the implication of each one of these rules.

Problem 2. The paper in Problem 1 lists also several key issues for the design of the network: (1) algorithms to prevent lost packets from permanently disabling communications and enabling them to be successfully retransmitted from the source; (2) providing for host-to-host "pipelining" so that multiple packets could be en-route from source to destination at the discretion of the participating hosts, if the intermediate networks allowed it; (3) the need for end-end checksums, reassembly of packets from fragments and detection of duplicates, if any; (4) the need for global addressing; and (5) techniques for host-to-host flow control.

Discuss how these issues were addressed by the TCP/IP network architecture.

Problem 3. Analyze the challenges of transition to IPv6. What will be in your view the effect of this transition on cloud computing?

Problem 4. Discuss the algorithms used to compute the TCP window size.

Problem 5. Creating a virtual machine (VM) reduces ultimately to copying a file, therefore the explosion of the number of VMs cannot be prevented, see Section 11.9. As each VM needs its own IP address, virtualization could drastically lead to an exhaustion of the IPv4 address space. Analyze the solution to this potential problem adopted by the IaaS cloud service delivery model.

Problem 6. Read the paper describing the stochastic fair queuing algorithm [176]. Analyze the similarities and dissimilarities of this algorithm and the start-time fair queuing discussed in Section 9.13.

Problem 7. The small-worlds networks were introduced by D. Watts and S. H. Strogatz. They have two desirable features, high clustering and small path length. Read [515] and design an algorithm to construct a small-worlds network.

Problem 8. The properties of scale-free networks are discussed in [14–16,51]. Discuss the important features of systems interconnected by scale-free networks discussed in these papers.

Problem* 9. Consider two 192 node fat-tree interconnect with two 96-way and twelve 24-way switches, the one in Figure 5.10 and the one in Figure 5.11. Compute the bisection bandwidth of the two interconnects.

CHAPTER

CLOUD DATA STORAGE

6

The volume of data generated by human activities is growing about 40% per year; 90% of the data in the world today has been gathered in the last two years alone (https://e27.co/tag/aureus-analytics/). Computer clouds provide the vast amounts of storage demanded by many applications using these data.

Several data sources contribute to the massive amounts of data stored on a cloud. A variety of sensors feed streams of data to cloud applications or simply generate content. An ever increasing number of cloud-based services collect detailed data about their services and information about the users of these services.

Big Data, discussed in depth in Chapter 12, reflects the reality that many applications use data sets so large that local computers, or even small to medium scale data centers, do not have the capacity to store and process such data. The consensus is that Big Data growth can be viewed as a three-dimensional phenomenon: (i) implies an increased volume of data; (ii) requires increased processing speed to process more data and produce more results; and (iii) involves a diversity of data sources and data types.

The network-centric data storage model is particularly useful for mobile devices with limited power reserves and local storage, now able to save and to access large audio and video files stored on computer clouds. Billions of Internet-connected mobile, as well as stationary devices, access data stored on computer clouds. It is predicted that: "annual global IP traffic will pass the zettabyte, 1000 exabytes [EB] threshold by the end of 2016, and will reach 2.3 ZB per year by 2020" (http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html).

Storage and processing on the cloud are intimately tied to one another. Data analytics uses the very large volumes of data collected by many organizations to optimize their businesses. An in-depth analysis allows these organizations to discover how to reach a larger population of customers, identify strength in their products or shortcomings in the organization, save energy, and, last but not least, protect the environment.

Applications in many areas of science, including genomics, structural biology, high energy physics, astronomy, meteorology, and the study of the environment, carry out complex analysis of data sets often of the order of terabytes.[1] In 2010, the four main detectors at the Large Hadron Collider (LHC) produced 13 PB of data; the Sloan Digital Sky Survey (SDSS) collects about 200 GB of data per night. As a result of this increasing appetite file systems, such as Btrfs, XFS, ZFS, exFAT, NTFS, HFS Plus, and ReFS, support disk formats with theoretical volume sizes of several exabytes.

While we emphasize the advantages of a concentration of resources we have to be acutely aware that a cloud is a large-scale distributed system with a very large number of components which must work in concert. The management of the large collection of storage systems poses significant challenges

[1]Terabyte, 1 TB = 10^{12} bytes; Petabyte, 1 PB = 10^{15} bytes; Exabyte, 1 EB = 10^{18} bytes; Zettabyte, 1 ZB = 10^{21} bytes.

and requires novel approaches to system design. Effective data replication and storage management strategies are critical to the computations performed on the cloud.

Sophisticated strategies to reduce the access time and to support multimedia access are necessary to satisfy the timing requirements of data streaming and content delivery. Data replication allows concurrent access to data from multiple processors and decreases the chances of data loss. Maintaining consistency among multiple copies of data records increases the data management software complexity and could negatively affect the storage system performance if data is frequently updated.

Nowadays large-scale systems are built with off-the-shelf components, while the distributed file systems of the past used custom-designed reliable components. The storage system design philosophy has shifted from performance-at-any-cost to reliability-at-the-lowest-possible-cost. This shift is evident in the evolution of ideas from the file systems of the 1980s, such as the Network File System (NFS), the Andrew File System (AFS), and the Sprite File System (SFS), to the Google File System (GFS), the Megastore, and the Colossus [173] developed during the last two decades.

The discussion of cloud storage starts with a review of the storage technology followed by an overview of storage models in Sections 6.1 and 6.2, respectively. The evolution of file systems from distributed file systems to parallel file systems, then to the file systems capable of handling massive amounts of data is presented in Sections 6.3, 6.4, and 6.5 which cover distributed file systems, the General Parallel File Systems, and the Google File System, respectively.

A locking service, Chubby, based on the Paxos algorithm is presented in Section 6.6 followed by a discussion of NoSQL databases and of transaction processing systems in Sections 6.7 and 6.8. Sections 6.9 and 6.10 analyze the BigTable and the Megastore system, respectively. Storage reliability at scale, data center disk locality, and database provenance are discussed in Sections 6.11, 6.12, and 6.13, respectively.

6.1 THE EVOLUTION OF STORAGE TECHNOLOGIES

During the last decades the storage technological has evolved at an accelerated pace and the volume of data stored every year has constantly increased [233]:

- 1986 – 2.6 EB; equivalent to less than one CD-ROM storing 730-MB per person.
- 1993 – 15.8 EB; equivalent to 4 CD-ROMs per person.
- 2000 – 54.5 EB; equivalent to 12 CD-ROMs per person.
- 2007 – 295 EB; equivalent to almost 61 CD-ROMs per person.

Storage technology. Though it pales in comparison with the evolution of processor technology, the evolution of the storage technology is astounding. A 2003 study [354] shows that during the 1980–2003 period the storage density of hard disk drives (HDD) has increased by four orders of magnitude from about 0.01 Gb/in^2 to about 100 Gb/in^2. During the same period the prices have fallen by five orders of magnitude to about 1 cent/Mbyte. HDD densities were projected to climb to 1 800 Gb/in^2 by 2016, up from 744 Gb/in^2 in 2011.

The density of DRAM (Dynamic Random Access Memory) increased from about 1 Gb/in^2 in 1990 to 100 Gb/in^2 in 2003. The cost of DRAM tumbled from about \$80/Mbyte to less than \$1/Mbyte during the same period. In 2010 Samsung announced the first monolithic 4 gigabit, low-power double-data-rate (LPDDR2) DRAM using a 30 nm process.

Recent advancements in storage technology have a broad impact on the storage systems used for cloud computing. The capacity of NAND flash-based devices outpaced DRAM capacity growth and the cost per gigabyte has significantly declined. The manufacturers of storage devices are investing in competing solid state technologies such as Phase-Change Memory.

While solid state memories are based on the charge of the electron, other fundamental property of an electron, its spin, is used to store information. A new field known as spintronics, an acronym for spin transport electronics, promises storage media based on antiferromagnetic materials insensitive to perturbations by stray fields and with much shorter switching times [528]. Solid state drives (SSDs) can support hundreds of thousands IOPS (I/O operations per second).

While the density of storage devices has increased and the cost has decreased dramatically, the access time has improved only slightly. The performance of I/O subsystems has not kept pace with the performance of processors. This performance gap affects multimedia, scientific and engineering, and other modern applications which process increasingly large volumes of data.

Storage systems face substantial pressure as the volume of data generated increased exponentially during the last decades. While in the 1980s and 1990s data was primarily generated by humans, nowadays machines generate data at an unprecedented rate. Mobile devices, such as smart phones and tablets, record static images as well as movies; they have limited local storage capacity and rely on transferring the data to cloud storage. Sensors, surveillance cameras, and digital medical imaging devices generate data at a high rate and dump it on storage systems accessible via the Internet. Online digital libraries, eBooks, and digital media, along with reference data add to the demand for massive amounts of storage. The term reference data is used for infrequently used data such as archived copies of medical or financial records, customer account statements, and so on.

As the volume of data increases, new methods and algorithms for data mining that require powerful computing systems have been developed. Only a concentration of resources could provide the CPU cycles along with the vast storage capacity necessary when performing such intensive computations and when accessing the very large volume of data.

The rapid technological advancements have changed the balance between the initial investment in the storage devices and the system management costs. Now, the cost of storage management is the dominant element of the total cost of a storage system. This effect favors the centralized storage strategy supported by a cloud; indeed, a centralized approach can automate some of the storage management functions such as replication and backup and thus, reduce substantially the storage management cost.

Disk technologies. *Hard disk drives* are ubiquitous secondary storage media for general-purpose computers. An HDD is a non-volatile random-access data storage device consisting of one or more rotating platters coated with magnetic material. Magnetic heads mounted on a moving actuator arm read and write data to the surface of the platters.

A typical HDD has a spindle motor that spins the disks and an actuator that positions the read/write head assembly across the spinning disks. The rotation speed of platters in today's HDDs ranges from 4 200 rpm for energy-efficient portable devices, to 15 000 rpm for high-performance servers. HDDs for desktop computers and laptops are 3.5-inch and 2.5-inch, respectively.

HDDs are characterized by capacity and performance. The capacity is measured in Megabytes (MB), Terabytes (TB), or Gigabytes (GB). The average access time is the most relevant HDD performance indicator. The access time includes the *seek time*, the time for the arm to reach to the

Table 6.1 The evolution of hard disk drive technologies from 1956 to 2016. The improvement ranges from astounding ratios such as 650×10^6 to one, 300×10^6 to one, and 2.7×10^6 to one for density, price, and capacity, respectively, to a modest 200 to one for average access time and 11 to one for MTBF, the mean time between failures.

Parameter	1956	2016
Capacity	3.75 MB	10 TB
Average access time	≈ 600 msec	2.5–10 ms
Density	200 bits/sq. inch	1.3 TB sq. inch
Average life span	≈ 2 000 hours/MTBF	≈ 22, 500 hours/MTBF
Price	\$9,200/MB	\$0.032/GB
Weight	910 Kg	62 g
Physical volume	1.9 m^3	34 cm^3

cylinder/track and the *search time*, the time to locate the record on a track. HDD technology has improved dramatically since the disk first introduced by IBM in 1956, as shown in Table 6.1.

At this time typical servers of a data center have up to six 2 TB disks; the physical space available on the rack limits this number. Increasing the disk space of a data center is costly.

Solid-state disks are persistent storage devices using integrated circuit assemblies as memory. SSD interfaces are compatible with the block I/O of HDDs, thus they can replace the traditional disks. SSDs do not have moving parts, are typically more resistant to physical shock, run silently, have lower access time, and lower latency than HDDs.

Lower-priced SSDs use triple-level or multi-level cell (MLC) flash memory, slower and less reliable than single-level cell (SLC) flash memory. The MLC to SLC ratios of persistence, sequential write, sequential read, and price are 1 : 10, 1 : 3, 1 : 1, and 1 : 1.3, respectively. Most SSDs use MLC NAND-based flash memory, a non-volatile memory that retains data when power is lost.

The latency of SLC NAND I/O operations is: 25 µsec to fetch a 4 KB page from the array to the I/O buffer on a read, 250 µsec to commit a 4 KB page from the IO buffer to the array on a write, and 2 msec to erase a 256 KB block. When multiple NAND devices operate in parallel the SSD bandwidth scales and the high latencies can be hidden, provided that the load is evenly distributed between NAND devices and sufficient outstanding operations are pending. Most SSD manufacturers use non-volatile NAND due to lower cost compared with DRAM and the ability to retain the data without a constant power supply.

Solid state hybrid disks (SSHDs) combine the features of SSDs and HDDs, they include a large HDD and an SSD cache to improve performance of frequently accessed data. The SSD cost-per-byte is reduced by about 50% every year, but it should be down by up to three orders of magnitude to be competitive with HDD.

6.2 STORAGE MODELS, FILE SYSTEMS, AND DATABASES

Storage models. A storage model describes the layout of a data structure in physical storage, while a data model captures the most important logical aspects of a data structure in a database. The physical storage can be a local disk, a removable media, or storage accessible via the network.

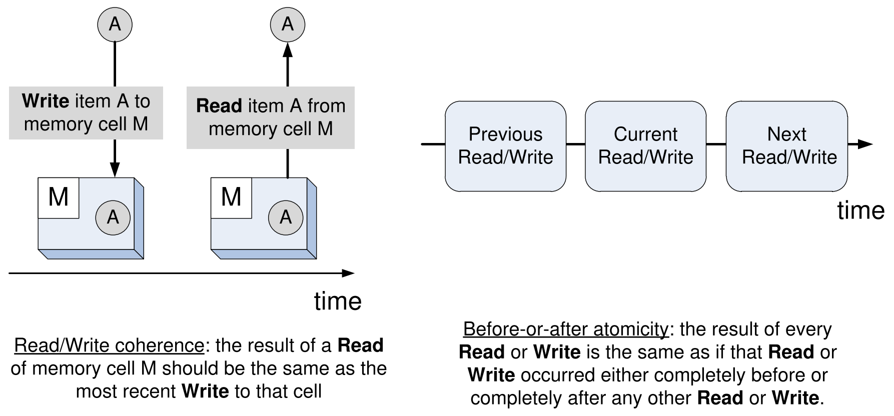

A
Write item A to memory cell M
M
A
A
Read item A from memory cell M
M
A
time
Read/Write coherence: the result of a Read of memory cell M should be the same as the most recent Write to that cell
Previous Read/Write
Current Read/Write
Next Read/Write
time
Before-or-after atomicity: the result of every Read or Write is the same as if that Read or Write occurred either completely before or completely after any other Read or Write.

FIGURE 6.1

Illustration capturing the semantics of read/write coherence and before-or-after atomicity.

Two abstract models of storage are commonly used: cell storage and journal storage. *Cell storage* assumes that the storage consists of cells of the same size and that each object fits exactly in one cell. This model reflects the physical organization of several storage media; the primary memory of a computer is organized as an array of memory cells and a secondary storage device, e.g., a disk, is organized in sectors or blocks read and written as a unit. Read/write coherence and before-or-after atomicity are two highly desirable properties of any storage model, and in particular, of cell storage, Figure 6.1.

Journal storage is a fairly elaborate organization for storing composite objects such as records consisting of multiple fields. Journal storage consists of a manager and a cell storage where the entire history of a variable is maintained, rather than just the current value. The user does not have direct access to the cell storage, instead it can request the journal manager to: (i) start a new action; (ii) read the value of a cell; (iii) write the value of a cell; (iv) commit an action; and (v) abort an action. The journal manager translates user requests to commands sent to the cell storage: (i) read a cell; (ii) write a cell; (iii) allocate a cell; and (iv) deallocate a cell.

In the context of storage systems a log contains the history of all variables in a cell storage. The information about the updates of each data item forms a record appended at the end of the log. A log provides authoritative information about the outcome of an action involving the cell storage; the cell storage can be reconstructed using the log which can be easily accessed, we only need a pointer to the last record.

An all-or-nothing action first records the action in a log in journal storage and then installs the change in the cell storage by overwriting the previous version of a data item, see Figure 6.2. The log is always kept on non-volatile storage, e.g., disk, and the considerably larger cell storage resides typically on non-volatile memory, but can be held in memory for real-time access or using a write-through cache.

File systems. A file system consists of a collection of directories and each directory provides information about a set of files. Today high-performance systems can choose among three classes of file systems: Network File Systems (NFS), Storage Area Networks (SAN), and Parallel File Systems (PFS).

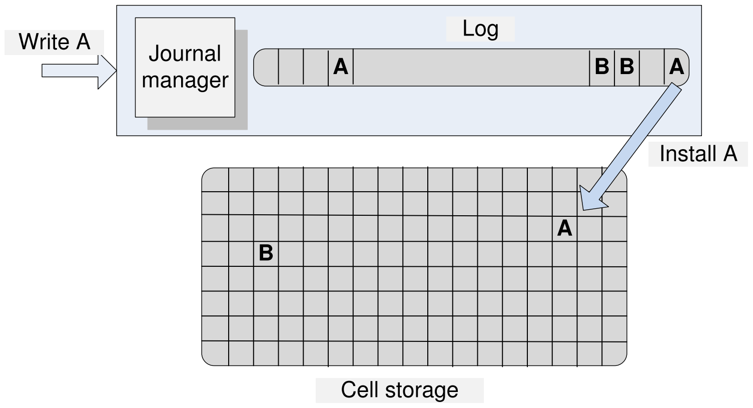


FIGURE 6.2

A log contains the entire history of all variables; the log is stored on a non-volatile media of a journal storage. If the system fails after the new value of a variable is stored in the log, but before the value is stored in the cell memory, then the value can be recovered from the log. If the system fails while writing the log, the cell memory is not updated. This guarantees that all actions are all-or-nothing. Two variables **A** and **B** in the log and the cell storage are shown. A new value of **A** is written first to the log and then installed on cell memory at the unique address assigned to **A**.

Network file systems are very popular and have been used for some time, but do not scale well and have reliability problems; an NFS server could be a single point of failure.

Advances in the networking technology allow the separation of the storage systems from the computational servers; the two can be connected by a SAN. SANs offer additional flexibility and allow cloud servers to deal with non-disruptive changes in the storage configuration. Moreover, the storage in a SAN can be *pooled* and then allocated based on the needs of the servers. Pooling requires additional software and hardware support and represents another advantage of a centralized storage system. A SAN-based implementation of a file system can be expensive, as each node must have a Fiber Channel adapter to connect to the network.

Parallel file systems are scalable, are capable of distributing files across a large number of nodes, and provide a global naming space. In a parallel file system several I/O nodes serve data to all computational nodes and include also a metadata server which contains information about the data stored in the I/O nodes. The interconnection network of a parallel file system could be a SAN.

Databases and database management systems. Most cloud applications do not interact directly with the file systems, but through an application layer which manages a database. A database is a collection of logically-related records. The software that controls the access to the database is called a Data Base Management System (DBMS). The main functions of a DBMS are: enforce data integrity, manage data access and concurrency control, and support recovery after a failure.

A DBMS supports a query language, a dedicated programming language used to develop database applications. Several database models, including the navigational model of the 1960s, the relational model of the 1970s, the object-oriented model of the 1980s, and the NoSQL model of the first decade of the 2000s, reflect the limitations of the hardware available at the time and the requirements of the most popular applications of each period.

In October 2015 Gartner (http://www.gartner.com) predicted that by 2017 a single DBMS platform will include multiple data models, relational and NoSQL and that the NoSQL label will no longer be used. The study classified DBMS offerings in four quadrants based on the ability to execute and the completeness of company vision:

1. *Leaders* – Oracle, Microsoft, AWS, IBM, MongoDB, SAP, DataStax, EnerpriseDB, InterSystems, MarkLogic, and Redis Las.
2. *Visionaries* – Couchbase, Fujitsu, MemSQL, and NuoDB.
3. *Challenges* – MariaDB and Percona.
4. *Niche players* – FairCom, Cloudera, MapR, Atibase, VoltDB, NeoTechnology, TmaxSoft, Clustrix, Actian, Aerospike, Hortonworks, Orient Technologies, and McObject.

Cloud databases. Most cloud applications are data-intensive, test the limitations of the existing cloud storage infrastructure, and demand database management systems capable of supporting rapid application development and short-time to the market. At the same time, cloud applications require low latency, scalability, high availability, and demand a consistent view of the data.

These requirements cannot be satisfied simultaneously by existing database models; for example, relational databases are easy to use for application development, but do not scale well. As its name implies, the NoSQL model does not support SQL as a query language and may not guarantee the ACID, Atomicity, Consistency, Isolation, and Durability properties of traditional databases. It usually guarantees an eventual consistency for transactions limited to a single data item.

The NoSQL model is useful when the structure of the data does not require a relational model and the amount of data is very large. Several types of NoSQL databases have emerged in the last few years. Based on the manner the NoSQL databases store the data, we recognize several types such as key-value stores, BigTable implementations, document store databases, and graph databases.

Replication, used to ensure fault-tolerance of large-scale systems built with commodity components, requires mechanisms to guarantee that replicas are consistent with one another. This is yet another example of increased complexity of modern computing and communication systems when the software has to support desirable properties of the physical systems. Section 6.6 contains an in-depth analysis of a service implementing a consensus algorithm to guarantee that replicated objects are consistent.

Many cloud applications support online transaction processing and have to guarantee the correctness of the transactions. Transactions consist of multiple actions; for example, the transfer of funds from one account to another requires withdrawing funds from one account and crediting it to another. The system may fail during or after each one of the actions and steps to ensure correctness must be taken. Correctness of a transaction means that the result should be guaranteed to be the same as if the actions were applied one after another regardless of the order. More stringent conditions must sometimes be observed; for example, banking transactions must be processed in the order they are issued, the so-called *external time consistency*. To guarantee correctness, a transaction processing system supports *all-or-nothing atomicity* discussed in Section 3.11.

6.3 DISTRIBUTED FILE SYSTEMS; THE PRECURSORS

The first distributed file systems were developed in the 1980s by software companies and universities. The systems covered are: NFS developed by Sun Microsystems in 1984, AFS developed at Carnegie Mellon University as part of the Andrew project, and SFS developed by John Osterhout's group at U. C. Berkeley as a component of the Unix-like distributed operating system called Sprite. Other systems developed at about the same time are Locus [507], Apollo [298], and the Remote File System (RFS) [46]. The main concerns in the design of these systems are scalability, performance, and security, see Table 6.2.

In 1980s many organizations, including research centers, universities, financial institutions, and design centers, considered that networks of workstations are an ideal environment for their operations. Diskless workstations were appealing due to reduced hardware costs and also to lower maintenance and system administration costs. Soon it became obvious that a distributed file system could be very useful for the management of a large number of workstations and Sun Microsystems, one of the main promoters of a distributed systems based on workstations, proceeded to develop the NFS in the early 1980s.

Network File System. NFS was the first widely used distributed file system; the development of this application based on the client-server model was motivated by the need to share a file system among a number of clients interconnected by a local area network.

A majority of workstations were running under UNIX; thus, many design decisions for the NFS were influenced by the design philosophy of the UNIX File System (UFS). It is not surprising that the NFS designers aimed to:

- Provide the same semantics as a local UFS to ensure compatibility with existing applications.
- Facilitate easy integration into existing UFS.
- Ensure that the system will be widely used; thus, support clients running on different operating systems.
- Accept a modest performance degradation due to remote access over a network with a bandwidth of several Mbps.

UFS has three important characteristics which enabled the extension from local to remote file management:

1. The layered design provides the necessary flexibility of the file system. Layering allows separation of concerns and minimization of the interaction among the modules necessary to implement the system. The addition of the vnode layer allowed UNIX file system to treat uniformly local and remote file access.
2. The hierarchical design supports file system scalability; it allows grouping of files into special files called directories, supports multiple levels of directories and collections of directories and files, the so-called file systems. The hierarchical file structure is reflected by the file naming convention.
3. The metadata supports a systematic rather than an ad hoc design philosophy of the file system. Inodes contain information about individual files and directories and are kept on persistent media together with the data. Metadata includes the file owner, the access rights, the creation time or the time of the last modification of the file, the file size, as well as information about the structure of the file and the persistent storage device cells where data is stored. Metadata also supports

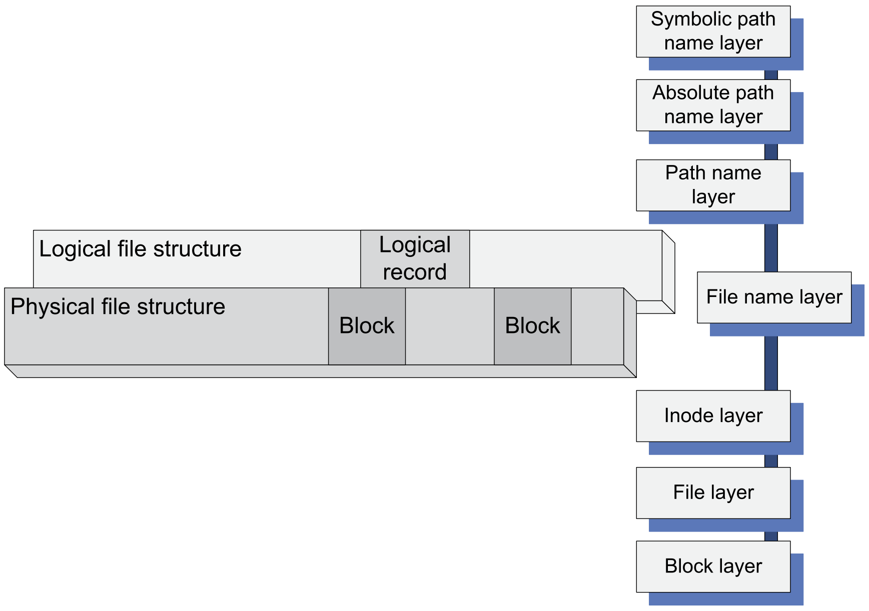


FIGURE 6.3

UFS layered design separates the physical file structure from the logical one. The lower three layers, block, file, and inode, are related to the physical file structure, while the upper three layers, path name, absolute path name, and symbolic path name reflect the logical organization. The file name layer mediates between the two groups.

device-independence, a very important objective due to the very rapid pace of storage technology development.

The *logical organization* of a file reflects the data model, the view of the data from the perspective of the application. The *physical organization* reflects the storage model and describes the manner the file is stored on a given storage media. The layered design allows UFS to separate the concerns for the physical file structure from the logical one.

Recall that a file is a linear array of cells stored on a persistent storage device; the *file pointer* identifies a cell used as a starting point for a read or write operation. This linear array is viewed by an application as a collection of logical records; the file is stored on a physical device as a set of physical records, or blocks, of size dictated by the physical media.

The lower three layers of the UFS hierarchy, the block, the file, and the inode layer, reflect the physical organization. The block layer allows the system to locate individual blocks on the physical device; the file layer reflects the organization of blocks into files; and the inode layer provides the metadata for the objects (files and directories). The upper three layers, the path name, the absolute path name, and the symbolic path name layer, reflect the logical organization. The file name layer mediates between the machine-oriented and the user-oriented views of the file system, see Figure 6.3.

Several control structures maintained by the kernel of the operating systems support the file handling by a running process; these structures are maintained in the user area of the process address space

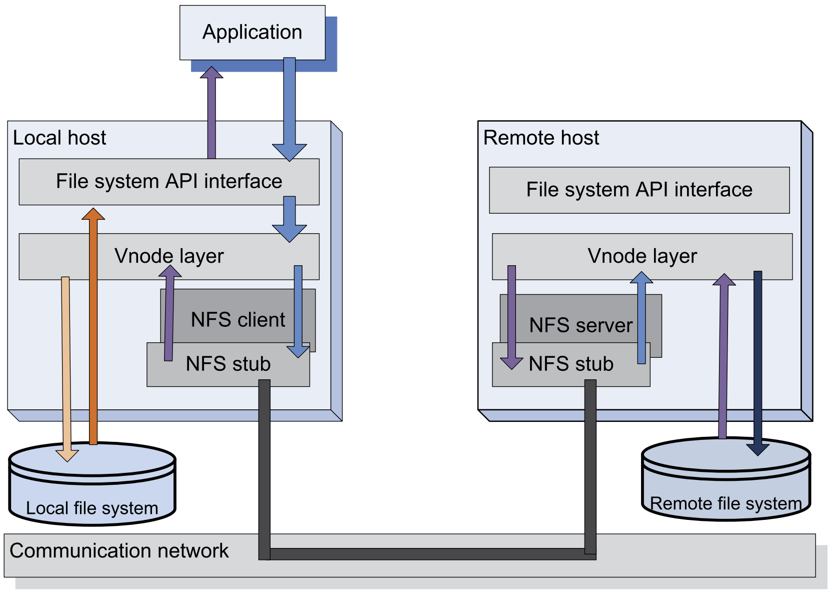


FIGURE 6.4

The NFS client-server interaction. The vnode layer implements file operation in a uniform manner, regardless of whether the file is local or remote. An operation targeting a local file is directed to the local file system, while one for a remote file involves NFS; an NFS client packages the relevant information about the target and the NFS server passes it to the vnode layer on the remote host which, in turn, directs it to the remote file system.

and can only be accessed in kernel mode. To access a file, a process must first establish a connection with the file system by opening the file; at that time a new entry is added to the file description table and the meta-information is brought in to another control structure, the open file table.

A *path* specifies the location of a file, or directory, in a file system. A *relative path* specifies the file location relative to the current/working directory of the process, while a *full path*, also called an absolute path, specifies the location of the file independently of the current directory, typically relative to the root directory. A local file is uniquely identified by a *file descriptor* (fd), generally, an index in the open file table.

NFS is based on the client-server paradigm. The client runs on the local host, while the server is at the site of the remote file system. The client and the server interact by means of Remote Procedure Calls (RPCs), Figure 6.4. The API interface of the local file system distinguishes file operations on a local file from the ones on a remote file and, in the later case, invokes the RPC client. Figure 6.5 shows the API for a UNIX file system, the calls made by the RPC client in response to API calls issued by a user program for a remote file system, as well as some of the actions carried out by the NFS server in response to an RPC call. NFS uses a *vnode* layer to distinguish between operations on local and remote files, as shown in Figure 6.4.

A remote file is uniquely identified by a *file handle* (fh), rather than a file descriptor. The file handle is a 32-byte internal name, a combination of the file system identification, an inode number, and a

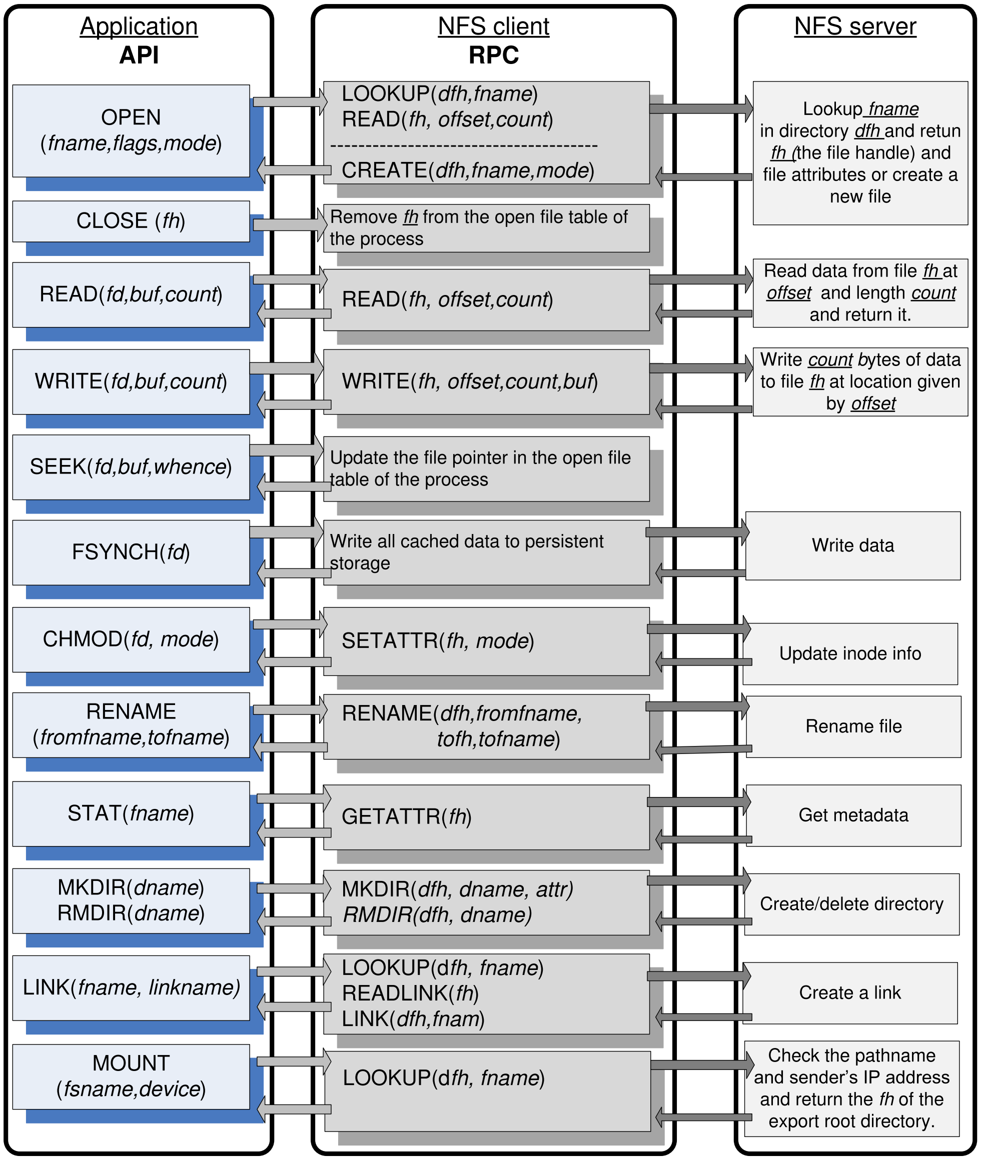

Application API
OPEN (fname,flags,mode)
CLOSE (fh)
READ(fd,buf,count)
WRITE(fd,buf,count)
SEEK(fd,buf,whence)
FSYNCH(fd)
CHMOD(fd, mode)
RENAME (fromfname,tofname)
STAT(fname)
MKDIR(dname) RMDIR(dname)
LINK(fname, linkname)
MOUNT (fsname,device)
NFS client RPC
LOOKUP(dfh,fname) READ(fh, offset,count) CREATE(dfh,fname,mode)
Remove fh from the open file table of the process
READ(fh, offset,count)
WRITE(fh, offset,count,buf)
Update the file pointer in the open file table of the process
Write all cached data to persistent storage
SETATTR(fh, mode)
RENAME(dfh,fromfname, tofh,tofname)
GETATTR(fh)
MKDIR(dfh, dname, attr) RMDIR(dfh, dname)
LOOKUP(dfh, fname) READLINK(fh) LINK(dfh,fnam)
LOOKUP(dfh, fname)
NFS server
Lookup fname in directory dfh and retun fh (the file handle) and file attributes or create a new file
Read data from file fh at offset and length count and return it.
Write count bytes of data to file fh at location given by offset
Write data
Update inode info
Rename file
Get metadata
Create/delete directory
Create a link
Check the pathname and sender's IP address and return the fh of the export root directory.

FIGURE 6.5

The API of UFS and the corresponding RPCs issued by an NFS client to the NFS server. The actions of the server in response to an RPC issued by the NFS client are too complex to be fully described. *fd* stands for file descriptor, *fh* for file handle, *fname* for file name, *dname* for directory name, *dfh* for the directory were the file handle can be found, *count* for the number of bytes to be transferred, *buf* for the buffer to transfer the data to/from, *device* for the device where the file system is located.

generation number. The file handle allows the system to locate the remote file system and the file on that system; the generation number allows the system to reuse the inode numbers and ensures a correct semantics when multiple clients operate on the same remote file.

While many RPC calls, such as *Read*, are idempotent[2] communication failures could sometimes lead to an unexpected behavior. Indeed, if the network fails to deliver the response to a *Read* RPC, then the call can be repeated without any side effects. By contrast, when the network fails to deliver the response to the *Rmdir* RPC, the second call returns an error code to the user if the call was successful the first time; if the server fails to execute the first call then the second call returns normally. Note also that there is no *Close* RPC because this action only makes changes to the open file data structure of the process and does not affect the remote file.

The NFS has undergone significant transformations along the years; it has evolved from Version 2 [437] discussed in this section, to Version 3 [395] in 1994, and then to Version 4 [396] in 2000, see Section 6.14.

Andrew File System. AFS is a distributed file system developed in the late 1980s at Carnegie Mellon University (CMU) in collaboration with IBM [353]. The designers of the systems envisioned a very large number of workstations interconnected with a relatively small number of servers; it was anticipated that each individual at CMU would have an Andrew workstation thus, the system would connect up to 10 000 workstations.

The set of trusted servers in AFS form a structure called Vice. The workstation OS, 4.2BSD UNIX, intercepts file system calls and forwards them to a user-level process called Venus which caches files from Vice and stores modified copies of files back on the servers they came from. Reading and writing operations are performed directly on the cached copy of the file and bypass Venus. Only when a file is opened or closed does Venus communicate with Vice.

The emphasis of the AFS design is on performance, security, and simple management of the file system [242]. The local disks of a workstations act as persistent cache ensuring scalability and reducing the response time. The master copy of a file residing on one of the servers is updated only when the file is modified. This strategy reduces the server load and improves the system performance.

Another major objective of the AFS design is improved security. The communications between clients and servers are encrypted and all file operations require secure network connections. When a user signs in to a workstation the password is used to obtain security tokens from an authentication server; these tokens are then used every time a file operation requires a secure network connection.

AFS uses Access Control Lists (ACLs) to allows control sharing of the data. An ACL specifies the access rights of an individual user or of a group of users. A set of tools support the management of ACLs. Another facet of the effort to reduce the user involvement in the file management is *location transparency*. Files could be accessed from any location and could be moved automatically, or at the request of system administrators, without user's involvement and inconvenience. The relatively small number of servers reduces drastically the efforts related to system administration as operations, such as backups, affect only the servers while workstations can be added, removed, or moved from one location to another without administrative intervention.

Sprite Network File System. SFS is a component of the Sprite network operating system [234]. SFS supports non-write-through caching of files on the client, as well as the server systems [358]. Processes

[2]An action is idempotent if repeating it several times has the same effect as if the action was executed only once.

running on all workstations enjoy the same semantics for file access as if they would run on a single system. This is possible due to a cache consistency mechanism which flushes portions of the cache and disables caching for shared files opened for read-write operations.

Caching not only hides the network latency, but also reduces the server utilization and obviously improves the performance by reducing the responser time. A file access request made by a client process could be satisfied at different levels. First, the request is directed to the local cache; if not satisfied there, it is passed to the local file system of the client. If it cannot be satisfied locally, then the request is sent to the remote server. Finally, when the request cannot be satisfied by the remote server's cache the request is sent to the file system running on the server.

The design decisions for the Sprite system were influenced by the resources available at a time when a typical workstation had a 1 to 2 MIPS processor and 4 to 14 Mbytes of physical memory. The main-memory caches allowed diskless workstations to be integrated in the system and enabled the development of unique caching mechanisms and policies for both clients and servers. The results of a file-intensive benchmark reported by [358] show that SFS was 30% to 35% faster than either NFS or AFS.

The file cache is organized as a collection of 4K blocks; a cache block has a virtual address consisting of a unique file identifier supplied by the server and a block number in the file. Virtual addressing allows the clients to create new blocks without the need to communicate with the server. File servers map virtual addresses to physical disk addresses. Note also that the page size of the virtual memory in Sprite is also 4K. The size of the cache available to an SFS client or a server system changes dynamically function of the needs. This is possible because the Sprite operating system ensures an optimal sharing of the physical memory between file caching by SFS and virtual memory management.

The file system and the virtual memory manage separate sets of physical memory pages and maintain a time-of-last-access for each block or page, respectively. Virtual memory uses a version of the clock algorithm [357] to implement a Least Recently Used (LRU) page replacement algorithm and the file system implements a strict LRU order since it knows the time of each read and write operation. Whenever the file system or the virtual memory management experience a file cache miss or a page fault it compares the age of its oldest cache block or page, respectively, with the age of the oldest one of the other system; the oldest cache block or page is forced to release the real memory frame.

An important design decision of the SFS was to delay write-backs; this means that a block is first written to cache and the writing to the disk is delayed for a time of the order of tens of seconds. This strategy speeds-up writing and also avoids writing when data is discarded before the time to write it to the disk. The obvious draw back of this policy is that data can be lost in case of a system failure. Write-through is the alternative to the delayed write-back. Write-through guarantees reliability as the block is written to the disk as soon as it is available on the cache, but it increases the time for a write operation.

Most network file systems guarantee that once a file is closed, the server will have the newest version on persistent storage. As far as concurrency is concerned we distinguish sequential write-sharing, when a file cannot be opened simultaneously for reading and writing by several clients, from concurrent write-sharing, when multiple clients can modify the file at the same time. Sprite allows both modes of concurrency and delegates the cache consistency to the servers. In case of concurrent write-sharing the client cashing is disabled and all reads and writes are carried out by the server.

Table 6.2 presents a comparison of caching, writing strategy, and consistency of NFS [437], AFS [353], Sprite [234], Locus [507], Apollo [298], and RFS [46].

Table 6.2 A comparison of several network file systems [353].

File system	Cache size and location	Writing policy	Consistency guarantees	Cache validation
NFS	Fixed, memory	On close or 30 sec. delay	Sequential	On open, with server consent
AFS	Fixed, disk	On close	Sequential	When modified server asks client
SFS	Variable, memory	30 sec. delay	Sequential, concurrent	On open, with server consent
Locus	Fixed, memory	On close	Sequential, concurrent	On open, with server consent
Apollo	Variable, memory	Delayed or on unlock	Sequential	On open, with server consent
RFS	Fixed, memory	Write-through	Sequential, concurrent	On open, with server consent

6.4 GENERAL PARALLEL FILE SYSTEM

Once the distributed file systems became ubiquitous, the natural next step in the file systems evolution was supporting parallel access. Parallel file systems allow multiple clients to read and write concurrently from the same file. Support for parallel I/O is essential for the performance of many applications [334]. Early supercomputers such as the Intel Paragon took advantage of parallel file systems to support data-intensive applications.

Concurrency control is a critical issue for parallel file systems. Several semantics for handling shared and concurrent file access are possible. One option is to have a *shared file pointer*. In this case successive reads issued by different clients advance the file pointer. Another semantics is to allow each client to have its own file pointer.

The General Parallel File System (GPFS) [444] was developed by IBM in early 2000s as a successor of the TigerShark multimedia file system [226]. GPFS is a parallel file system emulating closely the behavior of a general-purpose POSIX system running on a single system. GPFS was designed for optimal performance of large clusters. GPFS can support a file system of up to 4 petabytes consisting of up to 4 096 disks of 1 TB each, see Figure 6.6. The maximum file size is $2^{63} - 1$ bytes.

A file consists of blocks of equal size, ranging from 16 KB to 1 MB stripped across several disks. The system could support not only very large files, but also a very large number of files. GPFS directories use the *extensible hashing* techniques to access a file. A hash function is applied to the name of the file; then the n low-order bits of the hash value give the block number of the directory where the file information can be found, with n a function of the number of files in the directory. Extensible hashing is used to add a new directory block. The system maintains user data, file metadata, such as the time when last modified, and file system metadata, such as allocation maps. Metadata, such as file attributes and data block addresses, is stored in inodes and in indirect blocks.

Reliability is a major concern in a system with many physical components. To recover from system failures GPFS records all metadata updates in a write-ahead log file. *Write-ahead* means that updates are written to persistent storage only after the log records have been written. For example, when a new file is created, a directory bloc must be updated and an inode for the file must be created. These records

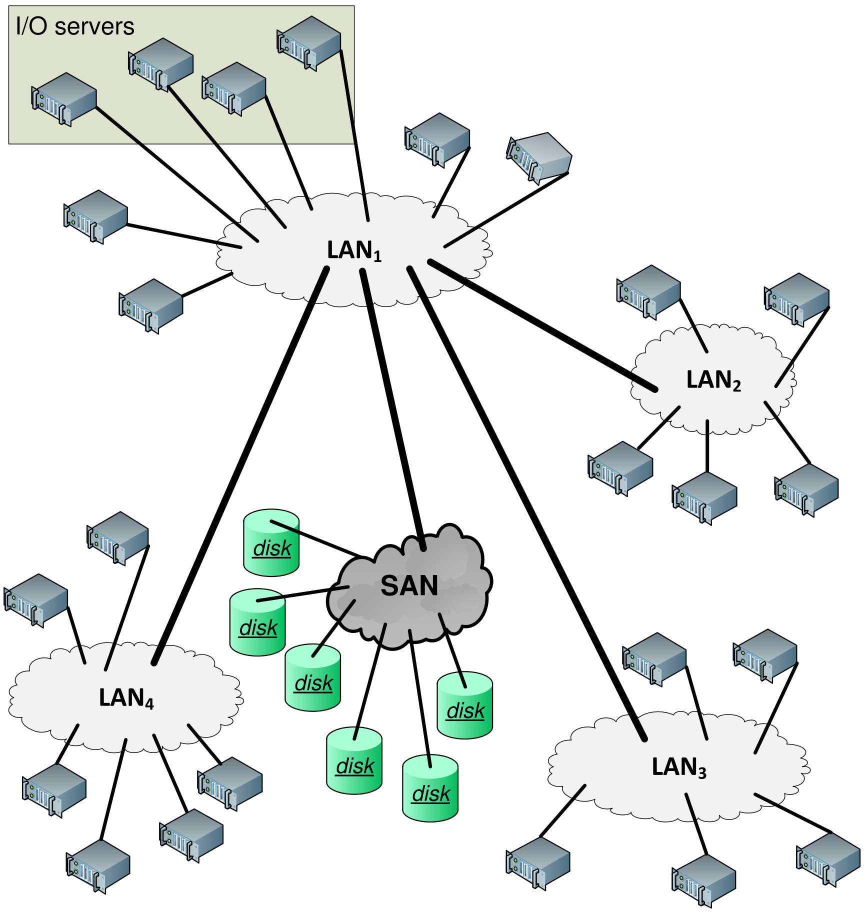


FIGURE 6.6

A GPFS configuration. The disks are interconnected by a SAN; compute servers are distributed in four LANs, LAN$_1$–LAN$_4$. The I/O nodes/servers are connected to LAN$_1$.

are transferred from cache to disk after the log records have been written. When the system ends up in an inconsistent state, the directory bloc is written and then if the I/O node fails before writing the inode, the log file allows the system to recreate the inode record. The log files are maintained by each I/O node for each file system it mounts and any I/O node is able to initiate recovery on behalf of a failed node. Disk parallelism is used to reduce the access time; multiple I/O read requests are issued in parallel and data is pre-fetched in a buffer pool.

Data striping allows concurrent access and improves performance, but can have unpleasant side-effects. Indeed, when a single disk fails, a large number of files are affected. To reduce the impact of such undesirable events, the system attempts to mask a single disk failure or the failure of the access path to a disk. The system uses RAID devices with the stripes equal to the block size and dual-attached RAID controllers. To further improve the fault tolerance of the system GPFS data files, as well as metadata, are replicated on two different physical disks.

Consistency and performance, critical for any distributed file system, are difficult to balance. Support for concurrent access improves performance, but faces serious challenges for maintaining consistency. GPFS consistency and synchronization are ensured by a distributed locking mechanism. A *central lock manager* grants *lock tokens* to *local lock managers* running in each I/O node. Lock tokens are also used by the cache management system.

Lock granularity has important implications on the performance of a file system and GPFS uses a variety of techniques for different types of data. *Byte-range tokens* are used for read and write operations to data files as follows: the first node attempting to write to a file acquires a token covering the entire file, $[0, \infty]$. This node is allowed to carry out all reads and writes to the file without any need for permission until a second node attempts to write to the same file; then, the range of the token given to the first node is restricted. More precisely, if the first node writes sequentially at offset fp_1 and the second one at offset $fp_2 > fp_1$, then the range of the tokens for the two tokens are $[0, fp_2]$ and $[fp_2, \infty]$, respectively, and the two nodes can operate concurrently without the need for further negotiations. Byte-range tokens are rounded to block boundaries.

Byte-range token negotiations among nodes use the *required range* and the *desired range* for the offset and for the length of the current and the future operations, respectively. The *data-shipping*, an alternative to byte-range locking, allows fine-grain data sharing. In this mode the file blocks are controlled by the I/O nodes in a round-robin manner. A node forwards a read or write operation to the node controlling the target block, the only one allowed to access the file.

A *token manager* maintains the state of all tokens; it creates and distributes tokens, collects tokens once a file is closed, downgrades/upgrades tokens when additional nodes request access to a file. Token management protocols attempt to reduce the load place on the token manager; for example, when a node wants to revoke a token it sends messages to all the other nodes holding the token and forwards the reply to the token manager.

Access to metadata is synchronized; for example, when multiple nodes write to the same file, the file size and the modification dates are updated using a *shared write lock* to access an inode. One of the nodes assumes the role of a *metanode* and all updates are channeled through it; the file size and the last update time are determined by the metanode after merging the individual requests. The same strategy is used for updates of the indirect blocks. GPFS global data such as ACLs (Access Control Lists), quotas, and configuration data are updated using the distributed locking mechanism.

GPFS uses *disk maps* for the management of the disk space. The GPFS block size can be as large as 1 MB and a typical block size is 256 KB. A block is divided into 32 sub-blocks to reduce disk fragmentation for small files thus, the block map has 32 bits to indicate if a sub-bloc is free or used. The system disk map is partitioned into n regions and each disk map region is stored on a different I/O node; this strategy reduces the conflicts and allows multiple nodes to allocate disk space at the same time. An *allocation manager* running on one of the I/O nodes is responsible for actions involving multiple disk map regions. For example, it updates free space statistics, helps with deallocation by sending periodically hints of the regions used by individual nodes.

A detailed discussion of system utilities and of the lessons learned from the deployment of the file system at several installations in 2002 can be found in [444]; the documentation of the GPFS is available from [247].

6.5 GOOGLE FILE SYSTEM

The Google File System, developed in late 1990s, uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs [193]. Thus, it should not be surprising that a main concern of the GFS designers was reliability of a system exposed to hardware failures, system software errors, application errors and, last but not least human errors.

The system was designed after a careful analysis of the file characteristics and of the access models. Some of the most important aspects of this analysis reflected in the GFS design are:

- Scalability and reliability are critical features of the system; they must be considered from the beginning, rather than at later design stages.
- The vast majority of files range in size from a few GB to hundreds of TB.
- The most common operation is to append to an existing file; random write operations to a file are extremely infrequent.
- Sequential read operations are the norm.
- Users process the data in bulk and are less concerned with the response time.
- To simplify the system implementation the consistency model should be relaxed without placing an additional burden on the application developers.

As a result of this analysis several design decisions were made:

1. Segment a file in large chunks.
2. Implement an atomic file *append* operation allowing multiple applications operating concurrently to append to the same file.
3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow; schedule the high-bandwidth data flow by pipelining the data transfer over TCP connections to reduce the response time. Exploit network topology by sending data to the closest node in the network.
4. Eliminate caching at the client site; caching increases the overhead for maintaining consistency among cashed copies at multiple client sites and it is not likely to improve performance.
5. Ensure consistency by channeling critical file operations through a master controlling the entire system.
6. Minimize master's involvement in file access operations to avoid hot-spot contention and to ensure scalability.
7. Support efficient checkpointing and fast recovery mechanisms.
8. Support efficient garbage collection mechanisms.

GFS files are collections of fixed-size segments called *chunks*; at the time of file creation each chunk is assigned a unique *chunk handle*. A chunk consists of 64 KB blocks and each block has a 32 bit checksum. Chunks are stored on Linux files systems and are replicated on multiple sites; a user may change the number of the replicas, from the standard value of three, to any desired value. The chunk size is 64 MB; this choice is motivated by the desire to optimize the performance for large files and to reduce the amount of metadata maintained by the system.

A large chunk size increases the likelihood that multiple operations will be directed to the same chunk thus, it reduces the number of requests to locate the chunk and, at the same time, it allows an

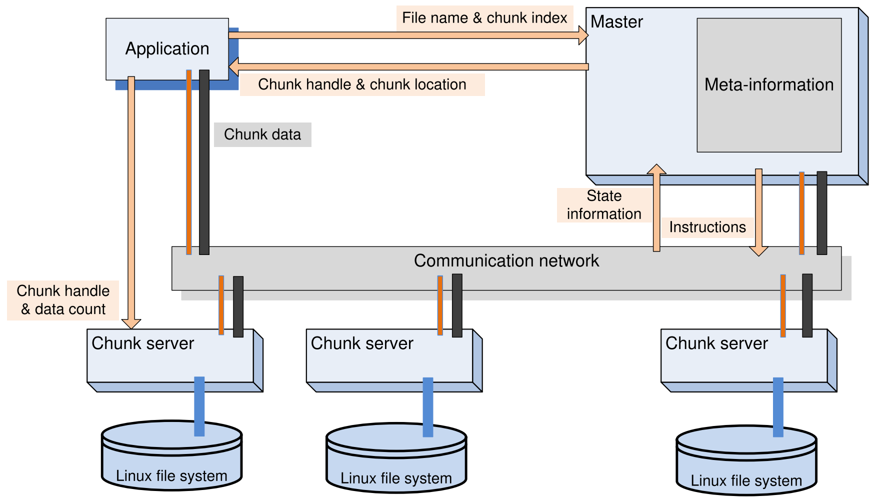


FIGURE 6.7

The architecture of a GFS cluster; the *master* maintains state information about all system components. The master controls a number of *chunk servers*. A chunk server runs under Linux and uses metadata provided by the master to communicate directly with an application. The data flow is decoupled from the control flow. The data and the control paths are shown separately, data paths with thick lines and the control paths with thin lines. Arrows show the flow of control between an application, the master and the chunk servers.

application to maintain a persistent network connection with the server where the chunk is located. Space fragmentation occurs infrequently as the chunk of a small file and the last chunk of a large file are only partially filled.

The architecture of a GFS cluster is illustrated in Figure 6.7. The *master* controls a large number of *chunk servers*; it maintains metadata such as the file names, access control information, the location of all the replicas for every chunk of each file, and the state of individual chunk servers. Some of the metadata is stored in persistent storage, e.g., the *operation log* records the file namespace, as well as the file-to-chunk-mapping.

The locations of the chunks are stored only in the control structure of the master's memory and are updated at the system start up, or when a new chunk server joins the cluster. This strategy allows the master to have up-to-date information about the location of the chunks.

System reliability is a major concern and the operation log maintains a historical record of metadata changes enabling the master to recover in case of a failure. As a result, such changes are atomic and are not made visible to the clients until they have been recorded on multiple replicas on persistent storage. To recover from a failure, the master replays the operation log. To minimize the recovery time, the master periodically checkpoints its state and at recovery time it replays only the log records after the last checkpoint.

Each chunk server is a commodity Linux system. A chunk server receives instructions from the master and responds with status information. For file read or write operations an application sends to the master the file name, the chunk index, and the offset in the file. The master responds with the chunk handle and the location of the chunk. Then the application *communicates directly* with the chunk server to carry out the desired file operation.

The consistency model is very effective and scalable. Operations, such as file creation, are atomic and are handled by the master. To ensure scalability, the master has a minimal involvement in file mutations, operations such as *write* or *append* which occur frequently. In such cases the master grants a lease for a particular chunk to one of the chunk servers called the *primary*; then, the primary creates a serial order for the updates of that chunk.

When data of a *write* operation straddles chunk boundary, two operations are carried out, one for each chunk. The following steps of a *write* request illustrate the process which buffers data and decouples the control flow from the data flow for efficiency:

1. The client contacts the master which assigns a lease to one of the chunk servers for the particular chunk, if no lease for that chunk exists; then, the master replies with the Ids of the primary and the secondary chunk servers holding replicas of the chunk. The client caches this information.
2. The client sends the data to all chunk servers holding replicas of the chunk; each one of the chunk servers stores the data in an internal LRU buffer and then sends an acknowledgment to the client.
3. The client sends the *write* request to the primary chunk server once it has received the acknowledgments from all chunk servers holding replicas of the chunk. The primary chunk server identifies mutations by consecutive sequence numbers.
4. The primary chunk server sends the *write* requests to all secondaries.
5. Each secondary chunk server applies the mutations in the order of the sequence number and then sends an acknowledgment to the primary chunk server.
6. Finally, after receiving the acknowledgments from all secondaries, the primary informs the client.

The system supports an efficient checkpointing procedure based on *copy-on-write* to construct system snapshots. A lazy garbage collection strategy is used to reclaim the space after a file deletion. As a first step, the file name is changed to a hidden name and this operation is time stamped. The master periodically scans the namespace, removes the metadata for the files with a hidden name older than a few days. This mechanism gives a window of opportunity to a user who deleted files by mistake to recover the files with little effort.

Periodically, chunk servers exchange with the master the list of chunks stored on each one of them; the master supplies them with the identity of orphaned chunks, whose metadata has been deleted and such chunks are then deleted. Even when control messages are lost, a chunk server will carry out the house cleaning at the next *heartbeat* exchange with the master. Each chunk server maintains in core the checksums for the locally stored chunks to guarantee data integrity.

CloudStore is an open source C++ implementation of GFS. CloudStore allows client access from C++, Java, and Python.

6.6 LOCKS; CHUBBY – A LOCKING SERVICE

Locks support the implementation of reliable storage for loosely-coupled distributed systems. Locks enable controlled access to shared storage and ensure atomicity of *read* and *write* operations. Consensus protocols are critical for the design of reliable distributed storage systems. The election of a leader or master from a group of data servers requires an effective consensus protocol because the master plays an important role in the management of a distributed storage system. For example, in GFS the master maintains state information about all systems components.

Locking and the election of a master can be done using a version of the Paxos algorithm for asynchronous consensus. The algorithm guarantees safety without any timing assumptions, a necessary condition in a large-scale system when communication delays are unpredictable. Nevertheless, the algorithm must use clocks to ensure liveliness and to avoid the impossibility of reaching consensus with a single faulty process [174]. Coordination and consensus using Paxos are discussed in depth in Sections 7.4 and 3.12, respectively.

Distributed systems experience communication problems, such as lost messages, messages out of sequence, or corrupted messages. There are solutions for handling these undesirable phenomena; for example, one can use virtual time, i.e. sequence numbers, to ensure that messages are processed in an order consistent with the time they were sent by all processes involved, but this complicates the algorithms and increases the processing time.

Advisory locks are based on the assumption that all processes play by the rules; advisory locks do not have any effect on processes that circumvent the locking mechanisms and access the shared objects directly. *Mandatory locks* block access to locked objects to all processes that do not hold the corresponding locks, regardless if a process uses locking primitives or not.

Locks that can be held for only a very short time are called *fine-grained*, while *coarse-grained* locks are held for a longer time. Some operations require meta-information about a lock, such as the name of the lock, whether the lock is shared or held in exclusivity, the generation number of the lock. This meta-information is sometimes aggregated into an opaque byte-string called a *sequencer*. The question how to most effectively support a locking and consensus component of a large-scale distributed system demands several design decisions.

A first decision is whether the locks should be mandatory or advisory. Mandatory locks have the obvious advantage of enforcing access control; a traffic analogy is that a mandatory lock is like a drawn bridge, once it is up all traffic is forced to stop. An advisory lock is like a stop sign, those who obey the traffic laws will stop, but some may not. The disadvantages of mandatory locks are added overhead and less flexibility. Once a data item is locked, even a high priority task related to maintenance or recovery cannot access the data unless it forces the application holding the lock to terminate. This is a very significant problem in large-scale systems where partial system failures are likely.

A second design decision is whether the system should be based on fine-grained or course-grained locking. Fine-grained locks allow more application threads to access shared data, but generate a larger workload for the lock server. Moreover, when the lock server fails for a period of time, a larger number of applications are affected. Advisory locks and course-grained locks seem to be a better choice for a system expected to scale to a very large number of nodes distributed in data centers interconnected via wide area networks with a higher communication latency.

A third design decision is how to support a systematic approach to locking. Two alternatives come to mind:

1. Delegate the implementation of the consensus algorithm to the clients and provide a library of functions needed for this task.
2. Create a locking service implementing a version of the asynchronous Paxos algorithm and provide a library to be linked with an application client to support service calls.

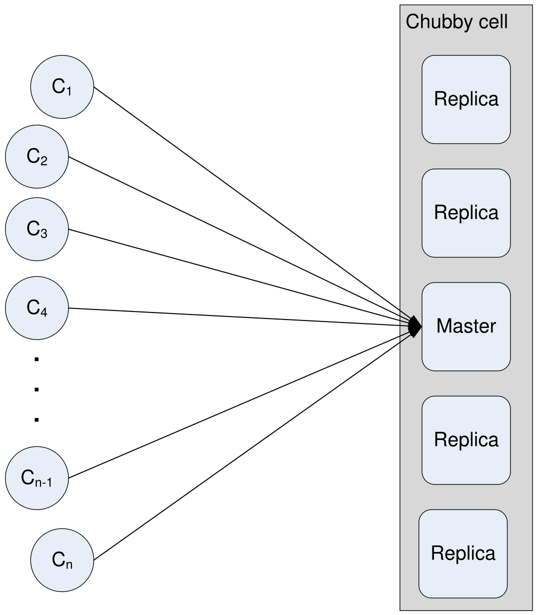


FIGURE 6.8

A Chubby cell consisting of five replicas, one of them is elected as the master. Clients $c_1, c_2, \ldots, c_n$ communicate with the master using RPCs.

Forcing application developers to invoke calls to a Paxos library is more cumbersome and more prone to errors than the service alternative. Of course, the lock service itself has to be scalable to support a potentially heavy load.

Another consideration when making this choice is flexibility, the ability of the system to support a variety of applications. A name service comes to mind as many cloud applications *read* and *write* small files. To allow atomic file operations the names of small files should be included in the namespace of the service. The choice should also consider the performance, a service can be optimized and clients can be allowed to cache control information. Lastly, the overhead and resources for reaching consensus should be considered. Again, the service alternative seems more advantageous as it needs fewer replicas for high availability.

In early 2000s, when Google started to develop a lock service called Chubby [83], it was decided to use advisory locks and coarse-grained locks. The service has been used since by several Google systems including the GFS discussed in Section 6.5 and the BigTable presented in Section 6.9.

A Chubby cell typically serves one data center. The cell server in Figure 6.8 includes several *replicas*; the standard number of replicas is five. To reduce the probability of correlated failures, the servers hosting replicas are distributed across the campus of a data center.

Chubby replicas use a distributed consensus protocol to elect a new *master* when the current one fails. The master is elected by a majority, as required by the asynchronous Paxos algorithm, accompanied by the commitment that another master will not be elected for a period of time, the *master lease.* A session is a connection between a client and the cell server maintained over a period of time. The

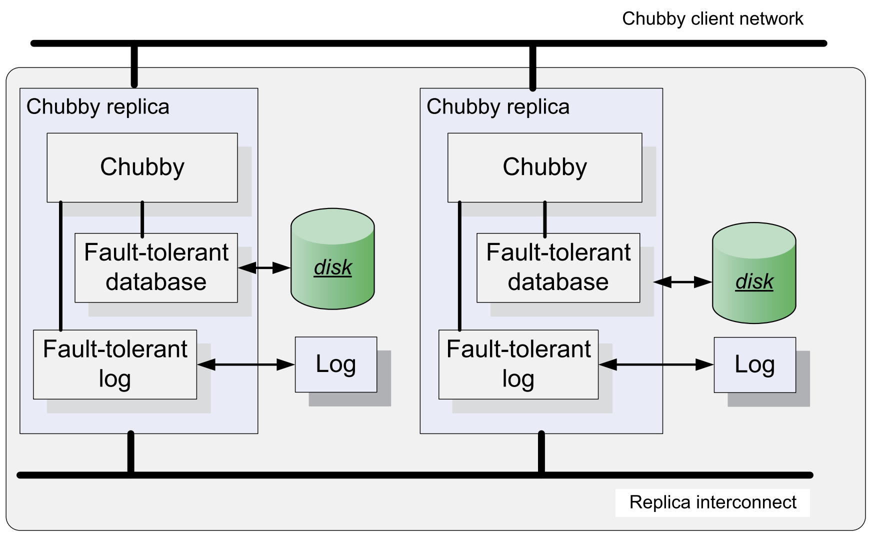


FIGURE 6.9

Chubby replica architecture; a Chubby component implements the communication protocol with the clients. The system includes a component to transfer files to a fault-tolerant database and a fault-tolerant log component to write log entries. The fault-tolerant log uses the Paxos algorithm to achieve consensus. Each replica has its own local file system. Replicas communicate with one another using a dedicated interconnect and communicate with the clients through a client network.

data cached by the client, the locks acquired, and the handles of all files locked by the client are only valid for the duration of the session. Clients use RPCs to request services from the master. When it receives a *write* request, the master propagates the request to all replicas and waits for a reply from a majority of replicas before responding. The master responds without consulting the replicas when receiving a *read* request.

The client interface of the system is similar, yet simpler, than the one supported by the Unix file system. It includes notification for events related to file or system status. A client can subscribe to events such as: file contents modification, change or addition of a child node, master failure, lock acquired, conflicting lock requests, and invalid file handle. The files and directories of the Chubby service are organized in a tree structure and use a naming scheme similar to Unix. Each file has a *file handle* similar to the file descriptor.

The master of a cell periodically writes a snapshot of its database to a GFS file server. Every file or directory can act as a lock. To *write* to a file the client must be the only one holding the file handle, while multiple clients may hold the file handle to *read* from the file. Handles are created by a call to *open()* function and destroyed by a call to *close()*. Other calls supporting the service are *GetContentsAndStat()*, to get the file data and meta-information, as well as *SetContents*, *Delete()*.

Several calls allow the client to acquire and release locks. Some applications may decide to create and manipulate a sequencer with calls to: *SetSequencer()* for associating a sequencer with a handle;

GetSequencer() for obtaining the sequencer associated with a handle; or *CheckSequencer()* for checking the validity of a sequencer.

The sequence of calls *SetContents()*, *SetSequencer()*, *GetContentsAndStat()*, and *CheckSequencer()* can be used by an application for the election of a master. In this process all candidate threads attempt to open a lock file, call it *lfile*, in exclusive mode. The one which succeeds to acquire the lock for *lfile*, becomes the master, writes its identity in *lfile*, creates a sequencer for the lock of *lfile*, call it *lfseq*, and passes it to the server. The other threads read the *lfile* and discover that they are replicas. Periodically, they check the sequencer *lfseq* to determine if the lock is still valid. This example illustrates the use of Chubby as a name server; in fact, this is one of the most frequent uses of the system.

Chubby locks and Chubby files are stored in a replicated database. The architecture of these replicas shows that the stack consists of: (i) the Chubby component implementing the Chubby protocol for communication with the clients; and (ii) the active components writing log entries and files to the local storage of the replica, see Figure 6.9.

An *atomicity log* for a transaction processing system allows a crash recovery procedure to undo all-or-nothing actions that did not complete, or finish all-or-nothing actions that committed but did not record all of their effects. Each replica maintains its own copy of the log; a new log entry is appended to the existing log and the Paxos algorithm is executed repeatedly to ensure that all replicas have the same sequence of log entries.

The next element of the stack is responsible for the maintenance of a fault-tolerant database, in other words to ensure that all local copies are consistent. Fault tolerance enables a system to continue operating properly in the event of one or more faults of its components. The database consists of the actual data, or the *local snapshot* in Chubby speak, and a *replay log* to allow recovery in case of failure. The state of the system is also recorded in the database.

The Paxos algorithm is used to reach consensus on sets of values, e.g., the sequence of entries in a replicated log. To ensure that the Paxos algorithm succeeds, in spite of the occasional failure of a replica, the following three phases of the algorithm are executed repeatedly:

1. Elect a replica to be the master/coordinator. When a master fails, several replicas may decide to assume the role of a master. To ensure that the result of the election is unique, each replica generates a sequence number larger than any sequence number it has seen, in the range $(1, r)$ where r is the number of replicas. Then it broadcasts a *propose* message with this sequence number. The replicas which have not seen a higher sequence number broadcast a *promise* reply and declare that they will reject proposals from other candidate masters. The replica who sent the *propose* message is elected as the master if the number of respondents represent a majority of replicas.
2. The master broadcasts to all replicas an *accept* message including the value it has selected and waits for replies, either *acknowledge* or *reject*.
3. Consensus is reached when the majority of the replicas send the *acknowledge* message; then the master broadcasts the *commit* message.

The implementation of the Paxos algorithm is far from trivial; while the algorithm can be expressed as a few tens of lines of pseudocode, its actual implementation could be several thousand lines of C++ code [94]. Moreover, the practical use of the algorithm cannot ignore the wide variety of failure modes, including algorithm errors, bugs in its implementation, and that testing a software system of a few thousands lines of codes is challenging.

6.7 NOSQL DATABASES

Before the age of cloud computing several data models were widely used: the *hierarchical* model for strictly hierarchical relations, the *network* model for many-to-many relationships, and the most ubiquitous of all, the *relational* model proposed by Codd [113]. Structured Query Language is a special-purpose language for managing structured data in a relational database system and the centerpiece of this storage technology. SQL has three components: a data definition language, a data manipulation language, and a data control language. Oracle, MySQL, SQLServer, and Postgres are the best known examples of Relational Database Management Systems (RDBMS).

Cloud computing brought along the demand for storing unstructured or semi-structured data thus, the need for a new database model. Convenience prevailed when naming this new model. An accurate description of this model, possibly *NO-Relational-database* lacked the appeal, so the name NoSQL was rapidly adopted by the community. This name is misleading. Michael Stonebreaker notes [467] that "blinding performance depends on removing overhead. Such overhead has nothing to do with SQL, but instead revolves around traditional implementations of ACID transactions, multi-threading, and disk management."

With the new model RDBMS names changed; a *partition* became a *shard*,[3] a *table* is a *document root element*, a *row* is an *aggregate/record* and a *column* is an *attribute/field/property*. There is no stand-alone query language for NoSQL databases.

Just to be clear, NoSQL does not reflect an advance in storing technology, but a response to practical needs to efficiently access very large datasets stored on large computer clusters [77]. Four flavors of the new database model can be distinguished:

1. Key-value model data as an index key and a value.
2. Aggregates/documents are similar to key-value, but the value associated with a key contains structured or semi-structured data.
3. Column-family are large sparse tables with a very large number of rows and only a few columns.
4. Graph databases where the nodes represent entities and the edges the relationships among the entities.

Cloud stores such as *document stores* and NoSQL databases are designed to scale well, do not exhibit a single point of failure, have built-in support for consensus-based decisions, and support partitioning and replication as basic primitives. Systems such as Amazon's *SimpleDB* discussed in Section 2.3, *CouchDB* (see http://couchdb.apache.org/), or *Oracle NoSQL database* [381] are very popular, though they provide less functionality than traditional databases. The *key-value* data model is very popular. Several such systems including Voldemort, Redis, Scalaris, and Tokyo cabinet are discussed in [90].

The *soft-state* approach in the design of NoSQL allows data to be inconsistent and transfers the task of implementing only the subset of the ACID properties required by a specific application to the application developer. The NoSQL systems ensure that data will be *eventually consistent* at some future point in time, instead of enforcing consistency at the time when a transaction is "committed."

It was suggested to associate NoSQL databases with the BASE acronym reflecting their relevant properties, Basically Available, Soft state, and Eventually consistent, whereas traditional databases are

[3] A shard is a horizontal partitioning of a database, a row in a table structured data.

characterized by ACID properties, see Section 6.2. Data partitioning among multiple storage servers and data replication are also tenets of the NoSQL philosophy; they increase availability, reduce the response time, and enhance scalability.

6.8 DATA STORAGE FOR ONLINE TRANSACTION PROCESSING SYSTEMS

Many cloud services are based on Online Transaction Processing (OLTP) and operate under tight latency constraints. Moreover, OLTP applications have to deal with extremely high data volumes and are expected to provide reliable services for very large communities of users. It did not take very long for organizations heavily involved in cloud computing such as Google and Amazon, eCommerce companies such as *eBay*, and social media networks such as Facebook, Twitter, or LinkedIn to discover that traditional relational databases are not able to handle the massive amount of data and the real-time demands of online applications critical for their business model.

The search for alternate models to store the data on a cloud is motivated by the need to decrease the latency by caching frequently used data in memory on dedicated servers, rather than fetching it repeatedly. Distributing data to a large number of servers allows multiple transactions to occur at the same time and decreases the response time. The relational schema is of little use for OLTP applications and conversion to key-value databases seems a much better approach. Of course, such systems do not store meaningful metadata information, unless they use extensions that cannot be exported easily.

Reducing the response time is a major concern of OLTP system designers. The term *memcaching* refers to a general purpose distributed memory system that caches objects in main memory. The system is based on a very large hash table distributed across many servers. A *memcached* system is based on a client-server architecture and runs under several operating systems including, Linux, Unix, Mac OS X, and Windows. The servers maintain a key-value associative array. The API allows clients to add entries to the array and to query it; a key can be up to 250 bytes long and a value can be not larger than 1 MB. A *memcached* system uses the LRU cache replacement strategy.

Scalability is the other major concern for cloud OLTP applications and implicitly for datastores. There is a distinction between *vertical scaling*, where the data and the workload are distributed to systems that share resources such as cores/processors, disks, and possibly RAM, and *horizontal scaling*, where the systems do not share either the primary or secondary storage [90].

The overhead of OLTP systems is due to four sources with equal contribution: logging, locking, latching, and buffer management. Logging is expensive because traditional databases require transaction durability thus, every write to the database can only be completed after the log has been updated. To guarantee atomicity, transactions lock every record and this requires access to a lock table.

Many operations require multi-threading and the access to shared data structures, such as lock tables, demands short-term latches[4] for coordination. The breakdown of the instruction count for these operations in existing DBMS is: 34.6% buffer management, 14.2% latching, 16.3% locking, 11.9% logging, and 16.2% for hand-coded optimization [224].

Today OLTP databases could exploit the vast amounts of resources of modern computing and communication systems to store the data in main memory rather than rely on disk-resident B-trees and heap

[4]A latch is a counter that triggers an event when it reaches zero. For example a master thread initiates a counter with the number of worker threads and waits to be notified when all of them have finished.

files, locking-based concurrency control, support for multi-threading optimized for computer technology of past decades [224]. Logless, single threaded, and transaction-less databases could replace the traditional ones for some cloud applications.

Data replication is critical not only for system reliability and availability, but also for its performance. In an attempt to avoid catastrophic failures due to power blackouts, natural disasters, or other causes (see also Section 1.6), many companies have established multiple data centers located in different geographic regions. Thus, data replication must be done over a wide area network. This could be quite challenging especially for log data, metadata, and system configuration information due to larger communication delays and an increased probability of communication failures. Several strategies are possible, some based on Master–Slave configurations, other based on homogeneous replica groups.

Master–Slave replication can be asynchronous or synchronous. In the first case the master replicates write-ahead log entries to at least one slave and each slave acknowledges appending the log record as soon as the operation is done, while in the second case the master must wait for the acknowledgments from all slaves before proceeding. Homogeneous replica groups enjoy shorter latency and higher availability than Master–Slave configurations, any member of the group can initiate mutations which propagate asynchronously.

In summary, the "one-size-fits-all" approach in the traditional storage system design is replaced by a flexible one, tailored to the specific requirements of the applications. Sometimes, the data management of a cloud computing environment integrates multiple databases. For example, Oracle integrates its NoSQL database with the HDFS discussed in Section 7.7, with the Oracle Database, and with the Oracle Exadata. Another approach, discussed in Section 6.10, partitions the data and guarantees full ACID semantics within a partition, while supporting eventual consistency among partitions.

6.9 BIGTABLE

BigTable is a distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of storage servers [96]. The system uses the GFS discussed in Section 6.5 to store user data, as well as system information. To guarantee atomic *read* and *write* operations, BigTable uses the Chubby distributed lock service, see Section 6.6. The directories and the files in the namespace of Chubby are used as locks. Client applications written in C++ can add/delete values, search for a subset of data, and lookup for data in a row.

BigTable is based on a simple and flexible data model and allows an application developer to exercise control on the data format and layout. It also reveals data locality information to the application clients. Column keys identify units of access control called *column families* including data of the same type. A column key consists of a string defining the family name, a set of printable characters, and an arbitrary string as qualifier. A row key is an arbitrary string of up to 64 KB and a row range is partitioned into *tablets* serving as units for load balancing. Any *read* or *write* row operation is atomic even when it affects more than one column. Time stamps used to index different versions of the data in a cell are 64-bit integers. The interpretation of time stamps can be defined by the application, while the default is the time of an event in microseconds. Figure 6.10 shows an example of BigTable, a sparse, distributed, multidimensional map for an Email application.

The system consists of three major components: a library linked to application clients to access the system, a master server and a large number of tablet servers. The master server controls the entire sys-

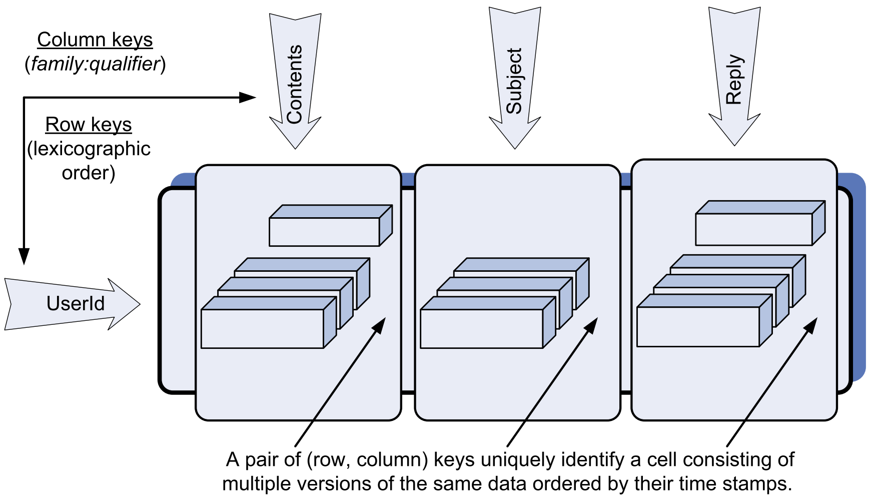

Column keys
(family:qualifier)
Row keys
(lexicographic order)
Contents
Subject
Reply
UserId
A pair of (row, column) keys uniquely identify a cell consisting of multiple versions of the same data ordered by their time stamps.

FIGURE 6.10

A BigTable example; the organization of an Email application as a sparse, distributed, multidimensional map. The slice of the BigTable shown consists of a row with the *UserId* key and three *family* columns; the *Contents* key identifies the cell holding the contents of Emails received, the one with the *Subject* key identifies the subject of Emails, and the one with the *Reply* key identifies the cell holding the replies; the version of records in each cell are ordered according to their time stamps. The row keys of this BigTable are ordered lexicographically; a column key is obtained by concatenating the *family* and the *qualifier* fields. Each value is an uninterpreted array of bytes.

tem, it assigns tablets to tablet servers and balances the load among them, manages garbage collection, and handles table and column family creation and deletion.

Internally, the space management is ensured by a three-level hierarchy: the *root tablet* whose location is stored in a Chubby file, points to entries in the second element, the *metadata* tablet which, in turn, points to *user* tablets, collections of locations of user's tablets. An application client searches through this hierarchy to identify the location of its tablets and then caches the addresses for further use.

The performance of the system reported in [96] is summarized in Table 6.3; the table shows the number of random and sequential *read* and *write* and scan operations for 1 000 bytes, when the number of servers increases from 1 to 50, then to 250, and finally to 500. Locking prevents the system from achieving a linear speedup, but the performance of the system is still remarkable due to a fair number of optimizations. For example, the number of scans on 500 tablet servers is $7\,843 \times 500$ instead of $15\,385 \times 500$. It is reported that only 12 clusters use more than 500 tablet servers, while some 259 clusters use between 1 and 19 tablet servers.

BigTable is used by a variety of applications including Google Earth, Google Analytics, Google Finance, and web crawlers. For example, Google Earth uses two tables, one for preprocessing and one for serving client data. The preprocessing table stores raw images; the table is stored on disk as it contains some 70 TB of data. Each row of data consists of a single imagery; adjacent geographic

Table 6.3 BigTable performance; the number of operations per tablet server.

Number of tablet servers	Random read	Sequential read	Random write	Sequential write	Scan
1	1 212	4 425	8 850	8 547	15 385
50	593	2 463	3 745	3 623	10 526
250	479	2 625	3 425	2 451	9 524
500	241	2 469	2 000	1 905	7 843

segments are stored in rows in close proximity to one another. The column family is very sparse, it contains a column for every raw image. The preprocessing stage relays heavily on MapReduce to clean and consolidate the data for the serving phase. The serving table is stored on GFS, its size is “only” 500 GB, and it is distributed across several hundred tablet servers which maintain in-memory column families. This organization enables the serving phase of *Google Earth* to provide a fast response time to tens of thousands of queries per second.

Google Analytics provides aggregate statistics such as the number of visitors of a web page per day. To use this service web servers embed a JavaScript code in their web pages to record information every time a page is visited. The data is collected in a *raw click* BigTable of some 200 TB with a row for each end-user session. A *summary* table of some 20 TB contains predefined summaries for a website.

6.10 MEGASTORE

Megastore is a scalable storage for online services. The system, distributed over several data centers, has a very large capacity and is highly available. Megastore is widely used internally at Google. In 2011 Megastore had a capacity of 1PB, handled some 23 billion transactions daily, 3 billion *write*, and 20 billion *read* transactions [48].

The basic design philosophy of the system is to partition the data into *entity groups* and replicate each partition independently in data centers located in different geographic areas. The system supports full ACID semantics within each partition and provides limited consistency guarantees across partitions, see Figure 6.11. Megastore supports only those traditional database features that allow the system to scale well and do not affect drastically the response time.

Another distinctive feature of the system is the use of the Paxos consensus algorithm discussed in Section 3.12 for replicating primary user data, metadata, and system configuration information across data centers and for locking. The version of the Paxos algorithm used by Megastore does not require a single master; instead, any node can initiate *read* and *write* operations to a write-ahead log replicated to a group of symmetric peers.

The entity groups are application-specific and store together logically related data; for example, an email account could be an entity group for an Email application. Data should be carefully partitioned to avoid excessive communication between entity groups. Sometimes, it is desirable to form multiple entity groups as is the case of blogs [48].

This middle ground between traditional and NoSQL databases taken by the Megastore designers is also reflected by the data model. The data model is declared in a *schema* consisting of a set of *tables*, composed of *entries*, each entry being a collection of named and typed *properties*. The unique primary

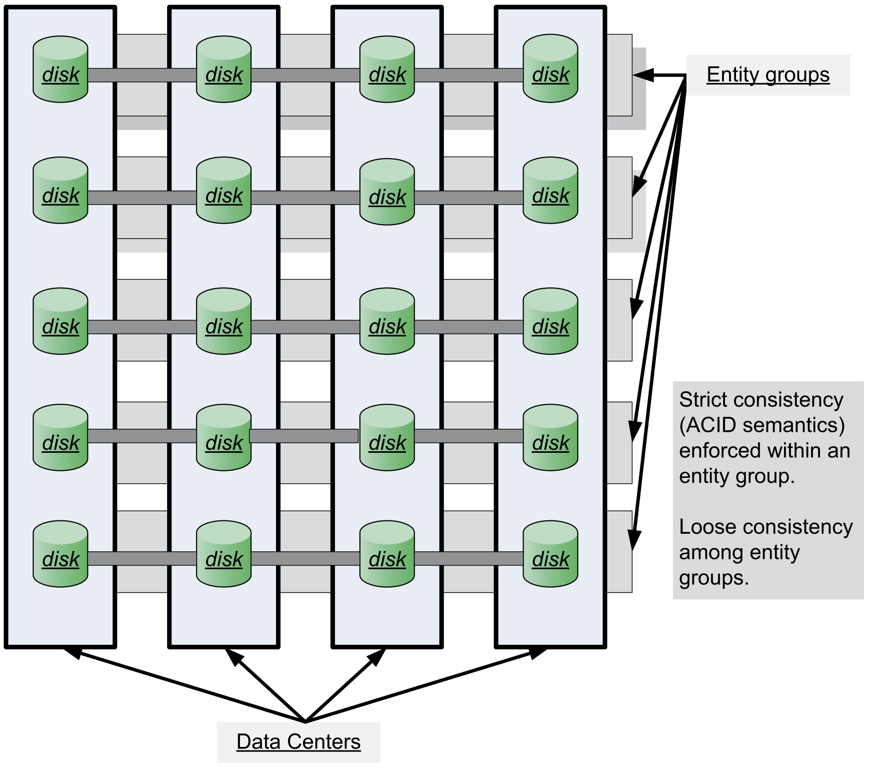

disk
Entity groups
Strict consistency (ACID semantics) enforced within an entity group.
Loose consistency among entity groups.
Data Centers

FIGURE 6.11

Megastore organization. The data is partitioned into *entity groups*; full ACID semantics within each partition and limited consistency guarantees across partitions are supported. A partition is replicated across data centers in different geographic areas.

key of an entity in a table is created as a composition of entry properties. A Megastore table can be a *root* or a *child* table; each *child entity* must reference a special entity, called *root entity* in its root table. An entity group consists of a primary entity and all the entities that reference it.

The system makes extensive use of BigTable. Entities from different Megastore tables can be mapped to the same BigTable row without collisions. This is possible because the BigTable column name is a concatenation of the Megastore table name and the name of a property. A BigTable row for the root entity stores the transaction and all metadata for the entity group. Multiple versions of the data with different time stamps can be stored in a cell as we have seen in Section 6.9.

Megastore takes advantage of this feature to implement *multi-version concurrency control*. When a mutation of a transaction occurs, this mutation is recorded along with its time stamp, rather than marking the old data as obsolete and adding the new version. This strategy has several advantages: *read* and *write* operations can proceed concurrently, a *read* always returns the last fully updated version.

A *write* transaction involves several steps: (1) get the time stamp and the log position of the last committed transaction; (2) gather the *write* operations in a log entry; (3) use the consensus algorithm to append the log entry and then commit; (4) update the BigTable entries; and (5) cleanup.

6.11 STORAGE RELIABILITY AT SCALE

Building reliable systems with unreliable components is a major challenge in system design identified and studied early on by John von Neumann [504]. This challenge is greatly amplified on one hand by the scale of the cloud computing infrastructure and by the use of off-the-shelf components that reduces the infrastructure cost and, on the other hand, by the latency constrains of many cloud applications.

Even though the mean time to failure of individual components can be of the order of month or years, it is unavoidable to witness a small, but significant number of server and network components that are failing at any given time. Data losses cannot be tolerated thus, the failure of storage devices is a major concern. It is left to the software to mask the failures of storage devices and avoid data loss.

Dynamo and DynamoDB. Amazon developed two database systems to support reliability at scale. Dynamo, a highly available key-value storage system has been solely used by AWS core services for in-house applications since 2007 [134]. In 2012 DynamoDB, a NoSQL database service for latency-sensitive applications that need consistent access at any scale, was offered to the AWS user community. Dynamo and DynamoDB use a similar data model, but Dynamo had a multi-master design requiring the client to resolve version conflicts, whereas DynamoDB uses synchronous replication across multiple data centers for high durability and availability.

DynamoDB is a fully-managed database service designed to provide an "always-on" experience. It supports both document and key-value store models and has been used for mobile, web, gaming, IoT, advertising, real-time analytics, and other applications. DynamoDB stores data on SSDs to support latency-sensitive applications; typical requests take milliseconds to complete. DynamoDB allows developers to specify the throughput capacity required for specific tables within their database using the *provisioned throughput* feature to deliver predictable performance at any scale. The service is integrated with other AWS services, e.g., it offers integration with Hadoop via Elastic MapReduce.

Design objectives. As opposed to BigTable, Dynamo's primary concern is high availability where updates are not rejected even in the wake of network partitions or server failures. Dynamo has to deliver predictive performance in addition to reliability and scalability. The services supported by Dynamo have stringent latency requirements and this precludes supporting ACID properties. Indeed, data stores providing ACID guarantees tend to exhibit poor availability.

Some of the most significant design considerations regard data replication and measures to increase availability in wake of failures. Strong consistency and high data availability cannot be achieved simultaneously. Availability can be increased by optimistic replication, allowing changes to propagate to replicas in the background, while disconnected work is tolerated.

In traditional data stores writes may be rejected if the data store cannot reach all, or a majority of the replicas at a given time. This approach is not tolerated by many AWS applications. As a consequence, rather than implementing conflict resolution during writes and keeping the read complexity simple, Dynamo increases the complexity of conflict resolution of the read operations.

Dynamo supports simple read and write operations to data items uniquely identified by a key. The *get(key)* operation locates the object replicas associated with the *key* in the storage system and returns a single object or a list of objects with conflicting versions along with a context. The *put(key, context, object)* operation determines where the replicas of the object should be placed based on the associated key and writes replicas to the secondary storage.

The *context* encodes system metadata about the object that is opaque to the caller and includes information such as the version of the object. The context information is stored along with the object so that the system can verify the validity of the context object supplied in the put request.

The main techniques used to achieve Dynamo's design objectives are:

1. *Incremental scalability* ensured by consistent hashing.
2. *High write availability* based on the use of vector clocks with reconciliation.
3. *Handling temporary failures* using sloppy quorum and hinted handoff. This provides high availability and durability guarantees when some of the replicas are not available.
4. *Permanent failure recovery* based on anti-entropy and Merkle trees. This technique synchronizes divergent replicas in the background.
5. *Gossip-based membership protocol and failure detection* for membership and failure detection. The advantage of this technique is that it preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

These techniques are discussed next.

Scaling, load balancing, and replication. The data partitioning scheme designed to support incremental scaling of the system is based on *consistent hashing*. The output of a hash function is treated as a ring and each node in the system is assigned a random value within this space representing its position on the ring.

Consistent hashing reduces the number of keys to be remapped when a hash table is resized. On average only K/n keys need to be remapped, with K the number of keys and n the number of slots. In most traditional hash tables a change in the number of slots causes nearly all keys to be remapped because the mapping between the keys and the slots is defined by a modular operation.

A data item identified by a key is assigned to a storage server by hashing the data item key to yield its position on the ring, and then walking the ring clockwise to find the first node with a position larger than the position of the item. Each storage server is responsible for the ring region between itself and its predecessor on the ring, see Figure 6.12A. In this example node B replicates key k at nodes C and D in addition to storing it locally. Node D will store the keys that fall in the ranges $(A, B]$, $(B, C]$, and $(C, D]$.

Instead of mapping a storage server to a single point in the ring, the system uses the concept of "virtual nodes" and assigns it to multiple points in the ring. A physical server is mapped to multiple nodes of the ring. This form of virtualization supports:

1. Load balancing. When a storage server is unavailable its load is dispersed among available servers. When the server comes back again, it is added to the system and accepts a load roughly equivalent to the load of other servers.
2. System heterogeneity. The number of virtual nodes a physical server is mapped to depends on its capacity.

A data item is replicated at N servers. Each key, k, is assigned to a coordinator charged with the replication of the data items in its range. In addition to locally storing each key within its range, the coordinator replicates these keys at the $N - 1$ clockwise successor nodes in the ring. In this manner

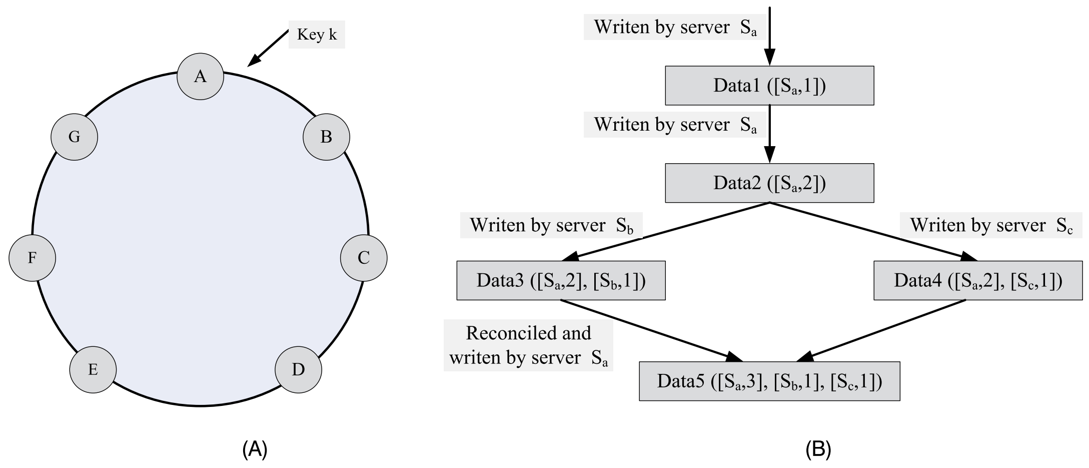


FIGURE 6.12

(A) The servers of the Dynamo service are organized as a ring. Ring nodes B, C, and D store keys in range (A, B), including the key k. (B) The evolution of an object in time using vector clocks.

each node is responsible for the region of the ring between it and its N-th predecessor. The *preference list* is the list of all nodes responsible for storing a particular key.

Eventual consistency. This strategy allows updates to be propagated to all replicas asynchronously. A versioning system allows multiple versions of a data object to be present in the data store at the same time. The result of each modification is a new and immutable version of the data. New versions often subsume the older ones and the system can use syntactic reconciliation to determine the authoritative version.

The system uses *vector clocks*, lists of (node, counter) pairs, to capture causality of each version of a data object. When a client updates an object, it must specify the version it is updating by passing the context it obtained from an earlier read operation, which contains the vector clock information. System failures combined with concurrent updates lead to conflicting versions of an object and version branching. Given two versions of the same object, the first is an ancestor of the second and can be forgotten if the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock; otherwise, the two changes are in conflict and require reconciliation.

Figure 6.12B illustrates the versioning mechanism for the following sequence of events. Data is written by server S_a and the object *Data1* with the associated clock $[S_a, 1]$ is created. The same server S_a writes again and the object *Data2* with the associated clock $[S_a, 2]$ is created. *Data2* is a descendent of *Data1* and overwrites it. There may be replicas of *Data1* at servers that have not yet seen *Data2*. Then the same client updates the object and server S_b handles the request; a new object data *Data3* and its associated clock $[(S_a, 2), (S_b, 1)]$ are created.

A different client reads *Data2* and then tries to update it, and this time server S_c handles her request. A new object *Data4*, a descendent of *Data2*, with version clock is $[(S_a, 2), (S_c, 1)]$ is created. Upon receiving *Data4* and its clock, a server aware of *Data1* or *Data2* could determine that both are overwritten by the new data and can be garbage collected.

A node aware of *Data3* and *Data4* will see that there is no causal relation between them as there are changes not reflected in each other. Both versions of the data must be kept and presented to a client for semantic reconciliation. If a client reads both *Data3* and *Data4* its context will be $[(S_a, 2), (S_b, 1), (S_c, 1)]$, the summary of the virtual clocks of both data objects. If the client performs a reconciliation and the write request is handled by server S_a then the vector clock of the new data, *Data5*, will be $[(S_a, 3), (S_b, 1), (S_c, 1)]$. The size of the vector clock grows, but in practice this growth is limited.

Sloppy quorum for handling failures. During server failures and network partitions a strict quorum membership is enforced by traditional systems. This conflicts with the durability requirement and in Dynamo all read and write operations are performed on the first N *healthy nodes* from the preference list. In this *sloppy quorum* the healthy nodes may not always be the first N nodes encountered while walking the consistent hashing ring.

For example, to maintain the desired availability and durability guarantees when node A in Figure 6.12A is unreachable during a write operation, a replica that would normally have been sent to A will be sent to D. The metadata of this replica will include a hint indicating the intended recipient of the replica.

Replica synchronization in case of permanent failures. Replica inconsistencies can be detected faster and with a minimum of data transferred using Merkle trees.[5] This allows each branch of the tree to be checked independently without the need to download the entire tree. For example, if the hash values of the root of two trees are equal, then the values of the leaf nodes in the trees are equal and the nodes require no synchronization.

Anti-entropy is a process of comparing the data of all replicas and updating each replica to the newest version; Merkle trees are used for anti-entropy. The key range is the set of keys covered by a virtual node. Each node maintains a separate Merkle tree for each key range and two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common to compare whether the keys within a key range are up-to-date.

Gossip-based node addition to the ring or removal from the ring. The node receiving the request writes the change and its time of issue to persistent store. A gossip-based protocol propagates membership changes and maintains an eventually consistent view of the membership. Each node contacts a peer chosen at random every second and the two nodes reconcile their persisted membership change histories. For example, consider the case when a new node Q is added between nodes A and B to the ring in Figure 6.12A. Now node Q will be storing keys in the ranges $(F, G]$, $(G, A]$ and $(A, Q]$ freeing nodes B, C and D from storing the keys in these ranges. Upon confirmation from Q, nodes B, C, and D will transfer the appropriate set of keys to it. When a node is removed from the system, the reallocation of keys proceeds as a reverse process.

[5]A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children.

6.12 DISK LOCALITY VERSUS DATA LOCALITY IN COMPUTER CLOUDS

Locality is critical for the performance of computing systems. Recall that a sequence of references is said to have spatial locality if the items referenced within a short time interval are close in space, e.g., they are at nearby memory addresses or nearby sectors on a disk. A sequence exhibits temporal locality if accesses to the same item are clustered in time.

Locality has major implications for memory hierarchies. The performance of a processor is highly dependent on the ability to access code and data in cache rather than memory. Virtual memory can only be effective if the code and the data exhibit spatial and temporal locality, so that page faults occur infrequently.

Optimizing locality in cloud computing a challenging problem. We shall see in Chapters 7, 8, and 9 that a fair amount of effort has been devoted to algorithms and systems designed to increase the fraction of the tasks of a job enjoying locality. Locality, the availability of data on the server where a task is running, improves the performance of cloud applications for two main reasons:

1. The bandwidth of disks is larger than the network bandwidth; moreover, the off-rack communication bandwidth is oversubscribed and affects the off-rack disk access.
2. The better performance of I/O-intensive applications when data are stored locally is due to the lower latency and the higher bandwidth of a local disk versus the latency and the bandwidth of a remote disk.

An important question for cloud resource management is whether *disk locality* is important. Intuitively, we expect the answer to the question whether tasks should be dispatched to the cluster nodes where their input data resides should be a resounding “Yes.” A slightly different view on the subject of disk locality is expressed in a paper with a blunt title “Disk-Locality in Datacenter Computing Considered Irrelevant” [28].

Maybe we tried to solve the wrong problem, instead of focusing on disk locality we should focus on data locality. In other words, we should look at the local memory as a “data cache” and make sure that data is stored in the local memory rather than the local disk of the processor where the task is scheduled to run. Of course, data transfer through the network may still be necessary, but why store it on the disk and then load it in memory? Several arguments support this thesis:

- The networking technology improves at a faster pace than hard disk technology. Switches with aggregate link speeds of 40 Gbps and 100 Gbps are available today. Server network interfaces will soon support rates of 10 Gbps and 25 Gbps.
- The bandwidth available to applications will increase as data centers adopt bisection topologies for their interconnects [56]. Recall that the bisection bandwidth is the sum of the bandwidths of the minimal number of links that are cut when splitting the system into two parts.
- The latency of a read access to a local disk is only slightly lower than the latency of a read to a disk in the same rack; local disk access is about 8% faster [56]. Access to a disk in a different rack will not increase drastically due to faster networks.
- Though the cost of solid state disk technology is climbing down at an impressive rate, 50% per year, SSDs are unlikely to replace hard disk drives any time soon due to the sheer volume of data stored on computer clouds. To be competitive with HDDs the cost-per-byte of SSD should be reduced by up to three orders of magnitude.

- Accessing data in local memory is two orders of magnitude faster than reading from a local disk. An increasing number of applications use the processor memory distributed across the nodes of large clusters to cache the data, rather than access the HDDs.
- There is at least a two orders of magnitude discrepancy between the capacity of the disks and the memory of today's clusters thus, it seems reasonable to use processor memory as a cache for the much larger volume of data stored on the disks.

It makes sense to use processor memory as a cache for large data sets used as the input of an application if and only if: (1) the size of the application's input is much larger than the memory size; and (2) the applications exhibit locality and have a modest sized working set, in other words if they access frequently only a relatively small fraction of data blocks.

The authors of [28] investigated Hadoop tasks running at a data center with some 3 000 machines interconnected by three-tiered networks. Most of the jobs at the Facebook center are data-intensive and spent most of their execution time reading input data. The analysis of job traces led to the conclusion that there is an order of magnitude discrepancy between the input size and the memory size and that some 75% of the data blocks are accessed only once.

The analysis also shows that workloads have a heavy tail distribution of block access. Moreover, 96% of the data inputs can fit into a fraction of the total cluster memory for a majority of jobs. Some 64% of all jobs investigated perform well under a least frequently used (LFU) replacement policy applied to all their tasks.

The *task progress report*, $\mathbb{T}$, measures the effect of locality on the duration of a task and is defined as the ratio

$$\mathbb{T} = \frac{data\_read + data\_written}{task\_duration}. \tag{6.1}$$

The results of measurements comparing node-local versus rack-local tasks show that 85% of the jobs have $0.9 \leq \mathbb{T} \leq 1.1$ and only 4% of jobs have $\mathbb{T} \leq 0.7$ thus, rack-local tasks are unlikely to significantly affect performance.

Data compression reduces the pressure on the data center disk space. Tasks read compressed data and uncompress it before processing. In spite of a network oversubscribed by a factor of ten at Facebook, data compression leads to very good results; running a task off-rack is only 1.2 times to 1.4 times slower compared with on-rack execution. The log analysis of Hadoop jobs suggests that prefetching data blocks could be beneficial because a large fraction of data blocks are accessed only once.

6.13 DATABASE PROVENANCE

Data *provenance* or *lineage* describes the origins and the history of data and adds value to data by explaining how it was obtained. The lineage of a tuple $\mathcal{T}$ in the result of a query is the set of items contributing to produce $\mathcal{T}$. The lineage is important for the extract-transform-load process and for incrementally adding and updating a database.

Before the Internet era the information in databases was trusted, it was assumed that the organization maintaining the database was trustworthy and that every effort to ensure the veracity of data was made. This assumption is no longer true, there is no centralized control over data integrity as the

Internet allows data to be created, copied, moved around, and combined indiscriminately. Establishing data provenance is necessary for all databases and is also critical for cloud databases as the data owners relinquish control of their data to the CSPs.

Data provenance has been practiced by the scientific and engineering community for some time, long before the disruptive effects of data democratization brought about by the Internet. Data collected by scientific experiments contains information about the experimental setup and the settings of measuring instruments for each batch of data. Ensuring that experiments can be replicated has always been an essential aspect of scientific integrity. The same requirements apply to engineering when test data, e.g., data collected during the testing of a new aircraft, include lineage information.

Data provenance could show inputs that explain *why* an output record was produced, describing in detail *how* the record was produced, and/or explaining *where* output data cames from. The why-, how-, and where-provenance are analyzed in [107] and discussed briefly in this section.

The *witness* of a database record is the subset of database records ensuring that the record is the output of a query. The *why-provenance* includes information about the witnesses to a query. The number of witnesses can be exponentially large. To limit this number the concept of *witness bases* of tuple $\mathcal{T}$ in query $\mathcal{Q}$ on database $\mathcal{D}$ is defined as the particular set of witnesses which can be calculated efficiently from $\mathcal{Q}$ and $\mathcal{D}$.

The *why-provenance* of an output tuple $\mathcal{T}$ is the witness basis of $\mathcal{I}$ according to $\mathcal{Q}$. The witness basis depends on the structure of the query and it is sensitive to the query formulation. Let us now take a short detour for introducing the *Datalog conjunctive query* notations before presenting an example showing that the why-provenance is sensitive to query rewriting.

Datalog is a declarative logic programming language. Query evaluation in Datalog is based on first order logic thus, it is sound and complete. A Datalog program includes *facts* and *rules*. A rule consists of two elements, the head and the body, separated by the ":-"symbol. A rule should be understood as: *"head" if it is known that "body"*. For example,

- the facts in the left box below mean: (1) Y is in relation R with X; (2) Z is in relation R with Y.
- the rules in the right box mean: (1) Y is in relation P with X if it is known that Y is in relation R with X; (2) Y is in relation P with X if it is known that Z is in relation R with X AND rule P is satisfied, i.e., Y is in relation P with Z.

```
R(X,Y).
R(Y,Z).
```

```
P(X,Y) :- R(X,Y).
P(X,Y) :- R(X,Z), P(Z,Y)
```

Consider now an instance I defining relation R and three tuples t, t', t'', two queries Q, Q' and the output of the two queries $Q(I)$, $Q'(I)$. The two queries are

$$\begin{aligned} Q &: Ans(x, y) : -R(x, y). \\ Q' &: Ans(x, y) : -R(x, y), R(x, z) \end{aligned} \tag{6.2}$$

Table 6.4 shows that queries Q and Q' are equivalent, they produce the same result. Table 6.5 shows that the why-provenance is sensitive to query rewriting. There exists a subset of the witness basis invariant under equivalent queries. This subset, called *minimal witness basis*, includes all *minimal*

Table 6.4 The two queries Q and Q' are equivalent under R.

Instance I			Output of Q(I)	
R	**A**	**B**	**A**	**B**
t:	1	2	1	2
t':	1	3	1	3
t'':	4	2	4	2

Table 6.5 The why-provenance of the two equivalent queries Q and Q' are different for the three tuples t, t' and t''.

Instance I			Output of $Q(I)$			Output of $Q'(I)$		
R	**A**	**B**	**A**	**B**	**why**	**A**	**B**	**why**
t:	1	2	1	2	$\{\{t\}\}$	1	2	$\{\{t\},\{t,t''\}\}$
t':	1	3	1	3	$\{\{t'\}\}$	1	3	$\{\{t'\},\{t,t''\}\}$
$(t'')^2$:	4	2	4	2	$\{\{(t'')^2\}\}$	4	2	$\{\{(t'')^2\}\}$

Table 6.6 The how-provenance of the two equivalent queries Q and Q' are different for the three tuples t, t' and t''.

Instance I			Output of $Q(I)$			Output of $Q'(I)$		
R	**A**	**B**	**A**	**B**	**how**	**A**	**B**	**how**
t:	1	2	1	2	t	1	2	$t^2 + t \cdot t'$
t':	1	3	1	3	t'	1	3	$(t')^2 + t \cdot t'$
t'':	4	2	4	2	t''	4	2	$(t'')^2$

witnesses in the witness basis. A witness is minimal if none of its proper sub-instances is also a witness in the witness basis. For example, $\{t\}$ is a minimal witness for the output tuple $(1, 2)$ in Table 6.5 but $\{t, t'\}$ is not a minimal witness since $\{t\}$ is a sub-instance of it and it is a witness to $(1, 2)$. Hence, the minimal witness basis is $\{t\}$ in this case.

The *why-provenance* describes the source tuples that witness the existence of an output tuple in the result of the query, but it does not show how an output tuple is derived according to the query. The how-provenance is more general than the why-provenance, thus, it is also sensitive to query formulation as shown in Table 6.6. The how-provenance of the tuple $(1, 2)$ in the output of Q is t but it is $t^2 + t \cdot t'$ for Q'.

The *where-provenance* describes the relationship between the source and the output locations, while the why-provenance describes the relationship between source and output tuples. Examples presented in [107] illustrate the fact that the where-provenance is also sensitive to query formulation.

6.14 HISTORY NOTES AND FURTHER READINGS

A 1989 survey of distributed file systems can be found in [440]. NFS Versions 2, 3, and 4 are defined in RFCs 1094, 1813, and 3010, respectively. NFS Version 3 added a number of features including:

support for 64-bit file sizes and offsets, support for asynchronous writes on the server, additional file attributes in many replies, and a *READDIRPLUS* operation. These extensions allowed the new version to handle files larger than 2 GB, to improve performance, and to get file handles and attributes along with file names when scanning a directory. NFS Version 4 borrowed a few features from the Andrew file system. WebNFS is an extension of NFS Versions 2 and 3; it enables operations through firewalls and is easier integrated into Web browsers.

AFS was further developed as an open-source system by IBM under the name OpenAFS in 2000. Locus [507] was initially developed at UCLA in the early 1980s and its development was continued by Locus Computing Corporation. Apollo [298] was developed at Apollo Computer Inc, established in 1980 and acquired in 1989 by HP. The Remote File System (RFS) [46] was developed at Bell Labs in mid 1980s. The documentation of a current version of GPFS and an analysis of the caching strategy are given in GPFS [247] and [444], respectively.

Several DBMS generations based on different models have been developed along the years. In 1968 IBM released the Information Management System (IMS) for the IBM 360 computers. IMS was based on the so-called *navigational model* supporting manual navigation in a linked data set where the data is organized hierarchically. In the RDBMS model, introduced by Codd, related records are linked together and can be accessed using a unique *key*. Codd also introduced a *tuple calculus* as a basis for a query model for a RDBMS; that led to the development of the Structured Query Language.

In 1973, the Ingres research project at U. C. Berkeley developed a relational data base management system. Several companies including Sybase, Informix, NonStop SQL, and Ingres were established to create SQL RDBMS commercial products based on the ideas generated by the Ingres project. IBM's DB2 and SQL/DS dominated the RDBMS market for mainframes during the later years of 1980s. Oracle Corporation, founded in 1977, was also involved in the development of RDBMS.

ACID properties of database transactions were defined by Jim Gray in 1981 [201] and the term ACID was introduced in [223]. The object-oriented programming ideas of the 1980s led to the development of Object-Oriented Data Base Management Systems (OODBMS) where the information is packaged as objects. The ideas developed by several research projects, including Encore-Ob/Server at Brown University, Exodus at the University of Wisconsin-Madison, Iris at HP, ODE at Bell Labs, and the Orion project at MCC-Austin, helped the development of several OODBMS commercial products [269].

NoSQL database management systems emerged in the 2000s. They do not follow the RDBMS model, do not use SQL as a query language, may not give ACID grantees, and have a distributed, fault-tolerant architecture.

Further readings. A 2011 article in the journal Science [233] discusses the volume of information stored, processed, and transferred through the networks. [354] is a comprehensive study of the storage technology until 2003. The evolution of storage technology is presented in [248].

Network File System Versions 2, 3, and 4 are discussed in [437], [395], and [396], respectively. [353] and [242] provide a wealth of information about the Andrew File System, while [234] and [358] discuss in detail the Sprite File System. Other file systems such as Locus, Apollo, and the Remote File System (RFS) are discussed in [507], [298], and [46], respectively. The recovery in the Calypso file system is analyzed in [144]. The Lustre file system is analyzed in [380].

The General Parallel File System (GPFS) developed at IBM and its precursor, the TigerShark multimedia file system are presented in [444] and [226]. A good source for information about the Google Files System is [193]. Main memory OLTP recovery is covered in [322]. The development of Chubby is covered by [83]. NoSQL databases are analyzed in several papers including [467], [224], and [90]. BigTable and Megastore developed at Google are discussed in [96] and [48]. An evaluation of distributed data store is reported [80].

Attaching cloud storage to a campus grid is the subject of [151]. A cost analysis of storage in enterprise is discussed in [412] and [464] is an insightful discussion of main-memory OLTP databases. [503] presents VMware storage.

6.15 EXERCISES AND PROBLEMS

Problem 1. Analyze the reasons for the introduction of storage area networks (SANs) and their properties. *Hint:* read [354].

Problem 2. Block virtualization simplifies the storage management tasks in SANs. Provide solid arguments in support of this statement. *Hint:* read [354].

Problem 3. Analyze the advantages of memory-based checkpointing. *Hint:* read [258].

Problem 4. Discuss the security of distributed file systems including SUN NFS, Apollo Domain, Andrew, IBM AIX, RFS, and Sprite. *Hint:* read [440].

Problem 5. The designers of the Google file system (GFS) have re-examined the traditional choices for a file system. Discuss the four observations regarding these choices that have guided the design of GFS. *Hint:* read [193].

Problem 6. In his seminal paper on the virtues and limitations of the transaction concept [201] Jim Gray analyzes logging and locking. Discuss the main conclusions of his analysis.

Problem 7. Michael Stonebreaker argues that “blinding performance depends on removing overhead...” Discuss his arguments regarding the NoSQL concept. *Hint:* read [467].

Problem 8. Discuss the Megastore data model. *Hint:* read [48].

Problem 9. Discuss the use of locking in the BigTable. *Hint:* read [96] and [83].

SECTION

3

CHAPTER 7

CLOUD APPLICATIONS

The users of large-scale computing systems discovered along the years how difficult it was to develop efficient data-intensive and computationally-intensive applications. It was equally difficult to locate systems best suited to run an application, to determine when an application was able to run on these systems, and to estimate when results could be expected. Porting an application from one system to another frequently represented a challenging endeavor. Often, an application optimized for one system performed poorly on other systems.

There were also formidable challenges for the providers of service because system resources could not be effectively managed and it was not feasible to deliver QoS guarantees. Accommodating a dynamic load, supporting security and rapid recovery after a system-wide failure, were daunting tasks due to the scale of the system. Any economic advantage offered by resource concentration was offset by the relatively low utilization of costly resources.

Cloud computing changed the views and perceptions of users and providers of service on how to compute more efficiently and at a lower cost. Most of the challenges faced by application developers and service providers either disappeared, or are significantly diminished. Cloud application developers enjoy the advantages of a *just-in-time infrastructure*, they are free to design an application without being concerned where the application will run.

Cloud elasticity allows applications to seamlessly absorb additional workload. Cloud users also benefit from the speedup due to parallelization. When the workload can be partitioned in n segments, the application can spawn n instances of itself and run them concurrently resulting in dramatic speedups. This is particularly useful for computer-aided design and for modeling complex systems when multiple design alternatives and multiple models of a physical system can be evaluated at the same time.

Cloud computing is focused on enterprise computing, this clearly differentiates it from the grid computing effort largely focused on scientific and engineering applications. An important advantage of cloud computing over grid computing is that the resources offered by a cloud service provider are in one administrative domain.

Cloud computing is beneficial for the providers of computing cycles as it typically leads to a more efficient resource utilization. It soon became obvious that a significant percentage of the typical workloads are dominated by frameworks such as MapReduce and that multiple frameworks must share the large computer clusters populating the cloud infrastructure.

The future success of cloud computing rests on the ability of companies promoting utility computing to convince an increasingly larger segment of user population of the advantages of network-centric computing and network-centric content. This translates into the ability to provide satisfactory solutions to critical aspects of security, scalability, reliability, quality of service, and the requirements agreed upon in SLAs.

Sections 7.1, 7.2, and 7.3 cover cloud application developments and provide insights into workflow management. Coordination based on a state machine model is presented in Section 7.4 followed by the

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00010-8

in-depth discussion of the MapReduce programming model and one of its applications in Sections 7.5 and 7.6. Hadoop, Yarn, and Tez are covered in Section 7.7, while systems such a Pig, Hive, and Impala are discussed in Section 7.8.

An overview of current cloud applications and new application opportunities is presented 7.9 followed by a discussion of cloud applications in science and engineering and in biology research in Sections 7.10 and 7.11, respectively. Social computing and software fault isolations are the subjects of Sections 7.12 and 7.13.

7.1 CLOUD APPLICATION DEVELOPMENT AND ARCHITECTURAL STYLES

Web services, database services, and transaction-based services are ideal applications for cloud computing. The cost-performance profiles of such applications benefit from an elastic environment where resources are available when needed and where a cloud user pays only for the resources consumed by her application.

Not all types of applications are suitable for cloud computing. Applications where the workload cannot be arbitrarily partitioned, or require intensive communication among concurrent instances are unlikely to perform well on a cloud. An application with a complex workflow and multiple dependencies, as is often the case in high-performance computing, could experience longer execution times and higher costs on a cloud. Benchmarks for high-performance computing discussed in Section 7.10 show that communication-intensive and memory-intensive applications may not exhibit the performance levels shown when running on supercomputers with low latency and high bandwidth interconnects.

Cloud application development challenges. The development of efficient cloud applications faces challenges posed by the inherent imbalance between computing, I/O, and communication bandwidth of processors. These challenges are greatly amplified due to the scale of the cloud infrastructure, its distributed nature, and by the very nature of data-intensive applications. Though cloud computing infrastructures attempt to automatically distribute and balance the workload, application developers are still left with the responsibility to identify optimal storage for the data, exploit spatial and temporal data and code locality, and minimize communication among running treads and instances.

A main attraction of cloud computing is the ability to use as many servers as necessary to optimally respond to the cost and the timing constraints of an application. This is only possible if the workload can be partitioned in segments of arbitrary size and can be processed in parallel by the servers available in the cloud. The *arbitrarily divisible load sharing model* describes workloads that can be partitioned into an arbitrarily large number of units and can be processed concurrently by multiple cloud instances. Applications most suitable for cloud computing enjoy this model.

The shared infrastructure, a defining quality of cloud computing, has side effects. Performance isolation discussed in Section 10.1 is nearly impossible under real conditions, especially when a system is heavily loaded. The performance of virtual machines fluctuates based on the workload and the environment. Security isolation is challenging on multi-tenant systems.

Reliability of the cloud infrastructure is also a major concern. The frequent failures of servers built with off-the-shelf components is a consequence of the sheer number of servers and communication systems. Choosing an optimal instance from those offered by the cloud infrastructure is another critical factor to be considered. Instances differ in terms of performance isolation, reliability, and security. Cost considerations also play a role in the choice of the instance type.

Many applications consist of multiple stages. In turn, each stage may involve multiple instances running in parallel on cloud servers and communicating among them. Thus, efficiency, consistency, and communication scalability are major concerns for an application developer. A cloud infrastructure exhibits latency and bandwidth fluctuations affecting the performance of all applications and, in particular, of data-intensive ones.

Data storage plays a critical role in the performance of any data-intensive application. The organization and the location of data storage, as well as the storage bandwidth must be carefully analyzed to ensure optimal application performance. Clouds support many storage options to set up a file system similar to the Hadoop file system discussed in Section 7.7. Among them are off-instance cloud storage, e.g. S3, mountable off-instance block storage, e.g., EBS, as well as storage persistent for the lifetime of the instance.

Many data-intensive applications use metadata associated with individual data records. For example, the metadata for an MPEG audio file may include the title of the song, the name of the singer, recording information, and so on. Metadata should be stored for easy access, the storage should be scalable, and reliable.

Another important consideration for the application developer is logging. The ability to identify the source of unexpected results and errors is helped by frequent logging, but performance considerations limit the amount of data logging. Logging is typically done using instance storage preserved only for the lifetime of the instance thus, measures to preserve the logs for a post-mortem analysis must be taken. Another challenge waiting resolution is related to software licensing discussed in Section 2.12.

Cloud application architectural styles. Cloud computing is based on the client-server paradigm discussed in Section 4.8. The vast majority of cloud applications take advantage of request-response communication between clients and stateless servers. A *stateless server* does not require a client to first establish a connection to the server, instead it views a client request as an independent transaction and responds to it.

The advantages of stateless servers are obvious. Recovering from a server failure requires a considerable overhead for a server which maintains the state of all its connections, but in case of a stateless server a client is not affected while a server goes down and then comes back up between two consecutive requests. A stateless system is simpler, more robust, and scalable. A client does not have to be concerned with the state of the server; if the client receives a response to a request, this means that the server is up and running, if not it should resend the request later. A connection-based service must reserve space to maintain the state of each connection with a client therefore, such a system is not scalable; the number of clients a server could interact with at any given time is limited by the storage space available to the server.

The Hypertext Transfer Protocol used by a browser to communicate with a web server is a request-response application protocol. HTTP uses TCP, a connection-oriented and reliable transport protocol. While TCP ensures reliable delivery of large objects, it exposes web servers to denial of service attacks. In such attacks malicious clients fake attempts to establish a TCP connection and force the server to allocate space for the connection. A basic web server is stateless; it responds to an HTTP request without maintaining a history of past interactions with the client. The client, a browser, is also stateless since it sends requests and waits for responses.

A critical aspect of the development of networked applications is how processes and threads running on systems with different architectures, possibly compiled from different programming languages, can *communicate structured information with one another*. First, the internal representation of the two structures at the two sites may be different, one system may use *Big-Endian* and the other *Little-Endian* representation. The character representations may also be different. A communication channel transmits a sequence of bits/bytes thus, the data structure must be serialized at the sending site and reconstructed at the receiving site.

Several considerations including neutrality, extensibility, and independence, must be analyzed before deciding the architectural style of an application. *Neutrality* refers to application-level protocol ability to use different transport protocols such as TCP and UDP and, in general, to run on top of a different protocol stack. For example, we shall see that SOAP can use as transport vehicles TCP, but also UDP, SMTP, and Java Message Service. *Extensibility* refers to the ability to incorporate additional functions such as security. *Independence* refers to the ability to accommodate different programming styles.

Very often application clients and the servers running on the cloud communicate use RPCs discussed in Section 4.8, yet other communication styles are possible. RPC-based applications use *stubs* to convert the parameters involved in an RPC call. A stub performs two functions, marshaling the data structures and serialization.

A more general concept is that of an Object Request Broker (ORB), a middleware facilitating communication of networked applications. The ORB at the sending site transforms the data structures used internally and transmits a byte sequence over the network. The ORB at the receiving site maps the byte sequence to the data structures used internally by the receiving process.

CORBA (Common Object Request Broker Architecture), developed in early 1990s, enables networked applications developed using different programming languages and running on systems with different architecture and system software to work with one another. At the heart of the system is the Interface Definition Language (IDL) used to specify the interface of an object. An IDL representation is then mapped to the set of programming languages including: C, C++, Java, Smalltalk, Ruby, Lisp, and Python. Networked applications pass CORBA by reference and pass data by value.

SOAP (Simple Object Access Protocol) was developed in 1998 for web applications. SOAP message format is based on the Extensible Markup Language (XML). SOAP uses TCP and more recently UDP transport protocols but it can also be stacked above other application layer protocols such as HTTP, SMTP, or JMS. The processing model of SOAP is based on a network consisting of senders, receivers, intermediaries, message originators, ultimate receivers, and message paths. SOAP is an underlying layer of Web Services.

WSDL (Web Services Description Language) (see http://www.w3.org/TR/wsdl) was introduced in 2001 as an XML-based grammar to describe communication between end points of a networked application. The abstract definitions of the elements involved include: *services*, collection of endpoints of communication; *types*, containers for data type definitions; *operations*, description of actions supported by a service; *port types*, operations supported by endpoints; *bindings*, protocols and data format supported by a particular port type; and *port*, an endpoint as a combination of a binding and a network address. These abstractions are mapped to concrete message formats and network protocols to define endpoints and services.

REST (Representational State Transfer) is a style of software architecture for distributed hypermedia systems. REST supports client communication with stateless servers, it is platform and language independent, supports data caching, and can be used in the presence of firewalls. REST almost always uses HTTP to support all four CRUD (*Create/Read/Update/Delete*) operations; it uses *GET*, *PUT*, and *DELETE* to read, write, delete the data, respectively.

REST is a much easier to use alternative to RPC, CORBA, or Web Services such as SOAP or WSDL. For example, to retrieve the address of an individual from a database a REST system sends an URL specifying the network address of the database, the name of the individual, and the specific attribute in the record, the client application wants to retrieve, in this case the address. The corresponding SOAP version of such a request consists of ten lines or more of XML. The REST server responds with the address of the individual. This justifies the statement that REST is a lightweight protocol. As far as usability is concerned, REST is easier to build from scratch and debug, but SOAP is supported by tools that use self-documentation, e.g., WSDL to generate the code to connect.

7.2 COORDINATION OF MULTIPLE ACTIVITIES

Many applications require the completion of multiple interdependent tasks [538]. The description of a complex activity involving such an ensemble of tasks is known as a *workflow*. In this section we discuss workflow models, the lifecycle of a workflow, the desirable properties of a workflow description. Workflow patterns, reachability of the goal state of a workflow, dynamic workflows, and a parallel between traditional transaction systems and cloud workflows are covered in Section 7.3.

Basic concepts. Workflow models are abstractions revealing the most important properties of the entities participating in a workflow management system. *Task* is the central concept in workflow modeling. A task is a unit of work to be performed on the cloud and it is characterized by several attributes, such as:

- Name – a string of characters uniquely identifying the task.
- Description – a natural language description of the task.
- Actions – an action is a modification of the environment caused by the execution of the task.
- Preconditions – boolean expressions that must be true before the action(s) of the task can take place.
- Postconditions – boolean expressions that must be true after the action(s) of the task do take place.
- Attributes – provide indications of the type and quantity of resources necessary for the execution of the task, the actors in charge of the tasks, the security requirements, whether the task is reversible or not, and other task characteristics.
- Exceptions – provide information on how to handle abnormal events. The exceptions supported by a task consist of a list of `<event, action>` pairs. The exceptions included in the task exception list are called *anticipated exceptions*, as opposed to unanticipated exceptions. Events not included in the exception list trigger re-planning. *Replanning* means restructuring of a process, redefinition of the relationship among various tasks.

A *composite task* is a structure describing a subset of tasks and the order of their execution. A *primitive task* is one that cannot be decomposed into simpler tasks. A composite task inherits some properties from workflows; it consists of tasks, has one start symbol, and possibly several end symbols. At the same time, a composite task inherits some properties from tasks; it has a name, preconditions, and postconditions.

A *routing task* is a special-purpose task connecting two tasks in a workflow description. The task that has just completed execution is called the *predecessor* task, the one to be initiated next is called the *successor task*. A routing task could trigger the sequential, concurrent, or iterative execution. Several types of routing tasks exist:

- A *fork routing task* triggers execution of several successor tasks. Several semantics for this construct are possible:
  1. All successor tasks are enabled.
  2. Each successor task is associated with a condition; the conditions for all tasks are evaluated and only the tasks with a `true` condition are enabled.
  3. Each successor task is associated with a condition; the conditions for all tasks are evaluated but, the conditions are mutually exclusive and only one condition may be `true` thus, only one task is enabled.
  4. Nondeterministic, k out of $n > k$ successors are selected at random to be enabled.
- A *join routing task* waits for completion of its predecessor tasks. There are several semantics for the join routing task:
  1. The successor is enabled after all predecessors end.
  2. The successor is enabled after k out of $n > k$ predecessors end.
  3. Iterative – the tasks between the fork and the join are executed repeatedly.

Process descriptions and cases. A *process description*, also called a workflow schema, is a structure describing the *tasks* or *activities* to be executed and the order of their execution; a process description contains one start symbol and one end symbol. A process description can be provided in a Workflow Definition Language (WFDL), supporting constructs for choice, concurrent execution, the classical *fork, join* constructs, and iterative execution. Clearly, a workflow description resembles a *flowchart*, a concept we are familiar with from programming.

The phases in the life-cycle of a workflow are creation, definition, verification, and enactment. There is a striking similarity between the life-cycle of a workflow and that of a traditional computer program, namely, creation, compilation, and execution, see Figure 7.1. The workflow specification by means of a workflow description language is analogous to writing a program. Planning is equivalent to automatic program generation. Workflow verification corresponds to syntactic verification of a program, and workflow enactment mirrors the execution of a compiled program.

A *case* is an instance of a process description. The start and stop symbols in the workflow description enable the creation and the termination of a case. An *enactment model* describes the steps taken to process a case. When all tasks required by a workflow are executed by a computer, the enactment can be performed by a program called an *enactment engine*.

The *state of a case* at time t is defined in terms of tasks already completed at that time. Events cause transitions between states. Identifying the states of a case consisting of concurrent activities is considerably more difficult than identifying the states of a strictly sequential process. Indeed, when several activities could proceed concurrently, the state has to reflect the progress made on each independent activity.

An alternative description of a workflow can be provided by a transition system describing the possible paths from the current state to a goal state. Sometimes, instead of providing a process description, we may specify only the goal state and expect the system to generate a workflow description that could lead to that state through a set of actions. In this case, the new workflow description is generated auto-

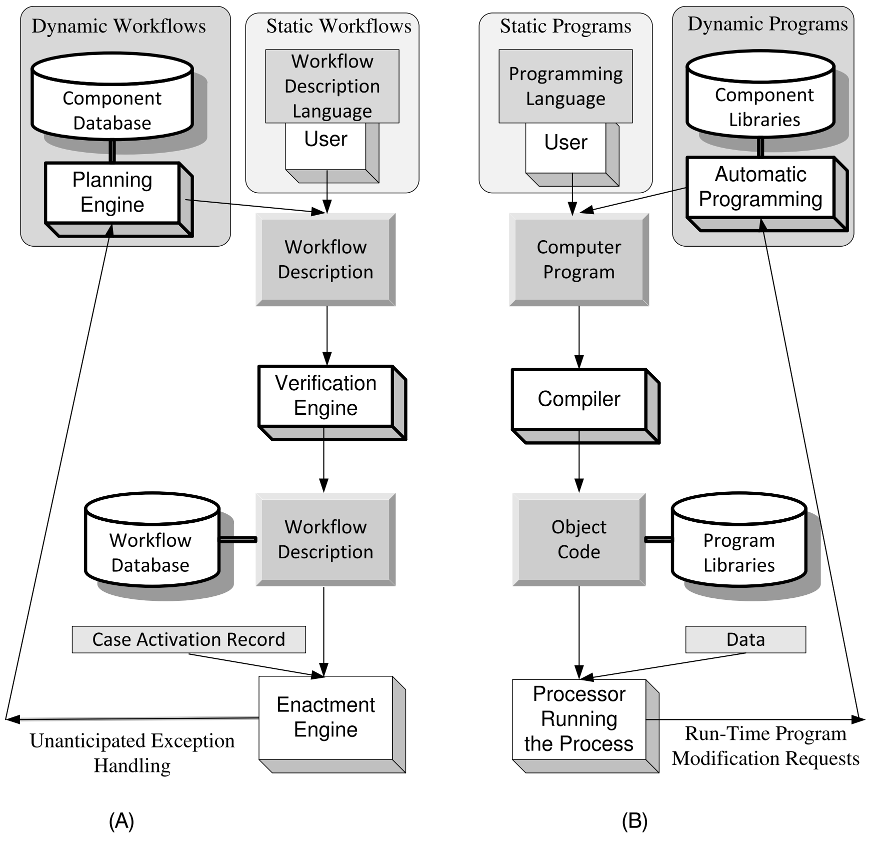


FIGURE 7.1

A parallel between workflows and programs. (A) The life-cycle of a workflow. (B) The life-cycle of a computer program. The workflow definition is analogous to writing a program. Planning is analogous to automatic program generation. Verification corresponds to syntactic verification of a program. Workflow enactment mirrors the execution of a program. A static workflow corresponds to a static program and a dynamic workflow to a dynamic program.

matically, knowing a set of tasks and the preconditions and postconditions for each one of them. In AI this activity is known as *planning*.

The state space of a process includes one initial state and one goal state. A transition system identifies all possible paths from the initial to the goal state. A case corresponds to a particular path in the transition system. The state of a case tracks the progress made during the enactment of that case.

Safety and liveness are the most desirable properties of a process description. Informally, *safety* means that nothing "bad" ever happens while *liveness* means that something "good" will eventually take place, should a case based on the process be enacted. Not all processes are safe and live. For example, the process description in Figure 7.2A violates the liveness requirement. As long as task C is chosen after completion of B, the process will terminate. However, if D is chosen, then F will never

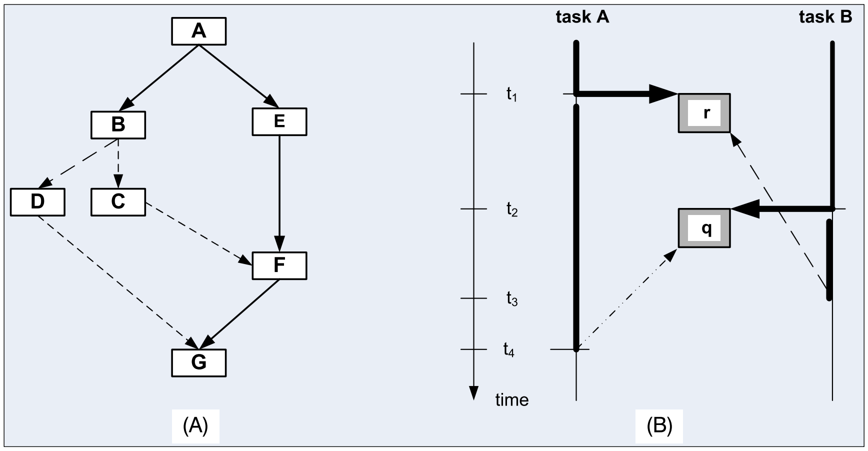


FIGURE 7.2

(A) A process description which violates the liveness requirement; if task C is chosen after completion of B, the process will terminate after executing task G; if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate because G requires completion of both D and F. (B) Tasks A and B need exclusive access to two resources r and q and a deadlock may occur if the following sequence of events occur: at time t_1 task A acquires r, at time t_2 task B acquires q and continues to run; then, at time t_3, task B attempts to acquire r and it blocks because r is under the control of A; task A continues to run and at time t_4 attempts to acquire q and it blocks because q is under the control of B.

be instantiated because it requires the completion of both C and E. The process will never terminate because G requires completion of both D and F.

A process description language should be unambiguous and should allow a verification of the process description before the enactment of a case. It is entirely possible that a process description may be enacted correctly in some cases, but could fail for others. Such enactment failures may be very costly and should be prevented by a thorough verification at the process definition time. To avoid enactment errors, we need to verify process description and check for desirable properties such as safety and liveness. Some process description methods are more suitable for verification than others.

A note of caution: although the original description of a process could be live, the actual enactment of a case may be affected by deadlocks due to resource allocation. To illustrate this situation, consider two tasks, A and B, running concurrently; each of them needs exclusive access to resources r and q for a period of time. Two scenarios are possible:

(1) either A or B acquires both resources and then releases them, and allows the other task to do the same;

(2) we face the undesirable situation in Figure 7.2B when at time t_1 task A acquires r and continues its execution; then at time t_2 task B acquires q and continues to run. Then at time t_3 task B attempts to acquire r and it blocks because r is under the control of A. Task A continues to run and at time t_4 attempts to acquire q and it blocks because q is under the control of B.

The deadlock illustrated in Figure 7.2B can be avoided by requesting each task to acquire all resources at the same time; the price to pay is under-utilization of resources; indeed, the idle time of each resource increases under this scheme.

7.3 WORKFLOW PATTERNS

The term *workflow pattern* refers to the temporal relationships among the tasks of a process. The workflow description languages and the mechanisms to control the enactment of a case must have provisions to support these temporal relationships. Workflow patterns are analyzed in [1], [323], and [540]. These patterns are classified in several categories: basic, advanced branching and synchronization, structural, state-based, cancellation, and patterns involving multiple instances. The basic workflow patterns in Figure 7.3 are:

- The *sequence* pattern occurs when several tasks have to be scheduled one after the completion of the other, Figure 7.3A.
- The *AND split* pattern requires several tasks to be executed concurrently. Both tasks B and C are activated when task A terminates, Figure 7.3B. In case of an *explicit AND split* the activity graph has a routing node and all activities connected to the routing node are activated as soon as the flow of control reaches the routing node. In the case of an *implicit AND split,* activities are connected directly and conditions can be associated with branches linking an activity with the next ones. The tasks are activated only when the conditions associated with a branch are true.
- The *synchronization* pattern requires several concurrent activities to terminate before an activity can start; in our example, task C can only start after both tasks A and B terminate, Figure 7.3C.
- The *XOR split* requires a decision; after the completion of task A, either B or C can be activated, Figure 7.3D.
- The *XOR join*; several alternatives are merged into one; in our example task C is enabled when either A or B terminates, Figure 7.3E.
- The *OR split* pattern is a construct to choose multiple alternatives out of a set. In our example, after completion of task A, one could activate either B or C, or both, Figure 7.3F.
- The *multiple merge* construct allows multiple activations of a task and does not require synchronization after the execution of concurrent tasks. Once A terminates, tasks B and C execute concurrently, Figure 7.3G. When the first of them, say B, terminates, then task D is activated; then, when C terminates, D is activated again.
- The *discriminator* pattern waits for a number of incoming branches to complete before activating the subsequent activity, Figure 7.3H; then it waits for the outgoing branches to finish without taking any action until all of them have terminated. Next, it resets itself.
- The *N out of M join* construct provides a barrier synchronization. Assuming that $M > N$ tasks run concurrently, N of them have to reach the barrier before the next task is enabled; in our example, any two out of the three tasks A, B, and C have to finish before E is enabled, Figure 7.3I.
- The *deferred choice* pattern is similar to the XOR split but this time the choice is not made explicitly and the run-time environment decides what branch to take, Figure 7.3J.

Next we discuss the reachability of the goal state and we consider the following elements:

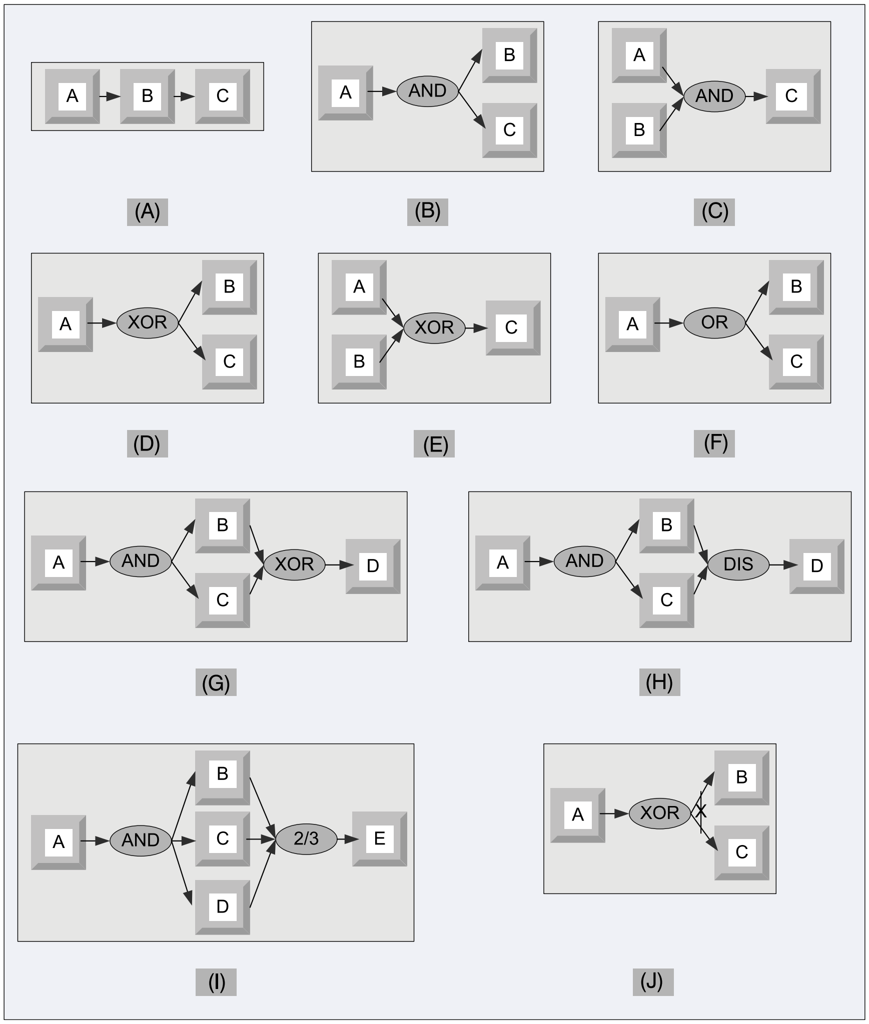


FIGURE 7.3

Basic workflow patterns. (A) Sequence; (B) AND split; (C) Synchronization; (D) XOR split; (E) XOR merge; (F) OR split; (G) Multiple Merge; (H) Discriminator; (I) N out of M join; (J) Deferred Choice.

- A system Σ, an initial state of the system, $\sigma_{initial}$, and a goal state, σ_{goal}.
- A process group, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$; each process p_i in the process group is characterized by a set of preconditions, $pre(p_i)$, postconditions, $post(p_i)$, and attributes, $atr(p_i)$.
- A workflow described by a directed activity graph $\mathcal{A}$ or by a procedure Π capable to construct $\mathcal{A}$ given the tuple $< \mathcal{P}, \sigma_{initial}, \sigma_{goal} >$. The nodes of $\mathcal{A}$ are processes in $\mathcal{P}$ and the edges define precedence relations among processes. $P_i \rightarrow P_j$ implies that $pre(p_j) \subset post(p_i)$.

- A set of constraints, $\mathcal{C} = \{C_1, C_2, \dots, C_m\}$.

The coordination problem for system Σ in state $\sigma_{initial}$ is to reach state σ_{goal}, as a result of postconditions of some process $P_{final} \in \mathcal{P}$ subject to constraints $C_i \in \mathcal{C}$. Here $\sigma_{initial}$ enables the preconditions of some process $P_{initial} \in \mathcal{P}$. Informally, this means that a chain of processes exists such that the postconditions of one process are preconditions of the next process in the chain.

Generally, the preconditions of a process are either the conditions and/or the events that trigger the execution of the process, or the data the process expects as input. The postconditions are the results produced by that process. The attributes of a process describe special requirements or properties of the process.

Some workflows are static, the activity graph does not change during the enactment of a case. *Dynamic workflows* are those that allow the activity graph to be modified during the enactment of a case. Some of the more difficult questions encountered in dynamic workflow management refer to: (i) how to integrate workflow and resource management and guarantee optimality or near optimality of cost functions for individual cases; (ii) how to guarantee consistency after a change in a workflow; (iii) how to create a dynamic workflow. Static workflows can be described in WFDL (the workflow definition language), but dynamic workflows need a more flexible approach.

We distinguish two basic models for the mechanics of workflow enactment:

1. *Strong coordination models* where the process group $\mathcal{P}$ executes under the supervision of a *coordinator* process. A coordinator process acts as an enactment engine and ensures a seamless transition from one process to another in the activity graph.
2. *Weak coordination models* where there is no supervisory process.

In the first case, we may deploy a *hierarchical coordination scheme* with several levels of coordinators. A supervisor at level i in a hierarchical scheme with $i+1$ levels coordinates a subset of processes in the process group. A supervisor at level $i-1$ coordinates a number of supervisors at level i and the root provides global coordination. Such a hierarchical coordination scheme may be used to reduce the communication overhead; a coordinator and the processes it supervises may be co-located.

An important feature of this coordination model is its ability of supporting dynamic workflows. The coordinator or the global coordinator may respond to a request to modify the workflow by first stopping all the threads of control in a consistent state, then investigating the feasibility of the requested changes, and finally implementing feasible changes.

Weak coordination models are based on peer-to-peer communication between processes in the process group by means of a societal service such as a *tuple space*. Once a process $p_i \in \mathcal{P}$ finishes, it deposits a token including possibly a subset of its postconditions, $post(p_i)$, in a tuple space. The consumer process p_j is expected to visit at some point in time the tuple space, examine the tokens left by its ancestors in the activity graph and, if its preconditions $pre(p_j)$ are satisfied, commence the execution. This approach requires individual processes to either have a copy of the activity graph or some timetable to visit the tuple space. An alternative approach is using an *active space*, a tuple space augmented with the ability to generate an event awakening the consumer of a token.

There are similarities and some differences between workflows of traditional transaction-oriented systems and cloud workflows; the similarities are mostly at the modeling level, whereas the differences affect the mechanisms used to implement workflow management systems. Some of the more subtle differences between them are:

- The emphasis in a transactional model is placed on the contractual aspect of a transaction; in a workflow the enactment of a case is sometimes based on a "best-effort" model where the agents involved will do their best to attain the goal state but, there is no guarantee of success.
- A critical aspect of the transactional model in database applications is that of maintaining a consistent state of the database; a cloud is an open system, thus, its state is considerably more difficult to define.
- The database transactions are typically short-lived; the tasks of a cloud workflow could be long lasting.
- A database transaction consists of a set of well-defined actions that are unlikely to be altered during the execution of the transaction. However, the process description of a cloud workflow may change during the lifetime of a case.
- The individual tasks of a cloud workflow may not exhibit the traditional properties of database transactions. For example, consider durability; at any instance of time, before reaching the goal state, a workflow may roll back to some previously encountered state and continue from there on an entirely different path. A task of a workflow could be either reversible or irreversible. Sometimes, paying a penalty for reversing an action is more profitable in the long run than continuing on a wrong path.
- Resource allocation is a critical aspect of the workflow enactment on a cloud without an immediate correspondent for database transactions.

The relatively simple coordination model discussed next is often used in cloud computing.

7.4 COORDINATION BASED ON A STATE MACHINE MODEL – THE ZOOKEEPER

Cloud computing elasticity requires the ability to distribute computations and data across multiple systems. In a distributed computing environment coordination among these systems is a critical function. The coordination model depends on the specific task, e.g., coordination of data storage, orchestration of multiple activities, blocking an activity until an event occurs, reaching consensus for the next action, or recovery after an error.

The entities to be coordinated could be processes running on a set of cloud servers or even running on multiple clouds. Servers running critical task are often replicated; when one primary server fails, a backup automatically continues the execution. This is only possible if the backup is in a *hot standby* mode, in other words, if the primary server shares at all times its state with the backup.

For example, in the distributed data store model discussed in Section 2.7 the access to data is mitigated by a proxy. An architecture with multiple proxies is desirable as the proxy is a single point of failure. The proxies should be in the same state so, whenever one of them fails, the client could seamlessly continue to access the data using another proxy.

Consider now an advertising service involving a large number of servers in a cloud. The advertising service runs on a number of servers specialized for tasks such as: database access, monitoring, accounting, event logging, installers, customer dashboards,[1] advertising campaign planners, scenario testing, and so on.

These activities can be coordinated using a configuration file shared by all systems. When the service starts, or after a system failure, all servers use the configuration file to coordinate their actions. This solution is static, any change requires an update and re-distribution of the configuration file. Moreover, in case of a system failure the configuration file does not allow recovery from the state each server was in prior to the system crash.

A solution for the coordination problem is to implement a proxy as a deterministic finite state machine transitioning from one state to the next in response to client commands. When P proxies are involved, all must be synchronized and execute the same sequence of state transitions upon receiving client commands. This scenario can be ensured when all proxies implement a version of the Paxos consensus algorithm described in Section 3.12.

ZooKeeper is a distributed coordination service based on this model. The high throughput and low latency service is used for coordination in large-scale distributed systems. The open-source software is written in Java and has bindings for Java and C. The information about the project is available at http://zookeeper.apache.org/.

The ZooKeeper software must first be downloaded and installed on several servers. Then clients can connect to any server and access the coordination service. The service is available as long as the majority of servers in the pack are available.

The servers in the pack communicate with one another and elect a *leader*. A database is replicated on each one of them and the consistency of the replicas is maintained. Figure 7.4A shows that the service provides a single system image, a client can connect to any server of the pack. A client uses TCP to connect to one server, then sends requests, receives responses, and watches events. A client synchronizes its clock with the server it is connected to. When a server fails, the TCP connections of all clients connected to it time-out, the clients detect the failure and connect to other servers.

Figures 7.4B and C show that a READ operation directed to any server in the pack returns the same result, while processing of a WRITE operation is more involved. The servers elect a *leader* and any *follower* server forwards to the leader requests from the clients connected to it. The leader uses atomic broadcast to reach consensus. When the leader fails the servers elect a new leader.

The system is organized as a shared hierarchical namespace similar to the organization of a file system. A name is a sequence of path elements separated by a backslash. Every name in ZooKeeper's name space is identified by a unique path, see Figure 7.5.

A *znode* of ZooKeeper, the equivalent of an *inode* of UFS, may have data associated with it. The system is designed to store state information, the data in each node includes version numbers for the data, changes of ACLs,[2] and time stamps. A client can set a watch on a znode and receive a notification when the znode changes. This organization allows coordinated updates. The data retrieved by a client contains also a version number. Each update is stamped with a number that reflects the order of the transition.

[1]A customer dashboard provides access to key customer information, such as contact name and account number, in an area of the screen that remains persistent as the user navigates trough multiple web pages.

[2]An Access Control List (ACL) is a list of pairs (subject,value) which define the list of access rights to an object; for example, read, write, execute permissions for a file.

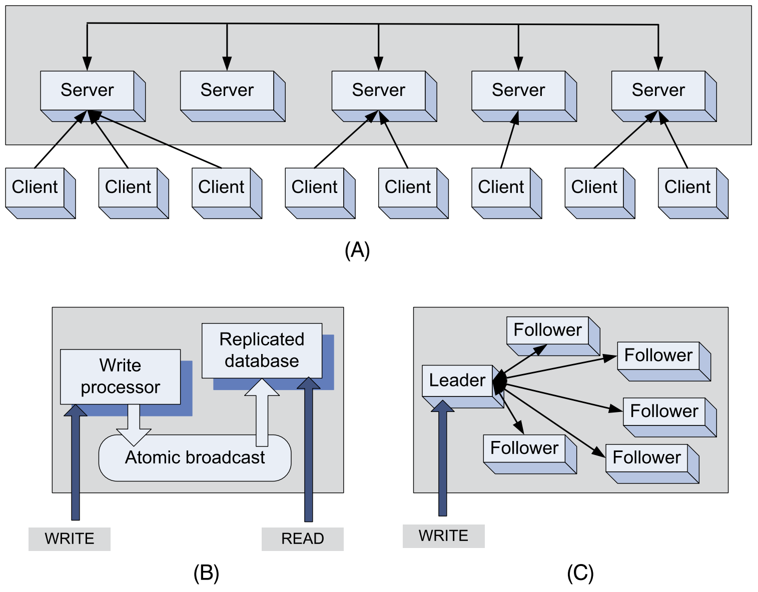


FIGURE 7.4

The ZooKeeper coordination service. (A) The service provides a single system image, clients can connect to any server in the pack. (B) The functional model of the ZooKeeper service; the replicated database is accessed directly by READ commands. (C) Processing a WRITE command: (1) a server receiving a command from a client, forwards it to the leader; (2) the leader uses atomic broadcast to reach consensus among all followers.

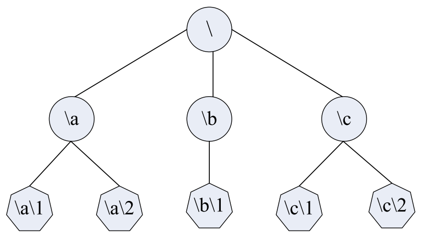


FIGURE 7.5

The ZooKeeper is organized as a shared hierarchical namespace; a name is a sequence of path elements separated by a backslash.

The data stored in each node is read and written atomically, a READ returns all data stored in a znode, while a WRITE replaces all data in the znode. Unlike a file system, the ZooKeeper data, the image of the state, is stored in the server memory. Updates are logged to disk for recoverability, and WRITEs are serialized to disk before they are applied to the in-memory database containing the entire tree. The ZooKeeper service guarantees:

1. Atomicity – a transaction either completes or fails.

2. Sequential consistency of updates – updates are applied strictly in the order they are received.
3. Single system image for the clients – a client receives the same response regardless of the server it connects to.
4. Persistence of updates – once applied, an update persists until it is overwritten by a client.
5. Reliability – the system is guaranteed to function correctly as long as the majority of servers function correctly.

READ requests are serviced from the local replica of the server connected to the client to reduce the response time. When the leader receives a WRITE request it determines the state of the system where the WRITE will be applied and then it transforms the state into a transaction capturing the new state.

The messaging layer is responsible for the election of a new leader when the current leader fails. The messaging protocol uses: *packets* – sequence of bytes sent through a FIFO channel, *proposals* – units of agreement, and *messages* – sequence of bytes atomically broadcast to all servers. A message is included into a proposal and it is agreed upon before it is delivered. Proposals are agreed upon by exchanging packets with a quorum of servers as required by the Paxos algorithm.

An atomic messaging system keeps all of the servers in pack in synch. The messaging system guarantees: (a) Reliable delivery: if a message m is delivered to one server, it will be eventually delivered to all servers; (b) Total order: if message m is delivered before message n to one server, it will be delivered before n to all servers; and (c) Causal order: if message n is sent after m has been delivered by the sender of n, then m must be ordered before n.

The ZooKeeper Application Programming Interface (API) is very simple, it consists of seven operations:

- *create* – add a node at a given location on the tree.
- *delete* – delete a node.
- *get data* – read data from a node.
- *set data* – write data to a node.
- *get children* – retrieve a list of the children of the node.
- *synch* – wait for the data to propagate.

The system also supports the creation of *ephemeral* nodes, nodes created when a session starts and deleted when the session ends.

This brief description shows that the ZooKeeper service supports the finite state machine model of coordination where a *znode* stores the state. The ZooKeeper service can be used to implement higher-level operations such as group membership, synchronization, and so on. Yahoo's Message Broker and many other applications use the ZooKeeper service.

7.5 THE MAPREDUCE PROGRAMMING MODEL

A main advantage of cloud computing is elasticity, the ability to use as many servers as necessary to optimally respond to the cost and the timing constraints of an application. In case of transaction processing system typically, a front-end system distributes incoming transactions to a number of back-end systems and attempts to balance the workload. As the workload increases new back-end systems are

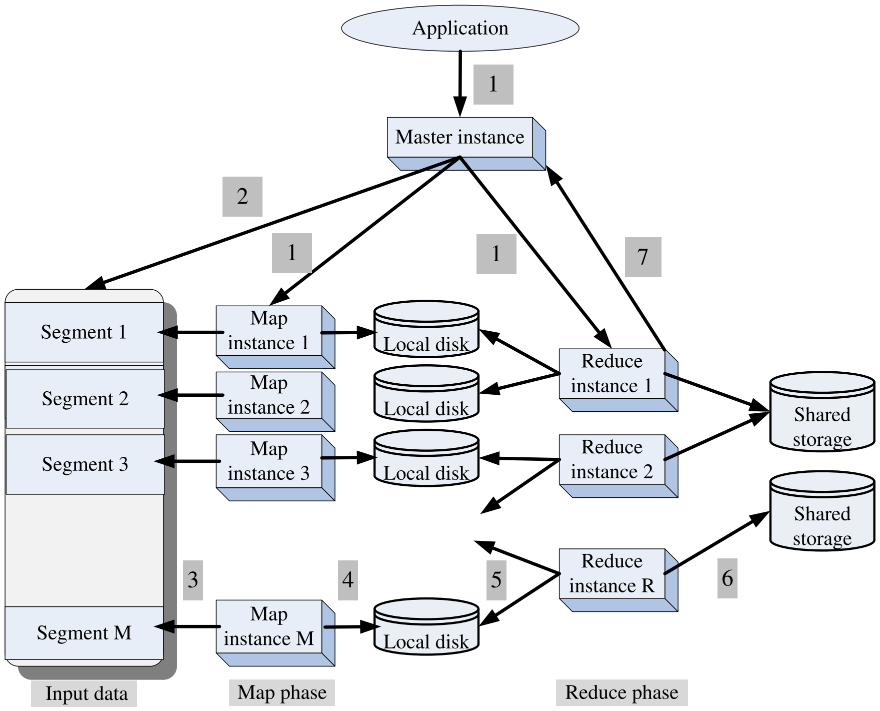


FIGURE 7.6

The MapReduce philosophy. (1) An application starts a Master instance and M worker instances for the Map phase and later R worker instances for the Reduce phase. (2) The Master partitions the input data in M segments. (3) Each Map instance reads its input data segment and processes the data. (4) The results of the processing are stored on the local disks of the servers where the Map instances run. (5) When all Map instances have finished processing their data the R Reduce instances read the results of the first phase and merges the partial results. (6) The final results are written by Reduce instances to a shared storage server. (7) The Master instance monitors the Reduce instances and when all of them report task completion the application is terminated.

added to the pool. Many realistic applications in physics, biology, and other areas of computational science and engineering obey the arbitrarily divisible load sharing model, the workload can be partitioned into an arbitrarily large number of smaller workloads of equal, or very close size. Yet, partitioning the workload of data-intensive applications is not always trivial.

MapReduce. MapReduce is based on a very simple idea for parallel processing of data-intensive applications supporting arbitrarily divisible load sharing, see Figure 7.6. First, split the data into blocks, assign each block to an instance/process and then run these instances in parallel. Once all the instances have finished the computations assigned to them, start the second phase and merge the partial results produced by individual instances. The Same Program Multiple Data (SPMD) paradigm, used since the early days of parallel computing, is based on the same idea, but assumes that a *Master* instance partitions the data and gathers the partial results.

MapReduce is a programming model inspired by the *map* and the *reduce* primitives of the Lisp programming language. It was conceived for processing and generating large data sets on computing clusters [130]. As a result of the computation, a set of input $< key, value >$ pairs is transformed into a set of output $< key, value >$ pairs.

Numerous applications can be easily implemented using this model. For example, one can process logs of web page requests and count the URL access frequency; the Map and Reduce functions produce the pairs $< URL, 1 >$ and $< URL, totalcount >$, respectively. Another trivial example is *distributed sort* when the Map function extracts the key from each record and produces a $< key, record >$ pair and the Reduce function outputs these pairs unchanged. The following example [130] shows the two user-defined functions for an application which counts the number of occurrences of each word in a set of documents.

```
map(String key, String value):
  // key: document name;  value: document contents
  for each word w in value:
  EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word;  values: a list of counts
  int result = 0;
  for each v in values:
  result += ParseInt(v);
  Emit(AsString(result));
```

Call M and R the number of Map and Reduce tasks, respectively, and N the number of systems used by the MapReduce. When a user program invokes the MapReduce function, the following sequence of actions take place:

- The run-time library splits the input files into M *splits* of 16 to 64 MB each, identifies a number N of systems to run, and starts multiple copies of the program, one of the system being a Master and the others Workers. The Master assigns to each idle system either a *map* or a *reduce* task. The Master makes $O(M + R)$ scheduling decisions and keeps $O(M \times R)$ worker state vectors in memory. These considerations limit the size of M and R; at the same time, efficiency considerations require that $M, R >> N$.
- A Worker being assigned a Map task reads the corresponding input split, parses $< key, value >$ pairs and passes each pair to a user-defined Map function. The intermediate $< key, value >$ pairs produced by the Map function are buffered in memory before being written to a local disk, partitioned into R regions by the partitioning function.
- The locations of these buffered pairs on the local disk are passed back to the Master, who is responsible for forwarding these locations to the Reduce Workers. A Reduce Worker uses remote procedure calls to read the buffered data from the local disks of the Map Workers; after reading all the intermediate data, it sorts it by the intermediate keys. For each unique intermediate key, the key and the corresponding set of intermediate values are passed to a user-defined Reduce function. The output of the Reduce function is appended to a final output file.
- The Master wakes up the user program when all Map and Reduce task finish.

The system is fault-tolerant; for each Map and Reduce task, the Master stores the state (idle, in-progress, or completed) and the identity of the worker machine. The Master pings every worker periodically and marks the worker as failed if it does not respond; a task in progress on a failed worker is reset to idle and becomes eligible for rescheduling. The Master writes periodic checkpoints of its control data structures and, if the task fails, it can be restarted from the last checkpoint. The data is stored using GFS, the Google File System discussed in Section 6.5.

An environment for experimenting with MapReduce is described in [130]: the computers are typically dual-processor *x86* processors running Linux, with 2–4 GB of memory per machine and commodity networking hardware typically 100–1 000 Mbps. A cluster consists of hundreds or thousands of machines. Data is stored on IDE[3] disks attached directly to individual machines. The file system uses replication to provide availability and reliability with unreliable hardware. To minimize network bandwidth the input data is stored on the local disks of each system.

MapReduce with FlumeJava. The Java library discussed in Section 3.14 supports a new operation called MapShuffleCombineReduce. This operation combines *ParallelDo, GroupByKey, CombineValues*, and *Flatten* operations into a single MapReduce [92]. This generalization of MapReduce supports multiple reducers and combiners and allows each reducer to produce multiple outputs, rather than enforcing the requirement that the reducer must produce outputs with the same key as its input. This solutions enables the FlumeJava optimizer to produce better results.

M input channels, each performing a Map operation, feed into R output channels, each optionally performing a shuffle, an optional combine, and a Reduce operation. The *executor* of FlumeJava will run an operation locally if the input is relatively small. It will run parallel MapReduce remotely, though the overhead of launching a remote execution is larger but the advantages for data larger inputs is significant. Temporary files for the outputs of all operations are created automatically and deleted when no longer necessary for later operations of the pipeline.

The system supports a *cached* execution mode when it first attempts to reuse the result of an operation from the previous run, if it was saved in a (internal or user-visible) file and if it was not changed since. A result is unchanged if the inputs and the operation's code and the saved state have not changed. This execution mode is useful for debugging an extended pipeline. Reference [92] reports that the largest pipeline had 820 unoptimized stages and 149 optimized stages.

7.6 CASE STUDY: THE GREPTHEWEB APPLICATION

Many applications process massive amounts of data using the MapReduce programming model. An application, GrepTheWeb [494], in production at Amazon, illustrates the power and the appeal of cloud computing. The application allows a user to define a regular expression and search the web for records that match it. GrepTheWeb is analogous to the *grep* Unix command used to search a file for a given regular expression.

This application performs a search of a very large set of records attempting to identify records that satisfy a regular expression. The source of this search is a collection of document URLs produced by the

[3]IDE (Integrated Drive Electronics) is an interface for connecting disk drives; the drive controller is integrated into the drive, as opposed to a separate controller on, or connected to, the motherboard.

Alexa Web Search, a software system that crawls the web every night. The inputs to the applications are: (i) the large data set produced by the web crawling software, and (ii) a regular expression. The output is the set of records that satisfy the regular expression. The user is able to interact with the application and get the current status, see Figure 7.7A.

The application uses message passing to trigger the activities of multiple controller threads which launch the application, initiate processing, shutdown the system, and create billing records. GrepTheWeb uses Hadoop MapReduce, an open source software package that splits a large data set into chunks, distributes them across multiple systems, launches the processing, and, when the processing is complete, aggregates the outputs from different systems into a final result. Apache Hadoop is a software library for distributed processing of large data sets across clusters of computers using a simple programming model.

GrepTheWeb workflow, illustrated in Figure 7.7B, consists of the following steps [494]:

1. *The start-up phase:* create several queues – launch, monitor, billing, and shutdown queues; start the corresponding controller threads. Each thread polls periodically its input queue and when a message is available, retrieves the message, parses it, and takes the required actions.
2. *The processing phase:* it is triggered by a *StartGrep* user request; then a launch message is enqueued in the launch queue. The launch controller thread picks up the message and executes the launch task; then, it updates the status and time stamps in the Amazon *Simple DB* domain. Lastly, it enqueues a message in the monitor queue and deletes the message from the launch queue. The processing phase consists of the following steps:
   (a) The launch task starts Amazon EC2 instances: it uses a Java Runtime Environment pre-installed Amazon Machine Image (AMI), deploys required Hadoop libraries and starts a Hadoop job (run Map/Reduce tasks).
   (b) Hadoop runs map tasks on EC2 slave nodes in parallel: a map task takes files from S3, runs a regular expression and writes locally the match results along with a description of up to five matches; then the combine/reduce task combines and sorts the results and consolidates the output.
   (c) Final results are stored on Amazon S3 in the output bucket.
3. *The monitoring phase:* the monitor controller thread retrieves the message left at the beginning of the processing phase, validates the status/error in Simple DB and executes the monitor task; it updates the status in the Simple DB domain, enqueues messages in the shutdown and the billing queues. The monitor task checks for the Hadoop status periodically, updates the Simple DB items with status/error and the S3 output file. Finally, it deletes the message from the monitor queue when the processing is completed.
4. *The shutdown phase:* the shutdown controller thread retrieves the message from the shutdown queue and executes the shutdown task which updates the status and time stamps in the Simple DB domain; finally, it deletes the message from the shutdown queue after processing. The shutdown phase consists of the following steps:
   (a) The shutdown task kills the Hadoop processes, terminates the EC2 instances after getting EC2 topology information from Simple DB and disposes of the infrastructure.
   (b) The billing task gets the EC2 topology information, Simple DB usage, S3 file and query input, calculates the charges, and passes the information to the billing service.
5. *The cleanup phase:* archives the Simple DB data with user info.

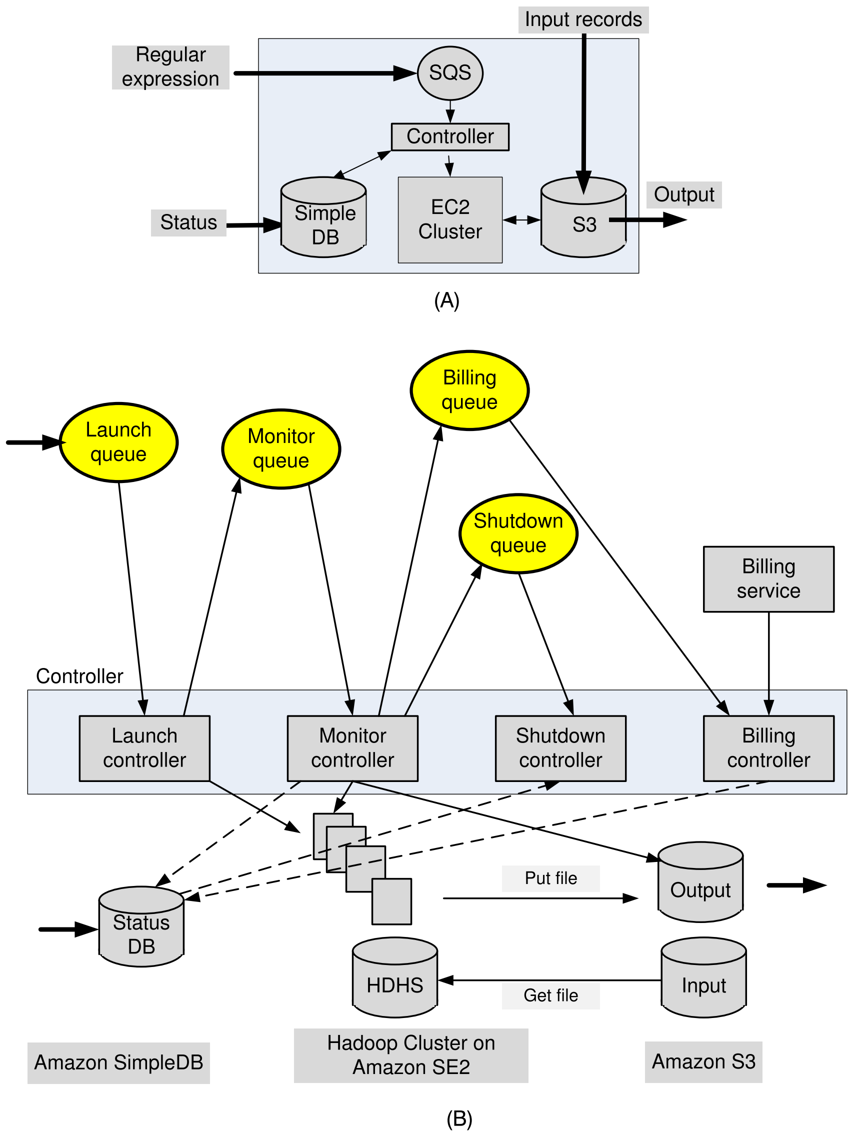


FIGURE 7.7

The organization of the GrepTheWeb application. The application uses the Hadoop MapReduce software and four Amazon services: EC2, Simple DB, S3, and SQS. (A) The simplified workflow showing the two inputs, the regular expression and the input records generated by the web crawler; a third type of input are the user commands to report the current status and to terminate the processing. (B) The detailed workflow; the system is based on message passing between several queues; four controller threads periodically poll their associated input queues, retrieve messages, and carry out the required actions.

6. *User interactions with the system:* get the status and output results. The *GetStatus* is applied to the service endpoint to get the status of the overall system (all controllers and Hadoop) and download the filtered results from S3 after completion.

Multiple S3 files are bundled up and stored as S3 objects to optimize the end-to-end transfer rates in the S3 storage system. Another performance optimization is to run a script and sort the keys, the URL pointers, and upload them in sorted order in S3. Multiple fetch threads are started to fetch the objects. This application illustrates the means to create an on-demand infrastructure and run it on a massively distributed system in a manner that allows it to run in parallel and scale up and down based on the number of users and the problem size.

7.7 HADOOP, YARN, AND TEZ

A wide range of data-intensive applications such as marketing analytics, image processing, machine learning, and web crawling use the Apache Hadoop, an open source, Java-based software system.[4] Hadoop supports distributed applications handling extremely large volumes of data. Many members of the community have contributed to the development and optimization of Hadoop and of several related Apache projects such as Hive and HBase.

Hadoop is used by many organizations from industry, government, and research. The long list of Hadoop users includes major IT companies e.g., Apple, IBM, HP, Microsoft, Yahoo, and Amazon, media companies e.g., New York Times and Fox, social networks including Twitter, Facebook, and LinkedIn, and government agencies such as the Federal Reserve. In 2012 the Facebook Hadoop cluster had a capacity of 100 petabytes and was growing at a rate of 0.5 petabytes a day. In 2013 more than half of Fortune 500 companies were using Hadoop. Azure HDInsight service deploys Hadoop on Microsoft Azure.

Hadoop. Apache Hadoop is an open-source software framework for distributed storage and distributed processing based on the MapReduce programming model. Recall that in MapReduce the Map stage processes the raw input data, one data item at a time, and produces a stream of data items annotated with keys. Next, a local sort stage orders the data produced during the map stage by key. The locally-ordered data is then passed to an (optional) combiner stage for partial aggregation by key. The shuffle stage then redistributes data among machines to achieve a global organization of data by key.

A Hadoop system has two components, a MapReduce engine and a database, see Figure 7.8. The database could be the Hadoop File System (HDFS), Amazon S3, or CloudStore, an implementation of GFS discussed in Section 6.5. HDFS is a highly performant distributed file system written in Java. HDFS is portable, but cannot be directly mounted on an existing operating system, it is not fully POSIX compliant.

The Hadoop engine on the master of a multi-node cluster consists of a *job tracker* and a *task tracker*, while the engine on a slave has only a *task tracker*. The *job tracker* receives a MapReduce job from a client and dispatches the work to the *task trackers* running on the nodes of a cluster. To increase efficiency, the *job tracker* attempts to dispatch the tasks to available slaves closest to the place where

[4]Hadoop requires JRE (Java Runtime Environment) 1.6 or higher.

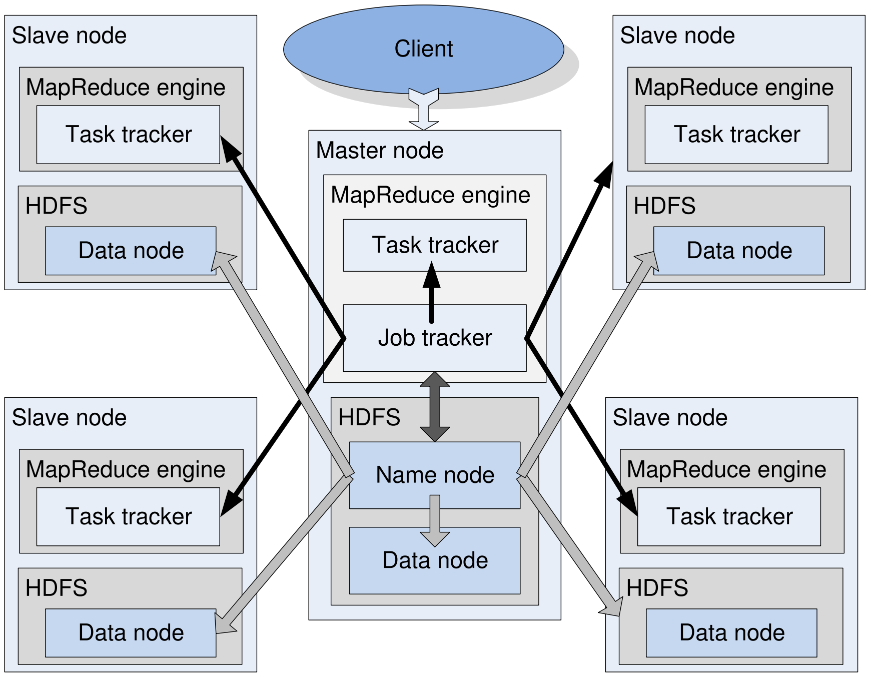


FIGURE 7.8

A Hadoop cluster using HDFS; the cluster includes a master and four slave nodes. Each node runs a MapReduce engine and a database engine, often HDFS. The *job tracker* of the master's engine communicates with the *task trackers* on all the nodes and with the *name node* of HDFS. The *name node* of HDFS shares information about the data placement with the *job tracker* to minimize communication between the nodes where data is located and the ones where it is needed.

had stored the task data. The *task tracker* supervises the execution of the work allocated to the node. Several scheduling algorithms for Hadoop engines have been implemented; Facebook's fair scheduler and Yahoo's capacity scheduler are examples of Hadoop schedulers.

HDFS replicates data on multiple nodes, the default is three replicas; a large dataset is distributed over many nodes. The *name node* running on the master manages the data distribution and data replication and communicates with *data nodes* running on all cluster nodes it shares with the *job tracker*. To minimize communication between the nodes where data is located and the ones where it is needed the name node shares information about data placement with the job traker. Although HDFS can be used for applications other than those based on the MapReduce model, its performance for such applications is not at par with the ones it was originally designed for.

Hadoop brings computations to the data on clusters built with off-the-shelf components. This strategy is pushed further by Spark which stores data in processor's memory instead of the disk. Data locality allows Hadoop and Spark to compete with traditional High Performance Computing (HPC) running on supercomputers with high-bandwidth storage and faster interconnection networks.

The Apache Hadoop framework has the following modules:

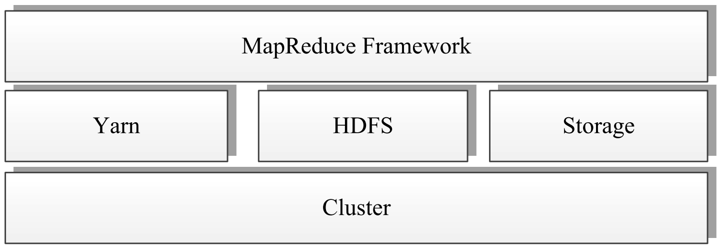


FIGURE 7.9

The resources needed by the MapReduce framework are provided by the cluster and are managed by Yarn. HDFS provides permanent, reliable, and distributed storage; different organizations of the storage system are supported, e.g., AWS implementation of Hadoop offers S3.

1. Common – contains libraries and utilities needed by all Hadoop modules.
2. Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster.
3. Yarn is a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications.
4. MapReduce, an implementation of the MapReduce programming model.

Additional software packages such as Apache Pig, Apache Hive, Apache HBase, Apache Phoenix, Apache Spark, Apache ZooKeeper, Cloudera Impala, Apache Flume, Apache Sqoop, Apache Oozie, and Apache Storm are also available.

Several Hadoop frameworks are used to manage and run deep analytics. SQL processing is particularly important to gain insights from large collections of data and the number of SQL-on-Hadoop systems has increased. There are many SQL engines running on Hadoop including BigSQL from IBM, Impala from Cloudera, and HAWQ from Pivotal. All these engines implement a standard language specification and compete on performance and extended services. Users are insulated from difficulties because the applications that talk to those engines are portable.

Systems such as Pig, Hive, and Impala discussed in Section 7.8 are native Hadoop-based systems, or database-Hadoop hybrids. HadoopOthers, such as *Hadapt* [4] exploits Hadoop scheduling and fault-tolerance, but uses a relational database, PostgreSQL, to execute query fragments.

Yarn. Yarn is a resource management system supplying CPU cycles, memory, and other resources needed by a single job or to a DAG of MapReduce applications. Resource allocation and job scheduling in Hadoop versions prior to 2.0 were done by the MapReduce framework. Yarn carries out these functions in the newer versions of Hadoop. This new system organization allows frameworks such Spark to share cluster resources.

The organization of Hadoop including Yarn is shown in Figure 7.9 where we see the three elements involved:(1) the MapReduce framework; (2) Yarn, HDFS and the storage substrate; and (3) the cluster where the application is running.

Figure 7.10 presents the organization of Yarn and shows the Resource Manager and the Node Managers running in each node. A Node Manager is responsible for containers, monitors their resource usage (CPU, memory, disk, network) and reports to the resource manager tasked to arbitrate resources

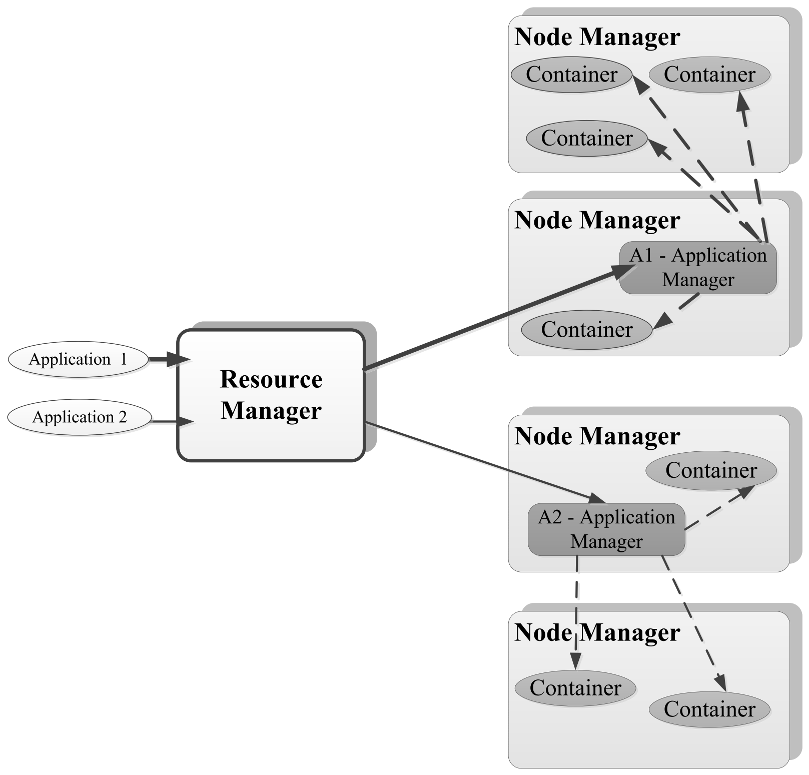


FIGURE 7.10

Yarn organization and application processing. Applications are submitted to the *Resource Manager*. The Resource Manager communicates with Node Managers running on every node of the cluster. The tasks of every application are managed by an Application Manager. Each task is packaged in a container.

sharing among all applications. Each application has an Application Manager which negotiates with the Resource Manager the access to resources needed by the application. Once resources are allocated, the Application Manager interacts with the Node Managers of each node allocated to the application to start the tasks and then monitors their execution.

The Scheduler component of the Resource Manager uses the resource Container abstraction which incorporates memory, CPU, disk, and network to make resource allocation decisions for an applications. The Scheduler performs no monitoring or tracking application status and offers no guarantees about restarting failed tasks. A pluggable policy is responsible for sharing cluster resources among applications. The Capacity Scheduler and the Fair Scheduler are examples of scheduler plug-ins. MapReduce in Hadoop-2.x maintains API compatibility with previous stable release thus, MapReduce jobs should still run unchanged on top of Yarn after a recompile.

The process of starting an application involves several steps illustrated in Figure 7.11:

1. The user submits an application to the Resource Manager.

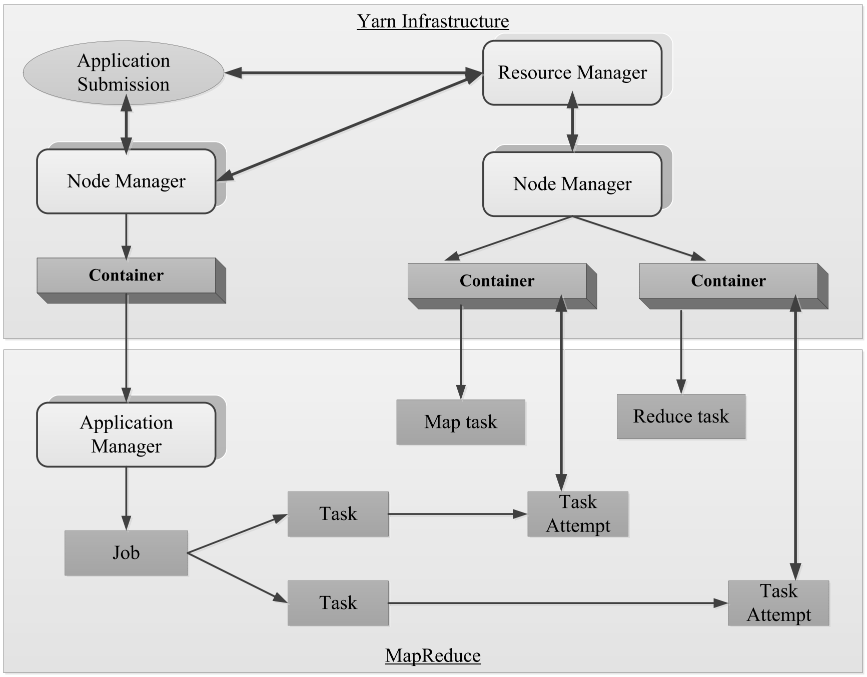


FIGURE 7.11

The interaction among the components of Yarn and MapReduce. Once an application is submitted Yarn's Resource Manager contacts a Node Manager to create a container for the Application Manager. Then the Application Manager is activated. The Resource Manager with the assistance of the Scheduler selects the Node Manager(s) where the containers for the application tasks are created. Then the containers start the execution of these tasks.

2. The Resource Manager invokes the Scheduler and allocates a container for the Application Manager.
3. The Resource Manager contacts the Node Manager where the container will be launched.
4. The Node Manager lunches the container.
5. The container executes the Application Manager.
6. The Resource Manager contacts the Node Manager(s) where the tasks of the application will run.
7. Containers for the tasks of the application are created.
8. The Application Manager monitors the execution of the tasks until termination.

Tez. Tez is an extensible framework for building high performance batch and interactive processing for Yarn-based applications in Hadoop. A job is decomposed into individual tasks and each task of the job runs as an Yarn process. Tez models data processing as a DAG; the graph vertices represent application logic and edges represent movement of data. A Java API is used to express the DAG representation of the workflow. The execution engine uses Yarn to acquire resources and reuses every component in the pipeline to avoid operation duplication. Apache Hive and Apache Pig use Apache Tez to improve the speed of MapReduce applications, see http://hortonworks.com/apache/tez/.

7.8 SQL ON HADOOP: PIG, HIVE, AND IMPALA

The landscape of parallel SQL database vendors prior to 2009 included IBM, Oracle, Microsoft, ParAccel, Greenplum, Teradata, Netezza, Vertica, but none of them were supporting SQL queries along with MapReduce jobs. From their early years in business Yahoo and Facebook used the MapReduce platform extensively to store, process, and analyze huge amounts of data. Both companies wanted a faster and easy-to-work-with platform supporting SQL queries.

MapReduce is a heavyweight, high-latency execution framework and does not support workflows, join operations for combined processing of several datasets, filtering, aggregation, top-k thresholding, and high-level operations. To address these challenges in 2009 Yahoo created a dataflow system called Pig, while Facebook built Hive on MapReduce because it was the shortest path to SQL on Hadoop. Pig and Hive have their own language, Pig Latin and HiveQL, respectively. In both systems a user types a query, then a parser reads the query, figures out what the user wants and runs a series of MapReduce jobs. That was a sensible decision given the requirements of the time.

In 2012 Cloudera developed Impala and published it under an Apache license. Later Facebook developed Presto, its next-generation query processing engine for real-time access to data via SQL. It was built, like Hive, new, from the ground up, as a distributed query processing engine.

Pig. The system presented in [187] supports workflows, join operations for combined processing of several datasets, filtering, aggregation, and the high-level operations. Pig compiles programs written in Pig Latin into a set of Hadoop jobs and coordinates their execution. It also supports several user interaction modes: (1) Interactive – using a shell for Pig commands; (2) Batch – a user submits a script containing a series of Pig commands; and (3) Embedded – commands are submitted via method invocation from a Java program. The job processing stages are:

1. Parsing – the parser performs syntactic verification, type checking, and schema inference and produces a DAG called a Logical Plan. In this plan an operator is annotated with the schema of its output data, with braces indicating a bag of tuples.
2. Logical optimization of the DAG and creation of a Physical Plan describing data distribution.
3. Compilation of the optimized Physical Plan into a set of MapReduce jobs, followed by optimization phase, e.g., partial aggregation, resulting in an optimized DAG. Distributive and algebraic aggregation functions, e.g., AVERAGE, are broken into series of three steps: initial (e.g. generate [sum, count] pairs), intermediate (e.g. combine n [sum, count] pairs into a single pair), final (e.g. combine n [sum, count] pairs and take the quotient). These steps are assigned to the map, combine, and reduce stages, respectively.
4. The DAG is topologically sorted and jobs are submitted to Hadoop for execution. The flow control is implemented using a pull model with a single threaded implementation and a simple API for user-defined functions for moving tuples through the execution pipeline. An operator can respond in one of three ways when asked to produce a tuple, return the tuple, declare that it has finished, or return a pause signal indicating that either it is not finished or unable to produce an output tuple.

Compilation and execution is triggered by the STORE command. If a Pig Latin program contains more than one STORE command, the generated physical plan contains a SPLIT physical operator.

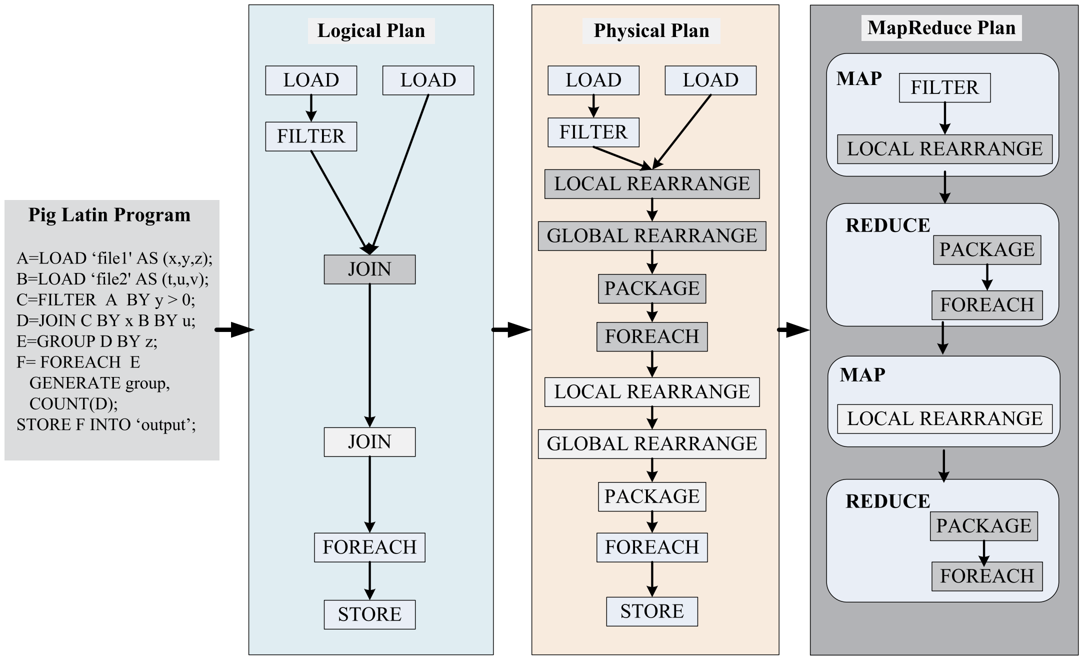


FIGURE 7.12

The transformation of a Pig Latin program into a Logical Plan, followed by the transformation of the Logical Plan into a Physical Plan, and finally the transformation of the Physical Plan into a MapReduce Plan. Each one of the two JOINs in the Logical Plan generate LOCAL REARRANGE, GLOBAL REARRANGE, PACKAGE, and FOREACH statements. A pair of MAP, REDUCE is then generated in the MapReduce Plan.

Figure 7.12 illustrates the transformation of a Pig Latin program into a Logical Plan, followed by the transformation of the Logical Plan into a Physical Plan, and finally the transformation of the Physical Plan into a MapReduce Plan.

Memory management is challenging because Pig is implemented in Java and program controlled memory allocation and deallocation is not feasible. A handler can be registered using the *Memory-PoolMXBean* Java class. The handler is notified whenever a configurable memory threshold is reached; then the system releases registered bags until enough memory is freed.

Pig Mix is a benchmark used at Yahoo and elsewhere to evaluate system's performance and to exercise a wide range of the system functionality. About 60% of the ad hoc Hadoop jobs at Yahoo use Pig. The use of the system outside Yahoo has been increasing.

Hive. Hive is an open-source system for data-warehousing supporting queries expressed in an SQL-like declarative language called HiveQL [484]. These queries are then compiled into MapReduce jobs executed on Hadoop. The system includes the Metastore, a catalog for schemas and statistics used for query optimization.

Hive supports data organized as tables, partitions, and buckets. *Tables* are inherited from relational databases and can be stored internally or externally in HDFS, NFS, or local directory. A table is serialized and stored in the files of an HDFS directory. The format of each serialized table is stored in the system catalog and it is accessed during query compilation and execution.

Partitions are components of tables described by the subdirectories of table directory. In turn, *buckets* consists of partition data stored as a file in the partition's directory and selected based on the hash of one of the columns of the table. The query language supports data definition statements to create tables with specific serialization formats, and partitioning and bucketing columns, as well as user defined column transformation and aggregation functions implemented in Java.

HiveQL accepts as input DDL, DML and allows user-defined MapReduce scripts written in any language using a simple row-based streaming interface. Data Description Language (DDL) has a syntax for defining data structures similar to a computer programming language and it is widely used for database schemas. Data Manipulation Language (DML) is used to retrieve, store, modify, delete, insert and update data in database; SELECT, UPDATE, INSERT statements or query statements are examples of DML.

The system has several components:

- *External Interface* – includes a command line (CLI), a Web user-interface (UI), and language APIs such as JDBC and ODBC.[5]
- *Thrift Server* – a client API for execution of HiveQL statements[6] Java and C++ clients can be used to build JDBC or ODBC common drivers, respectively.
- *Metastore* – the system catalog. It contains several objects: (1) Database – a namespace for tables; (2) Table – contains the list of columns and their types, owner, storage, the location of the table data, data formats and bucketing information; (3) Partition – each partition can have its own columns and storage information.
- *Driver* – manages the compilation, optimization, and execution of HiveQL statements.
- *Database* – the namespace for tables.
- *Table* – the metadata for tables containing the list of columns and their types, owner, storage, and a wealth of other data including the location of the table data, data formats and bucketing information. It also includes SerDe metadata regarding the implementation class of serializer and deserializer methods and information required by their implementation.
- *Partition* – information about the columns in a partition including SerDe and storage information.

The HiveQL compiler has several components: the *Parser* transforms an input string into a parse tree, then the *Semantic Analyzer* transforms this tree into an internal representation, the *Logical Plan Generator* converts this internal representation into a *Logical Plan*, and finally the *Optimizer* rewrites the logical plan.

Facebook's Hive warehouse contains over 700 TB of data and supports more than 5000 daily queries. Hive is an Apache project, with an active user and developer community.

Impala. Impala is a query engine exploiting a shared-nothing parallel database architecture written in C++ and Java and designed to use standard Hadoop components (HDFS, HBase, Metastore, Yarn,

[5]Java Database Connectivity (JDBC) is an API for Java, defining how a client may access a database. Open Database Connectivity (ODBC) is an open standard application API for accessing a database.
[6]Thrift is a framework for cross-language services, see Apache Thrift, http://incubator.apache.org/thrift.

Sentry) [280]. The system developed by Cloudera and published under an Apache license in 2012 is designed from the ground up for SQL query execution on Hadoop, rather than on a general-purpose distributed processing system. It delivers better performance than Hive because it does not translate an SQL query into another processing framework as Hive. It supports most of the SQL-92 SELECT statement syntax and SQL-2003 analytic functions. It does not support UPDATE or DELETE, but supports bulk insertions $INSERTINTO \dots, SELECT \dots$.

Impala code, installed on every node of a Cloudera cluster alongside MapReduce, Apache HBase, and third-party engines like SAS and Apache Spark, a customer may choose to deploy, waits for SQL queries to execute. All these engines have access to the same data, and users can choose one of them depending on the application. Impala runs queries using long-running daemons on every HDFS Data Node, and pipelines intermediate results between computation stages.

The I/O layer of Impala spawns one I/O thread per disk on each node to read data stored in HDFS and achieves high utilization of both CPU and disks by decoupling asynchronous read requests from the synchronous actual reading of data. The system exploits Intel's SSE instructions discussed in Section 4.3 to parse and process textual data efficiently. Impala requires the working set of a query to fit in the aggregate physical memory of the cluster.

The system operates basic services offered by three daemons:

1. *Impalad* – accepts, plans, and coordinates query execution. An impalad daemon is deployed on every server and operates a query planner, a query coordinator, and a query executor. The front-end compiles SQL text into query plans executable by the back-ends. In this stage a parse tree is translated into a single-node plan tree including: HDFS/HBase scan, hash join, cross join, union, hash aggregation, sort, top-n, and analytic evaluation nodes. A second phase transforms the single-node plan into a distributed execution plan; the goal of this process is to minimize data movement and maximize scan locality.
2. *Statestored* – provides a metadata publish-subscribe service and disseminates cluster-wide metadata to all processes. It maintains a set of topics, (key, value, version) triplets defined by an application. Processes wishing to receive updates to any topic express their interest by registering at start-up and providing a list of topics.
3. *Catalogd* – is the catalog repository and metadata access gateway. It pulls information from third-party metadata stores and aggregates it. Only a skeleton entry for each table it discovers is loaded at startup, then table metadata is loaded in the background from third-party stores.

A recent paper [175] compares the performance of Impala and Hive. Both systems input their data from columnar storage in the Parquet and the ORC (Optimized Row Columnar) formats. The Apache columnar storage formats are shared by the software in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language. Columnar formats improve the performance of queries in the context of relational databases.

Parquet stores data grouped together in logical horizontal partitions called *row groups*. Every row group contains a *column chunk* for each column in the table. A column chunk consists of multiple *pages* and is guaranteed to be stored contiguously on disk. Compression and encoding schemes work at a page level. Metadata is stored at all the levels in the hierarchy i.e., file, column chunk, and page. An ORC file stores multiple groups of row data as *stripes* and has a *footer* containing the list of the stripes in the file, the number of rows stored in each stripe, the data type of each column, and column-level aggregates, such as: count, sum, min, and max.

The experiments reported in [175] used Hive version 0.12 (Hive-MR) and Impala version 1.2.2 on top of Hadoop 2.0.0-cdh4.5.0, Hive version 0.13 (Hive-Tez) on top of Tez 0.3.0, and Apache Hadoop 2.3.0.1. Hadoop was configured to run 12 containers per node, 1 per core. The HDFS replication factor was set to three and the maximum JVM heap size was set to 7.5 GB per task. Impala uses MySQL as the metastore. One impalad process ran on each compute node and had access to 90 GB of memory.

One of the nodes of a 21 node cluster used for the measurements hosted the HDFS Name Node and there were 20 compute nodes. Each node ran 64-bit Ubuntu Linux 12.04 and had: one Intel Xeon CPUs @ 2.20 GHz with 6 cores, eleven SATA disks (2TB, 7k RPM), one 10 Gigabit Ethernet card, and 96 GB of RAM. One out of the eleven SATA disks in each node, hosted the OS and the rest were used for HDFS.

The experiments discussed in [175] ran the 22 TPC-H[7] queries for Hive and Impala and reported the execution time for each query. The file cache was flushed before each run in all compute nodes. The results show that Impala outperforms Hive-MR and Hive-Tez for all file formats with, or without compression. The performance gains vary from $1.5X$ to $13.5X$. Several factors contribute to this drastic performance boost:

- A more efficient I/O subsystem than Hive-MR or Hive-Tez.
- Long running daemon processes process queries in each node thus, the overhead of the job initialization and scheduling due to MapReduce in Hive-MR is eliminated.
- The query execution is pipelined, while in Hive-MR data is written out at the end of each step and read in by subsequent step(s).

Impala eliminates overheads of virtual function calls, inefficient instruction branching due to large switch statements, and other sources of inefficiency using code generation. When this feature is enabled at runtime the query execution time improves by about $1.3X$ for all 21 TPC-H queries combined. TPC-H Query 1 shows the largest improvement, $5.5X$, while the remaining queries improve up to $1.75X$.

A second set of experiments involve the TPC-DS benchmark.[8] Results show that Impala is on average $8.2X$ faster than Hive-MR and $4.3X$ faster than Hive-Tez and $10X$ faster than Hive-MR and $4.4X$ faster than Hive-Tez on a second workload. The first workload consisted of 20 queries which access a single fact table and six dimension tables, while the second used the same workload but removed the explicit partitioning predicate and used the correct predicate values.

7.9 CURRENT CLOUD APPLICATIONS AND NEW OPPORTUNITIES

Existing cloud applications can be divided in several broad categories: (i) processing pipelines; (ii) batch processing systems; and (iii) web applications [494].

[7]The TPC BenchmarkH (TPC-H) is a decision support benchmark consisting of a suite of business-oriented ad hoc queries and concurrent data modifications chosen to have broad industry-wide relevance. This benchmark is relevant for applications examining a large volume of data and executing queries with a high degree of complexity.

[8]TPC-DS is a de-facto industry standard benchmark for assessing the performance of decision support systems.

Processing pipelines are data-intensive and sometimes compute-intensive applications and represent a fairly large segment of applications currently running on a cloud. Several types of data processing applications can be identified:

- Indexing; the processing pipeline supports indexing of large datasets created by web crawler engines.
- Data mining; the processing pipeline supports searching very large collections of records to locate items of interests.
- Image processing; a number of companies allow users to store their images on the cloud, e.g., Flicker (flickr.com) and Picasa (http://picasa.google.com/). The image processing pipelines support image conversion, e.g., enlarge an image or create thumbnails; they can also be used to compress or encrypt images.
- Video transcoding; the processing pipeline transcodes from one video format to another, e.g., from AVI to MPEG.
- Document processing; the processing pipeline converts very large collection of documents from one format to another, e.g., from Word to PDF or encrypt the documents; they could also use OCR (Optical Character Recognition) to produce digital images of documents.

Batch processing systems also cover a broad spectrum of data-intensive applications in enterprise computing. Such applications typically have deadlines and the failure to meet these deadlines could have serious economic consequences; security is also a critical aspect for many applications of batch processing. A non-exhaustive list of batch processing applications includes:

- Generation of daily, weekly, monthly, and annual activity reports for organizations in retail, manufacturing, and other economical sectors.
- Processing, aggregation, and summaries of daily transactions for financial institutions, insurance companies, and healthcare organizations.
- Inventory management for large corporations.
- Processing billing and payroll records.
- Management of the software development, e.g., nightly updates of software repositories.
- Automatic testing and verification of software and hardware systems.

Lastly, but of increasing importance, are cloud applications in the area of web access. Several categories of web sites have a periodic or temporary presence. For example, the web site for conferences or other events. There are also web sites active during a particular season (e.g., the Holidays Season) or supporting a particular type of activity, such as income tax reporting with the April 15 deadline each year. Other limited-time web site are used for promotional activities, or web sites that "sleep" during the night and auto-scale during the day.

It makes economic sense to store the data in the cloud close to where the application runs; as we have seen in Section 2.3 the cost per GB is low and the processing is much more efficient when the data is stored close to the computational servers. This leads us to believe that several new classes of cloud computing applications could emerge in the years to come; for example, batch processing for decision support systems and other aspects of business analytics.

Another class of new applications could be parallel batch processing based on programming abstractions such as MapReduce discussed in Section 7.5. Mobile interactive applications which process large volumes of data from different types of sensors; services that combine more than one data source, e.g., mashups,[9] are obvious candidates for cloud computing.

Science and engineering could greatly benefit from cloud computing as many applications in these areas are compute-intensive and data-intensive. Similarly, a cloud dedicated to education would be extremely useful. Mathematical software, e.g., MATLAB and Mathematica, could also run on the cloud.

7.10 CLOUDS FOR SCIENCE AND ENGINEERING

For more than two thousand years science was empirical. Several hundred years ago theoretical methods based on models and generalization were introduced and this allowed a substantial progress in human knowledge. In the last few decades, we have witnessed the explosion of computational science based on the simulation of complex phenomena.

In a talk delivered in 2007 and posted on his web site just before he went missing in January 2007, Jim Gray discussed the *eScience* as a transformative scientific method [231]. Today, the *eScience* unifies the experiment, theory, and simulation; data captured from measuring instruments, or generated by simulations is processed by software systems, data and knowledge are stored by computer systems and analyzed with statistical packages.

The generic problems in virtually all areas of science are: (1) collection of experimental data; (2) management of a very large volumes of data; (3) building and execution of models; (4) integration of data and literature; (5) documentation of the experiments; and (6) sharing the data with others; data preservation for long periods of time. All these activities require powerful computing systems.

A typical example of a problem faced by agencies and research groups is data discovery in large scientific data sets. Examples of such large collections are the biomedical and genomic data at NCBI,[10] the astrophysics data from NASA,[11] or the atmospheric data from NOAA,[12] and NCAR.[13] The process of online data discovery can be viewed as an ensemble of several phases: (i) recognition of the information problem; (ii) generation of search queries using one or more search engines; (iii) evaluation of the search results; (iv) evaluation of the web documents; and (v) comparing information from different sources. The web search technology allows the scientists to discover text documents related to such data but, the binary encoding of many of them poses serious challenges.

High performance computing on AWS. Reference [254] describes the set of applications used at NERSC (National Energy Research Scientific Computing Center) and presents the results of a comparative benchmark of EC2 and three supercomputers. NERSC is located at Lawrence Berkeley National

[9]A mashup is an application that uses and combines data, presentations, or functionality from two or more sources to create a service. The fast integration, frequently using open APIs and multiple data sources, produces results not envisioned by the original services; combination, visualization, and aggregation are the main attributes of mashups.

[10]NCBI is the National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/.

[11]NASA is the National Aeronautics and Space Administration, http://www.nasa.gov/.

[12]NOAA is the National Oceanic and Atmospheric Administration, www.noaa.gov.

[13]NCAR is the National Center for Atmospheric Research, https://ncar.ucar.edu/.

Laboratory and serves a diverse community of scientists; it has some 3 000 researchers and involves 400 projects based on some 600 codes. Some of the codes used are:

CAM (Community Atmosphere Mode), the atmospheric component of CCSM (Community Climate System Model) is used for weather and climate modeling.[14] The code developed at NCAR uses two two-dimensional domain decompositions, one for the dynamics and the other for re-mapping. The first is decomposed over latitude and vertical level and the second is decomposed over longitude–latitude. The program is communication-intensive; on-node/processor data movement and relatively long MPI[15] messages that stress the interconnect point-to-point bandwidth are used to move data between the two decompositions.

GAMESS (General Atomic and Molecular Electronic Structure System) is used for ab initio quantum chemistry calculations. The code developed by the Gordon research group at the Department of Energy's Ames Lab at Iowa State University has its own communication library, the Distributed Data Interface (DDI) and is based on the SPMD (Same Program Multiple Data) execution model. DDI presents the abstraction of a global shared memory with one-side data transfers even on systems with physically distributed memory. On the cluster systems at NERSC the program uses socket communication; on the Cray XT4 the DDI uses MPI and only one-half of the processors compute, while the other half are data movers. The program is memory- and communication-intensive.

GTC (Gyrokinetic[16]) is a code for fusion research.[17] It is a self-consistent, gyrokinetic tri-dimensional Particle-in-cell (PIC)[18] code with a non-spectral Poisson solver; it uses a grid that follows the field lines as they twist around a toroidal geometry representing a magnetically confined toroidal fusion plasma. The version of GTC used at NERSC uses a fixed, one-dimensional domain decomposition with 64 domains and 64 MPI tasks. Communication is dominated by nearest neighbor exchanges that are bandwidth-bound. The most computationally intensive parts of GTC involve gather/deposition of charge on the grid and particle "push" steps. The code is memory intensive, as the charge deposition uses indirect addressing.

IMPACT-T (Integrated Map and Particle Accelerator Tracking Time) is a code for the prediction and performance enhancement of accelerators; it models the arbitrary overlap of fields from beamline elements, and uses a parallel, relativistic PIC method with a spectral integrated Green function solver. This object-oriented *Fortran90* code uses a two-dimensional domain decomposition in the y–z directions and dynamic load balancing based on the domains. Hockney's FFT (Fast Fourier Transform) algorithm is used to solve Poisson's equation with open boundary conditions. The code is sensitive to the memory bandwidth and MPI collective performance.

[14] See http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization.

[15] MPI, Message Passing Interface is a communication library based on a standard for a portable message-passing system.

[16] The trajectory of charged particles in a magnetic field is a helix that winds around the field line; it can be decomposed into a relatively slow motion of the guiding center along the field line and a fast circular motion called cyclotronic motion. Gyrokinetics describes the evolution of the particles without taking into account the circular motion.

[17] See http://www.scidacreview.org/0601/html/news4.html.

[18] PIC is a technique to solve a certain class of partial differential equations; individual particles (or fluid elements) in a Lagrangian frame are tracked in continuous phase space, whereas moments of the distribution such as densities and currents are computed simultaneously on Eulerian (stationary) mesh points.

MAESTRO is a low Mach number hydrodynamics code for simulating astrophysical flows.[19] Its integration scheme is embedded in an adaptive mesh refinement algorithm based on a hierarchical system of rectangular non-overlapping grid patches at multiple levels with different resolution; it uses a multigrid solver. Parallelization is via a tri-dimensional domain decomposition using a coarse-grained distribution strategy to balance the load and minimize communication costs. The communication topology tends to stress simple topology interconnects. The code has a very low computational intensity, it stresses memory latency, the implicit solver stresses global communications; the message sizes range from short to relatively moderate.

MILC (MIMD Lattice Computation) is a QCD (Quantum Chromo Dynamics) code used to study “strong” interactions binding quarks into protons and neutrons and holding them together in the nucleus.[20] The algorithm discretizes the space and evaluates field variables on sites and links of a regular hypercube lattice in four-dimensional space–time. The integration of an equation of motion for hundreds or thousands of time steps requires inverting a large, sparse matrix. The CG (Conjugate Gradient) method is used to solve a sparse, nearly-singular matrix problem. Many CG iterations steps are required for convergence; the inversion translates into tri-dimensional complex matrix–vector multiplications. Each multiplication requires a dot product of three pairs of tri-dimensional complex vectors; a dot product consists of five multiply-add operations and one multiply. The MIMD computational model is based on a four-dimensional domain decomposition; each task exchanges data with its eight nearest neighbors and is involved in the *all-reduce* calls with very small payload as part of the CG algorithm; the algorithm requires *gather* operations from widely separated locations in memory. The code is highly memory- and computational-intensive and it is heavily dependent on pre-fetching.

PARATEC (PARAllel Total Energy Code) is a quantum mechanics code; it performs ab initio total energy calculations using pseudo-potentials, a plane wave basis set and an all-band (unconstrained) conjugate gradient (CG) approach. Parallel three-dimensional FFTs transform the wave functions between real and Fourier space. The FFT dominates the runtime; the code uses MPI and is communication-intensive. The code uses mostly point-to-point short messages. The code parallelizes over grid points, thereby achieving a fine-grain level of parallelism. The BLAS3 and one-dimensional FFT use optimized libraries, e.g., Intel’s MKL or AMD’s ACML, and this results in high cache reuse and a high percentage of per-processor peak performance.

The authors of [254] use the HPCC (High Performance Computing Challenge) benchmark to compare the performance of EC2 with the performance of three large systems at NERSC. HPCC[21] is a suite of seven synthetic benchmarks: three targeted synthetic benchmarks which quantify basic system parameters that characterize individually the computation and communication performance; four complex synthetic benchmarks which combine computation and communication and can be considered simple proxy applications. These benchmarks are:

[19]See http://www.astro.sunysb.edu/mzingale/Maestro/.
[20]See http://physics.indiana.edu/~sg/milc.html.
[21]For more information see http://www.novellshareware.com/info/hpc-challenge.html.

- DGEMM[22] – the benchmark measures the floating point performance of a processor/core; the memory bandwidth does little to affect the results, as the code is cache friendly. Thus, the results of the benchmark are close to the theoretical peak performance of the processor.
- STREAM[23] – the benchmark measures the memory bandwidth.
- The network latency benchmark.
- The network bandwidth benchmark.
- HPL[24] – a software package that solves a (random) dense linear system in double precision arithmetic on distributed-memory computers; it is a portable and freely available implementation of the High Performance Computing Linpack Benchmark.
- FFTE – measures the floating point rate of execution of double precision complex one-dimensional DFT (Discrete Fourier Transform)
- PTRANS – parallel matrix transpose; it exercises the communications where pairs of processors communicate with each other simultaneously. It is a useful test of the total communications capacity of the network.
- RandomAccess – measures the rate of integer random updates of memory (GUPS).

The systems used for the comparison with cloud computing are:

Carver – a 400 node IBM iDataPlex cluster with quad-core Intel Nehalem processors running at 2.67 GHz and with 24 GB of RAM (3 GB/core). Each node has two sockets; a single Quad Data Rate (QDR) IB link connects each node to a network that is locally a fat-tree with a global two-dimensional mesh. The codes were compiled with the Portland Group suite version 10.0 of and Open MPI version 1.4.1.

Franklin – a 9 660-node Cray XT4; each node has a single quad-core 2.3 GHz AMD Opteron “Budapest” processor with 8 GB of RAM (2 GB/core). Each processor is connected through a 6.4 GB/s bidirectional HyperTransport interface to the interconnect via a Cray SeaStar-2 ASIC. The SeaStar routing chips are interconnected in a tri-dimensional torus topology, where each node has a direct link to its six nearest neighbors. Codes were compiled with the Pathscale or the Portland Group version 9.0.4.

Lawrencium – a 198-node (1 584 core) Linux cluster; a compute node is a Dell Poweredge 1950 server with two Intel Xeon quad-core 64 bit, 2.66 GHz Harpertown processors with 16 GB of RAM (2 GB/core). A compute node is connected to a Dual Data Rate InfiniBand network configured as a fat tree with a 3 : 1 blocking factor. Codes were compiled using Intel 10.0.018 and Open MPI 1.3.3.

The virtual cluster at Amazon had four EC2 CUs (Compute Units), two virtual cores with two CUs each, and 7.5 GB of memory (an `m1.large` instance in Amazon parlance); a Compute Unit is approximately equivalent to a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor. The nodes are connected with gigabit Ethernet. The binaries were compiled on Lawrencium. The results reported in [254] are summarized in Table 7.1.

[22] For more details see https://computecanada.org/?pageId=138.
[23] For more details see http://www.streambench.org/.
[24] For more details see http://netlib.org/benchmark/hpl/.

Table 7.1 The results of the measurements reported in [254].

System	DGEMM Gflops	STREAM GB/s	Latency µs	Bndw GB/S	HPL Tflops	FFTE Gflops	PTRANS GB/s	RandAcc GUP/s
Carver	10.2	4.4	2.1	3.4	0.56	21.99	9.35	0.044
Frankl	8.4	2.3	7.8	1.6	0.47	14.24	2.63	0.061
Lawren	9.6	0.7	4.1	1.2	0.46	9.12	1.34	0.013
EC2	4.6	1.7	145	0.06	0.07	1.09	0.29	0.004

The results in Table 7.1 give us some ideas about the characteristics of scientific applications likely to run efficiently on the cloud. Communication intensive applications will be affected by the increased latency (more than 70 times larger then *Carver*) and lower bandwidth (more than 70 times smaller than *Carver*).

7.11 CLOUD COMPUTING FOR BIOLOGY RESEARCH

Biology is one of the scientific fields in need of vast amounts of computing power and data storage and was one of the first to take advantage of cloud computing. Molecular dynamics computations are CPU-intensive while protein alignment is data-intensive.

An experiment carried out by a group from Microsoft Research illustrates the importance of cloud computing for biology research [315]. The authors carried out an "all-by-all" comparison to identify the interrelationship of the 10 million protein sequences (4.2 GB size) in NCBI's non-redundant protein database using Azure BLAST, a version of the BLAST[25] program running on the Azure platform [315].

Azure offers VM with four levels of computing power depending on the number of cores: small (one core), medium (two cores), large (eight cores), and extra large (more than eight cores). The experiment used eight core CPUs with 14 GB RAM and a 2 TB local disk. It was estimated that the computation would take six to seven CPU-years thus, the experiment was allocated 3 700 weighted instances or 475 extra-large VMs from three data centers. Each data center hosted three Azure BLAST deployments, each with 62 extra large instances. The 10 million sequences were divided into multiple segments, each segment was submitted for execution by one Azure BLAST deployment. With this vast amount of resources allocated, it took 14 days to complete the computations which produced 260 GB of compressed data spread across over 400 000 output files.

A post-experiment analysis led to a few conclusions useful for many scientific applications running on Azur. When a task runs for more than two hours, a message automatically reappears in the queue requesting the task to be scheduled, leading to repeated computations. The simple solution to this problem is to check if the result of a task has been generated before launching its execution.

[25]The Basic Local Alignment Search Tool (BLAST) finds regions of local similarity between sequences; it compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches; can be used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families. More information available at http://blast.ncbi.nlm.nih.gov/Blast.cgi.

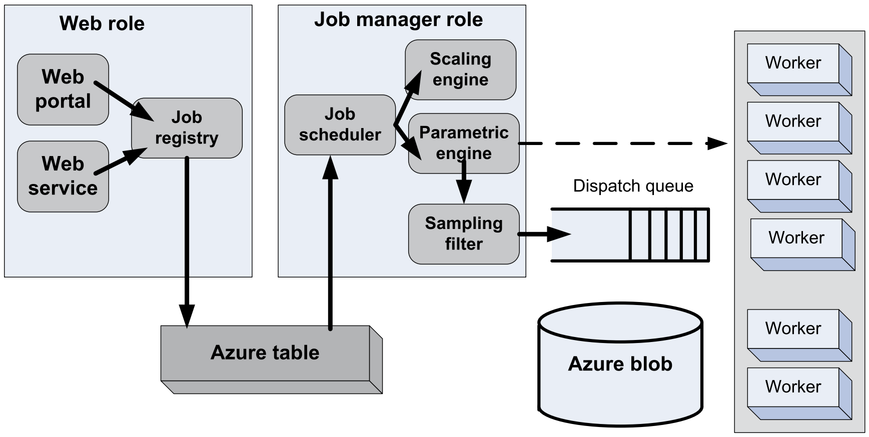


FIGURE 7.13

Cirrus – a general platform for executing legacy Windows applications on the cloud.

Many applications, including BLAST, allow for the setting of some parameters but, the computational effort for finding optimal parameters is prohibitive. To meet budget limitations each user is expected to decide on an optimal balance between cost and the number of instances.

A number of inefficiencies were observed: many VMs were idle for extended periods of time. When a task finished execution all worker instances waited for the next task. When all jobs use the same set of instances, resources are either under or over utilized. Load imbalance is another source of inefficiency; some of the tasks required by a job take considerably longer than others and delay the completion time of the job.

The analysis of the logs shows unrecoverable instance failures; some 50% of active instances lost connection to the storage service but were automatically recovered by the fabric controller. System updates caused several ensembles of instances to fail.

Another observation is that a computational science experiment requires the execution of several binaries thus, the creation of workflows, a challenging task for many domain scientists. To address this challenge the authors of [302] developed a general platform for executing legacy Windows applications on the cloud. In the Cirrus system a job has a description consisting of a prologue, a set of commands, and a set of parameters. The prologue sets up the running environment; the commands are sequences of shell scripts including Azure-storage-related commands to transfer data between Azure blob storage and the instance.

After the Windows Live ID service authenticates the user, it can submit and track a job through the portal provided by the web role, see Figure 7.13; the job is added to a table called *job registry*. The execution of each job is controlled by a *job manager instance* which first scales the size of the worker based on the job configuration, then, the parametric engine starts exploring the parameter space; if this is a test-run, the parameter sweeping result is sent to the sampling filter.

Each task is associated with a record in the task table and this state record is updated periodically by the worker instance running the task; the progress of the task is monitored by the manager. The

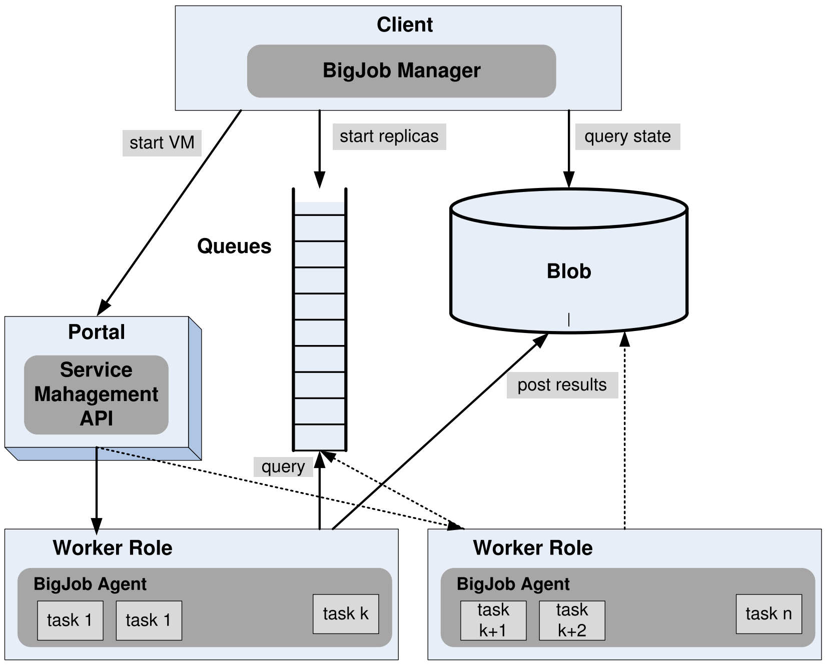


FIGURE 7.14

The execution of loosely-coupled workloads using the Azure platform.

dispatch queue feeds into a set of worker instances. A worker periodically updates the task state in the task table and listens for any control signals from the manager.

A loosely-coupled workload for an ensemble-based simulation on the Azure cloud is reported in [316]. A *role* in Azure is an encapsulation of an application; as noted earlier, there are two kinds of roles: (i) the web roles for web applications and front-end code; and (ii) the worker roles for background processing. Scientific applications, such as Azure BLAST use worker roles for the compute tasks and they implement their API which provides a run method and an entry point for the application and the state or configuration change notifications. The applications use the Blob Storage (ABS) for large raw data sets, the Table Storage (ATS) for semi-structured data, and the Queue Storage (AQS) for message queues; these services provide strong consistency guarantees but, the complexity is moved to the application space.

Figure 7.14 illustrates the use of a software system called BigJob to decouple resource allocation from resource binding for the execution of loosely coupled workloads on an Azure platform [316]; this software eliminates the need for the application to manage individual VMs. The results of measurements show a noticeable overhead for starting VMs and for launching the execution of an application task on a remote resource; increasing the computing power of the VM decreases the completion time for long-running tasks.

7.12 SOCIAL COMPUTING, DIGITAL CONTENT, AND CLOUD COMPUTING

Social networks play an increasingly important role in people's lives; they have expanded in terms of the size of the population involved and of the function performed. A promising solution for analyzing large-scale social networks data is to distribute the computation workload over a large number of cloud servers. Traditionally, the importance of a node or a relationship in a network is done using sampling and surveying but, in a very large network structural properties cannot be inferred by scaling up the results from small networks. The evaluation of social closeness is computationally intensive.

Social intelligence is another area where social and cloud computing intersect. Indeed, the process of knowledge discovery and techniques based on pattern recognition demand high-performance computing and resources that can be provided by computing clouds. Case-based reasoning (CBR), the process of solving new problems based on the solutions of similar past problems, is used by context-aware recommendation systems; it requires similarity-based retrieval. As the case base accumulates, such applications must handle massive amount of history data and this can be done by developing new reasoning platforms running on the cloud. CBR is preferable to rule-based recommendation systems for large-scale social intelligence applications. Indeed, the rules can be difficult to generalize or apply to some domains, all triggering conditions must be strictly satisfied, scalability is a challenge as data accumulate, and the systems are hard to maintain as new rules have to be added when the amount of data increases.

The *BetterLife 2.0* [243], a CBR-based system, consists of a cloud layer, a case-based reasoning engine, and an API. The cloud layer uses Hadoop clusters to store application data represented by cases, as well as social network information, such as relationship topology, and pairwise social closeness information. The CBR engine calculates similarity measures between cases to retrieve the most similar ones and also stores new cases back to the cloud layer. The API connects to a master node which is responsible for handling user queries, distributes the queries to server machines, and receives results.

A *case* consists of a problem description, solution, and optional annotations about the path to derive the solution. The CBR uses MapReduce; all the cases are grouped by their *userId*, and then a *breath first search* (BFS) algorithm is applied to the graph where each node corresponds to one user. MapReduce is used to calculate the closeness according to pairwise relationship weight. A reasoning cycle has four steps: (a) retrieve the most relevant or similar cases from memory to solve the case; (b) reuse – map the solution from the prior case to the new problem; (c) revise – test the new solution in the real world or in a simulation and, if necessary, revise; and (d) retain – if the solution was adapted to the target problem, store the result as a new case.

In the past, social networks have been constructed for a specific application domain, e.g., MyExperiment and nanoHub for biology and nanoscience, respectively. These networks enable researchers to share data and provide a virtual environment supporting remote execution of workflows. Another form of social computing is the *volunteer computing* when a large population of users donate resources such as CPU cycles and storage space for a specific project. The Mersenne Prime Search initiated in 1996, followed in the late 1990s by the SETI@Home, the Folding@home, and the Storage@Home, a project to back up and share huge data sets from scientific research, are well-known examples of volunteer computing. Information about these projects is available online at: www.myExperiment.org, www.nanoHub.org, www.mersenne.org, setiathome.berkeley.edu, and at folding.stanford.edu.

Volunteer computing cannot be used for applications where users require some level of accountability. The PlanetLab project is a credit based system in which users earn credits by contributing resources and then spend these credits when using other resources. The Berkeley Open Infrastructure for Network Computing (BOINC) aims to develop middleware for a distributed infrastructure suitable for different applications.

An architecture designed as a Facebook application for a social cloud is presented in [100]. Methods to get a range of data including friends, events, groups, application users, profile information, and photos are available through a Facebook API. The Facebook Markup Language is a subset of HTML with proprietary extensions and the Facebook JavaScript is a version of JavaScript. The prototype uses Web Services to create a distributed and decentralized infrastructure. There are numerous examples of cloud platforms for social networks. There are scalable cloud applications hosted by commercial clouds.

The new technologies supported by cloud computing favor the creation of digital content. *Data mashups* or *composite services* combine data extracted by different sources; *event-driven mashups*, also called Svc, interact through events rather than the request-response traditional method. A recent paper [462] argues that "the *mashup* and the cloud computing worlds are strictly related because very often the services combined to create new Mashups follow the SaaS model and more, in general, rely on cloud systems." The paper also argues that the Mashup platforms rely on cloud computing systems, for example, the IBM Mashup Center and the JackBe Enterprise Mashup server.

There are numerous examples of monitoring, notification, presence, location, and map services based on the Svc approach including: Monitor Mail, Monitor RSSFeed, Send SMS, Make Phone Call, GTalk, Fireeagle, and Google Maps. As an example, consider a service to send a phone call when a specific Email is received; the Mail Monitor Svc uses input parameters, such as User ID, Sender Address Filter, Email Subject Filter, to identify an Email and generates an event which triggers the *Make TTS Call* action of a *Text To Speech Call* Svc linked to it.

The system in [462] supports creation, deployment, activation, execution and management of Event Driven Mashups; it has a user interface, a graphics tool called Service Creation Environment that supports easily the creation of new Mashups, and a platform called *Mashup Container* that manages Mashup deployment and execution. The system consists of two subsystems, the *service execution platform* for Mashups execution and the *deployer* module that manages the installation of Mashups and Svcs. A new Mashup is created using the graphical development tool and it is saved as an XML file; it can then be deployed into a *Mashup Container* following the Platform as a Service (PaaS) approach. The *Mashup Container* supports a primitive SLA allowing the delivery of different levels of service.

The prototype uses the JAVA Message Service (JMS) which supports an asynchronous communication; each component sends/receives messages and the sender does not block waiting for the recipient to respond. The system's fault tolerance was tested on a system based on the VMware vSphere. In this environment, the fault tolerance is provided transparently by the VMM and neither the VMs nor the applications are aware of the fault tolerance mechanism; two VMs, a Primary and a Secondary one, run on distinct hosts and execute the same set of instructions such that, when the Primary fails, the Secondary continues the execution seamlessly.

Table 7.2 The features of the SFI for the Native Client on the x86-32, x86-64 , and ARM; ILP stands for Instruction Level Parallelism.

Feature/Architecture	x86-32	x86-64	ARM
Addressable memory	1 GB	4 GB	1 GB
Virtual base address	any	44GB	0
Data model	ILP32	ILP32	ILP 32
Reserved registers	0 of 8	1 of 16	0 of 16
Data address mask	None	Implicit in result width	Explicit instruction
Control address mask	Explicit instruction	Explicit instruction	Explicit instruction
Bundle size (bytes)	32	32	16
Data in text segment	forbidden	forbidden	allowed
Safe address registers	all	rsp, rbp	sp
Out-of-sandbox store	trap	wraps mod 4 GB	No effect
Out-of-sandbox jump	trap	wraps mod 4 GB	wraps mod 1 GB

7.13 SOFTWARE FAULT ISOLATION

Software fault isolation (SFI) offers a technical solution for sandboxing binary code of questionable provenance that can affect security in cloud computing. Insecured and tampered VM images is one of the security threats; binary codes of questionable provenance for native plugins to a web browser can pose a security threat as web browsers are used to access cloud services.

A recent paper [448] discusses the application of the sandboxing technology for two modern CPU architectures, ARM and x86-64. ARM is a load/store architecture with 32-bit instruction, 16 general purpose registers. It tends to avoid multi-cycle instructions and it shares many of the RISC architecture features but: (a) it supports a "thumb" mode with 16-bit instruction extensions; (b) has complex addressing modes and a complex barrel shifter; and (c) condition codes can be used to predicate most instructions. In the x86-64 architecture general purpose registers are extended to 64-bits, with an *r* replacing the *e* to identify the 64 versus 32-bit registers, e.g., *rax* instead of *eax*; there are eight new general purpose registers named $r8$–$r15$. To allow legacy instructions to use these additional registers, x86-64 defines a set of new prefix bytes to use for register selection.

This SFI implementation is based on the previous work of the same authors on Google Native Client (NC). This implementation assumes an execution model where a trusted runtime system shares a process with the untrusted multi-threaded plugin. The rules for binary code generation of an untrusted plugin are:

1. The code section is read-only and it is statically linked.
2. The code is divided into 32 byte *bundles* and no instruction or pseudo-instruction crosses the bundle boundary.
3. The disassembly starting at the bundle boundary reaches all valid instructions.
4. All indirect flow control instructions are replaced by pseudo-instructions that ensure address alignment to bundle boundaries.

The features of the SFI for the Native Client on the *x86*-32, *x86*-64, and *ARM* are summarized in Table 7.2 [448]. The control flow and store sandboxing for the ARM SFI incur less then 5% average overhead and the ones for x86-64 SFI incur less than 7% average overhead.

7.14 FURTHER READINGS

MapReduce is discussed in [130] and [494] presents the GrepTheWeb application. Cloud applications in biology are analyzed in [315] and [316] and social applications of cloud computing are presented in [100], [243], and [462]. Benchmarking of cloud services is analyzed in [108], [254], and [186]. The use of structured data is covered in [319]. An extensive list of publications related to Divisible Load Theory is at http://www.ece.sunysb.edu/~tom/dlt.html.

There are several cluster programming models. Data flow models of MapReduce and Dryad [253] support a large collection of operations and share data through stable data. High-level programming languages such as DryadLINQ [539] and FlumeJava [92] allow users to manipulate parallel collections of datasets using operators such as *map* and *join*. There are several systems providing high-level interfaces for specific applications such as *HaLoop* [79]. There are also caching systems including Spark [542] and *Tachyon* [301], discussed in Chapter 8, and *Nectar* [209].

Hive [484] was the first SQL over Hadoop to use another framework such as MapReduce or Tez to process SQL-like queries. Shark uses another framework, Spark [531] as its runtime. Impala [175] from Cloudera, LinkedIn Tajo (http://tajo.incubator.apache.org/), MapR *Drill* (http://www.mapr.com/resources/community-resources/apache-drill) and Facebook *Presto* (http://prestodb.io/), resemble parallel databases and use long-running custom-built processes to execute SQL queries in a distributed fashion. Hadapt [4] uses a relational database (PostgreSQL) to execute query fragments. Microsoft PolyBase [145] and Pivotal [99] use database query optimization and planning to schedule query fragments, and read HDFS data into database workers for processing.

A discussion of cost effective cloud high performance computing and a comparison with supercomputers [486] is reported in [88]. Scientific computing on clouds is discussed in [526]. Service level checking is analyzed in [101].

7.15 EXERCISES AND PROBLEMS

Problem 1. Download and install the *ZooKeeper* from the site http://zookeeper.apache.org/. Use the API to create the basic workflow patterns shown in Figure 7.3.

Problem 2. Use the *AWS Simple Workflow Service* to create the basic workflow patterns shown in Figure 7.3.

Problem 3. Use the *AWS CloudFormation* service to create the basic workflow patterns shown in Figure 7.3.

Problem 4. Define a set of keywords ordered based on their relevance to the topic of cloud security, then search the web using these keywords to locate 10–20 papers and store the papers in an *S3* bucket. Create a MapReduce application modeled after the one discussed in Section 7.6 to rank the papers based on the incidence of the relevant keywords. Compare your ranking with the rankings of the search engine you used to identify the papers.

Problem 5. Use the *AWS MapReduce* service to rank the papers in Problem 4.

Problem 6. The paper [85] describes the *elasticLM*, a commercial product which provides license and billing Web-based services. Analyze the merits and the shortcomings of the system.

Problem 7. Search the web for reports of cloud system failures and discuss the causes of each incident.

Problem 8. Identify a set of requirements you would like to be included in a SLA. Attempt to express them using the Web Service Agreement Specification (WS-Agreement) [31] and determine if it is flexible enough to express your options.

Problem 9. Consider the workflow for your favorite cloud application. Use XML to describe this workflow including the instances and the storage required for each task. Translate this description into an file that can be used for the *Elastic Beanstalk* AWS.

Problem 10. In Section 7.10 we analyze cloud computing benchmarks and compare them with the results of the same benchmarks performed on a supercomputer. Discuss the reasons why we should expect the poor performance of fine-grained parallel computations on a cloud.

Problem 11. An IT company decides to provide free access to a public cloud dedicated to higher education. Which one of the three cloud computing delivery models, SaaS, PaaS, or IaaS should it embrace and why? What applications would be most beneficial for the students? Will this solution have an impact on distance learning? Why?

CHAPTER

CLOUD HARDWARE AND SOFTWARE

8

This chapter presents the computing hardware and the software stack for a cloud computing infrastructure. In their quest to provide reliable, low-cost services, cloud service providers exploit the latest computing, communication, and software technologies to offer a highly available, easy to use, and efficient cloud computing infrastructure.

The cloud infrastructure is built with inexpensive off-the-shelf components to deliver cheap computing cycles. The millions of servers operating in today's cloud data centers deliver the computing power necessary to solve problems that in the past could only be solved by large supercomputers assembled from expensive, one-of-a-kind components.

Investing in large-scale computing systems can only be justified if the systems can efficiently accommodate a mix of workloads. The management of large-scale systems poses significant challenges and has triggered a flurry of developments in hardware and software systems. For example, virtual machines (VMs) and containers are key components of the cloud infrastructure. They exploit different embodiments of resource virtualization, a concept discussed in depth in Chapter 10.

Virtualization means to abstract a physical system and the access to it. A VM abstracts a physical processor and exploits virtualization by multiplexing. Containers exploit virtualization by aggregation and bridge the gap between a clustered infrastructure and assumptions made by applications about their environments and hide the complexity of the system.

Containers abstract an OS and include applications or tasks, as well as all their dependencies. Containers are portable, independent objects that can be easily manipulated by the software layer managing a large virtual computer. This virtual computer exposes to users the vast resources of a physical cluster with a very large number of independent processors.

There is not a single *killer application* for cloud computing. Modern cluster management systems address the problems posed by a mix of workloads, in addition to scalability challenges. Typical cloud workloads include not only coarse-grained, batch applications, but also fine-grained, long running applications with strict timing constrains. Only strict performance isolation and sophisticated scheduling can eliminate the undesirable effects of the long tail distribution of the response time for latency-sensitive jobs.

Several milestones in the evolution of ideas in cluster architecture along with algorithms and policies for resource sharing and effective implementation of the mechanisms to enforce these policies are analyzed in this chapter. Our analysis of the cloud software stack is complemented by the discussion of software systems closely related to applications presented in Chapter 7 and by topics related to Big Data applications discussed in Chapter 12.

Section 8.1 looks deeper into the challenges and benefits of virtualization and containerization. The next two Sections, 8.2 and 8.3, analyze Warehouse Scale Computers (WSCs) and their performance. Then the focus is switched to the software as the presentation concentrates on VMs and hypervisors in

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00011-X

Section 8.4 and then on frameworks such as Dryad, Mesos, Borg, Omega, and Quasar in Sections 8.5, 8.6, 8.7, 8.8, and 8.9, respectively.

Dryad is an execution engine for coarse-grained data parallel applications, Mesos is used for fine-grained cluster resource sharing, Omega is based on state sharing, and Quasar supports QoS-aware cluster management. Resource isolation discussed in Section 8.10 is followed by an analysis of in-memory cluster computing with Spark and Tachyon in Section 8.11. Docker containers and Kubernetes are covered in Sections 8.12 and 8.13.

8.1 CHALLENGES; VIRTUAL MACHINES AND CONTAINERS

Computing systems have evolved from single processors to multiprocessors, to multicore multiprocessors, and to clusters. Warehouse-scale computers (WSCs) with hundreds of thousands of processors are no longer a fiction, but serve millions of users, and are analyzed in computer architecture textbooks [56,228].

WSCs are controlled by increasingly complex software stacks. Software helps integrate a very large number of system components and contributes to the challenge of ensuring efficient and reliable operation. The scale of the cloud infrastructure combined with the relatively low mean-time to failure of the off-the-shelf components used to assemble a WSC make the task of ensuring reliable services quite challenging.

At the same time, long-running cloud services require a very high degree of availability. For example, a 99.99% availability means that the services can only be down for less than one hour per year. Only a fair level of hardware redundancy combined with software support for error detection and recovery can ensure such a level of availability [228].

Virtualization. The goal of virtualization is to support portability, improve efficiency, increase reliability, and shield the user from the complexity of the system. For example, threads are virtual processors, abstractions that allow a processor to be shared among different activities thus, increasing its utilization and effectiveness. RAIDs are abstractions of storage devices designed to increase reliability and performance.

Processor virtualization, running multiple independent instances of one or more operating systems, pioneered by IBM in early 1970, was revived for computer clouds. Cloud Virtual Machines run applications inside a guest OS which runs on virtual hardware under the control of a hypervisor. Running multiple VMs on the same server allows applications to better share the server resources and achieve higher processor utilization. The instantaneous demands for resources of the applications running concurrently are likely to be different and complement each other; the idle time of the server is reduced.

Processor virtualization by multiplexing is beneficial for both users and cloud service providers. Cloud users appreciate virtualization because it allows a better isolation of applications from one another than the traditional process sharing model. CSPs enjoy larger profits due to the low cost for providing cloud services.

Another advantage is that an application developer can chose to develop the application in a familiar environment and under the OS of her choice. Virtualization also provides more freedom for the system resource management because VMs can be easily migrated. The VM migration proceeds as follows: the VM is stopped, its state is saved as a file, the file is transported to another server, and the VM is restarted.

On the other hand, virtualization contributes to increased complexity of the system software and has undesirable side-effects on application performance and security. Processor sharing is now controlled by a new layer of software, the *hypervisor*, also called a Virtual Machine Monitor. It is often argued that a hypervisor is a more compact software with only a few hundred thousand lines of code versus the million lines of code of a typical OS, thus the hypervisor is less likely to be faulty.

Unfortunately, though the footprint of the hypervisor is small, a server must run a management OS in addition to the hypervisor. For example, Xen, the hypervisor used by AWS and others, invokes initially *Dom0*, a privileged domain that starts and manages unprivileged domains called *DomU*. *Dom0* runs the Xen management *toolstack*, is able to access the hardware directly, and provides Xen with virtual disks and network access for guests.

Containers. Containers are based on *operating-system-level virtualization* rather than *hardware virtualization*. An application running inside a container is isolated from another application running in a different container and both applications are isolated from the physical system where they run. Containers are portable and the resources used by a container can be limitted. Containers are more transparent than VMs thus, easier to monitor and manage. Containers have several other benefits including:

1. Streamline the creation and the deployment of applications.
2. Applications are decoupled from the infrastructure; application container images are created at build time rather than deployment time.
3. Support portability; containers run independently of the environment.
4. Support an application-centric management.
5. Have an optimal philosophy for application deployment; applications are broken into smaller, independent pieces that can be managed dynamically.
6. Support higher resource utilization.
7. Lead to predictable application performance.

Containers were initially designed to support the isolation of the *root* file system. The concept can be traced back to the *chroot* system call implemented in 1979 in Unix to: (i) change the root directory for the running process issuing the call and for its children; and (ii) to prohibit access to files outside the directory tree. Later, BSD and Linux adopted the concept and in 2000, FreeBSD expanded it and introduced the *jail* command. The environment created with *chroot* was used to create and host a new virtualized copy of the software system.

Container technology has emerged as an ideal solution combining isolation with increased productivity for application developers who need no longer be aware of the details of the cluster organization and management. Container technology is now ubiquitous and has a profound impact on cloud computing. Docker's containers gained widespread acceptance for ease of use, while Google's Kubernetes are performance-oriented.

Cluster management systems have evolved and each system has benefited from the experience gathered from the previous generation. Mesos, a system developed at U.C. Berkeley is now widely used by more than 50 organizations and has also morphed in a variety of systems such as Aurora used by Twitter, Marathon offered by Mesospheres,[1] and Jarvis used by Apple. Borg, Omega, and Kubernetes are the milestones in Google's cluster management development effort discussed in this chapter.

[1]Mesospher is a startup selling the Datacenter Operating System, a distributed OS, based on Apache Mesos.

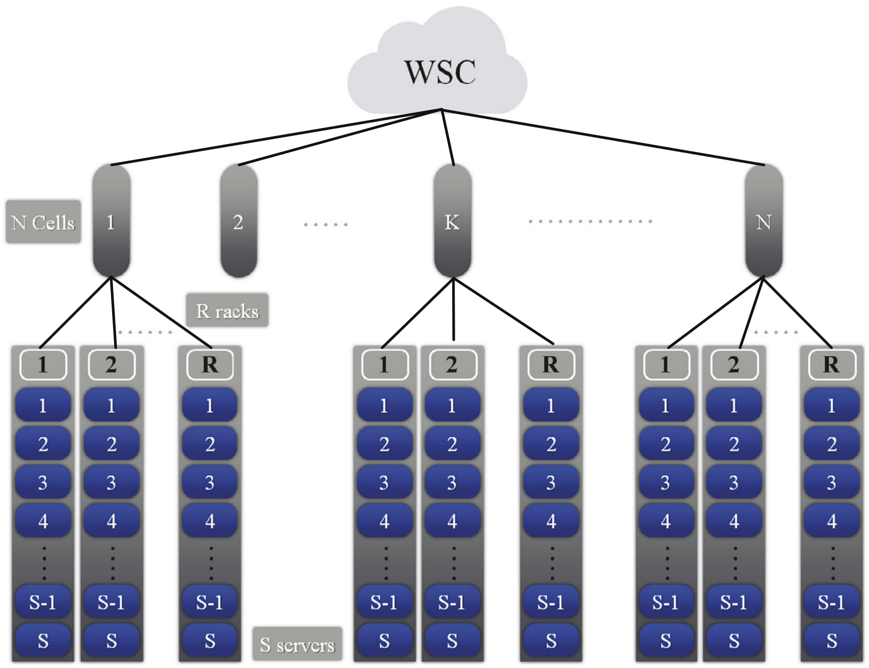


FIGURE 8.1

The organization of a WSC with N cells, R racks, and S servers per rack.

8.2 CLOUD HARDWARE; WAREHOUSE-SCALE COMPUTERS

Cloud computing had an impact on large-scale systems architecture. The WSCs [56,228] form the backbone of the cloud infrastructure of Google, Amazon, and other CSPs. WSCs are hierarchically organized systems with 50 000–100 000 processors capable of exploiting request-level and data-level parallelism.

At the heart of a WSC is a *hierarchy of networks* which connect the system components, servers, racks, and cells/arrays, together as in Figure 8.1. Typically, a *rack* consists of 48 servers interconnected by a 48 port, 10 Gbps Ethernet (GE) switch. In addition to the 48 ports, the GE switch has two to eight uplink ports connecting a rack to a cell. Thus, the level of *oversubscription*, the ratio of internal to external ports, is between $48/8 = 6$ and $48/2 = 24$. This has serious implications on the performance of an application; two communicating processes running on servers in the same rack have a much larger bandwidth and lower latency than the same processes running on servers in different racks.

The next component is a *cell*, sometimes called an array, consisting of a number of racks. The racks in a cell are connected by an *array switch*, a rather expensive communication hardware with a cost two orders of magnitude higher then that of a rack switch. The cost is justified by the fact that the bandwidth of a switch with n ports is of order n^2. To support a 10 times larger bandwidth for 10 times as many ports, the cost increases by a factor of 10^2. An array switch can support up to 30 racks.

Table 8.1 The memory hierarchy of a WSC with the latency given in microseconds, the bandwidth in MB/sec, and the capacity in GB [56].

Location type	DRAM			HDD		
	Latency	Bandwidth	Capacity	Latency	Bandwidth	Capacity
Local	0.1	20 000	16	10 000	200	2 000
Rack	100	100	1 040	11 000	100	160 000
Cell	3 000	10	31 200	12 000	10	4 800 000

WSCs support both interactive and batch workloads. The communication latency and the bandwidth within a server, a rack, and a cell are different thus, the execution time and the costs for running an application are affected by the volume of data, the placement of data, and by the proximity of instances. For example, the latency, the bandwidth, and the capacity of the memory hierarchy of a WSC with 80 servers/rack and 30 racks/cell are shown in Table 8.1 based on the data from [56].

The DRAM latency increases by more than three orders of magnitude, while the bandwidth decreases by a similar factor. The latency and the bandwidth of the HDDs follow the same trend, but the variation is less dramatic. To put this in perspective, the memory-to-memory transfer of 1 000 MB takes 50 msec within a server, 10 seconds within the rack, and 100 seconds within a cell, while the transfers between disks take 5, 10, and 100 seconds, respectively.

WSCs, though expected to supply cheap computing cycles, are by no means inexpensive. The cost of a WSC is of the order of \$150 million, but the cost-performance is what that makes WSCs appealing. The capital expenditures for a WSC include the costs for servers, for the interconnect, and for the facility. A case study reported in [228] shows a capital expenditure of \$167 510 000 including \$66 700 000 for 45 978 servers, \$12 810 000 for an interconnect with 1 150 rack switches, 22 cell switches, 2 layer 3 switches, and 2 border routers. In addition to the initial investment the operation cost of the cloud infrastructure including the cost of energy is significant. In this case study the facility is expected to use 8 MW.

We should now take a closer look at the WSC servers and ask ourselves what type of processors are best suited as server components. Unquestionably, multicore processors are ideal components of WSC servers as they support not only data-level parallelism for search and analysis of very large data sets, but also request-level parallelism for systems expected to support a very large number of transactions per second. Data-parallel and request-parallel applications are the two major components of the workloads experienced by cloud service providers such as Google.

There are two basic groups of multicore processors often called *browny* and *wimpy* cores [239]. The single core performance of a browny core is impressive, but so is the power dissipation. On the other hand, the wimpy cores are less powerful but they also consume less power. Power consumption is a major concern for cloud as we shall see in Section 9.2; for solid-state technologies the power dissipation is approximately $\mathcal{O}(f^2)$ with f the clock frequency.

Using wimpy cores poses several problems. When running on wimpy cores rather than on browny cores, a task needs to spawn a larger number of threads. This has two major implications: first, it complicates the software development process as it requires an explicit parallelization of the application thus, increasing the cost of application development. A second, equally important implication, is that running a larger number of threads increases the response time. Very often all threads have to finish before the next step of an algorithm, the well known problem posed by barrier-synchronization. This means that all threads have to wait for the slowest one.

The cost of systems using wimpy core may increase, e.g., the cost for DRAM will increase as the kernel and the system processes consume more aggregate memory. Data structures used by applications might need to be loaded into memory on multiple wimpy-core machines instead of being loaded into a single brawny-core machine memory; this has a negative effect on performance. Lastly, managing a larger number of threads will increase the scheduling overhead and diminish performance.

Hölzle [239] concludes "Once a chip's single-core performance lags by more than a factor of two or so behind the higher end of current-generation commodity processors, making a business case for switching to the wimpy system becomes increasingly difficult because application programmers will see it as a significant performance regression: their single-threaded request handlers are no longer fast enough to meet latency targets."

8.3 WSC PERFORMANCE

The central question addressed in this section is how to extract the maximum performance from a warehouse scale computer, what are the main sources of WSC inefficiency, and how these inefficiencies could be avoided. Even slight WSC performance improvements translate to large cost savings for the CSPs and noticeable better service for cloud users.

The workload of WSCs is very diverse, there are no typical, or "killer" applications that would drive the design decisions and, at the same time, guarantee optimal performance for such workloads. It is not feasible to experiment with systems at this scale or to simulate them effectively under realistic workloads.

The only alternative is to profile realistic workloads and analyze carefully the data collected during production runs, but this is only possible if low-overhead monitoring tools which minimize intrusion on the workloads are available. Monitoring tools minimize intrusion by random sampling and by maintaining counters of relevant events, rather than detailed event records.

Google-Wide-Profiling (GWP) is a low-overhead monitoring tool used to gather the data through random sampling. GWP randomly selects a set of servers to profile every day and uses mostly Perf[2] to monitor their activity for relatively short periods of time, then collects the *callstacks*[3] of the samples, aggregates the data and stores it in a database [418].

Data collected at Google with GWP over a period of 36 months is presented in [262] and discussed in this section. Only data for C++ codes was analyzed because C++ codes dominate the CPU cycle consumption though, the majority of codes are written in Java, Python, and Go. The data was collected from some 20 000 servers built with Intel Ivy Bridge processors.[4]

The analysis is restricted to 12 application binaries with distinct execution profiles: batch versus latency sensitive, low-level versus high-level services. The applications are: (1) Gmail and Gmail-fe,

[2]Perf is a profiler tool for Linux 2.6+ systems; it abstracts CPU hardware differences in Linux performance measurements.

[3]A *callstack*, also called execution stack, program stack, control stack, or run-time stack, is a data structure that stores information about the active subprograms invoked during the execution of a program.

[4]The Ivy Bridge-E family is made in three different versions with the postfix -E, -EN, and -EP, with up to six, ten, and twelve cores per chip, respectively.

the back-end and front-end Gmail application; (2) BigTable, a storage system discussed in Section 6.9; (3) *disk*, low-level distributed storage driver; (4) *indexing1* and *indexing2* of the indexing pipeline; (5) *search1, 2, 3* application for searching leaf nodes; (6) *ads*, an application targeting ads based on web page contents; (7) *video*, a transcoding and feature extraction application; and (8) *flights-search*, the application used to search and price airline flights.

In spite of the workload diversity, there are common procedures used by a vast majority of applications running on WSCs. Data-intensive applications run multiple tasks distributed across several servers and these tasks communicate frequently with one another. Cluster management software is also distributed and daemons running on every node of the cluster communicate with one or more schedulers making system-wide decisions.

It is thus, not unexpected, that common procedures that account for a significant percentage of CPU cycles are dedicated to communication. This is the case of RPCs (Remote Procedure Calls), as well as serialization, deserialization, and compression of buffers used by communication protocols. A typical communication pattern involves the following steps: (a) serialize the data to the protocol buffer; (b) execute an RPC and pass the buffer to the callee; and (c) caller deserializes the buffers received in response to the RPC. Data collected over a period of 11 months shows that these common procedures translate into an unavoidable "WSC architecture tax" and consume 22–27% of the CPU cycles.

About one third of RPCs are used by cluster management software to balance the load, encrypt the data, and detect failures. The balance of the RPC is used to move data between application tasks and system procedures. Data movement is also done using library functions such as *memmove* and *memcpy* with descriptive names[5] which account for 4–5% of the "tax."

Compression, decompression, hashing, and memory allocation and reallocation procedures are also common and account for more than one fourth of this "tax." Applications spend about one fifth of their CPU cycles in the scheduler and the other components the kernel. An optimization effort focused on these common procedures will undoubtedly lead to a better WSC utilization.

Most cloud applications have a substantial memory footprint, binaries of hundreds of MB are not uncommon and some do not exhibit either spatial or temporal locality. Moreover, the memory footprint of applications shows a significant rate of increase, about 30% per year. Also the instruction-cache footprints grow at a rate of some 2% per year. As more Big Data applications run on computer clouds neither the footprint nor the locality of these applications are likely to limit the pressure on cache and memory management. These functions represent a second important target for the performance optimization effort.

Memory latency rather than memory bandwidth affects a processor ability to deliver a higher level of performance through Instruction Level Parallelism (ILP). The performance of such processors is significantly affected by stall cycles due to cache misses. It is reported that data cache misses are responsible for 50–60% of the stall cycles and together with instruction cache missing contribute to a lower IPC (Instructions Per Clock cycles).

There are patterns of software that hinder execution optimization through pipelining, hardware threading, out-of-order execution, and other architectural features designed to increase the level of ILP. For example, linked data structures cause indirect addressing that can defeat hardware prefetching and build bursts of pipeline idleness when no other instructions are ready to execute.

[5] In Linux *memmove* and *memcpy* copy n bytes from one memory area to another; the areas may overlap for the former but do not overlap for the later.

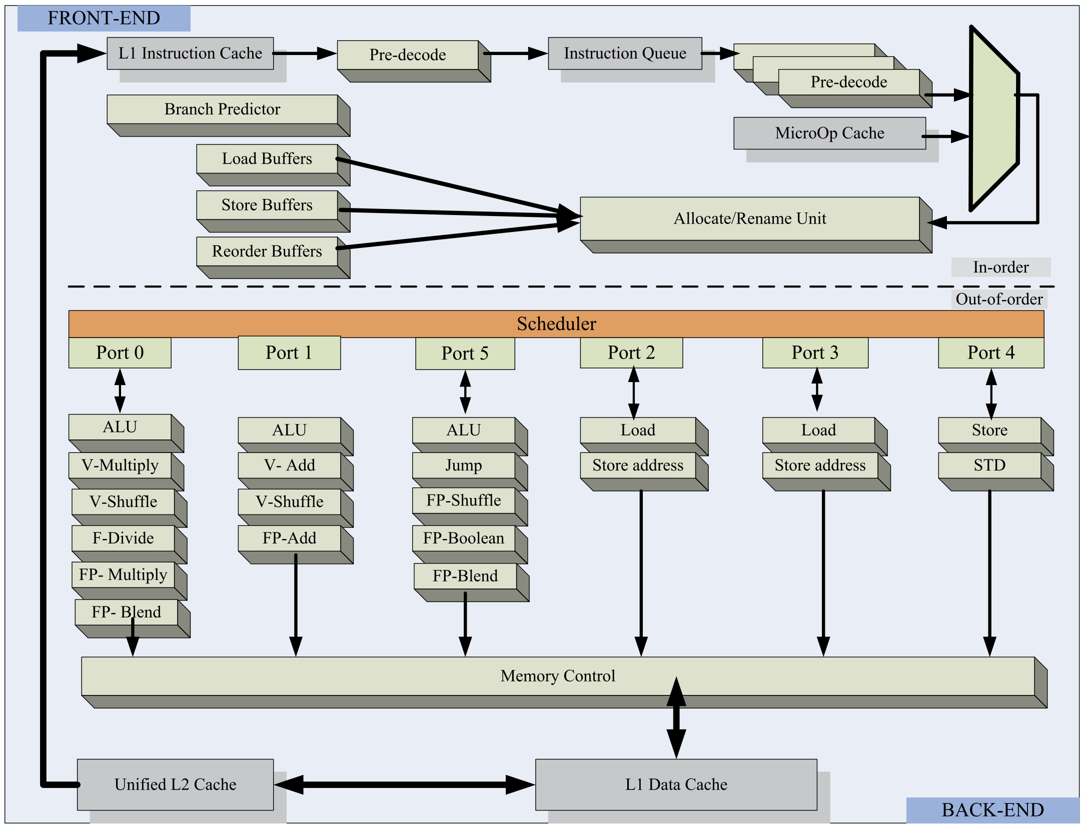


FIGURE 8.2

Schematic illustration of a modern processors core microarchitecture. Shown are the front-end and back-ends, L1 instruction and data caches, the unified L2 cache and the microoperation cache. Branch prediction unit, load/store and reorder buffers are components of the front-end. The instruction scheduler manages the dynamic instruction execution. The five ports of the instruction scheduler dispatch micro instructions to ALU and to load and store units. Vector (V) and floating-point operations (FP) are dispatched to ALU units.

Understanding cache misses and stalls requires a microarchitecture-level analysis. A generic organization of the microarchitecture of a modern core is illustrated in Figure 8.2. The core front-end processes instructions in order, while the instruction scheduler of the back-end is responsible for dynamic instruction scheduling and feeds the instruction to multiple execution units including Arithmetic and Logic Units (ALU)s and Load/Store units.

The microarchitecture-level analysis is based on a Top-Down methodology [535]. According to https://software.intel.com/en-us/top-down-microarchitecture-analysis-method: “The Top-Down characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application. Its aim is to show, on average, how well the CPU’s pipeline(s) were being utilized while running an application.”

This methodology identifies the *micro-op* (μop) queue as the separator between the front-end and the back-end components of a microprocessor core. The μop pipeline slots are then classified as *Retiring, Front-end bound, Bad speculation, Back-end bound*, with only the first one doing useful work. *Front-end bound* includes overheads associated with fetching, instruction caches, decoding and some other shorter-penalties and *Back-end bound* includes overheads due to the data cache hierarchy and the lack of ILP; *Bad speculation* is self-explanatory.

In a typical SPEC CPU 2006 benchmark the front-end wasted execution slots are typically 2–3 times lower than those reported for the Google workload which account for 15–30% of all wasted slots. One of the reasons for this behavior is that SPEC applications[6] do not exhibit the combination of low retirement rates[7] and high front-end boundedness of WSC ones.

Data shows that core back-end dominates the overhead and limit the ILP. The back-end and the front-end stalls limit the number of cores active during an execution cycle. To be more precise, only 1 or 2 cores of a 6 core Ivy Bridge processor are active in 72% of execution cycles, while 3 cores are active during the balance of 28% of cycles.

The observation that memory latency is more important than memory bandwidth is a consequence of the low memory bandwidth utilization at an average of 31% and a maximum of 68% with a heavy tail distribution. In turn, the low memory utilization is due in part to low CPU utilization. A surprising result reported in [262] is that the median CPU bandwidth utilization is 10% while [56] reports a median CPU utilization in a much higher range, 40–70%. A low CPU utilization is also reported for the CloudSuite [170].

Several conclusions regarding optimal processor architecture can be reached from the Top-Down data analysis. Data analysis shows that cloud workloads display access patterns involving bursts of computations intermixed with bursts of stall cycles. Processors supporting a higher level of *simultaneous multithreading* (SMT) are better equipped than current generations of 2-wide SMP processors to hide the latency by overlapping stall cycles. SMT is an architectural feature allowing instructions from more than one thread to be executed in any given pipeline stage at a time. SMT requires the ability to fetch instructions from multiple threads in a cycle; it also requires a larger register file to hold data from multiple threads.

The large working sets of the codes are responsible for the high rate of instruction cache misses. L2 caches show that MPKI (misses per kilo instructions) are particularly high. Larger caches would alleviate this problem but at the cost of higher cache latency. Separate cache policies which give priority to instructions over data or separate L2 caches for instructions and data could help in this regard.

[6]The following applications in the SPEC CPU2006 suite are used: *400.perlbench* which has high IPC and the largest instruction cache working set; *445.gobmk*, an application with hard-to-predict branches; *429.mcf* and *471.omnetpp* memory-bound applications which stress memory latency; and *433.milc* a memory-bound application which stresses memory bandwidth.

[7]In a modern processor the Completed Instruction Buffer holds instructions that have been speculatively executed. Associated with each executed instruction in the buffer are its results in rename registers and any exception flags. The retire unit removes from the buffer the executed instructions in program order, at a rate of up to four instructions per cycle. The retire unit updates the architected registers with the computed results from the rename registers. The retirement rate measures the rate of these updates.

8.4 HYPERVISORS

A hypervisor is the software that securely partitions the resources of a computer system into one or more VMs. A *guest OS* is an operating system that runs under the control of a hypervisor rather than directly on the hardware. A hypervisor runs in kernel mode, while a guest OS runs in user mode. Sometimes the hardware supports a third mode of execution for the guest OS.

Hypervisors allow several operating systems to run concurrently on a single hardware platform. A hypervisor controls how the guest OS uses the hardware resources; the events occurring in one VM do not affect any other VM running under the same hypervisor. At the same time, a hypervisor enables:

- Multiple services to share the same platform.
- The movement of a service from one platform to another, a process called *live migration*.
- System modification, while maintaining backward compatibility with the original system.

When a guest OS attempts to execute a privileged instruction the hypervisor traps the operation and enforces operation correctness and safety. The hypervisor guarantees the isolation of the individual VMs and thus, ensures security and encapsulation, a major concern in cloud computing. At the same time, the hypervisor monitors the system performance and takes corrective actions to avoid performance degradation. For example, to avoid memory thrashing a hypervisor may swap out a VM by copying to the disk all its pages and releasing the physical memory frames for paging by other VMs.

A hypervisor virtualizes the CPU and the memory. For example, a hypervisor traps interrupts and dispatches them to the individual guest operating systems. When a guest OS disables the interrupts, the hypervisor buffers them until the guest OS re-enables them. A hypervisor maintains a *shadow page table* for each guest OS and replicates any modification made by the guest OS in its own shadow page table. This shadow page table points to the actual page frame and it is used by the hardware component called the Memory Management Unit (MMU) for dynamic address translation.

Memory virtualization has important implications for performance. Hypervisors use a range of optimization techniques. For example, VMware systems avoid page duplication among different VMs, they maintain only one copy of a shared page and use copy-on-write policies, while Xen imposes total isolation of the VM and does not allow page sharing. Hypervisors control the virtual memory management and decide what pages to swap out. For example, when the ESX VMware Server wants to swap out pages it uses a *balloon process* inside a guest OS and requests it to allocate more pages to itself and thus, swaps-out pages of some of the processes running under that VM. Then it forces the balloon process to relinquish control of free page frames.

8.5 AN ENGINE FOR COARSE-GRAINED DATA-PARALLEL APPLICATIONS

When can we talk about coarse-grained parallelism and why is it important for the design of cloud software? The answer to the first question is that application developers have used the SPMD (Same-Program-Multiple-Data) paradigm for several decades. The name SPMD illustrates perfectly the idea behind the concept – a large dataset is split into several segments processed independently, and often concurrently, using the same program.

For example, converting a large number of images, e.g., 10^9, from one format to another can be done by splitting the set into one thousand segments with 10^6 images each, and then running concurrently the conversion program on one thousand processors. The answer to the second question is now obvious: to compute we need a large infrastructure, in this case one thousand processors, and as a result, the computing time is cut by almost three orders of magnitude.

It is easy to see that such applications are ideal for cloud computing, they need a large computing infrastructure and can keep the systems busy for a fair amount of time. To CSPs delight such jobs increase system utilization as no scheduling decisions have to be made until the time-consuming job has finished. This was noticed early on and, in 2004, the MapReduce idea was born [130].

MapReduce and Apache Hadoop, an open-source software framework consisting of a storage part, the Hadoop Distributed File System (HDFS), and the processing part called MapReduce are discussed in Chapter 7. MapReduce is a two-phase process. Computations on data segments are carried out during the Map phase; partial results are then merged during the Reduce phase. This extends the scope of SPMD for computations that are not totally independent.

The previous example can be slightly changed to benefit from MapReduce; instead of converting 10^9 images we now search during the Map phase for an object that could appear in any of them. After the search in each data segment is completed we combine the partial results during the Reduce phase to further refine the information about the object from the set of images selected during the Map phase.

Dryad is a general-purpose distributed execution engine developed in 2007 by Microsoft for coarse-grained data-parallel applications. Microsoft wanted to use Dryad for running Big Data applications on its clustered server environment as a proprietary alternative to Hadoop, a widely used platform for coarse-grained data-parallel applications.

A Dryad application combines computational *vertices* with communication links to form a dataflow graph [253]. Then it runs the application by executing the vertices of this graph on a set of available computers, communicating through files, TCP pipes, and shared-memory.

The system is centrally controlled by a *Job Manager* (JM) running either on one of the nodes of the cluster or outside the cluster, on the user's computer. The JM uses a *Name Server* (NS) to locate the nodes of the cluster where the work is actually done and an application-specific description to construct the dataflow graph of the application. A daemon running on each cluster node communicates with the JM and controls the execution of the code for the vertex of the graph assigned to that node. Daemons communicate directly among themselves without the intervention of the JM and use the information provided by the dataflow graph to carry out the computations.

A detailed description of the Dryad dataflow graph given in [253] presents a set of simpler graphs used to construct more complex one. The graph nodes are annotated to show the input and output datasets. Two connection operations are the pointwise composition and the complete bipartite composition.

The DryadLINQ (DLNQ) system is the product of a related Microsoft project [539]. It exploits the Language INtegrated Query (LINQ), a set of .NET constructs for performing arbitrary side effect-free transformations on datasets to automatically translate data-parallel codes into an workflow for distributed execution. The distributed execution is then executed on the Dryad platform. The development of DryadLINQ was motivated by the fact that parallel databases implement only declarative sides of SQL queries and do not support imperative programming. The LINQ-expression execution in DryadLINQ follows several steps:

1. The .NET application runs on the client machine and creates a DLNQ object.
2. The DLNQ object is passed to the DLNQ system via a *ToDryadTable* call.

3. DLNQ compiles the LINQ expression into a Dryad execution plan and invokes the Dryad JM which in turn creates the dataflow and then schedules the execution of the dataflow.
4. Each Dryad node daemon initiates the execution of the vertex allocated to it. When the execution of the vertex is completed DLNQ creates local *DryadTable* objects encapsulating the outputs of the execution that may be used as inputs to subsequent expressions in the user program.
5. Control returns to the user application and the *DryadTable* objects are accessible to the user via .NET.

Dryad is not scalable and to no one's surprise, soon after announcing plans to release Windows Azure- and Windows Server-based implementations of open source Apache Hadoop, Microsoft discontinued the project.

8.6 FINE-GRAINED CLUSTER RESOURCE SHARING

Mesos is a light-weight framework for fine-grained cluster resource sharing developed in late 2010s at U. C. Berkeley. Mesos consists of only some 10 000 lines of C++ code [237]. A novelty of the system is a two-level scheduling strategy for large clusters with workloads consisting of a mix of frameworks.

The term "framework" in this context means a widely-used, large consumer of CPU cycles, software system such as Hadoop, discussed in Chapter 7, and MPI (Message Passing Interface), a portable message-passing system used by the parallel computing community since 1990s. Another novelty is the concept of *resource offer*, an abstraction for a bundle of resources a framework can allocate to run its tasks on a cluster node.

The motivation for supporting two-level scheduling in Mesos is that centralized scheduling is not scalable due to its complexity. Centralized scheduling does not, and cannot, perform well for fine-grained resource sharing. Framework jobs consisting of short tasks are mapped to resource *slots* and the fine-grained matching has a high overhead and prevents sharing across frameworks. For example, a fair scheduler at Facebook allocates the resources of a 2 000 node cluster dedicated to Hadoop jobs. MPI and MapReduceOnline (a streaming of MapReduce discussed in Chapter 7) jobs for ad targeting need the data stored on the Hadoop cluster, but the frameworks cannot be mixed. Mesos allows multiple frameworks to share resources in a fine-grained manner and achieve data locality. It can isolate a production framework from experiments with a new version undergoing testing and experimentation.

Mesos runs on Linux, Solaris and OS X, and supports frameworks written in C++, Java, and Python. Mesos can use Linux containers to isolate tasks, CPU cores, and memory. Mesos is organized as follows: a *master* process manages daemons running on all cluster nodes, while frameworks run the tasks on cluster nodes. The master implements the fair-sharing of resource offers among frameworks and lets each framework manage resource sharing among its tasks.

Each framework has a *scheduler* which receives resource offers from the master. An *executor* on each machine launches the tasks of a framework. The scheduler runs

- *callbacks*,[8] such as *resourceOffer, offerRescinded, statusUdate, slaveLost*, and
- *actions*, such as *replyToOffer, setNeedsOffers, setFilters, getGuaranteedShare, killTask*.

[8]A callback is executable code passed as an argument to other code; the callee is expected to execute the argument either immediately or at a later time for synchronous and, respectively, asynchronous callbacks.

The executor functions are also *callbacks*, such as *launchTask, killTask* and *actions*, such as *sendStatus*.

Framework requests differentiate mandatory from preferred resources. A resource is *mandatory* if the framework cannot run without it, e.g., a GPU is a mandatory resource for applications using CUDA.[9] A resource is *preferred* if a framework performs better using a certain resource, but could also run using another one.

The two-level scheduling strategy keeps Mesos simple and scalable and, at the same time, gives the frameworks the power to optimally manage a cluster. The system is flexible and supports pluggable *isolation modules* to limit the CPU, memory, network bandwidth, and I/O usage of a process tree. Allocation modules can select framework-specific policies for resource management. For example, the killing of a task of a greedy or buggy framework is aware whether the tasks of a framework are interdependent, as in case of MPI, or independent, as in case of MapReduce.

The system is robust, there are replicated masters in hot-standby state. When the active master fails, a ZooKeeper service[10] is used to elect a new master. Then the daemons and the schedulers re-connect to the newly elected master.

The limitations of the distributed scheduling implemented by Mesos are also discussed in [237]. Sometimes, the collection of frameworks is not able to optimize bin packing as well as a centralized scheduler. Another type of fragmentation occurs when the tasks of a framework request relatively small quantities of resources and, upon completion, resources released by the tasks are insufficient to meet the demands of tasks from a different framework requesting larger quantities of resources. Resource offers could increase the complexity of framework scheduling. Centralized scheduling is also not immune to this problem.

It is reported that porting Hadoop to run on Mesos required relatively few modifications. Indeed, the *JobTracker* and the *TaskTrackers* components of Hadoop map naturally as Mesos framework scheduler and executor, respectively. Apache Mesos is an open-source system adopted by some 50 organizations including Twitter, Airbnb, and Apple (see http://mesos.apache.org/documentation/latest/powered-by-mesos/).

Several frameworks based on Mesos have been developed along the years. Apache Aurora was developed in 2010 by Twitter and now is open-source. Chronos is a cron-like[11] system, elastic and able to express dependencies between jobs. Apple uses a Mesos framework called Jarvis[12] to support Siri. Jarvis is an internal PaaS cloud service to answer IOS user's voice queries.

The utilization analysis of a production cluster with several thousand servers used to run production jobs at Twitter shows that the average CPU utilization is below 20% and the average memory utilization is below 50% [237]. Reservations improve the CPU utilization up to 80%. Some 70% of the reservations overestimate the resources they need by one order of magnitude, while 20% of the reservations underestimate resource needs by a factor of 5.

In summary, one can view Mesos as the opposite of virtualization. A VM is based on an abstraction layer encapsulating an OS together with an application inside a physical machine. Mesos abstracts

[9]CUDA is a parallel computing platform supporting graphics processing units (GPUs) for general purpose processing.

[10]ZooKeeper is a distributed coordination service implementing a version of the Paxos consensus algorithm, see Chapter 7.

[11]*Cron* is a job scheduler for Unix-like systems used to periodically schedule jobs; often it is used to automate system maintenance and administration.

[12]Jarvis is short for *Just A Rather Very Intelligent Scheduler.*

physical machines as pools of indistinguishable servers and allows a controlled and redundant distribution of tasks to these servers.

8.7 CLUSTER MANAGEMENT WITH BORG

A computer cluster may consist of tens of thousands of processors. For example, a cell of a WSC is in fact a cluster consisting of multiple racks, each with tens of processors, as shown in Figure 8.1. There are two sides of cluster management: one reflects the views of application developers who need simple means to locate resources for an application and then to control the use of resources; the other is the view of service providers concerned with system availability, reliability, and resource utilization.

These views drove the design of Borg, a cluster management software developed at Google [502]. Borg's design goals were:

- Manage effectively workloads distributed to a large number of machines and be highly reliable and available.
- Hide the details of resource management and failure handling thus, allow users to focus on application development. This is important as the machines of a cluster differ in terms of processor type and performance, number of cores per processor, RAM, secondary storage, network interface, and other capabilities.
- Support a range of long-running, highly-dependable applications. A first group of applications are long-running, production jobs and a second group are non-production, batch jobs.

A Borg cluster consists of tens of thousands of machines co-located and interconnected by a data center-scale network fabric. A cluster managed by Borg is called a *cell*. The architecture of the system shown in Figure 8.3 consists of a logically centralized controller, the *BorgMaster*, and a set of processes running on each machine in the cell, the *Borglets*. All Borg components are written in C++.

The main *BorgMaster* has five replicas and each replica maintains an in-memory copy of the state of the cell. The state of a cell is also recorded in a Paxos-based store on local disks of each replica. An elected master serves as Paxos leader and handles operations that change the state of a cell, e.g., submission a job or termination a task. The master process performs several actions:

- Handles client RPCs that change state or looks-up data.
- Manages state machines for machines, tasks, allocs, and other system objects.
- Communicates with *Borglets*.
- Responds to requests submitted to a web-based user interface.

Borglets start, stop, and restart failing tasks, manipulate the OS kernel setting to manage local resources, and report the local state to the *BorgMaster*.

Users interact with running processes by means of RPCs to *BorgMaster* and trigger task state transitions. Users request actions such as *submit, kill*, and *update*. The task state is also changed by system actions such as: *reject, evict, lost*, see Figure 8.4.

The *scheduler* is the other main component of the *BorgMaster*. The scheduler scans periodically in a round-robin order a priority queue of pending tasks. The *feasibility* component of the scheduling al-

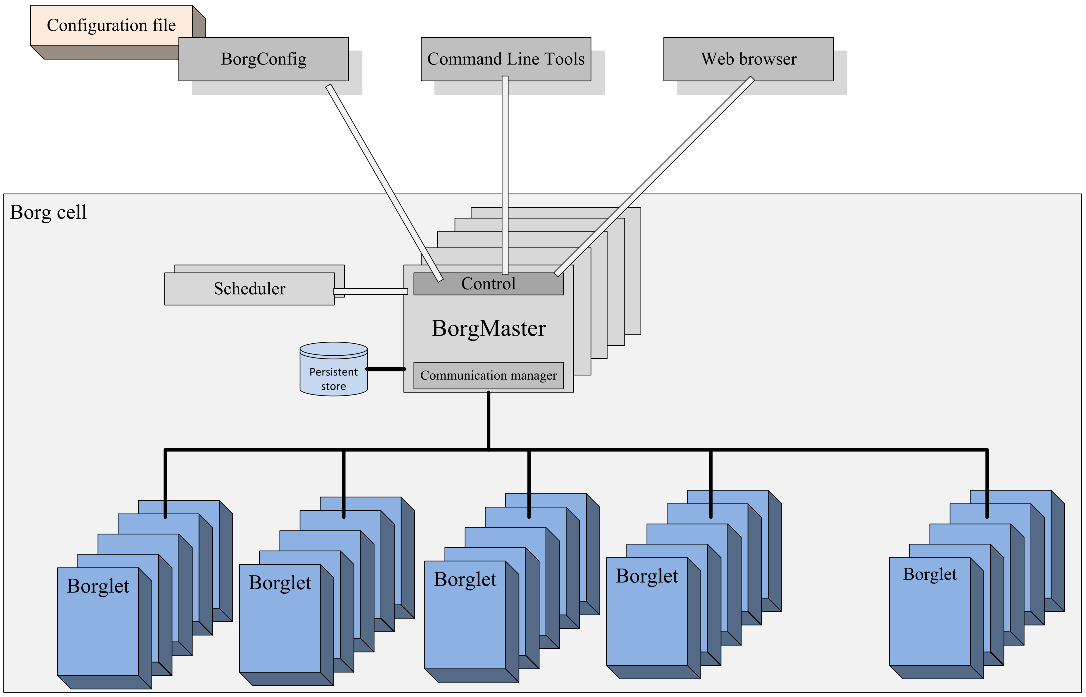


FIGURE 8.3

The architecture of Borg. A replicated *BorgMaster* interacts with *Borglets* running on each machine in the cell.

gorithm attempts to locate systems where to run tasks. The *scoring* component identifies the machine(s) to actually run the task.

Alloc and *alloc sets* reserve resources on a machine and, respectively, on multiple machines. Jobs have priorities; distinct *priority bands* are defined for activities such as monitoring, production, batch, and testing. A *quota system* for job scheduling uses a vector including the quantity of resources such as CPU, RAM, disk for specified periods of time. Higher-priority quota cost more than lower-priority ones. To simplify resource management and balance the load, a system similar with the one described in [27] is used to generate a single cost value per vector and minimize the cost, while balancing the workload and leaving room for spikes in demand.

Production jobs are allocated about 70% of CPU resources and 55% of the total memory [502]. A Borg job could have multiple tasks and runs in a single cell. The majority of jobs do not run inside a VM. Tasks map to Linux processes running in containers.

To manage large cells, the scheduler spawns several concurrent processes to interact with the *BorgMaster* and *Borglets*. These processes operate on cached copies of the cell state. Several other design decisions are important for system scalability. For example, to avoid frequent time-consuming machine and task scoring, Borg caches the score until the properties of the machine or task change significantly.

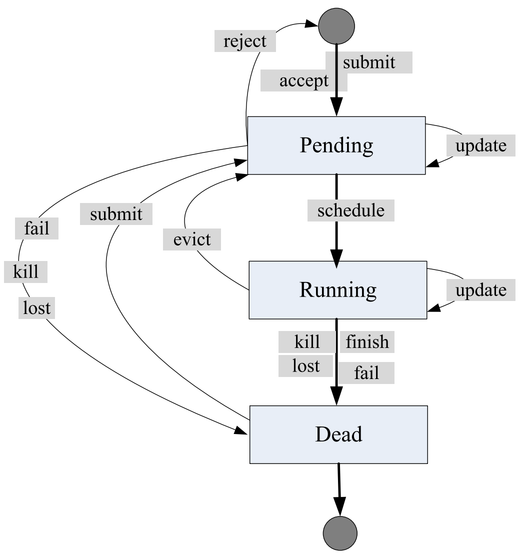


FIGURE 8.4

The states of a Borg task. Task state changes as a result of either user requests or system actions.

To avoid determining the feasibility of each pending task on every machine, Borg computes feasibility and scoring per equivalence classes of tasks, tasks with similar requirements. Moreover, this evaluation is not done for every machine in the cell but on random machines until enough suitable machines have been found.

The state of the *BorgMaster* can be saved as a *checkpoint* file and later used for studies of system performance and effectiveness, or to restore the state at a early point in time. A simulator, the *FauxMaster*, designed to identify system errors and performance problems by replaying checkpoint file facilitated the effort to improve the Borg system.

Results collected for a 12 000 server cluster at Google show an aggregate CPU utilization of 25–35% and an aggregate memory utilization of 40%. A reservation system raises these figures to 75% and 60%, respectively [416].

8.8 SHARED STATE CLUSTER MANAGEMENT

Could multiple independent schedulers do a better cluster management job than monolithic or two-level schedulers? The designers of the Omega system understood that efficient scheduling of large clusters is a very hard problem due to the scale of the system combined with the workload diversity and that only a novel approach should be conceived [446].

The workload of Google systems targeted by Omega was the mix of production/service and batch jobs discussed in Section 8.7. More than 80% of the workload are short batch jobs, spawning a large number of tasks. A larger share of resources, 55–80%, is allocated to production jobs running for extended periods of time and with fewer tasks than the batch jobs. The scheduling requirements are:

short turnaround time for the batch jobs and strict availability and performance target for the production jobs.

The workload of the scheduler increases with the cluster size and with the granularity of the tasks to be scheduled. The finer the task granularity, the more scheduler decisions have to be made, thus the higher is the likelihood of spatial and temporal resource fragmentation and lower resource utilization. The solution adopted in Omega's design is to allow multiple independent schedulers to access a *shared cluster state* protected with a lock-free optimistic concurrency control algorithm.

In Omega there is no central resource allocator and each scheduler has access to all cluster resources. A scheduler has its own private and frequently updated copy of the *shared cluster state*, a resilient master copy of the state of all cluster resources. Whenever it makes a resource allocation decision, a scheduler updates the shared cluster state in an *atomic transaction*.

In case of conflict among tasks, at most one commit succeeds and the resource is allocated to the winner. Then, the shared cluster state re-synchs with local copies of all schedulers. The losers may then retry at a later time to gain access to the resource and can be successful after the resource was released by the task holding it.

Multiple schedulers may attempt to allocate the same resource at the same time thus, there is the possibility of conflict. An optimal solution to this problem depends upon the frequency of conflicts. An optimistic approach increases parallelism and assumes that conflicts seldom occur and, when detected, they can be resolved efficiently. A pessimistic approach used by Mesos is to ensure that a resource is available to only one framework scheduler at a time.

Jobs typically spawn multiple tasks and the next question is whether all tasks of a job should be allocated all the resources they need at the same time when the job starts execution, a strategy called co-scheduling or gang scheduling. The alternative is to allocate resources only at the time when a task needs them. In the former case resources end up being idle until the tasks actually need them and the average resource utilization decreases. In the later case there is a chance of deadlock as some tasks need resources allocated to other tasks, while those holding these resources need the resources held by the first group of tasks.

Several metrics are useful to compare the effectiveness of large cluster schedulers. When we view scheduling as a service and a scheduling request as a transaction, the time elapsed from the instance a job is submitted until the time when the scheduler attempts to schedule a job is the waiting time, while the time required to schedule the job is the service time of the transaction. The waiting time and the service time are two important metrics for scheduler effectiveness. Conflict resolution is a component of the scheduler service time for a shared-state scheduler like Omega. The *conflict fraction* is the average number of conflicts per successful transaction.

The service time has two components, t_{sch_job}, the overhead for scheduling a job and t_{sch_task}, the time to schedule a task of the job; thus, the total time to make a scheduling decision for a job with n tasks is

$$t_{sch} = t_{wait} + t_{sch\_job} + n \times t_{sch\_tks}. \tag{8.1}$$

A monolithic scheduler using a centralized scheduling algorithm does not scale up. Another solution is to statically partition a cluster into sub-clusters allocated to different types of workloads. This policy is far from optimal due to fragmentation because the balance between different types of workload changes in time. A two-level scheduler has its own limitations, as we have seen in Section 8.6.

Table 8.2 A side-by-side comparison of four types of schedulers. The schedulers differ in terms of: (a) scope, the set of resources controlled; (b) possibility of conflict and conflict resolution method; (c) allocation granularity, gang co-scheduling versus task-by-task; and (d) scheduling policy.

Scheduler	Resources	Conflict	Granularity	Policy
Monolithic, e.g. Borg	All available resources	None	Global	Priority & preemption
Static partition, e.g. Dryad	Fixed subset of resources	None	Per-partition policy	Scheduler-dependent
Two-level, e.g. Mesos	Dynamic subset of resources	Pessimistic	Gang scheduling	Strict fairness
Shared-state, e.g. Omega	All available	Optimistic	Per-scheduler policy	Priority & preemption

Table 8.2 provides a side-by-side comparison of monolithic schedulers, schedulers for static-partitioned clusters, two-level schedulers, and multiple independent shared-state schedulers.

A light-weight simulator was used for a comparative study of Omega and the other schedulers. The two-level scheduling model in this simulator emulates Mesos and achieves fairness by alternately offering all available cluster resources to different schedulers. It assumes low-intensity tasks, thus resources become available frequently and scheduler decisions are quick. As a result, a long scheduler decision time means that subsets of cluster resources are unavailable to other schedulers for extended periods of time. The simulation results show that Omega is scalable and at realistic workloads there is little interference among independent schedulers. The simulator was also used to investigate gang scheduling [39] useful for MapReduce applications.

A more accurate, trace-driven scheduler can be used to gain further insights into scheduler conflicts. Trace-driven simulation is quite challenging; moreover, a large number of simplifying assumptions limit the accuracy of the simulator results. Neither the machine failures, nor the disparity between resource requests and the actual usage of those resources in the traces are simulated by the trace-driven simulator designed for Omega. The results produced by the trace-driven simulator are consistent with the ones provided by the light-weight simulator.

8.9 QOS-AWARE CLUSTER MANAGEMENT

Quality of Service (QoS) guarantees are important for the designers of cloud applications and for the users of computer clouds who wish to enforce a well-defined range of response time, execution time, or other significant performance metrics for their cloud workloads. Enforcing workload constraints is far from trivial thus, few cluster management systems could legitimately claim adequate QoS support.

Two aspects of cluster management, resource allocation and resource assignment, play a key role for supporting QoS guarantees. *Resource allocation* is the process of determining the amount of resources needed by a workload while *resource assignment* means identifying the location of resources that satisfy an allocation. Both aspects of resource management require the ability to classify a given workload as a member of one of several distinct classes. Once classified, allocate to the workload precisely the amount of resources typical for that class, assign the resources, monitor the workload execution, and adjust this amount if needed.

QoS and workload classification. Workload classification is a challenging problem due to the wide spectrum of system workloads. An effective filtering mechanism is thus needed to support a classification algorithm capable to make real-time decisions. Classification is widely used by recommender systems such as the one used by Netflix. A recommender system seeks to predict the preference of a user for an item by filtering information from multiple users regarding the item. Such systems are used to recommend research articles, books, movies, music, news, and any imaginable item.

A short diversion to the Netflix Challenge [60,136] could help understanding the basis of the classification technique for a QoS-aware cluster management. The Netflix Challenge uses Singular Value Decomposition (SVD) and PQ-reconstruction [57,508]. The input to SVD is a sparse matrix A of rank r describing a system of n viewers and m movies to be decomposed as the product of three matrices U, Σ, and V:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} = U \Sigma V^t \tag{8.2}$$

with

$$U_{m,r} = \begin{pmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,r} \\ u_{2,1} & u_{2,2} & \dots & u_{2,r} \\ \vdots & \vdots & \vdots & \vdots \\ u_{m,1} & u_{m,2} & \dots & u_{m,r} \end{pmatrix} \quad \text{and} \quad V_{r,n} = \begin{pmatrix} v_{1,1} & v_{1,2} & \dots & v_{1,n} \\ v_{2,1} & v_{2,2} & \dots & v_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ v_{r,1} & v_{r,2} & \dots & v_{r,n} \end{pmatrix} \tag{8.3}$$

Σ, the diagonal matrix of singular values of matrix A is

$$\Sigma_{r,r} = \begin{pmatrix} \sigma_{1,1} & 0 & \dots & 0 \\ 0 & \sigma_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_{r,r} \end{pmatrix} \tag{8.4}$$

The ratings are the matrix elements $a_{ij} \in A,\ 1 \le i \le m,\ 1 \le j \le n$. SVD decomposes matrix A as $A = U \cdot \Sigma \cdot V^t$ with U the matrix of left singular vectors representing correlation between rows of A and the similarity values, Σ is the matrix of similar values, and V the matrix of right singular vectors representing the correlation between columns of A and the similarity values.

PQ-reconstruction with Stochastic Gradient Descent (SGD) [57] uses matrices U and V to reconstruct the missing entries in matrix A. The initial reconstruction of A uses $P^t = \Sigma \cdot V^t$ and $V = U$. SGD iterates over the elements $R = [r_{ui}]$ of $R = Q \cdot P^t$ until convergence. The iteration process uses two empirically determined values, η and λ, the *learning rate* and the SGD *regularization factor*, respectively. Two parameters, μ and b_u, the average rating and a user bias that account for the divergence of some viewers from the norm, respectively, are also used. The error ϵ_{ui} and the values q_i and p_u at each iteration are computed as

$$\begin{aligned} \epsilon_{ui} &\leftarrow r_{ui} - \mu - b_u - q_i \cdot p_u^t \\ q_i &\leftarrow q_i + \eta(\epsilon_{ui} p_u - \lambda q_i) \\ p_u &\leftarrow p_u + \eta(\epsilon_{ui} q_i - \lambda p_u) \end{aligned} \tag{8.5}$$

The iteration continues until the L2 norm of the error becomes arbitrarily small

$$| \epsilon |_{L2} = \sqrt{\sum_{u,i} | \epsilon_{ui} |^2} \leq \epsilon. \tag{8.6}$$

The convergence speed of stochastic gradient descent is limited by the noisy approximation of the true gradient. When the gains decrease too slowly, the variance of the parameter estimate decreases equally slowly. When the gains decrease too quickly, the expectation of the parameter estimate takes a very long time to approach the optimum. SVD complexity is $\mathcal{O}(\min[n^2 m, m^2 n])$ and the complexity of PQ-reconstruction with SGD is $\mathcal{O}(n \times m)$.

Quasar. A performance-centric approach for cluster management is implemented by two systems developed at Stanford University, Quasar [137] and its predecessor Paragon [135]. While Paragon handles only resource assignment, Quasar implements resource allocation, as well as resource assignment.

Quasar is implemented in C, C++, and Python, runs on Linux and OS X, and supports applications written in C/C++, Java, and Python. The applications run unaltered, there is no need to modify them for running under Quasar.

Quasar is based on several innovative ideas. One of them regards reservation systems and the realization that users are seldom able to accurately predict the resource needs of their applications. Moreover, performance isolation, though highly desirable, is hard to implement. As a result, the execution time and the resources used by an application can be affected by other applications sharing the same physical platform(s). This is the reason why reservation systems often lead to underutilization of resources and why QoS guarantees are seldom offered.

Quasar presents a high-level interface to allow users, as well as schedules integrated in widely-used frameworks, to express constraints for the workloads they manage. These constraints are then translated into actionable resource allocation decisions. Classification techniques are then used to evaluate the impact of these decisions on all system workloads.

Performance constraints differ for different types of workloads. For transaction processing systems the system bandwidth, expressed as number of queries per second, represents a meaningful constraint as it reflects the response time experienced by users. For large batch workloads, e.g., the ones involving frameworks such as Hadoop, the execution time captures the expectations of the end-users.

To operate efficiently, the classification algorithms is based on four parameters: resources per node, the number of nodes for allocation, the server type, and the degree of interference for assignment. The results of the four independent classifications are combined by a greedy algorithm used to determine as accurately as feasible the set of resources needed to satisfy the performance constraints. The system constantly monitors the performance of the workload and adjusts the allocations if feasible.

Rather than relying exclusively on user's characterization of the workload, the classification system combines information gathered about the workload from prescreening with information from a database about past workloads. Once accepted in the system, a workload is profiled during a short execution on a few servers with two randomly-selected scale up allocations.

For example, Hadoop workloads are profiled for a small number of map tasks and two configurations of parameters, such as the number of mappers per node, the size of Java AM heap, the block size, the amount of memory per task, the replication factor, and the compression factor. The configuration data is then uploaded into a matrix with workloads as rows and scale up configurations as columns. To constrain the number of columns the vectors are quantized to integer multiples of cores and blocks of memory and storage. A configuration includes all relevant data for the workload.

The classification engine and the scheduler. The classification engine distinguishes between the allocation of more servers to a workload, called *resource scale out*, and additional resources from servers already allocated to the workload, called *resource scale up*. The Quasar classification engine carries out for each workload four classifications for scale up, scale out, heterogeneity, and interference. Some workloads may require both types of scaling, others one type or another. For example, Quasar may monitor the number of queries per second and the latency for a web server and apply both types of scaling. Initially, Quasar was focused on compute cores, memory and storage capacity with the expectation to cover also the network bandwidth soon.

In addition to scale up and scale out, there are two other types of classifications, heterogeneity and interference. To operate with a matrix of small dimensions and thus, reduce computational complexity, the four types of classifications are done independently and concurrently. The greedy scheduler combines data from the four classifications.

The scale up classification evaluates how the number of cores, the cache and the memory size affect performance. The scale out classification is only applied to several types of workloads that can use multiple servers and profiling is done with the same parameters as for the scale up classification. Heterogeneity classification is the result of profiling the workload on several randomly selected servers. Lastly, interference classification reflects the sensitivity and tolerance of other workloads using shared cores, cache, memory, and communication bandwidth.

The objective of the greedy scheduler is to allocate to each workload the least amount of resources allowing it to meet its SLO (Service Level Objectives).[13] For each allocation request the scheduler ranks available servers based on the resource quality, e.g., the sustainable throughput combined with minimal interference. First, it attempts vertical scaling, allocating more resource on each node, then, if necessary, switches to horizontal scaling allocating additional nodes, while keeping the total number of nodes as low as feasible.

Resources are allocated to applications on a FCFS basis. This could led to sub-optimal assignments but such assignments can be easily detected by sampling a few workloads. The scheduler also implements admission control to prevent oversubscription.

In summary, Quasar provides QoS guarantees and, at the same time, increases resource utilization. The process starts with an initial profiling of an application with short runs. The information from the initial profiling is expanded with information regarding four factors that can affect performance, scale up, scale out, heterogeneity, and interference. Then, the greedy scheduler uses the classification output to allocate resources that allow SLO compliance and maximize resource utilization.

8.10 RESOURCE ISOLATION

A recurring theme of this chapter is that cluster management systems must perform well for a mix of applications and deliver the performance promised by the strict SLOs for each workload. The dominant

[13]A service level objective is a key element of a SLA. SLOs are agreed upon as a means of measuring the performance of the CSP and are a way of avoiding disputes between the users and the CSP based on misunderstanding.

components of the application mix are *latency-critical* (LC) workloads, e.g., web search, and *best-effort* (BE) batch workloads, e.g., Hadoop. The two types of workloads share the servers and compete with one another for their resources.

The resource management systems discussed up to now act at the level of a cluster, but cannot be very effective at the level of individual servers or processors. First, they cannot have accurate information simply because the state of processors changes rapidly and communication delays prohibit a timely reaction to these changes. Second, a centralized, or even a distributed system for fine-grained server-level resource tuning would not be scalable.

Each server should react to changing demands and dynamically alter the balance of resources used by co-located workloads. A system with feedback is needed to implement an *iso-latency policy*, in other words to supply sufficient resource so that SLOs are met. More bluntly, this means allowing LC workloads to expand their resource portfolio at the expense of co-located BE workloads.

This is the basic philosophy of the Heracles system developed at Stanford and Google [311]. In this section we discuss the realtime mechanisms used by Heracles controller to isolate co-located workloads. In this context the term "isolate" means to prevent the best-effort workload to interfere with the SLO of the latency-critical workload.

Latency-critical workloads. A closer look at three Google latency-critical workloads, *websearch, ml_cluster*, and *memkeyval* helps us better understand why resource isolation is necessary for co-located workloads. The first, *websearch*, is the query component of the web search service. Every query has a large fan-out to thousands of leaf nodes, each one of them processing the query on a shard of the search index stored in DRAM. Each leaf node has a strict SLO of tens of milliseconds. This task is compute- intensive as it has to rank search hits and has a small working set of instructions, a large memory footprint, and a moderate DRAM bandwidth.

The second, *ml_cluster*, is a standalone service using machine-learning for assigning a snippet of text to a cluster. Its SLO is also of tens of milliseconds. It is slightly less CPU intensive, requires a larger memory bandwidth and lower network bandwidth than *memkeyval*. Each request for this service has a small cache footprint but a high rate of pending requests put pressure on the cache and DRAM.

The third, *memkeyval*, is an in-memory key-value store used by the back-end of the web service. Its SLO latency is of hundreds of microseconds. The high request rate makes this service compute-intensive mostly due to the CPU cycles needed for network protocol processing.

Sharing resources of individual servers is complicated because the intensity of LC workloads at any given time is unpredictable, therefore their latency constrains are unlikely to be satisfied at times of peak demand unless special precautions are taken. Resource reservation at the level needed for peak demand of LC workloads may come to mind first. But this naive solution is wasteful, it leads to low or extremely low resource utilization, thus, the need for better alternatives.

Processor resources. Processor resources subject to dynamic scaling and the mechanisms for resource isolation for each one of them are discussed next. Physical cores, caches, DRAM, power supplied to the processor, and network bandwidth are all resources affecting the ability of an LC workload to satisfy its SLO constraints. Individual resource isolation is not sufficient, cross-resource interactions deserve close scrutiny. For example, contention for cache affects DRAM bandwidth; a large network bandwidth allocated to query processing affects CPU utilization as communication protocols consume a large number of CPU cycles.

Processor cores are the engines delivering CPU cycles and an obvious target for dynamic, rather than static allocation for co-located workloads. This problem is complicated by Hyper-threading (HT) in multicore Intel processors. HT is a proprietary form of SMT (simultaneous multi-threading) discussed in Section 8.3. HT takes advantage of superscalar architecture and increases the number of independent instructions in the pipeline. The OS uses two virtual cores for each physical core and shares the workload between them whenever possible. Sharing between the two virtual cores interferes with the instruction execution, shared caches, and TLB [14] operations.

Dynamic frequency scaling is a technique for adjusting the clock rate for cores sharing a socket. The higher the frequency, the more instructions are executed per unit of time by each core, and the larger is the processor power consumption. Clock frequency is related to the operating voltage of the processor. The *dynamic voltage scaling* is a power conservation technique often used together with frequency scaling, thus the name *dynamic voltage and frequency scaling* (DVFS).

The *overclocking* techniques based on DVFS opportunistically increase the clock frequency of processor cores above the nominal rate when the workload increases. To allow the cores of an Intel processor to adjust their clock frequency independently, the *Enhanced Intel SpeedStep* technology option should be enabled in the BIOS.[15] To lower best-effort workloads, Heracles reduces the number of cores assigned to best-effort tasks.

The cycle stalls limit the effective IPC (instructions per clock cycle) of individual cores. This means that the shared *last-level cache*[16] is another critical resource shared by the LC and BE co-located workloads and should be dynamically allocated. Lastly, the DRAM bandwidth can greatly affect the performance of applications with a large memory footprint.

One answer to the question on how to implement an isolation mechanism allowing LC workloads to scale up could be to delegate this task to the local scheduler. Why not use existing work-conserving[17] real-time schedulers such as SCHED_FIFO or CFS, the Completely Fair Scheduler? A short detour to the world of real-time schedulers used by most operating systems should convince ourselves that these schedulers are designed to support data streaming and cannot satisfy SLO requirements of LC tasks.

The SCHED_FIFO scheduler allocates the CPU to a high priority process for as long the process wants it, subject only to the needs of higher-priority realtime processes. It uses the concept of "real-time bandwidth," *rt_bandwidth*, to mitigate the conflicts between several classes of processes; once a process exceeds its allocated *rt_bandwidth*, it is suspended. The bandwidth of LC tasks changes so SCHED_FIFO scheduler cannot satisfy the SLO requirements of LC tasks.

CFS uses a red-black tree[18] in which the nodes are structures derived from the general *task_struct* process descriptor with additional information. CFS is based on the "sleeper fairness" concept enforcing the rule that interactive tasks which spend most of their time waiting for user input or other events

[14]TLB, (translation look-aside buffer) can be viewed as a cache for dynamic address translation; it holds the physical address of recently used pages in virtual memory.

[15]The basic input/output system (BIOS) is invoked after a computer system is powered on to load the OS and later to manage the data flow between the OS and devices such as keyboard, mouse, disk, video adapter, and printer.

[16]Multicore processors have multiple level caches. The last level cache (LLC) is the cache called before accessing memory. Each core has its own L1 I-cache (instruction cache) and D-cache (data cache). Sometimes two cores share the same unified (instruction + data) L2 cache and all cores share an L3 cache. In this case the highest shared LLC is L3.

[17]A work-conserving scheduler tries to keep the resources busy, if there is work to be done, while a non-work conserving scheduler may leave resources idle while there is work to be done.

[18]A red-black tree is a self-balancing binary search tree where each node has a "color" bit (read or black) to ensure the tree remains approximately balanced during insertions and deletions.

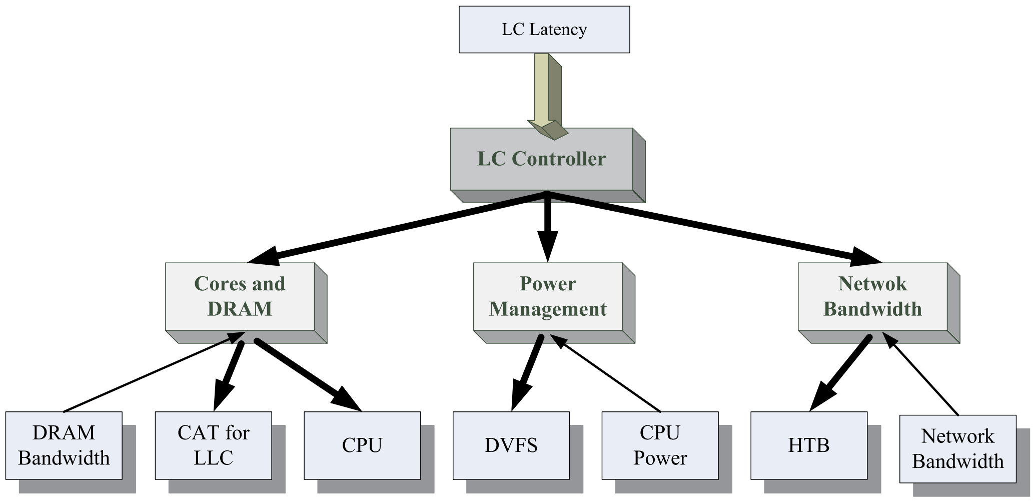


FIGURE 8.5

The organization of Heracles. The system runs on each server and controls isolation of a single LC workload and multiple best-effort jobs. The LC controller acts based on information regarding latency constraints of the LC workload and manages three resource controllers for: (A) Core and DRAM – uses CAT for LLC management and acts based on DRAM bandwidth data; (B) Power management – acts on DVFS using information on CPU power consumption; and (C) Network bandwidth – enforces bandwidth limits for outgoing traffic from the best-effort tasks using the *qdisc* scheduler with HTB (token bucket queuing) discipline with information provided by the network bandwidth monitoring system.

get a comparable share of CPU time as other processes, when they need it. As threads are spawned for every new query request we see that this approach is not satisfactory either.

Communication bandwidth sharing inside the server is controlled by the OS. Linux can be configured to guarantee outgoing bandwidth for the latency-critical workload. For incoming traffic it is necessary to throttle the core allocation until flow-control mechanisms of communication protocols are triggered. Communication among servers is supported by the cluster interconnection fabric and can be ensured by communication protocols that give priority to short messages typical for the latency-critical workloads.

Architectural support is needed for workload isolation at the microarchitecture level. Newer generations of Intel processors such as Xeon E5-2600 v3 family provide the hardware framework to manage a shared resource, like last level cache through Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT). CMT allows an OS or a hypervisor to determine the usage of cache by applications running on the platform. It assigns a Resource Monitoring ID (RMID) to each of the applications or VMs scheduled to run on a core and monitors cache occupancy on a per-RMID basis. CAT allows access to portions of the cache according to the *class of service* (COS).

Heracles organization and operation. Heracles runs as a separate instance on each server and manages the local interactions between the latency-critical and best-effort jobs. Figure 8.5 shows the latency-critical controller and the three resource controllers for cores, cache, and DRAM, for power management, and for communication bandwidth. The controller uses the *slack*, the difference between the SLO target and the tail of the measured performance index. A negative value of the latency slack means that the latency-critical workload has increased in intensity and is getting close to exceeding its SLO latency thus, it requires more resources. The operation of the latency-critical controller is described by the following pseudocode:

```
while True
     latency = Poll_Latency-critical -AppLatency
     load  = Poll_latency-critical -AppLoad
     slack = (target - latency)/target
     if slack < 0
          Disable Best-effort
          EnterCoolDown()
     elseif load > 0.85
          Disable best-effort
     elseif load < 0.80
          Enable Best-effort
     else if slack < 0.10
          Dissallow Best-effort Growth
          if slack < 0.05
          Best-effort_core.RemoveTwoCores
sleep{15}
```

The latency-critical controller is activated every 15 units of time and uses as input the latency-critical application latency and its load. When the slack is negative or when the latency-critical load is larger than 80% of capacity, the best-effort application is disabled. If the slack is less than 10% the best-effort task is not allowed to grow and when the slack further decreases to 5% then two cores allocated to best-effort tasks are removed. Best-effort is enabled when the latency-critical load is less than 80%.

There is a strong interaction between the number of cores allocated to a workload, its LLC cache, and DRAM requirements. This strong correlation explains why a single specialized controller is dedicated to the management of cores, LLC, and DRAM. The main objective of this controller is to avoid memory bandwidth saturation. The high-water mark that triggers action is 90% of the peak streaming DRAM bandwidth measured by the value of hardware counters for per-core memory traffic. When this limit is reached cores are removed from best-effort tasks.

When the DRAM bandwidth is not saturated, the gradient descent method is used to find the largest number of cores and cache partitions by alternating between increasing the number of cores and increasing the number of cache partitions allocated to best-effort tasks subject to the condition that the SLO of latency-critical tasks are satisfied. The power controller determines if there is sufficient power slack to guarantee latency-critical SLO with the lowest clock frequency. The system was evaluated with the three latency-critical workloads and the results show an average 90% resource utilization without any SLO violations.

8.11 IN-MEMORY CLUSTER COMPUTING FOR BIG DATA

The distinction between system and application software is blurred in cloud computing. The software stack includes components based on abstractions that combine aspects of applications and system management. Two systems developed at U. C. Berkeley, Spark [542] and Tachyon [301] are perfect examples of such elements of a cloud software stack.

It is unrealistic to assume that very large clusters could accommodate in-memory storage of Petabytes or more in the foreseeable future. Even if storage costs will decline dramatically, the intensive communication among the servers will limit the performance. There are iterative and other classes of Big Data applications where a stable subset of the input data is used repeatedly. In such cases dramatic performance improvements can be expected if a *working set* of input data is identified, loaded in memory, and kept for future use.

Obvious examples of such applications are those involving multiple databases and multiple queries across them and interactive data mining involving multiple queries over the same subset of data. Another well-known example of such an iterative algorithm is the *PageRank* algorithm [75] where data sharing is more complex. At each iteration i a document with *rank* $r^{(i)}$ and n neighbors sends a contribution of $\frac{r^{(i)}}{n}$ to each one of them. Then it updates its own rank as

$$r^{i+1} = \frac{\alpha}{N} + (1-\alpha)\sum_{j=1}^{n} c_j \tag{8.7}$$

with α a dumping factor, N the number of the documents in the database, and the sum over all contributions it received.

A distributed shared-memory (DSM) is a solution to in-memory data reuse. DSM allows fine-grained operations yet, access to individual data elements is not particularly useful for the class of applications discussed in this section. DSM does not support effective fault-recovery and data distribution, and does not lead to significant performance improvements. Ad hoc solutions to in-memory data reuse for different frameworks have been implemented, e.g., HaLoop [79] for MapReduce.

The question is whether a data sharing abstraction suitable for a broad class of applications and use cases can be developed for supporting a restricted form of shared memory based on *coarse-grained* transformations. This abstraction should provide a simple, yet expressive, user-interface allowing the end-user to describe data transformations, as well as powerful behind-the-scene mechanisms to carry out the data manipulations in a manner consistent with the system configuration and the current state of the system.

A data sharing abstraction. The concept of *Resilient Distributed Dataset* (RDD), for fault-tolerant, parallel data structures was introduced in [541]. RDD allows a user to keep intermediate results and optimizes their placement in the memory of a large cluster. The user interface of RDD exposes: (1) partitions, atomic pieces of the dataset; (2) dependencies on parent RDD; (3) a function for constructing the dataset; and (4) metadata about data location.

Spark provides a set of operators to efficiently manipulate such persistent datasets using a set of coarse-grained operations such as *map, union, sample* and *join*. *Map* creates an object with the same partitions and preferred locations as its parent, but applies the function used as an argument to the call to the *iterator* method applied to the parent's records. *Union* applied to two RDDs returns an RDD whose partitions are the union of the partitions of the two parents. *Sampling* is similar to *map*, but the RDD stores for each partition a random number generator to deterministically sample parent records. *Join* creates an RDD with either two narrow, two wide or mixed dependencies.

At run time, a driver program created by the user launches multiple workers to read data from a distributed file system such as HDFS and distribute it across multiple RDD partitions. Tasks locality is ensured by the delay scheduling algorithm [541] used by the Spark's scheduler. If a task fails the system restarts it on a different node if the parent is available. Partitions too large to fit in memory are stored on the secondary storage, possibly on solid state disks, if available.

Spark [542] and RDDs are restricted to I/O-intensive applications performing bulk writing. Spark and Tachyon [301], discussed later in this section, share the concept of *lineage* to support error recovery without the need to replicate the data. Lineage means to trace back descendents from a common ancestor and in the context of this discussion it means that lost output is recovered by carrying out again the tasks that created the lost data in the first place.

Spark driver programs. Imagine that an application wants to access a large log file stored in HDFS as a collection of lines of text. We wish to: (i) create a persistent dataset called *errors* of lines starting with the prefix "ERROR" distributed over the memory of the cluster; (ii) count the number of lines in this persistent dataset; (iii) count errors containing the string "MySQL;" and (iv) return the time of the errors, assuming that time is the third field in a tab-separated format of an array called HDFS. The following self-explanatory Spark code will do the job

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.count
errors.filter(_.contains("MySQL")).count()
errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

Notice that the *lines* dataset is not stored, only the much smaller *errors* dataset is stored in memory and used for the three actions. Behind the scene, the Spark scheduler will dispatch a set of tasks carrying the last two transformations to the nodes where the cached partitions of *errors* reside.

The Spark code for the *PageRank* algorithm summarized in Equation (8.7) is

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
// Build RDD of (targetURL, float) pairs with contributions sent by each page
val contribs = links.join(ranks).flatMap {
   (url, (links, rank)) =>links.map(dest => (dest, rank/links.size))
   }
// Sum contributions by URL and get new ranks
ranks = contribs.reduceByKey((x,y) => x+y).mapValues(sum => a/N + (1-a)*sum)
}
```

The *map, reduce*, and *join* operations and the lineage datasets of the graph of the several iterations of the *PageRank* algorithm are shown in Figure 8.6.

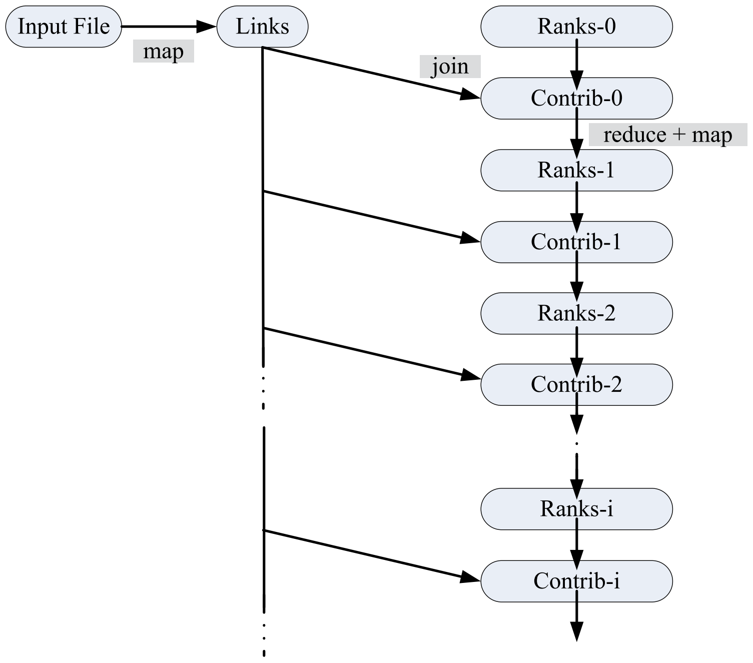


FIGURE 8.6

The lineage dataset for several iterations of the *PageRank* algorithm and the *map, reduce*, and *join* operations.

A new *ranks* dataset is created at each iteration and it is wise to call the *persist* action to save the dataset on secondary storage using the *RELIABLE* argument and reduce the recovery time in case of system failure. The *ranks* can be partitioned in the same way as the *links* to ensure that the *join* operation requires no communication.

Spark dependencies. An important design decision in Spark was to distinguish between *narrow* and *wide* dependencies. The former allow only one RDD partition to use the parent RDD and allow pipelined execution on one cluster to compute all parent partitions; for example, marrow partitions allow the application of a *map* followed by a *filter* on an element-by-element basis. In addition, recovery after a node failure is more efficient for narrow dependencies, only the lost parent needs to be recomputed and this can be done in parallel.

On the other hand, wide dependencies allow multiple children to depend on a single parent. Data from all parent partitions must be available and shuffled across the nodes for MapReduce operations. This also complicates recovery after a node failure. Figure 8.7 shows the effect of dependencies on RDD partitions as a result of *map, filter, union, join* and *group* operations.

Persistent RDD can be stored in memory either as deserialized Java objects or as serialized data and can also be stored on disk. Lineage is a very effective tool to recover RDDs after a node failure. Spark also supports checkpointing and this is particularly useful for large lineage graphs.

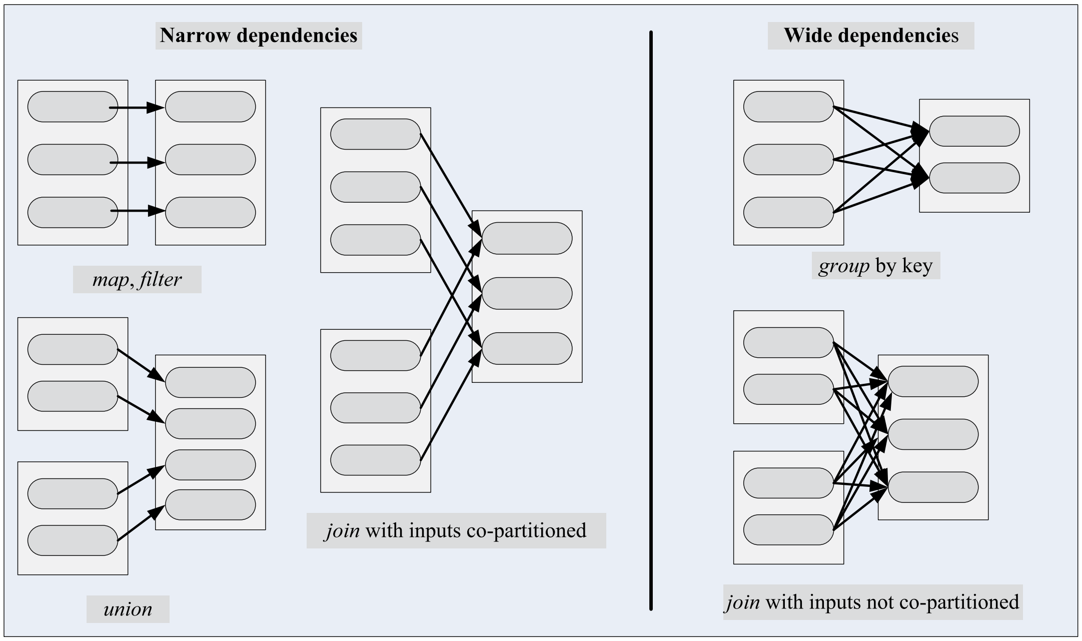


FIGURE 8.7

Narrow and wide dependencies in Spark. Arrows show the transformation of the partitions of one RDD due to *map, filter*, two RDDs for *union*, and two RDDs for *join* with inputs co-partitioned for narrow dependencies, when only one partition of RDD is allowed to use the parent RDD. For wide dependencies, when multiple children depend upon a single parent, two transformations are presented: a *group* by key of one RDD and *join* with input not co-partitioned for two RDDs.

According to [542] "Spark is up to 20 times faster than Hadoop for iterative applications, speeds-up a data analytics report 40 times and can be used interactively to scan a 1 TB dataset with 5–7 seconds latency." It is also very powerful, only 200 lines of Spark code implement the HaLoop model for MapReduce applications. HaLoop [79] extends MapReduce with programming support for iterative applications and improves efficiency by adding various caching mechanisms and by making the task scheduler loop-aware.

These results show that caching improves dramatically the performance of Big Data applications running on storage systems such HDFS which support only append operations. Lost data can be recovered by lineage, there is no need for replication of immutable data. On the other hand, fault-tolerance of applications involving write operations is more challenging. Fault tolerance based on data replication incurs a significant performance penalty when a data item is written on multiple nodes of a cluster.

The write bandwidth throughput for both hard disks and solid-state disks is three orders of magnitude lower than the memory bandwidth. Only the random access latency of solid-state disks is much lower than the latency of hard disks, their sequential I/O bandwidth is not larger. The network bandwidth is also orders of magnitude lower than the memory bandwidth. The system discussed next addresses precisely the problem of supporting fault-tolerance for in-memory datasets where both read and write operations must be supported.

Tachyon. The system was designed for high throughput in-memory storage for applications performing both extensive reads and writes [301]. The name of the system targeting Big Data workloads discussed in Chapter 12, "Tachyon,"[19] was most likely chosen to reflect the performance of the system, in Greek "tachy" means "fast." To recover lost data the system exploits the lineage concept used also by Spark and avoids data replication which could dramatically affect its performance.

To support read, as well as write operations, fault-tolerant, in-memory caching of datasets requires answers to several challenging questions:

1. How to recover the lost data due to server failures?
2. How to limit the resources and the time needed to recover lost data?
3. How to identify frequently used files and recover them with high priority when lost?
4. How to avoid recovering temporary files?
5. How to share resources among the two activities, running the jobs and re-computations?
6. How to ensure the system fault-tolerance?
7. How to manage the storage for binaries necessary for data recovery?
8. How to choose the files to be evicted when their cumulative size exceeds the available storage space?
9. How to deal with file name changes?
10. How to deal with changes in the cluster's run time environment?
11. How to support different frameworks?

The answers to these questions given by the designers of the system are discussed next. Checkpointing alone is not a solution for re-computation of lost data because periodic checkpointing leads to an unbounded recovery time. Lineage alone is also not feasible because the depth of the lineage graphs keeps growing and the time to recompute the entire path to a leaf of the graph is prohibitive. The solution adopted by Tachyon is based on combined checkpointing and lineage.

The Edge algorithm introduced in [301] checkpoints only the leaves of the lineage directed acyclic graph (DAG). This strategy reduces the number of checkpointed files and limits the resources needed for recovery. The implicit assumption of this approach is that data is immutable between consecutive checkpoints. Data should be versioned if modified between two consecutive checkpoints, different files produced from the same parent should have different IDs.

A read counter associated with every file is used as file priority. Frequently read files have a high priority while temporary files with a low read count are avoided. Re-computation is done asynchronously in the background using the linage thus, the interference with jobs running on the cluster is minimized and their SLOs can be guaranteed. The system is controlled by a *Tachyon Master* with several stand-by replicas ready to take over if the current master fails. If the master fails, a Paxos algorithm is used to elect the next master.

Computing the lineage of a file requires re-running the binaries of all applications executed from the instance the parent was created until the lost data was created. The workflow manager, a component of the Tachyon Master, uses a DAG for each file and does a depth-first search (DFS) of nodes to reach

[19]Tachyon is also the name given to a hypothetical particle that moves faster than the speed of light. In modern physics "tachyon" refers to imaginary mass fields rather than to faster-than-light particles.

the targeted files; it stops as soon as it reaches a node representing a file already in storage. So it is fair to ask ourselves how much storage is necessary for the binaries of all the jobs that can potentially be executed during the recovery process. Data gathered by Microsoft shows that a typical data center running some 1 000 jobs daily needs about 1 TB of storage for all binaries executed over a period of one year [209].

Data eviction is based on a LRU (Least Recently Used) policy. This policy is justified by the access frequency and the by temporal locality. According to a cross-industry study [105] the file access in a large data center often follows a Zipf-like distribution and 75% of re-accessing occurs within 6 hours. A file is uniquely identified by an immutable ID in the lineage information record to address file name changes. This ensures that the re-computation done according to the ordered list prescribed by lineage reads the same files and in the same order as in the original execution.

Among the most frequent changes in the cluster's runtime environment are changes of the version of a framework used to re-compute lost data and changes of the OS version. To address this problem the system runs in a *Synchronous mode* before any such change when all un-replicated files are check-pointed and the new data is saved. Once this is done, this mode is disabled. Lastly, Tachyon requires a program written in a framework to provide information before writing a new file. This information is used: (i) to decide if the file should be in memory only and (ii) to recover a lost file using its lineage.

Edge algorithm. The algorithm assumes that the lineage is represented by a DAG with files as vertices and edges representing the transformation of a parent to all of its descendants. The algorithm checkpoints the leafs of the graph. For example, assume a lineage chain for a file A_0 including files $\{A_0, A_1, A_2, \ldots, A_i, \ldots, A_j, \ldots\}$. Then if there is a checkpoint of A_i and A_j is lost, the re-computation starts with the latest checkpoint, in this case A_i, rather than A_0. Figure 8.8 shows the lineage DAGs of two files $A1$ and $B1$ and the leafs checkpointed at several instances of time; $A1$ and $B1$ are checkpointed first, then $A4$, $B4$, $B5$, and $B6$.

The Edge algorithm does not take into account priorities. A balanced algorithm alternates edge-driven checkpointing with priority-based checkpointing and allocates a fraction c of time to the former and a fraction $(1 - c)$ of time to the latter. To guarantee applications SLOs the recovery time for any file must be bounded.

Call W_i the time to checkpoint an edge i of the DAG and G_i the time to generate edge i from its ancestors. Then two bounds on the recovery time for any file are proven in [301]: (1) Edge checkpointing alone leads to the following bound of the recovery time

$$\mathcal{T}^{edge} = 3 \times M \quad \text{with} \quad M = \max_i\{T_i\}, \quad \text{and} \quad T_i = \max(W_i, G_i). \tag{8.8}$$

This shows that the re-computation time is independent of the DAGs depth. (2) The bound for the recovery time for alternating edge and priority checkpointing is

$$\mathcal{T}^{edge,priority} = \frac{3 \times M}{c} \quad \text{with} \quad M = \max_i\{T_i\}, \quad \text{and} \quad T_i = \max(W_i, G_i)\,. \tag{8.9}$$

Resource management. The resource management policies should address several concerns. For example, when several files have to be recovered at the same time the system should consider data dependencies to avoid recursive task launching. Dynamic file checkpointing priorities are also necessary; the low priority assigned to a file requested by a low priority job should be automatically

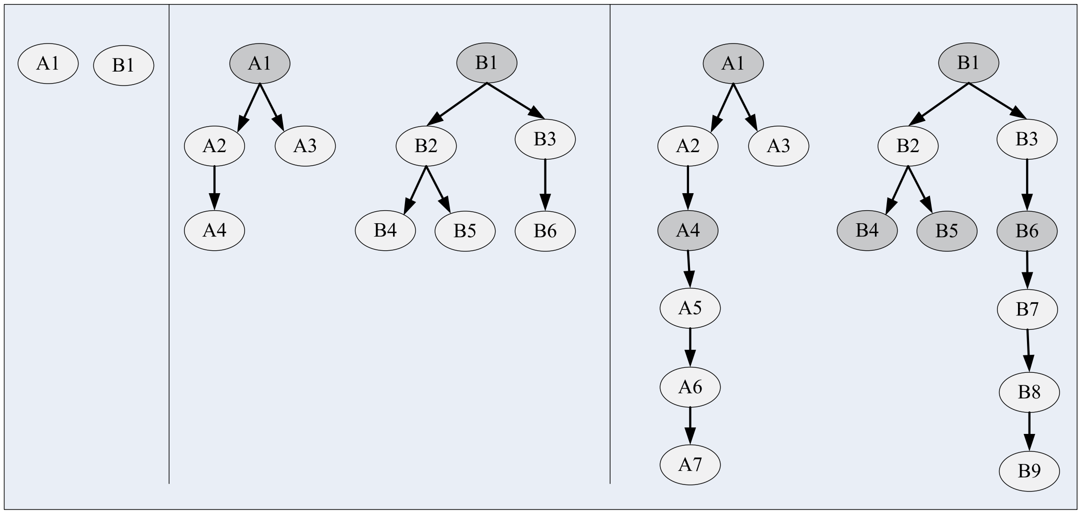


FIGURE 8.8

Illustration of the Edge algorithm. Dark-filled ellipses represent checkpointed files and light-filled ones represent un-checkpointed files. Lineage is represented by a DAG with files as vertices and edges representing the transformation of a parent to all of its descendants. Shown are the lineage DAGs of two files, $A1$ and $B1$. At each stage only the leaf nodes of the lineage DAG are checkpointed. $A1$ and $B1$ are checkpointed first, then $A4$, $B4$, $B5$, and $B6$. In the next stage, not shown in the figure, only $A7$ and $B9$ will be checkpointed.

increased when the same file is requested by a high priority job. The lineage record should be deleted after checkpointing to save space.

All resources should be dedicated to normal work when no recovery is necessary. Typical average server utilization rarely exceeds 30%, so most of the time there are sufficient resources available for data recovery. But what to do when the system is near its capacity? Then the priority of the jobs and of the recovery come into play and is used by the cluster scheduler.

Tachyon could accommodate the two most frequently used scheduling policies for cluster resource management, priority based and fair-shared based scheduling. In case of *priority-based* scheduling all re-computation jobs are given the lowest priority by default. Deadlock may occur unless precautions are taken.

For example, assume that a job $\mathcal{J}_i$ has a higher priority than file $\mathcal{F}$ it needs to recover and that job $\mathcal{J}_i$ is scheduled to run. At the time when $\mathcal{J}_i$ needs access to $\mathcal{F}$ another job, $\mathcal{R}_{\mathcal{F}}$, also needing to recover $\mathcal{F}$, cannot run as it inherits the lower priority of $\mathcal{F}$. The solution is *priority inheritance*. In this case $\mathcal{J}_i$ and, implicitly the job $\mathcal{R}_{\mathcal{F}}$, should inherit the priority of $\mathcal{J}_i$ that needs $\mathcal{F}$. If another job with an even higher priority needs the file $\mathcal{F}$, its priority, hence the priority of $\mathcal{R}_{\mathcal{F}}$, should increase again.

Assume now a *fair-share scheduler*. In this case $W_1, W_2, \ldots, W_i, \ldots$ are the weights of resources allocated to jobs $\mathcal{J}_1, \mathcal{J}_2, \ldots, \mathcal{J}_i, \ldots$, respectively. The minimal share unit is, $W_g = 1$. When files $\mathcal{F}_{i,1}, \mathcal{F}_{i,2}, \mathcal{F}_{i,3}$ of job $\mathcal{J}_i$ are lost the scheduler allocates the minimum weight W_g from the W_i to the three recovery jobs $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_1}, \mathcal{R}_{\mathcal{J}_i,\mathcal{F}_2}$ and $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_3}$. At the time job $\mathcal{J}_i$ needs to access the file $\mathcal{F}_{i,2}$ the scheduler allocates the fractions $(1-\alpha)$ and α of W_i to $\mathcal{J}_i$ and $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_2}$, respectively.

Tachyon implementation. Tachyon has two layers: the *lineage layer*, which tracks the sequence of jobs that have created a data output; the *persistence layer* which manages data in storage and is mainly used for asynchronous checkpoints. The persistence layer can be any replication-based storage, such as HDFS. Tachyon has a master-slave architecture with several passive replicas of the master and worker daemons running on every cluster node and managing local resources. The lineage information is tracked by a workflow manager running inside the master. The workflow manager computes the order of checkpoints and interacts with the cluster resource manager to get resources needed for re-computations.

Each worker uses a RAM disk for storing memory-mapped files. The concept of wide and narrow dependences inherited from Spark is used to carry out the operations discussed earlier in this section. The system was implemented in about 36 000 lines of Java code and uses the *ZooKeeper* for election of a new master when one fails.

Tachyon lineage can capture the requirements of MapReduce, and SQL, as well as Hadoop. Spark can run on top of Tachyon. According to [301] Tachyon has a 110 times higher write throughput and for realistic workloads improves end-to-end latency four times compared with in-memory HDFS. It can also reduce network traffic by up to 50% because many temporary files are deleted before being checkpointed. Data from Facebook and Bing show that it consumes not more than 1.65% of cluster resources for re-computations.

8.12 CONTAINERS; DOCKER CONTAINERS

This idea of containers was extended from file systems supported by *chroot* to other namespaces including the process IDs. Initially named *control groups*, the *cgroups* concept was implemented in the Linux kernel in 2006 at Google. *cgroups* isolate, control, limit, prioritize, and account for resources such CPU, memory, disk I/O, and network bandwidth, available to a set of processes. Control allows freezing groups of processes, manages checkpointing, and restarting. Resource limiting enforces the target set for resource utilization, while prioritization allows a group of processes to get a larger share of CPU cycles or a higher disk I/O throughput.

Docker made containers easy to use, created a set of tools with standard APIs, and made the same container portable across all environments. According to https://docs.docker.com/: "Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment." Figure 8.9 depicts the organization of both VM- and container-based systems.

Docker containers are light-weight, cost-effective in a public cloud environment, provide better performance, and require fewer hardware resources from a private cloud. Containers isolate an application from the underlying infrastructure and from other applications and support performance and security isolation. Multiple containers running on the same machine share the OS kernel thus, have a smaller memory footprint and a shorter start-up time than VMs.

Another major advantage of Docker containers is increased productivity. Containerization allows developers to choose the most suitable programming languages and software systems and eliminates the need to make copies of production code and install the same configuration in different environments. It also supports efficient up and down scaling of an application.

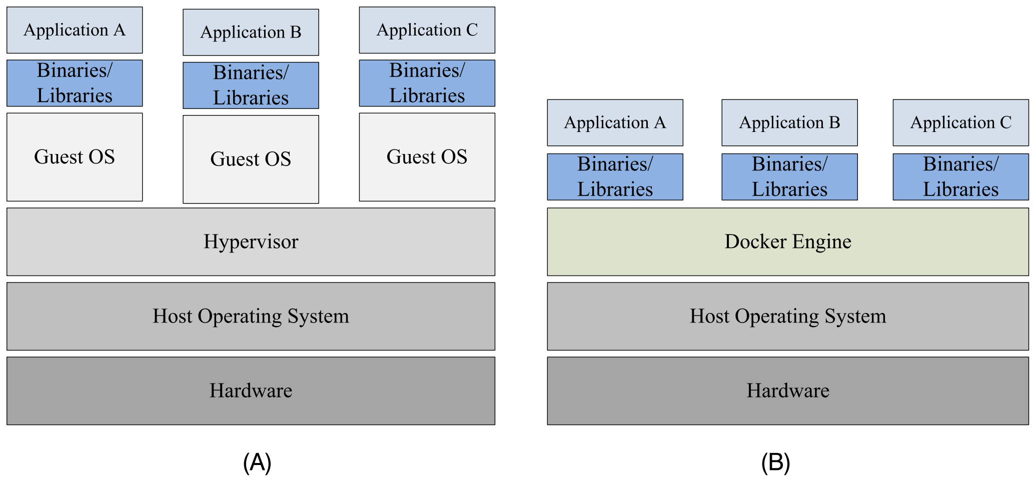


FIGURE 8.9

Organization of VMs and Docker containers. (A) VM; (B) Multiple Docker containers running on the same machine share the OS kernel, thus, have a smaller memory footprint and a shorter start-up time than VMs.

The Docker ecosystem is built around a few concepts discussed next. An *image* is a blueprint of an application. A *container* consists of one or more images and runs the actual application. A *daemon* is a background service running on the host that manages building, running, and distributing Docker containers. The daemon is the process that runs under the operating system to which clients talk to. A *client* is a command line tool used by a user to interact with the daemon. A *hub* is a registry of Docker images, a directory of all available Docker images.

There are two types of images, base and child. *Base images* have no parent image, usually images with an OS like Ubuntu, or BusyBox.[20] *Child images* are build on base images with additional functionality. *Official images* are maintained and supported by Docker, and have one word name; for example, *python, ubuntu* are base official images. *User images* are created and shared by users. A *Dockerfile* is a facility to automate the image creation, a text-file including Linux-like commands invoked by clients to create an image.

Docker Swarm exposes standard Docker API. Docker tools including Docker CLI, Docker Compose, Dokku, and Krane, work in this native Docker clustering. The distribution of it is packed as a Docker container and to set it up one only needs to install one of the service discovery tools and run the *swarm* container on all nodes, regardless of the OS.

Cloud computing has embraced containerization. Containers-as-a-Service (CaaS) is geared toward efficiently running a single application. Several CSPs including Heroku, OpenShift, dotCloud and CloudFoundry use containers to support PaaS delivery model. Amazon, Google, Microsoft, Open-

[20]BusyBox is software providing several stripped-down Unix tools in a single executable file and running in environments such as Linux, Android, FreeBSD, or Debian.

Stack, Cloudstack and other CSPs offering the IaaS cloud delivery model support containers. The container support from Amazon, Google, and Microsoft is overviewed next.

ECS is Amazon EC2 Container Service. It is straightforward for AWS users to create and manage ECS clusters, as ECS is integrated with existing services such as IAM for permissions, CloudTrail to get data regarding resources used by a container, CloudFormation for cluster lunching, and other services. AWS uses a custom scheduler/cluster manager for containers.

Container hosts are regular EC2 instances. To deploy a containerized application on AWS one has to first publish the image on an AWS accessible registry, e.g., the Docker Hub. Amazon ECS is free, while Google Container Engine discussed next is free up to five nodes. AWS charging granularity is one hour, while Google and Microsoft charge for the actual time used.

Google Container Engine (GKE). GKE is based on Kubernetes an open source cluster manager available to Google developers and the community of outside users. Google's approach to containerization is slightly different, its emphasis is more on performance than ease of use as discussed in Section 8.13. Two billion containers are started at Google every week according to http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

GKE is integrated with other services including Google Cloud Logging. Google users have access by default to private Docker registry and a JSON-based declarative syntax for configuration. The same syntax can be used to define what happens with the hosts. GKE can launch and terminate containers on different hosts as defined in the configuration file.

Microsoft Azur Container Service. The Azure Resources Manager API supports multiple orchestrations including Docker, and Apache Mesos.

In recent years the Open Container Initiative (OCI) was involved in an effort to create industry-standard container format and runtime systems under the Linux Foundation. The 40+ members of the OCI work together to make Docker more accessible to the large cloud users community. OCI's approach is to break Docker into small reusable components. In 2016 OCI announced Docker Engine 1.11 which uses a daemon, *containers*, to control *runC*[21] or other OCI compliant run-time systems to run containers. Users access the Docker Engine via a set of commands and a user interface.

8.13 KUBERNETES

First, the origin of the word Kubernetes. In ancient Greek "kubernan" means to steer and "kubernetes" is helmsman. In Latin "gubernare" means to steer/govern and "gubernator" means governor. Kubernetes is an open source software system developed and used at Google for managing containerized applications in a clustered environment.

Kubernetes bridges the gap between a clustered infrastructure and assumptions made by applications about their environments. Kubernetes is a cluster manager for containers. Mesos is adding several

[21] *runC* is an implementation of the Open Containers Runtime specification and the default executor bundled with Docker Engine. Its open specification allows developers to specify different executors without any changes to Docker itself.

Kubernetes ideas and expected to support Kubernetes API. Kubernetes is written in *Go*[22] and it is designed to work well with operating systems that offer lightweight virtual computing nodes. The system is lightweight, modular, portable and extensible.

Kubernetes is an open-ended system and its design allowed a number of other systems to be build atop Kubernetes. The same APIs used by its control plane are also available to developers and users who can write their own controllers, schedulers, etc., if they choose so. The system does not limit the type of applications, does not restrict the set of runtimes of languages supported and allows users to choose the logging, monitoring, and alerting systems of their choice.

Kubernetes does not provide built-in services including message buses, data-processing frameworks, databases (e.g., mysql), or cluster storage systems and does not require a comprehensive application configuration language. It provides deployment, scaling, load balancing, logging, monitoring, etc., services common to PaaS Kubernetes. Kubernetes is not monolithic, and default solutions are optional and pluggable.

Kubernetes organization. A master server manages a Kubernetes cluster and provides services to manage the workload and to support communication for a large number of relatively unsophisticated *minions* servers that do the actual work. *Etcd* is a lightweight, distributed key-value store to share configuration data with the cluster nodes. The master also provides an API service; an HTTP/JSON API is used for service discovery. The Kubernetes scheduler tracks resources available and those allocated to the workloads on each host.

Minions use a *docker* service to run encapsulated application containers. A *kuberlet* service allows minions to communicate with the master server and with the *etcd* store to get configuration details and update the state. A proxy service of the minions interacts with the containers and provides a primitive load balance.

In Kubernetes *pods* are groups of containers scheduled onto the same host and serve as units of scheduling, deployment, horizontal scaling, and replication. Pods share fate and share resources such as storage volumes. The *run* command is used to create a single container pod and the *Deployment* which monitors that pod. All applications in a pod share a network namespace, including the IP address and the port space thus, can communicate with each other using *localhost*.

Pods manage co-located support software including: content management systems, controllers, managers, configurators, updaters, logging and monitoring adapters, and event publishers. Pods also manage file and data loaders, local cache managers, log and checkpoint backup, compression, rotation, snapshotting, data change watchers, log trailers, proxies, bridges, and adapters.

Arguably, pods are preferable to running multiple applications in a single Docker container for several reasons:

1. Efficiency – containers can be lightweight as the infrastructure takes on more responsibility.
2. Transparency and user convenience – the containers within a pod are visible to the infrastructure and allow it to provide process management, resource monitoring, and other services. Users do not need to run their own process managers, or deal with signal and exit-code propagation.
3. Decoupling software dependencies – individual containers may be versioned, rebuilt and redeployed independently.

[22]*Go* or *Golang* is open source compiled, statically typed language like Algol and C; has garbage collection, limited structural typing, memory safety features, and CSP-style concurrent programming.

Kubernetes *replication controllers* – handle the lifecycle of containers. The pods maintained by a replication controller are automatically replaced if they fail, get deleted, or are terminated. A replication controller supervises multiple pods across multiple nodes. *Labels* provide the means to find and query containers and *services* identify a set of containers performing a common function.

Kubernetes has different CLI, API, and YAML[23] definitions than Docker and has a steep learning curve. Kubernetes setup is more complicated than Docker Swarm and its installation differs for different operating systems and service providers.

8.14 FURTHER READINGS

There is little doubt that Amazon has a unique position in the cloud computing world. There is a wealth of information on how to use AWS services [18–25], but little has been published about the algorithms and the mechanisms for resource allocation and the software used by AWS. The effort to maintain a shroud of secrecy probably reflects the desire to maintain Amazon's advantage over its competitors. There are only a few papers published in leading journals or in the proceeding of top conferences describing research results related to Microsoft's Azur cloud platform such as [253,539].

In stark contrast with Amazon and Microsoft, Google's research teams publish often and have made a significant contribution to understanding the challenges posed by very large systems. The evolution of ideas and Google's perspective in cloud computing is presented in [132]. An early discussion of Google cluster architecture is presented in [54]. The current hardware infrastructure and the Warehouse Scale Computers are analyzed in a 2013 book [56] and in a chapter of a classical computer architecture book [228]. A very interesting analysis of WSC performance is due to a team including Google researchers [262]. The discussion of multicores best suited for typical Google workloads is presented in [239].

There is a wealth of information regarding cluster management and the systems developed at Google: Borg [502], Omega [446], Quasar [137], Heracles [311], and Kubernetes [82]. Controlling latency is discussed in [131]. Performance analysis of large-scale systems using Google trace data is reported in [416]. Important research results related to cloud computing have been reported at U.C. Berkeley where Mesos was designed [237], Stanford [135,137,310,311], and Harvard [262]. Cloud energy consumption is analyzed in many publications including [7,36,50,55,56,327,501,506].

A very positive development is the dominance of open software. Open software is available from Apache, Linux Foundation, and others. Docker software and tutorials can be downloaded from https://www.docker.com/ and https://www.digitalocean.com/community/tags/docker?type=tutorials and http://prakhar.me/docker-curriculum/, respectively.

Detailed information about Kubernetes are at http://kubernetes.io/ and https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes. The Open Containers Initiative of the Linux Foundation has developed technologies for container-based applications, see https://www.opencontainers.org/news/news/2016/04/docker-111-first-runtime-built-containerd-and-based-oci-technology. A distributed main memory processing system is presented in [259].

[23]CLI stands for Command Line Interface and provides the means for a user to interact with a program; YAML is a human-readable data serialization language.

8.15 EXERCISES AND PROBLEMS

Problem 1. The average CPU utilization is an important measure of performance for the cloud infrastructure. [56] reports a median CPU utilization in the 40–70% range while [262] reports a median utilization around 10% consistent with utilization reported for the CloudSuite [170]. Read the three references. Discuss the results in [262] and explain the relation between CPU utilization and memory bandwidth and latency.

1. Discuss the effects of cycle stalls and of the ILP (Instruction Level Parallelism) on processor utilization. Analyze the data on cycle stalls and ILP reported in [262].
2. Identify the reasons for the high ratio of cache stalls and the low ILP for data-intensive WSC workloads reported in [262].
3. Why WSC workloads exhibit the high ratio of cache stalls and the low ILP?
4. What conclusions regarding memory bandwidth and latency can be drawn from the results reported in [262]? Justify your answers.

Problem 2. Discuss the results regarding simultaneous multithreading (SMT) reported in [262].

1. For what type of workloads is SMT most effective? Explain your answer.
2. The efficacy of SMT can be estimated by comparing specific per-hyperthread performance counters with ones aggregated on a per-core basis. This is very different from measuring the speedup that a single application experiences from SMT. Why?
3. Why is it difficult to measure the SMT efficiency in a cloud environment?

Problem 3. Mesos is a cluster management system designed to be robust and tolerant to failure. Read [237] and answer the following questions:

1. What are the specific means to achieve these design goals?
2. It is critical to make the Mesos *master* fault-tolerant because all frameworks depend on it. What are the special precautions to make the *master* fault-tolerant?

Problem 4. Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines. Read [502] and answer the following questions:

1. Does Borg have an admission control policy? If so describes its mechanism.
2. What are the elements that make the Borg scheduler scalable?
3. How does the job mix on Borg cells affect the CPI (Cycles per Instruction)?

Problem 5. Omega is a scalable cluster management system based on a parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control. Read [446] and answer the following questions:

1. What scheduler performance metrics are used for Omega, why each one of them is relevant, and how it is actually measured?
2. Trace-driven simulation was used to gain insights into the system. What are the benefits of trace-driven simulation and how was it used to investigate conflicts?
3. One of the simulation results refers to gang scheduling. What is gang scheduling and why it is beneficial for MapReduce applications?

Problem 6. Quasar is a QoS-aware cluster management system using a fast classification techniques to determine the impact of different resource allocations and assignments on workload performance.

1. Read [136] to understand the relationship between the Netflix challenge and the cluster resource allocation problem.
2. Quasar [137] classifies resource allocation for scale up, scale out, heterogeneity, and interference. Why are classification criteria important, and how are they applied?
3. What are stragglers and how does Quasar deal with them?

Problem 7. Cluster management systems must perform well for a mix of applications and deliver the performance promised by the SLOs for each workload. Resource isolation is critical for achieving strict SLOs.

1. What are the mechanisms used by Heracles [311] for mitigating interference?
2. Discuss the results related to latency of Latency Sensitive workload (LS) from Heracles experiments.
3. Discuss the results related to Effective Machine Utilization (EMU) from Heracles experiments.

Problem 8. Effective cloud resource management require understanding the interaction between the workloads and the cloud infrastructure. An analysis of both sides of this equation uses a trove of trace data provided by Google. This analysis is reported in [416].

1. What conclusions can you draw from the analysis of the trace data regarding the schedulers used for cluster management?
2. What are the reasons for scheduler behavior revealed by the trace analysis?
3. What characteristics of the Google workloads are most notable?

Problem 9. Tachyon is a distributed file system enabling reliable data sharing at memory speed across cluster computing frameworks. Read [301] and answer the following questions:

1. Does the evolution of memory, storage, and networking technologies support the argument that cloud storage systems should achieve fault-tolerance without replication?
2. What is file popularity, what is the distribution of file popularity for big data workloads?
3. How is this distribution used?
4. For what type of events is this distribution particularly important?

CHAPTER

CLOUD RESOURCE MANAGEMENT AND SCHEDULING 9

Resource management is a core function of any man-made system, it affects the three basic criteria for the evaluation of a system: performance, functionality, and cost. An efficient resource management has a direct effect on performance and cost and an indirect effect on the functionality of the system as some of the functions may be avoided due to the poor performance, and/or cost.

A cloud is a complex system with a very large number of shared resources subject to unpredictable requests and affected by external events it cannot control. Cloud resource management requires complex policies and decisions for multi-objective optimization. Effective resource management is extremely challenging due to the scale of the cloud infrastructure and to the unpredictable interactions of the system with a large population of users. The scale makes it impossible to have accurate global state information and the large user population makes it nearly impossible to predict the type and the intensity of the system workload.

Resource management becomes even more complex when resources are oversubscribed and users are uncooperative. In addition to external factors, resource management is affected by internal factors, such as heterogeneity of hardware and software systems, the scale of the system, the failure rates of different components, and other factors.

The strategies for resource management associated with the basic cloud delivery models, IaaS, PaaS, SaaS, and DBaasS are different. In all cases the cloud service providers are faced with large fluctuating loads which challenge the claim of cloud elasticity. In some cases, when a spike can be predicted, the resources can be provisioned in advance, e.g., for web services subject to seasonal spikes. For an unplanned spike the situation is slightly more complicated.

Auto-scaling can be used for unplanned spikes of the workload provided that: (a) there is a pool of resources that can be released or allocated on demand; and (b) there is a monitoring system enabling the resource management system to reallocate resources in real time. Auto-scaling is supported by PaaS services, such as Google AppEngine. Auto-scaling for IaaS discussed in Section 9.3 is complicated due to the lack of standards.

Centralized control cannot provide adequate solutions for management policies when changes in the environment are frequent and unpredictable. Distributed control poses its own challenges since it requires some form of coordination between the entities in control. Autonomic policies are of great interest due to the scale of the system and the unpredictability of the load when the ration of peak to mean resource demands can be very large.

Throughout this text we use the term *bandwidth* in a broad sense to mean the number of operations or the amount of data transferred per time unit. For example Mips (Million Instructions Per Second) or Mflops (Million Floating Point Instructions Per Second) measure the CPU speed, Mbps (Mega bits per second) measure the speed of a communication channel. The *latency* is defined as the time elapsed from the instance an operation is initiated and the instance its effect is sensed. Latency is context dependent. For example, the latency of a communication channel is the time it takes a bit to traverse the communication channel from its source to its destination; the memory latency is the time elapsed from the instance a memory *read* instruction is issued until the time the data becomes available in a memory register. The demand for computing resources such as CPU cycles, primary and secondary storage, and network bandwidth depend heavily on the volume of data processed by an application.

This chapter presents research topics related to cloud resource management and scheduling. The overview of policies and mechanisms for cloud resource management in Section 9.1 is followed by a presentation of energy efficiency and cloud resource utilization and the impact of application scaling on resource management in Sections 9.2 and 9.3, respectively. A control theoretic approach to resource allocations is discussed in Sections 9.4, 9.5, and 9.6 followed by a machine learning algorithm for coordination of specialized autonomic performance managers in Section 9.7.

A utility model for resource allocation for a web service is then presented in Section 9.8. The discussion of scheduling algorithms for computer clouds in Section 9.9 is followed by an analysis of delay scheduling and of data-aware scheduling in Sections 9.10 and 9.11, respectively. The Apache capacity scheduler is presented in Section 9.12. The start-time fair queuing and the borrowed virtual time scheduling algorithms are analyzed in Sections 9.13 and 9.14, respectively.

9.1 POLICIES AND MECHANISMS FOR RESOURCE MANAGEMENT

A *policy* refers to the principles guiding decisions, while *mechanisms* represent the means to implement policies. Separation of policies from mechanisms is a guiding principle in computer science. Butler Lampson [294] and Per Brinch Hansen [221] offer solid arguments for this separation in the context of operating system design and their arguments can be extended to computer clouds.

Cloud resource management policies can be loosely grouped into five classes: admission control, capacity allocation, load balancing, energy optimization, and QoS guarantees. The explicit goal of an admission control policy is to prevent the system from accepting workload in violation of high-level system policies. For example, a system may not accept additional workload which would prevent it from completing work already in progress or contracted.

Limiting the workload requires some knowledge of the global state of the system; in a dynamic system such knowledge, when available, is at best obsolete. Capacity allocation means to allocate resources for individual instances; an instance is an activation of a service. Locating resources subject to multiple global optimization constraints requires a search of a very large search space when the state of individual systems changes rapidly.

Load balancing and energy optimization can be done locally, but global load balancing and energy optimization policies encounter the same difficulties as the ones we have already discussed. Load balancing and energy optimization are correlated and affect the cost for providing services [146].

The common meaning of the term "load balancing" is evenly distribute the load among the set of servers. For example, consider the case of four identical servers, A, B, C and D whose relative loads are $80\%, 60\%, 40\%$ and 20%, respectively, of their capacity. As a result of a perfect load balancing all servers would end with the same relative workload, 50% of each server's capacity. An important goal of cloud resource management is minimization of the cost for providing cloud service and, in particular, minimization of cloud energy consumption.

Table 9.1 The normalized performance and energy consumption, function of the processor speed; the performance decreases at a lower rate than does the energy when the clock rate decreases.

CPU speed (GHz)	Normalized energy (%)	Normalized performance (%)
0.6	0.44	0.61
0.8	0.48	0.70
1.0	0.52	0.79
1.2	0.58	0.81
1.4	0.62	0.88
1.6	0.70	0.90
1.8	0.82	0.95
2.0	0.90	0.99
2.2	1.00	1.00

This leads to a different meaning of the term "load balancing;" instead of having the load evenly distributed amongst all servers, we wish to concentrate it and use the smallest number of servers while switching the others to a standby mode, a state where a server uses very little energy. In our example, assuming that the servers have the same capacity the load from D will migrate to A and the load from C will migrate to B; thus, A and B will be loaded at full capacity while C and D will be switched to standby mode. In practice workloads larger than 80% of system capacity are not desirable. QoS is the aspect of resource management probably the most difficult to address and, at the same time, possibly the most critical for the future of cloud computing.

As we shall see in this section, often resource management strategies jointly target performance and power consumption. The Dynamic Voltage and Frequency Scaling (DVFS)[1] techniques such as Intel's SpeedStep and AMD's PowerNow lower the voltage and the frequency to decrease the power consumption.[2] Motivated initially by the need to save power for mobile devices, these techniques have migrated virtually to all processors including the ones used for high performance servers.

Processor performance decreases, but at a substantially lower rate than the energy consumption, as a result of lower voltages and clock frequencies [300]. Table 9.1 shows the dependence of the normalized performance and the normalized energy consumption of a typical modern processor on the clock rate. As we can see, at 1.8 GHz we save 18% of the energy required for maximum performance, while the performance is only 5% lower than the peak performance, achieved at 2.2 GHz. This seems a reasonable energy-performance trade-off!

Virtually all optimal, or near-optimal, mechanisms to address the five classes of policies do not scale up and typically target a single aspect of resource management, e.g., admission control, but ignore energy conservation. Many require complex computations that cannot be done effectively in the time available to respond. The performance models are very complex, analytical solutions are intractable,

[1]Dynamic voltage and frequency scaling is a power management technique to increase or decrease the operating voltage or frequency of a processor to increase the instruction execution rate and, respectively, to reduce the amount of heat generated and to conserve power.

[2]The power consumption P of a CMOS-based circuit is: $P = \alpha \cdot C_{eff} \cdot V^2 \cdot f$ with: α – the switching factor, C_{eff} – the effective capacitance, V – the operating voltage, and f – the operating frequency.

and the monitoring systems used to gather state information for these models can be too intrusive and unable to provide accurate data.

Many techniques are concentrated on system performance in terms of throughput and time in system, but they rarely include energy trade-offs or QoS guarantees. Some techniques are based on unrealistic assumptions. For example, capacity allocation is viewed as an optimization problem, but under the assumption that servers are protected from overload.

Cloud resource allocation techniques must be based on a systematic approach, rather than on ad hoc methods. The four basic mechanisms for the implementation of resource management policies are:

- *Control theory.* Control theory uses feedback mechanisms to guarantee system stability and to predict transient behavior [260], [285]. Feedback can only be used to predict local, rather than global behavior. Kalman filters have been used for unrealistically simplified models.
- *Machine learning.* Machine learning techniques do not need a performance model of the system [488], a major advantage. This technique can be applied for coordination of several autonomic system managers, as discussed in [265].
- *Utility-based.* Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost, as discussed in [9].
- *Market-oriented mechanisms.* Such mechanisms do not require a model of the system, e.g., combinatorial auctions for bundles of resources discussed in [465].

A distinction should be made between interactive and non-interactive workloads. The management techniques for interactive workloads, e.g., web services, involve flow control and dynamic application placement, while those for non-interactive workloads are focused on scheduling. A fair amount of work reported in the literature is devoted to resource management of interactive workloads, some to non-interactive ones, and only a few, e.g., [476], to heterogeneous workloads, a combination of the two.

9.2 CLOUD RESOURCE UTILIZATION AND ENERGY EFFICIENCY

According to Moore's Law the number of transistors on a chip, thus, the computing power of microprocessors doubles approximately every 1.5 years. A recent study [279] reports that electrical efficiency of computing devices doubles also about every 1.5 years. Thus, performance growth rate and improvements in electrical efficiency almost cancel out. It follows that the energy used for computing scales linearly with the number of computing devices. The number of computing devices continues to grow and many are now housed in large cloud data centers.

The energy consumption of cloud data centers is growing and has a significant ecological impact. It also affects the cost of cloud services. The energy costs are passed on to the users of cloud services and differ from one country to another and from one region to another. For example, the published rates for two AWS regions, US East and South America are: upfront for a year \$2,604 versus \$5,632 and hourly \$0.412 versus \$0.724, respectively. Higher energy and communication costs are partially responsible for the significant difference in this example; the energy costs for the two regions differ by about 40%.

All these facts justify the need to take a closer look at cloud energy consumption, a complex subject extensively discussed in the literature [7,36,50,55,56,327,501,506]. The topics to be covered are how to define the energy efficiency, how energy-efficient are the processors, the storage devices, the networks,

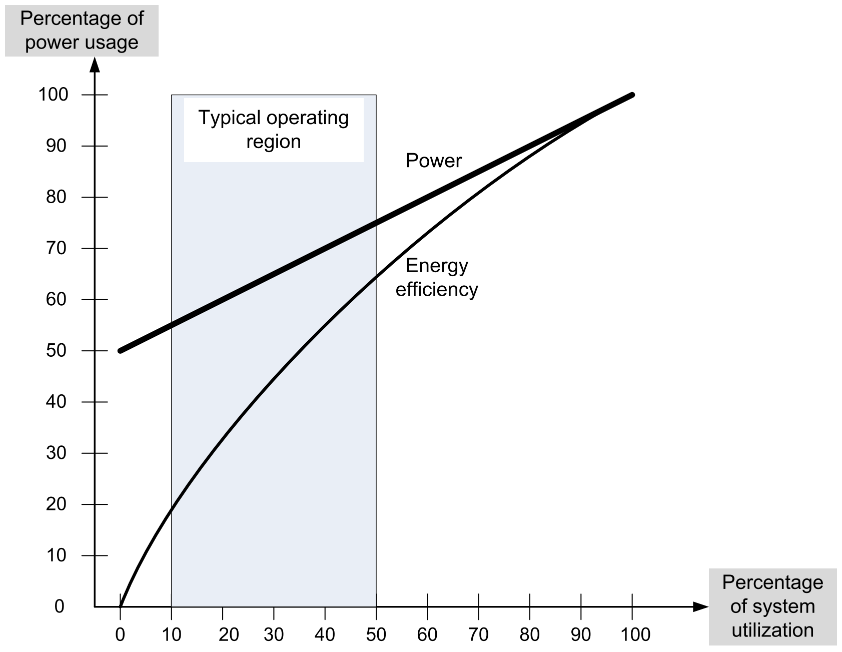


FIGURE 9.1

Even when power requirements scale linearly with the load, the energy efficiency of a computing system is not a linear function of the load. When idle, a system may use 50% of the power corresponding to the full load. Data collected over a long period of time shows that the typical operating region for the servers at a data center is the range 10% to 50% of system utilization [55].

and the other physical elements of the cloud infrastructure, and what are the constraints and how well are these resources managed.

Cloud elasticity and overprovisioning. One of the main appeals of utility computing is elasticity. *Elasticity* means that additional resources are guaranteed to be allocated when an application needs them and that resources will be released when no longer needed. A user ends up paying only for resources actually used.

Overprovisioning means having capacity in excess of normal or average needs. This implies that a cloud service provider has to invest in an infrastructure larger than *typical* cloud workload warrants. It follows that the average cloud server utilization is low [7,68,327]. Low server utilization negatively affects the performance per watt of power, a common measure of energy efficiency, and the ecological impact of cloud computing. Overprovisioning is not economically sustainable [97].

Elasticity is based on overprovisioning and on the assumption that there is an effective admission control mechanism. Another assumption is that the likelihood of all running applications dramatically increasing their resource consumption at the same time is extremely low. This assumption is realistic, though we have seen cases when a system is overloaded due to concurrent access by large crowds, e.g., the phone system in case of a catastrophic event such as an earthquake. A possible solution is to request cloud users to specify in their service request the type of workloads and to pay for access accordingly, e.g., a low rate for slow varying and a high rate for workloads with sudden peaks.

Energy efficiency and energy-proportional systems. An energy-proportional system consumes no power when idle, very little power under a light load and, gradually, more power as the load increases. By definition, an ideal energy-proportional system is always operating at 100% efficiency. Humans are a good approximation of an ideal energy proportional system; the human energy consumption is about 70 W at rest, 120 W on average on a daily basis, and can go as high as 1 000–2 000 W during a strenuous, short time effort [55].

In real life, even systems whose power requirements scale linearly, when idle use more than half the power consumed at full load, see Figure 9.1. Indeed, a 2.5 GHz Intel E5200 dual-core desktop processor with 2 GB of RAM consumes 70 W when idle and 110 W when fully loaded; a 2.4 GHz Intel Q6600 processor with 4 GB of RAM consumes 110 W when idle and 175 W when fully loaded [50].

Different subsystems of a computing system behave differently in terms of energy efficiency; while many processors have relatively good energy-proportional profiles, significant improvements in memory and disk subsystems are necessary. The processors used in servers consume less than one-third of their peak power at very low load and have a dynamic range of more than 70% of peak power; the processors used in mobile and/or embedded applications are better in this respect.

The dynamic power range[3] of other components of a system is much narrower [55]: less than 50% for DRAM, 25% for disk drives, and 15%for networking switches. The power consumption of such devices is: 4.9 KW for a 604.8 TB, HP 8100 EVA storage server, 3.8 KW for the 320 Gbps Cisco 6509 switch, 5.1 KW for the 660 Gbps Juniper MX-960 gateway router [50].

The alternative to the wasteful resource management policy when the servers are *always on*, regardless of their load, is to develop *energy-aware load balancing and scaling* policies. Such policies combine *dynamic power management* with load balancing and attempt to identify servers operating outside their optimal energy regime and decide if and when they should be switched to a sleep state or what other actions should be taken to optimize the energy consumption.

Energy saving. The effort to reduce the energy use is focused on the computing, networking, and storage activities of a data center. A 2010 report shows that a typical Google cluster spends most of its time within the 10–50% CPU utilization range; there is a mismatch between server workload profile and server energy efficiency [7]. A similar behavior is also seen in the data center networks; these networks operate in a very narrow dynamic range, the power consumed when the network is idle is significant compared to the power consumed when the network is fully utilized.

A strategy to reduce energy consumption is to concentrate the workload on a small number of disks and allow the others to operate in a low-power mode. One of the techniques to accomplish this is based on replication. A replication strategy based on a sliding window is reported in [506]. Measurements results indicate that it performs better than LRU, MRU, and LFU[4] policies for a range of file sizes, file availability, and number of client nodes and the power requirements are reduced by as much as 31%.

[3]The dynamic range in this context is determined by the lower and the upper limit of the power consumption. A large dynamic range means that the device is better, it is able to operate at a lower fraction of its peak power when its load is low.

[4]LRU (Least Recently Used), MRU (Most Recently Used), and LFU (Least Frequently Used) are replacement policies used by memory hierarchies for caching and paging.

Another technique is based on data migration. The system in [225] uses data storage in virtual nodes managed with a distributed hash table. Migration is controlled by two algorithms, a short-term optimization algorithm used for gathering or spreading virtual nodes according to the daily variation of the workload so that the number of active physical nodes is reduced to a minimum, and a long-term optimization algorithm used for coping with changes in the popularity of data over a longer period, e.g., a week.

A number of proposals have emerged for *energy proportional* networks [8]; the energy consumed by such networks is proportional with the communication load. For example, a data center interconnection network based on a flattened butterfly topology is more energy and cost efficient according to [7]. High-speed channels typically consist of multiple serial lines with the same data rate. A physical unit is stripped across all active lines. Channels commonly operate plesiochronously[5] and are always on, regardless of the load, because they must still send idle packets to maintain byte and lane alignment across the multiple lines. An energy proportional network, *InfiniBand*, is discussed in Section 5.7.

Many proposals argue that dynamic resource provisioning is necessary to minimize power consumption. Two issues are critical for energy saving: the amount of resources allocated to each application and the placement of individual workloads. A resource management framework combining a utility-based dynamic VM provisioning manager with a dynamic VM placement manager to minimize power consumption and reduce Service Level Agreement violations is presented in [497].

Energy optimization is an important policy for cloud resource management, but it cannot be considered in isolation; energy optimization should be coupled with admission control, capacity allocation, load balancing, and quality of service. Existing mechanisms cannot support concurrent optimization of all policies. Mechanisms based on a solid foundation such as control theory are too complex and do not scale well, those based on machine learning are not fully developed, and the others require a model of a system with a dynamic configuration operating in a fast-changing environment.

9.3 RESOURCE MANAGEMENT AND DYNAMIC APPLICATION SCALING

The demand for resources can be a function of the time of day, can monotonically increase or decrease in time, or can experience predictable or unpredictable peaks. For example, a new web service will experience a low request rate at the very beginning and the load will exponentially increase if the service is successful. A service for income tax processing will experience a peak around the tax filling deadline, while access to a service provided by FEMA (Federal Emergency Management Agency) will increase dramatically after a natural disaster.

The elasticity of a public cloud, the fact that it can supply precisely the amount of resources an application needs, and that a cloud user pays only for resources consumed are serious incentives to migrate to a public cloud. The question we address is how scaling can be implemented in a cloud with a very large number of applications exhibiting an unpredictable behavior [84,331,496]. To make matter worse, in addition to an unpredictable external workload the cloud resource management has to deal with relocation of running applications due to server failures.

[5]Different parts of the system are almost, but not quite perfectly, synchronized; in this case, the core logic in the router operates at a frequency different from that of the I/O channels.

We distinguish two scaling strategies, vertical and horizontal. *Vertical scaling* keeps the number of VMs of an application constant, but increases the amount of resources allocated to each one of them. This can be done either by migrating VMs to more powerful servers, or by keeping VMs on the same servers, but increasing their share of the CPU time. The first alternative involves additional overhead; the VM is stopped, a snapshot of it is taken, the file is transported to a more powerful server, and, finally, the VM is restated at the new site.

Horizontal scaling is the most common scaling strategy on a cloud; it is done by increasing the number of VMs as the load increases and reducing this number when the load decreases. Often, this leads to an increase of communication bandwidth consumed by the application. Load balancing among the running VMs is critical for this mode of operation. For a very large application multiple load balancers may need to cooperate with one another. In some instances load balancing is done by a front-end server which distributes incoming requests of a transaction-oriented system to backend servers.

An application should be designed to support scaling. Workload partitioning of a *modularly divisible* application is static as we have seen in Section 7.5. Static workload partitioning is decided a priori and cannot be changed thus, the only alternative is vertical scaling. The workload of an *arbitrarily divisible* application can be partitioned dynamically. As the load increases, the system can allocate additional VMs to process the additional workload. Most cloud applications belong to this class and this justifies the statement that horizontal scaling is the most common scaling strategy.

Mapping a computation means to assign suitable physical servers to the application. A very important first step in application processing is to identify the type of application and map it accordingly. For example, a communication-intensive application should be mapped to a powerful server to minimize the network traffic. Such a mapping may increase the cost per unit of CPU usage, but it will decrease the computing time and, probably, reduce the overall user cost. At the same time, it will reduce network traffic, a highly desirable effect from the perspective of the CSP.

To scale up or down a compute-intensive application a good strategy is to increase/decrease the number of VMs or instances. As the load is relatively stable, the overhead of starting up or terminating an instance does not increase significantly the computing time or the cost.

Several strategies to support scaling exist. *Automatic VM scaling* uses pre-defined metrics, e.g., CPU utilization to make scaling decisions. Automatic scaling requires *sensors* to monitor the state of the VMs and servers and *controllers* to make decisions based on the information about the state of the cloud.

Controllers often use a state machine model for decision making. Amazon and Rightscale (http://www.rightscale.com) offer automatic scaling. The AWS CloudWatch service supports applications monitoring and allows a user to set up conditions for automatic migrations.

Non-scalable or single load balancers are also used for horizontal scaling. The Elastic Load Balancing AWS service automatically distributes incoming application traffic across multiple EC2 instances. Another service, the Elastic Beanstalk allows dynamic scaling between a low and a high number of instances specified by the user, see Section 2.3. The cloud user usually has to pay for the more sophisticated scaling services such as Elastic Beanstalk.

9.4 CONTROL THEORY AND OPTIMAL RESOURCE MANAGEMENT

Control theory has been used to design adaptive resource management for many classes of applications including power management [265], task scheduling [314], QoS adaptation in web servers [3], and load balancing [350,398]. The classical feedback control methods are used in all these cases to regulate the key operating parameters of the system based on measurement of the system output. The feedback control for these methods assumes a linear time-invariant system model, and a closed-loop controller. This controller is based on an open-loop system transfer function which satisfies stability and sensitivity constraints.

A technique to design self-managing systems based on concepts from control theory is discussed in [512]. This technique allows multiple QoS objectives and operating constraints to be expressed as a cost function. The technique can be applied to stand-alone or distributed web servers, database servers, high performance application servers, and embedded systems.

The following discussion considers a single processor serving a stream of input requests with the goal of minimizing a cost function reflecting the response time and the power consumption. The goal is to illustrate the methodology for optimal resource management based on control theory concepts. The analysis is intricate and cannot be easily extended to a collection of servers.

Control theory principles. An overview of control theory principles used for optimal resource allocation is presented next. Optimal control generates a sequence of control inputs over a look-ahead horizon, while estimating changes in operating conditions. A convex cost function has as arguments $x(k)$, the state at step k, and $u(k)$, the control vector. The cost function is minimized subject to the constraints imposed by system dynamics. The discrete-time optimal control problem is to determine the sequence of control variables $u(i), u(i+1), \ldots, u(n-1)$ to minimize the expression

$$J(i) = \Phi(n, x(n)) + \sum_{k=i}^{n-1} L^k(x(k), u(k)) \tag{9.1}$$

where $\Phi(n, x(n))$ is the cost function of the final step, n, and $L^k(x(k), u(k))$ is a time-varying cost function at the intermediate step k over the horizon $[i, n]$. The minimization is subject to the constraints

$$x(k+1) = f^k(x(k), u(k)), \tag{9.2}$$

where $x(k+1)$, the system state at time $k+1$, is a function of $x(k)$, the state at time k, and of $u(k)$, the input at time k; in general, the function f^k is time-varying thus, its superscript.

One of the techniques to solve this problem is based on the *Lagrange multiplier* method of finding the extremes (minima or maxima) of a function subject to constrains. More precisely, if we wish to maximize the function $g(x, y)$ subject to the constraint $h(x, y) = k$ we introduce a Lagrange multiplier λ. Then we study the function

$$\Lambda(x, y, \lambda) = g(x, y) + \lambda \times \left[h(x, y) - k\right]. \tag{9.3}$$

A necessary condition for optimality is that (x, y, λ) is a stationary point for $\Lambda(x, y, \lambda)$, in other words

$$\nabla_{x,y,\lambda} \Lambda(x, y, \lambda) = 0 \quad \text{or} \quad \left(\frac{\partial \Lambda(x, y, \lambda)}{\partial x}, \frac{\partial \Lambda(x, y, \lambda)}{\partial y}, \frac{\partial \Lambda(x, y, \lambda)}{\partial \lambda} \right) = 0. \tag{9.4}$$

The Lagrange multiplier at time step k is $\lambda(k)$ and we solve Equation (9.4) as an unconstrained optimization problem. We define an adjoint cost function which includes the original state constrains

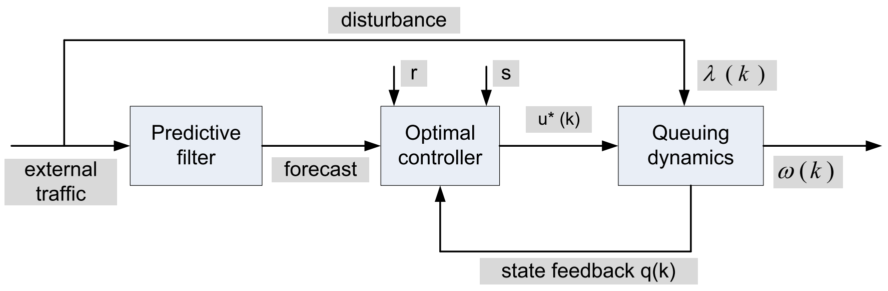


FIGURE 9.2

The structure of the optimal controller in [512]. The controller uses the feedback regarding the current state, as well as, the estimation of the future disturbance due to environment to compute the optimal inputs over a finite horizon. The two parameters r and s are the weighting factors of the performance index.

as the Hamiltonian function H, then we construct the adjoint system consisting of the original state equation and the *costate equation*[6] governing the Lagrange multiplier. Thus, we define a two-point boundary problem;[7] the state x_k develops forward in time while the costate occurs backward in time.

A model capturing QoS and energy consumption for a single server system. We now turn our attention to the case of a single processor serving a stream of input requests. To compute the optimal inputs over a finite horizon the controller in Figure 9.2 uses the feedback regarding the current state and the estimation of the future disturbance due to the environment. The control task is solved as a state regulation problem updating the initial and final states of the control horizon.

We use a simple queuing model to estimate the response time; requests for service at processor P are processed on an FCFS basis. We do not assume a priori distributions of the arrival process and of the service process; instead, we use the estimate, $\hat{\Lambda}(k)$ of the arrival rate $\Lambda(k)$ at time k. We also assume that the processor can operate at frequencies $u(k)$ in the range $u(k) \in [u_{min}, u_{max}]$ and call $\hat{c}(k)$ the time to process a request at time k when the processor operates at the highest frequency in the range, u_{max}. Then we define the scaling factor $\alpha(k) = u(k)/u_{max}$ and we express an estimate of the processing rate $N(k)$ as $\alpha(k)/\hat{c}(k)$.

The behavior of a single processor is modeled as a non-linear, time-varying, discrete-time state equation. If T_s is the sampling period, defined as the time difference between two consecutive observations of the system, e.g., the one at time $(k+1)$ and the one at time k, then the size of the queue at time $(k+1)$ is

$$q(k+1) = \max\left\{\left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], 0\right\} \tag{9.5}$$

[6]The costate equation is related to the state equation in optimal control.

[7]A boundary value problem has conditions specified at the extremes of the independent variable while an initial value problem has all of the conditions specified at the same value of the independent variable in the equation. The common case is when boundary conditions are supposed to be satisfied at two points – usually the starting and ending values of the integration.

The first term, $q(k)$, is the size of the input queue at time k and the second one is the difference between the number of requests arriving during the sampling period, T_s, and those processed during the same interval. The response time $\omega(k)$ is the sum of the waiting time and the processing time of the requests

$$\omega(k) = (1 + q(k)) \times \hat{c}(k). \tag{9.6}$$

Indeed, the total number of requests in the system is $(1 + q(k))$ and the departure rate is $1/\hat{c}(k)$.

We wish to capture both the QoS and the energy consumption, as both affect the cost of providing the service. A utility function, such as the one depicted in Figure 9.5, captures the rewards, as well as the penalties specified by the SLA for the response time. In the queuing model the utility is a function of the size of the queue and can be expressed as a quadratic function of the response time

$$S(q(k)) = 1/2\left(s \times (\omega(k) - \omega_0)^2\right) \tag{9.7}$$

with ω_0, the response time set point and $q(0) = q_0$, the initial value of the queue length. The energy consumption is a quadratic function of the frequency

$$R(u(k)) = 1/2\left(r \times u(k)^2\right). \tag{9.8}$$

The two parameters s and r are weights for the two components of the cost, derived from the utility function and from the energy consumption, respectively. We have to pay a penalty for the requests left in the queue at the end of the control horizon, a quadratic function of queue length

$$\Phi(q(N)) = 1/2\left(v \times q(n)^2\right). \tag{9.9}$$

The performance measure of interest is a cost expressed as

$$J = \Phi(q(N)) + \sum_{k=1}^{N-1}\left[S(q(k)) + R(q(k))\right]. \tag{9.10}$$

The problem is to find the optimal control u^* and the finite time horizon $[0, N]$ such that the trajectory of the system subject to optimal control is q^*, and the cost J in Equation (9.10) is minimized subject to the following constraints

$$q(k+1) = \left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], \quad q(k) \geq 0, \quad \text{and} \quad u_{min} \leq u(k) \leq u_{max}. \tag{9.11}$$

When the state trajectory $q(\cdot)$ corresponding to the control $u(\cdot)$ satisfies the constraints

$$\Gamma 1: \ q(k) > 0, \quad \Gamma 2: \ u(k) \geq u_{min}, \quad \Gamma 3: \ u(k) \leq u_{max}, \tag{9.12}$$

then the pair $\left[q(\cdot), u(\cdot)\right]$ is called a *feasible state*. If the pair minimizes the Equation (9.10) then the pair is *optimal*.

The Hamiltonian H in our example is

$$H = S(q(k)) + R(u(k)) + \lambda(k+1) \times \left[q(k) + \left(\Lambda(k) - \frac{u(k)}{c \times u_{max}} \right) T_s \right] \\ + \mu_1(k) \times (-q(k)) + \mu_2(k) \times (-u(k) + u_{min}) + \mu_3(k) \times (u(k) - u_{max}). \tag{9.13}$$

According to Pontryagin's minimum principle[8] the necessary condition for a sequence of feasible pairs to be optimal pairs is the existence of a sequence of costates λ and a Lagrange multiplier $\mu = [\mu_1(k), \mu_2(k), \mu_3(k)]$ such that

$$H(k, q^*, u^*, \lambda^*, \mu^*) \leq H(k, q, u^*, \lambda^*, \mu^*), \ \forall q \geq 0 \tag{9.14}$$

where the Lagrange multipliers, $\mu_1(k), \mu_2(k), \mu_3(k)$, reflect the sensitivity of the cost function to the queue length at time k and the boundary constraints and satisfy several conditions

$$\mu_1(k) \geq 0, \ \mu_1(k)(-q(k)) = 0, \tag{9.15}$$
$$\mu_2(k) \geq 0, \ \mu_2(k)(-u(k) + u_{min}) = 0, \tag{9.16}$$
$$\mu_3(k) \geq 0, \ \mu_3(k)(u(k) - u_{max}) = 0. \tag{9.17}$$

A detailed analysis of the methods to solve this problem and the analysis of the stability conditions is beyond the scope of our discussion and can be found in [512].

The extension of the techniques for optimal resource management from a single system to a cloud with a very large number of servers is a rather challenging area of research. The problem is even harder when, instead of transaction-based processing, the cloud applications require the implementation of a complex workflow.

9.5 STABILITY OF A TWO-LEVEL RESOURCE ALLOCATION ARCHITECTURE

The discussion in Section 9.4 shows that a server can be assimilated with a closed-loop control system and that we can apply theoretical control principles to resource allocation. We now discuss a two-level resource allocation architecture based on control theory concepts for the entire cloud, see Figure 9.3. The automatic resource management is based on two levels of controllers, one for the service provider and one for the application.

The main components of a control system are: the inputs, the control system components, and the outputs. The inputs in such models are: the offered workload and the policies for admission control, the capacity allocation, the load balancing, the energy optimization, and the QoS guarantees in the cloud. The system components are *sensors* used to estimate relevant measures of performance and *controllers* which implement various policies. The output is the resource allocations to the individual applications.

The controllers use the feedback provided by sensors to stabilize the system; stability is related to the change in the output. If the change is too large then the system may become unstable. In our context

[8]Pontryagin's principle is used in the optimal control theory to find the best possible control which leads a dynamic system from one state to another, subject to a set of constrains.

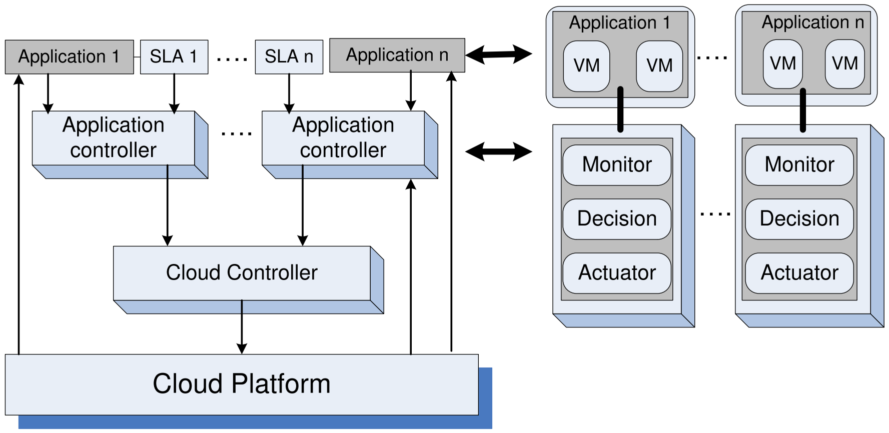


FIGURE 9.3

A two-level control architecture; application controllers and cloud controllers work in concert.

the system could experience thrashing, the amount of useful time dedicated to the execution of applications becomes increasingly smaller and most of the system resources are occupied by management functions.

There are three main sources of instability in any control system:

1. The delay in getting the system reaction after a control action.
2. The granularity of the control, the fact that a small change enacted by the controllers leads to very large changes in the output.
3. Oscillations, when the changes in the input are too large and the control is too weak, such that the changes in the input propagate directly to the output.

Two types of policies are used in autonomic systems: (i) threshold-based policies and (ii) sequential decision policies based on Markovian decision models. In the first case, upper and lower bounds on performance trigger adaptation through resource reallocation; such policies are simple and intuitive but require setting per-application thresholds.

Lessons learned from the experiments with two levels of controllers and the two types of policies are discussed in [157]. A first observation is that the actions of the control system should be carried out in a rhythm that does not lead to instability; adjustments should only be carried out after the performance of the system has stabilized. The controller should measure the time for an application to stabilize and adapt to the manner in which the controlled system reacts.

If an upper and a lower threshold are set, then instability occurs when the thresholds are too close to one another and when the variation of the workload is large enough and the time required to adapt does not allow the system to stabilize. The actions consist of allocation/deallocation of one or more VMs. Sometimes allocation/deallocation of a single VM required by one of the thresholds may cause crossing of the other threshold, another source of instability.

9.6 FEEDBACK CONTROL BASED ON DYNAMIC THRESHOLDS

The elements involved in a control system are sensors, monitors, and actuators. The *sensors* measure the parameter(s) of interest, then transmit the measured values to a *monitor* which determines if the system behavior must be changed, and, if so, it requests the *actuators* to carry out the necessary actions. Often, the parameter used for admission control policy is the current system load; when a threshold, e.g., 80%, is reached the cloud stops accepting additional load.

The implementation of such a policy is challenging, or outright infeasible in practice. First, due to the very large number of servers and to the fact that the workload changes rapidly in time, the estimation of the current system workload is likely to be inaccurate. Second, the ratio of average to maximal resource requirements of individual users specified in a SLA is typically very high. Once an agreement is in place user demands must be satisfied; a user's request for additional resources within the SLA limits cannot be denied.

Thresholds. A threshold is the value of a parameter related to the state of a system that triggers a change in the system behavior. Thresholds are used in control theory to keep critical parameters of a system in a predefined range. The threshold could be *static*, defined once and for all, or could be *dynamic*. A dynamic threshold could be based on an average of measurements carried out over a time interval, a so called *integral control*; the dynamic threshold could also be a function of the values of multiple parameters at a given time, or a mixture of the two.

A *high* and a *low* threshold are often defined to maintain the system parameters in a given range. The two thresholds determine different actions; for example, a high threshold could force the system to limit its activities and a low threshold could encourage additional activities. *Control granularity* refers to the level of detail of the information used to control the system. *Fine control* means that very detailed information about the parameters controlling the system state is used, while *coarse control* means that the accuracy of these parameters is traded off for the efficiency of implementation.

Proportional thresholding. Application of these ideas to cloud computing, in particular to the IaaS delivery model, and a strategy for resource management called *proportional thresholding* are discussed in [305]. The questions addressed are:

- Is it beneficial to have two types of controllers: (1) *application controllers* which determine if additional resources are needed and (2) *cloud controllers* which arbitrate requests for resources and allocate the physical resources?
- Is it feasible to consider *fine control*? Is *coarse control* more adequate in a cloud computing environment?
- Are dynamic thresholds based on time averages better than static ones?
- Is it better to have a high and a low threshold, or it is sufficient to define only a high threshold?

The first two questions are related. It seems more appropriate to have two controllers, one with knowledge of the application and one aware of the state of the cloud. In this case a coarse control is more adequate for many reasons. As mentioned earlier, the cloud controller can only have a very rough approximation of the global cloud state. Moreover, to simplify the resource management policies the cloud service provider may wish to hide some of the information available to them. For example, CSPs may not allow a VM to access information available to hypervisor-level sensors and actuators.

To answer the last two questions one has to define a measure of "goodness." In the experiments reported in [305] the parameter measured is the average CPU utilization and a strategy is better than another if it reduces the number of requests made by the application controllers to add or remove VMs to the pool of those available to the application.

A control theoretical approach to address these questions is challenging. The authors of [305] adopt a pragmatic approach and provide qualitative arguments; they also report simulation results using a synthetic workload for a transaction-oriented application, a web server.

The essence of the proportional thresholding is captured by the following algorithm:

1. Compute the integral value of the high and the low threshold as averages of the maximum and, respectively, the minimum of the processor utilization over the process history.
2. Request additional VMs when the average value of the CPU utilization over the current time slice exceeds the high threshold.
3. Release a VM when the average value of the CPU utilization over the current time slice falls below the low threshold.

The conclusions reached based on experiments with three VMs are: (a) dynamic thresholds perform better than the static ones and (b) two thresholds are better than one. While confirming our intuition, such results have to be justified by experiments in a realistic environment. Moreover, convincing results cannot be based on empirical values for some of the parameters required by integral control equations.

9.7 COORDINATION OF AUTONOMIC PERFORMANCE MANAGERS

Can specialized autonomic performance managers cooperate to optimize power consumption and, at the same time, satisfy the requirements of SLAs? This is the question examined by a group from IBM Research in a 2007 paper [265]. The paper reports on actual experiments carried out on a set of blades mounted on a chassis. Figure 9.4 shows the experimental setup. Extending the techniques discussed in this report to a large-scale farm of servers poses significant problems; computational complexity is just one of them.

Virtually all modern processors support Dynamic Voltage Scaling as a mechanism for energy saving; indeed, the energy dissipation scales quadratically with the supply voltage. The power management controls the CPU frequency thus, the rate of instruction execution. For some compute-intensive workloads the performance decreases linearly with the CPU clock frequency, while for others the effect of lower clock frequency is less noticeable or non-existent. The clock frequency of individual blades/servers is controlled by a power manager typically implemented in the firmware; it adjusts the clock frequency several times a second.

The approach to coordinating power and performance management in [265] is based on several ideas:

- Use a joint utility function for power and performance. The joint performance-power utility function, $U_{pp}(R, P)$, is a function of the response time, R, and the power, P, and it can be of the form

$$U_{pp}(R, P) = U(R) - \epsilon \times P \quad \text{or} \quad U_{pp}(R, P) = \frac{U(R)}{P}, \tag{9.18}$$

  with $U(R)$ the utility function based on response time only and ϵ a parameter to weigh the influence of the two factors, response time and power.

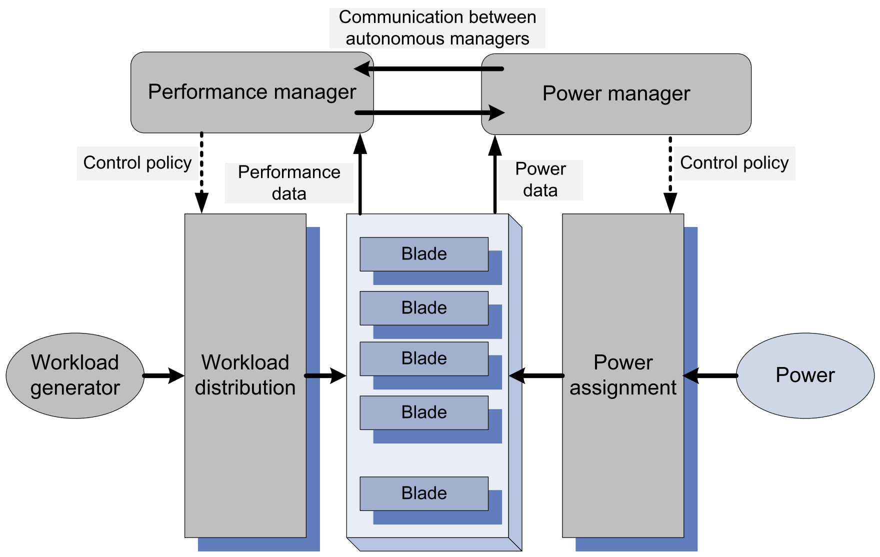


FIGURE 9.4

Autonomous performance manager and power manager cooperate to ensure SLA prescribed performance and energy optimization; they are fed with performance and power data and implement the performance and power management policies, respectively.

- Identify a minimal set of parameters to be exchanged between the two managers.
- Set up a power cap for individual systems based on the utility-optimized power management policy.
- Use a standard performance manager modified only to accept input from the power manager regarding the frequency determined according to the power management policy. The power manager consists of TCL and C programs to compute the per-server (per-blade) power caps and send them via IPMI[9] to the firmware controlling the blade power. The power manager and the performance manager interact but no negotiation between the two agents is involved.
- Use standard software systems. For example, use the WXD (WebSphere Extended Deployment), a middleware which supports setting performance targets for individual web applications and for the monitor response time, and periodically recompute the resource allocation parameters to meet the targets set. Use the Wide-Spectrum Stress Tool from IBM Web Services Toolkit as a workload generator.

For practical reasons the utility function is expressed in terms of n_c, the number of clients, and p_κ the powercap, as in

$$U'(p_\kappa, n_c) = U_{pp}(R(p_\kappa, n_c), P(p_\kappa, n_c)). \tag{9.19}$$

[9]Intelligent Platform Management Interface (IPMI) is a standardized computer system interface developed by Intel and used by system administrators to manage a computer system and monitor its operation.

The optimal powercap, p_κ^{opt} is a function of the workload intensity expressed by the number of clients, n_c,

$$p_\kappa^{opt}(n_c) = \arg\max U'(p_\kappa, n_c). \tag{9.20}$$

The hardware used for these experiments were blades with an Intel Xeon processor running at 3 GHz with 1 GB of level 2 cache and 2 GB of DRAM and with hyper-threading enabled. A blade could serve 30 to 40 clients with a response time at or better than 1 000 msec limit. When p_κ is lower than 80 watts, the processor runs at its lowest frequency, 375 MHz, while for p_κ at or larger than 110 Watts, the processor runs at its highest frequency, 3 GHz.

Three types of experiments were conducted: (i) with the power management turned off; (ii) when the dependence of the power consumption and the response time were determined through a set of exhaustive experiments; and (iii) when the dependency of the powercap p_κ on n_c is derived via reinforcement-learning models.

The second type of experiments led to the conclusion that both the response time and the power consumed are non-linear functions of the powercap, p_κ, and the number of clients, n_c. More specifically, the conclusions of these experiments are:

- At a low load the response time is well below the target of 1 000 msec.
- At medium and high load the response time decreases rapidly when p_k increases from 80 to 110 watts.
- For a given value of the powercap, the consumed power increases rapidly as the load increases.

The machine learning algorithm used for the third type of experiments was based on the Hybrid Reinforcement Learning algorithm described in [483]. In the experiments using the machine learning model, the powercap required to achieve a response time lower than 1 000 msec for a given number of clients was the lowest when $\epsilon = 0.05$ and the first utility function given by Equation (9.18) was used; for example, when $n_c = 50$ then $p_\kappa = 109$ watts when $\epsilon = 0.05$, while $p_\kappa = 120$ when $\epsilon = 0.01$.

9.8 A UTILITY MODEL FOR CLOUD-BASED WEB SERVICES

A *utility function* relates the "benefits" of an activity or service to the "cost" to provide the service. For example, the benefit could be revenue and the cost could be the power consumption.

An SLA often specifies the rewards as well as penalties associated with specific performance metrics. Sometimes the quality of services translates into average response time; this is the case of cloud-based web services when the SLA often specifies explicitly this requirement. For example, Figure 9.5 shows the case when the performance metrics is R, the response time. The largest reward can be obtained when $R \le R_0$; a slightly lower reward corresponds to $R_0 < R \le R_1$; when $R_1 < R \le R_2$, instead of gaining a reward, the provider of service pays a small penalty; the penalty increases when $R > R_2$. A utility function, $U(R)$, which captures this behavior is a sequence of step functions; the utility function is sometimes approximated by a quadratic curve as discussed in Section 9.4.

In this section we discuss a utility-based approach for autonomic management. The goal is to maximize the total profit computed as the difference between the revenue guaranteed by an SLA and the total cost to provide the services. Formulated as an optimization problem, the solution discussed in [9]

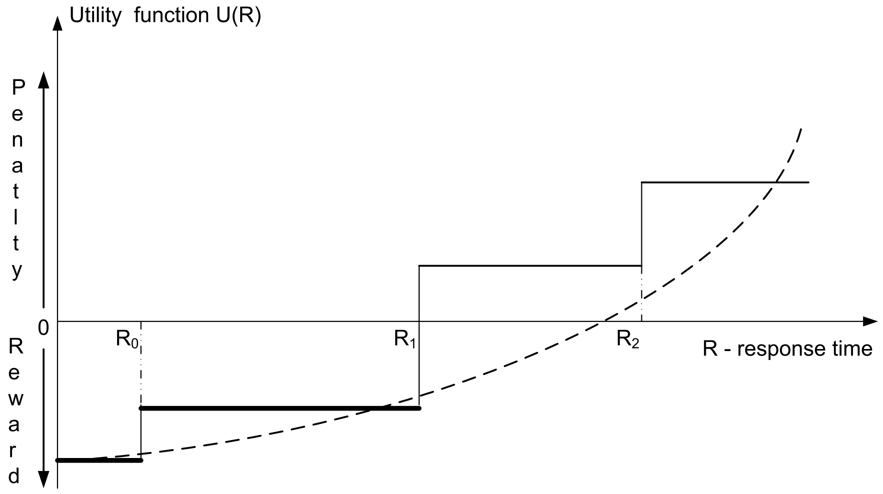


FIGURE 9.5

The utility function $U(R)$ is a series of step functions with jumps corresponding to the response time, $R = R_0|R_1|R_2$, when the reward and the penalty levels change according to the SLA. The dotted line shows a quadratic approximation of the utility function.

addresses multiple policies, including QoS. The cloud model for this optimization is quite complex and requires a fair number of parameters.

We assume a cloud providing $|K|$ different classes of service, each class k involving N_k applications. For each class $k \in K$ call v_k the revenue (or the penalty) associated with a response time r_k and assume a linear dependency for this utility function of the form $v_k = v_k^{max}\left(1 - r_k/r_k^{max}\right)$, see Figure 9.6A; call $m_k = -v_k^{max}/r_k^{max}$ the slope of the utility function.

The system is modeled as a network of queues with multi-queues for each server and with a delay center which models the think time of the user after the completion of service at one server and the start of processing at the next server, see Figure 9.6B. Upon completion, a class k request either completes with probability $1 - \sum_{k' \in K} \pi_{k,k'}$, or returns to the system as a class k' request with transition probability $\pi_{k,k'}$. Call λ_k the external arrival rate of class k requests and Λ_k the aggregate rate for class k, $\Lambda_k = \lambda_k + \sum_{k' \in K} \Lambda_{k'} \pi_{k,k'}$.

Typically, CPU and memory are considered as representative for resource allocation. For simplicity we assume a single CPU which runs at a discrete set of clock frequencies and a discrete set of supply voltages according to a DVFS model; the power consumption on a server is a function of the clock frequency. The scheduling of a server is work-conserving[10] and is modeled as a Generalized Processor Sharing (GPS) scheduling [545]. Analytical models [5], [386] are too complex for large systems.

The optimization problem formulated in [9] involves five terms: A and B reflect revenues, C is the cost for servers in a low power, stand-by mode, D is the cost of active servers given their operating

[10]A scheduling policy is work-conserving if the server cannot be idle while there is work to be done.

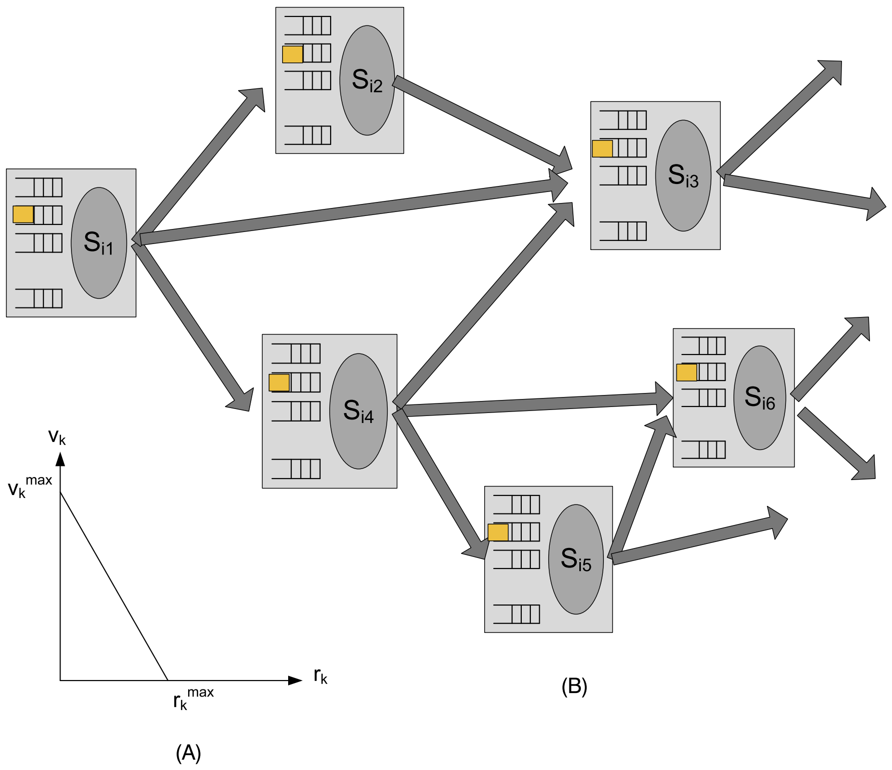


FIGURE 9.6

(A) The utility function, v_k the revenue (or the penalty) associated with a response time r_k for a request of class $k \in K$; the slope of the utility function is $m_k = -v_k^{max}/r_k^{max}$. (B) A network of multiqueues; at each server S_i there are $|K|$ queues for each one of the $k \in K$ classes of requests. A tier consists of all requests of class $k \in K$ at all servers S_{ij}, $i \in I$, $1 \le j \le 6$.

frequency, E is the cost for switching servers from low-power, stand-by mode, to active state, and F is the cost for migrating VMs from one server to another. There are 9 constraints $\Gamma_1, \Gamma_2, \ldots, \Gamma_9$ for this mixed integer non-linear programming problem. The decision variables for this optimization problem are listed in Table 9.2 and the parameters used are shown in Table 9.3.

The expression to be maximized is:

$$(A + B) - (C + D + E + F) \tag{9.21}$$

with

$$A = \max \sum_{k \in K} \left(-m_k \sum_{i \in I, j \in N_k} \frac{\lambda_{i,k,j}}{\sum_{h \in H_i} \left(C_{i,h} \times y_{i,h}\right) \mu_{k,j} \times \phi_{i,k,j} - \lambda_{i,k,j}} \right), \quad B = \sum_{k \in K} v_k \times \Lambda_k, \tag{9.22}$$

Table 9.2 Decision variables for the optimization problem.

Name	Description
x_i	$x_i = 1$ if server $i \in I$ is running, $x_i = 0$ otherwise
$y_{i,h}$	$y_{i,h} = 1$ if server i is running at frequency h, $y_{i,h} = 0$ otherwise
$z_{i,k,j}$	$z_{i,k,j} = 1$ if application tier j of a class k request runs on server i, $z_{i,k,j} = 0$ otherwise
$w_{i,k}$	$w_{i,k} = 1$ if at least one class k request is assigned to server i, $w_{i,k} = 0$ otherwise
$\lambda_{i,k,j}$	rate of execution of applications tier j of class k requests on server i
$\phi_{i,k,j}$	fraction of capacity of server i assigned to tier j of class k requests

Table 9.3 The parameters used for the A, B, C, D, E and F terms and the constraints Γ_i of the optimization problem.

Name	Description
I	the set of servers
K	the set of classes
Λ_k	the aggregate rate for class $k \in K$, $\Lambda_k = \lambda_k + \sum_{k' \in K} \Lambda_{k'} \pi_{k,k'}$
a_i	the availability of server $i \in I$
A_k	minimum level of availability for request class $k \in K$ specified by the SLA
m_k	the slope of the utility function for a class $k \in K$ application
N_k	number of applications in class $k \in K$
H_i	the range of frequencies of server $i \in I$
$C_{i,h}$	capacity of server $i \in I$ running at frequency $h \in H_i$
$c_{i,h}$	cost for server $i \in I$ running at frequency $h \in H_i$
$\bar{c}_i$	average cost of running server i
$\mu_{k,j}$	maximum service rate for a unit capacity server for tier j of a class k request
cm	the cost of moving a VM from one server to another
cs_i	the cost for switching server i from the stand-by mode to an active state
$RAM_{k,j}$	the amount of main memory for tier j of class k request
$\overline{RAM}_i$	the amount of memory available on server i

$$C = \sum_{i \in I} \bar{c}_i, \quad D = \sum_{i \in I, h \in H_i} c_{i,h} \times y_{i,h}, \quad E = \sum_{i \in I} cs_i \max(0, x_i - \bar{x}_i), \tag{9.23}$$

and

$$F = \sum_{i \in I, k \in K, j \in N_j} cm \max(0, z_{i,j,k} - \bar{z}_{i,j,k}). \tag{9.24}$$

The nine constraints are:

(Γ_1) $\sum_{i \in I} \lambda_{i,k,j} = \Lambda_k, \quad \forall k \in K, j \in N_k, \quad \Rightarrow$ the traffic assigned to all servers for class k requests equals the predicted load for the class.

(Γ_2) $\sum_{k \in K, j \in N_k} \phi_{i,k,j} \leq 1, \quad \forall i \in I, \quad \Rightarrow$ server i cannot be allocated an workload more than its capacity.

(Γ_3) $\sum_{h \in H_i} y_{i,h} = x_i, \quad \forall i \in I, \quad \Rightarrow$ if server $i \in I$ is active it runs at one frequency in the set H_i, only one $y_{i,h}$ is non-zero.

(Γ_4) $z_{i,k,j} \le x_i, \quad \forall i \in I, k \in K, j \in N_k \quad \Rightarrow$ requests can only be assigned to active servers.

(Γ_5) $\lambda_{i,k,j} \le \Lambda_k \times z_{i,k,j}, \quad \forall i \in I, k \in K, j \in N_k \quad \Rightarrow$ requests may run on server $i \in I$ only if the corresponding application tier has been assigned to server i.

(Γ_6) $\lambda_{i,k,j} \le \left(\sum_{h \in H_i} C_{i,h} \times y_{i,h}\right) \mu_{k,j} \times \phi_{i,k,j}, \quad \forall i \in I, k \in K, j \in N_k \quad \Rightarrow$ resources cannot be saturated.

(Γ_7) $RAM_{k,j} \times z_{i,k,j} \le \overline{RAM}_i, \quad \forall i \in I, k \in K \quad \Rightarrow$ the memory on server i is sufficient to support all applications running on it.

(Γ_8) $\Pi_{j=1}^{N_k}\left(1 - \Pi_{i=1}^{M}(1 - a_i^{w_{i,k}}\right) \ge A_k, \quad \forall k \in K \quad \Rightarrow$ the availability of all servers assigned to class k request should be at least equal to the minimum required by the SLA.

(Γ_9) $\sum_{j=1}^{N_k} z_{i,k,j} \ge N_k \times w_{i,k}, \quad \forall i \in I, k \in K$
$\lambda_{i,j,k}, \phi_{i,j,k} \ge 0, \quad \forall i \in I, k \in K, j \in N_k$
$x_i, y_{i,h}, z_{i,k,j}, w_{i,k} \in \{0,1\}, \quad \forall i \in I, k \in K, j \in N_k \quad \Rightarrow$ constraints and relations among decision variables.

Clearly, this approach is not scalable to clouds with a very large number of servers. Moreover, the large number of decision variables and parameters of the model make this approach unfeasible for a realistic cloud computing resource management strategy.

9.9 SCHEDULING ALGORITHMS FOR COMPUTER CLOUDS

Scheduling is a critical component of the cloud resource management responsible for resource sharing/multiplexing at several levels. A server can be shared among several VMs, each VM can support several applications, and each application may consist of multiple threads. CPU scheduling supports the virtualization of a processor, the individual threads acting as virtual processors; a communication link can be multiplexed among a number of virtual channels, one for each flow.

In addition to the need to meet its design objectives, a scheduling algorithm should be efficient, fair, and starvation-free. The objectives of a scheduler for a batch system are to maximize the throughput (the number of jobs completed in one unit of time, e.g., in one hour) and to minimize the turnaround time (the time between job submission and its completion). The objectives of a real-time system scheduler are to meet the deadlines and to be predictable.

Schedulers for systems supporting a mixture of tasks, some with hard real-time constraints, others with soft, or no timing constraints, are often subject to contradictory requirements. Some schedulers are *preemptive*, allowing a high-priority task to interrupt the execution of a lower priority one, others are *non-preemptive*.

Two distinct dimensions of resource management must be addressed by a scheduling policy: (a) the amount/quantity of resources allocated; and (b) the timing when access to resources is granted. Figure 9.7 identifies several broad classes of resource allocation requirements in the space defined by these two dimensions: best-effort, soft requirements, and hard requirements. Hard real-time requirements are the most challenging as they require strict timing and precise amounts of resources.

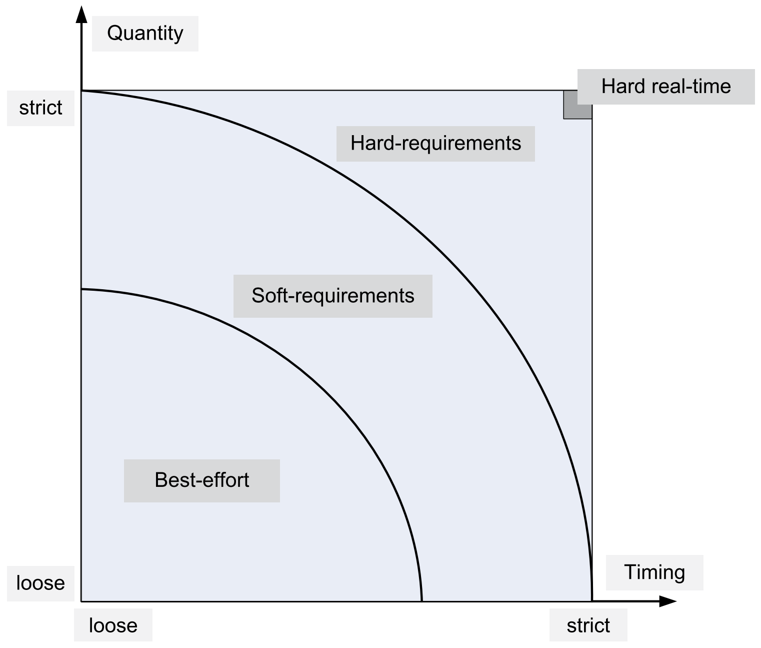


FIGURE 9.7

Resource requirement policies. *Best-effort* policies do not impose requirements regarding either the amount of resources allocated to an application, or the timing when an application is scheduled. *Soft*-requirements policies require statistically guaranteed amounts of resources and timing constraints. *Hard*-requirements policies demand strict timing and precise amounts of resources.

There are multiple definitions of a fair scheduling algorithm. First, we discuss the *max–min fairness criterion* [180]. Consider a resource with bandwidth B shared among n users who have equal rights; each user requests an amount b_i and receives B_i. Then, according to the max–min criterion, the following conditions must be satisfied by a fair allocation:

- C_1 – the amount received by any user is not larger than the amount requested, $B_i \le b_i$.
- C_2 – if the minimum allocation of any user is B_{min} no allocation satisfying condition C_1 has a higher B_{min} than the current allocation.
- C_3 – when we remove the user receiving the minimum allocation B_{min} and then reduce the total amount of the resource available from B to $(B - B_{min})$, the condition C_2 remains recursively true.

A fairness criterion for CPU scheduling [200] requires that the amount of work $\Omega_a(t_1, t_2)$ and $\Omega_b(t_1, t_2)$ in the time interval from t_1 to t_2 of two runnable threads a and b minimize the expression

$$\left| \frac{\Omega_a(t_1, t_2)}{w_a} - \frac{\Omega_b(t_1, t_2)}{w_b} \right|, \tag{9.25}$$

where w_a and w_b are the weights of the threads a and b, respectively.

The QoS requirements differ for different classes of cloud applications and demand different scheduling policies. Best-effort applications, such as batch applications and analytics[11] do not require QoS guarantees. Multimedia applications such as audio and video streaming have soft real-time constraints and require statistically guaranteed maximum delay and throughput. Applications with hard real-time constrains do not use a public cloud at this time, but may do so in the future.

Round-robin, first-come-first-serve (FCFS), shortest-job-first (SJF), and priority algorithms are among the most common scheduling algorithms for best-effort applications. Each thread is given control of the CPU for a definite period of time, called a *time-slice*, in a circular fashion in case of round-robin scheduling; the algorithm is fair and starvation-free. The threads are allowed to use the CPU in the order they arrive in the case of the FCFS algorithms and in the order of their running time in the case of SJF algorithms.

Earliest Deadline First (EDF) and Rate Monotonic Algorithms (RMA) are used for real-time applications. Integration of scheduling for the three classes of applications are discussed in [74] and two new algorithms for integrated scheduling, the Resource Allocation/Dispatching (RAD) and the Rate-Based Earliest Deadline (RBED) are proposed.

Several algorithms of special interest for computer clouds are discussed below. These algorithms illustrate the evolution in thinking regarding the fairness of scheduling and the need to accommodate multi-objective scheduling, in particular scheduling for Big Data and for multimedia applications.

9.10 DELAY SCHEDULING

How to simultaneously ensure fairness and maximize resource utilization without compromising locality and throughput for Big Data applications running on large computer clusters? This was one of the questions faced early on in the cloud computing era by Facebook, Yahoo, and other large IT service providers. Facebook's 600 node Hadoop cluster was running 7 500 MapReduce jobs per day using a data store of 2 PB and growing at a rate of 15 TB per day. Yahoo was facing similar problems for its 3 000 node cluster used for data analytics and ad hoc queries.

Hadoop scheduler. Each Hadoop job consists of multiple Map and Reduce tasks and the question is how to allocate resources to the tasks of newly submitted jobs. Recall from Section 7.7 that the *job tracker* of the Hadoop master manages a number of slave servers running under the control of *task trackers* with slots for Map and Reduce tasks. A FIFO scheduler with five priority levels assigns slots to tasks based on their priority. The fewer the number of tasks of a job already running on slots of all servers, the higher is the priority of the remaining tasks.

An obvious problem with this policy is that priority-based allocation does not consider data locality, namely the need to place tasks close to their input data. The network bandwidth in a large cluster is considerably lower than the disk bandwidth; also the latency for local data access is much lower than the latency of a remote disk access. Locality affects the throughput, *server locality*, i.e., getting data

[11]The term *analytics* is overloaded; sometimes it means discovery of patterns in the data; it could also mean statistical processing of the results of a commercial activity.

from the local server, is significantly better in terms of time and overhead than *rack locality*, i.e. getting input data from a different server in the same rack.

In steady-state, priority scheduling leads to the tendency to assign the same slot repeatedly to the next task(s) of the same job. As one of the job's tasks completes execution its priority decreases and the available slot is allocated to the next task of the same job. Input data for every job are striped over the entire cluster so spreading the tasks over the cluster can potentially improve data locality, but the priority scheduling favors the occurrence of *sticky slots*.

According to [541] sticky slots did not occur in Hadoop at the time of the report "due to a bug in how Hadoop counts running tasks. Hadoop tasks enter a *commit pending* state after finishing their work, where they request permission to rename their output to its final filename. The job object in the master counts a task in this state as running, whereas the slave object doesn't. Therefore, another job can be given the task's slot." Data gathered at Facebook shows that only 5% of the jobs with a low number, 1–25, of Map tasks achieve server locality and only 59% show rack locality.

Task locality and average job locality. A task assignment satisfies the locality requirement if the input task data are stored on the server hosting the slot allocated to the task. We wish to compute the expected locality of job $\mathcal{J}$ with the fractional cluster share $f_{\mathcal{J}}$ assuming that a server has L slots and that each block of the files system has R replicas. By definition $f_{\mathcal{J}} = n/N$ with n the number of slots allocated to job $\mathcal{J}$ and N the number of servers in the cluster.

The probability that a slot does not belong to $\mathcal{J}$ is $(1 - f_{\mathcal{J}})$, there are R replicas of block $\mathcal{B}_{\mathcal{J}}$ of job $\mathcal{J}$ and each replica is on a node with L slots. Thus, the probability that none of the slots of job $\mathcal{J}$ has a copy of block $\mathcal{B}_{\mathcal{J}}$ is $(1 - f_{\mathcal{J}})^{RL}$. It follows that $\mathcal{L}_{\mathcal{J}}$, the locality of job $\mathcal{J}$, can be at most

$$\mathcal{L}_{\mathcal{J}} = 1 - (1 - f_{\mathcal{J}})^{RL}. \tag{9.26}$$

How should a fair scheduler operate on a shared cluster? What is the number n of slots of a shared cluster the scheduler should allocate to jobs assuming that tasks of all jobs take an average of T seconds to complete? A sensible answer is that the scheduler should provide enough slots such that the response time on the shared cluster should be the same as $R_{n,\mathcal{J}}$, the completion time of job $\mathcal{J}$ would experience on a fictitious private cluster with n available slots for the n tasks of $\mathcal{J}$ as soon as job $\mathcal{J}$ arrives.

The job completion time can be approximated by the sum of the job processing time for all but the last task plus the waiting time until a slot becomes available for the last task of the job. There is no waiting time for running on the private cluster with n slots, while on a shared cluster the last task will have to wait before a slot is available.

A slot allocated to a task on the shared cluster will be free on average every T/N seconds thus, the time the job will have to wait until all its n tasks have found a slot on the shared cluster will be $n \times T/N$. This implies that a fair scheduler should guarantee that the waiting time of job $\mathcal{J}$ on the shared cluster is much smaller than the completion time, $R_{n,\mathcal{J}}$, on the fictitious private cluster

$$R_{n,\mathcal{J}} \gg f_{\mathcal{J}} \times T. \tag{9.27}$$

Equation (9.27) is satisfied if one of the following three conditions is satisfied:

1. $f_{\mathcal{J}}$ is small – there are many jobs sharing the cluster and the fraction of slots allocated to each job is small;
2. T is small – the individual tasks are short;
3. $R_{n,\mathcal{J}} \gg T$ – the completion time of a job is much larger than the average task completion time; the large jobs dominate the workload.

The cumulative distribution function of running time of MapReduce jobs at Facebook resembles a sigmoid function[12] with the middle stage of a job duration starting at about 10 seconds and ending at about 1 000 seconds. The median completion time of a Map task is much shorter that the median completion time of a job, 19 versus 84 seconds. There are fewer Reduce tasks, but of a longer average duration, 231 seconds. 83% of the jobs are launched within 10 seconds. Results reported in [541] show that delay scheduling performs well when most tasks are short relative to job duration, and when a running task can read a given data block from multiple locations.

Delay scheduling. A somewhat counterintuitive scheduling policy, *delay scheduling*, is proposed in [541]. As the name implies, the new policy delays scheduling the tasks of a new job for a relatively short time to address the conflict between fairness and locality.

The new policy skips a task of the job at the head of the priority queue if the input data are not available on the server where the slot is located and repeats this process up to D times as specified by the delay scheduling algorithm showed in the next box. An almost doubling of the throughput under the new policy, while ensuring fairness for workloads at Yahoo and Facebook is a good indication of the merits of the delay scheduling policy.

An analysis of the new policy assumes a cluster with N servers and L slots per server, thus with a total number of slots $S = NL$. A job $\mathcal{J}$ prefers slots on servers where its data are stored, call this set of slots $\mathcal{P}_{\mathcal{J}}$. Call $p_{\mathcal{J}}$ the probability that a task of job $\mathcal{J}$ has data on the server with the slot allocated to it

$$p_{\mathcal{J}} = \frac{|\mathcal{P}_{\mathcal{J}}|}{N}. \tag{9.28}$$

It is easy to see that the probability that a task being skipped D times does not have the input data on the server where the slot allocated to it runs, decreases exponentially with D. Indeed, after being skipped D times the probability of not having data on the slot allocated to it is $(1 - p_q\mathcal{J})^D$. For example, if $p_{\mathcal{J}} = 0.1$ and $D = 40$ then the probability of having data on the slot allocated to the task is $1 - (1 - p_{\mathcal{J}})^D = 0.99$, a 99% chance.

The pseudocode for the implementation of the delay scheduling algorithm is shown next.

[12]A sigmoid function $S(t)$ is "S-shaped." It is defined as $S(t) = \frac{1}{1-e^{-t}}$, and its derivative can be expressed as function of itself, $S'(t) = S(t)(1 - S(t))$. $S(t)$ can describe biological evolution with an initial segment describing early/childhood stage, a median stage representing maturity, and a third stage describing the late life stage.

```
Delay scheduling algorithm.

Initialize $j.skipcount$ to 0 for all jobs $j$
    when a heartbeat is received from node $n$
        if $n$ has a free slot then
            sort jobs in increasing order of number of running tasks
            for $j$ in jobs do
                if $j$ has unlaunched task $t$ with data on $n$ then
                    launch $t$ on $n$
                    set $j.skipcount = 0$
                else if $j$ has unlaunched task $t$ then
                    if $j.skipcount > D+1$ then
                        launch $t$ on $n$
                    else
                        set $j.skipcount = j.skipcount + 1$
                    end if
                end if
            end for
        end if
```

How to achieve the desired level of locality for job $\mathcal{J}$ with n tasks? An approximate analysis reported in [541] assumes that all tasks are of the same length and that the preferred location sets $\mathcal{P}_{\mathcal{J}}$ are uncorrelated. If $\mathcal{J}$ has k task left to launch and the replication factor is as before equal to R then

$$p_{\mathcal{J}} = 1 - \left(1 - \frac{k}{N}\right)^{R} \tag{9.29}$$

and the probability of launching a task of $\mathcal{J}$ after D skips is

$$p_{\mathcal{J},D} = 1 - (1 - p_{\mathcal{J}})^{D} = 1 - \left(1 - \frac{k}{N}\right)^{RD} \geq 1 - e^{-RDk/N}. \tag{9.30}$$

The expected value of $p_{\mathcal{J},D}$ is

$$\mathcal{L}_{\mathcal{J},D} = \frac{1}{N}\sum_{k=1}^{N}\left(1 - e^{-RDk/N}\right) = 1 - \frac{1}{N}\sum_{k=1}^{N} e^{-RDk/N}. \tag{9.31}$$

Then

$$\mathcal{L}_{\mathcal{J},D} \geq 1 - \frac{1}{N}\sum_{k=1}^{\infty} e^{-RDk/N} = 1 - \frac{e^{-RD/N}}{N(1 - e^{-RD/N})} \tag{9.32}$$

It follows that a locality $\mathcal{L}_{\mathcal{J},D} \geq \lambda$ requires job $\mathcal{J}$ to forgo D times its turn for a new slot while the head of the priority queue, with D satisfying the following condition

$$D \geq -\frac{N}{R}\ln\frac{n(1-\lambda)}{1+n(1-\lambda)} \quad \text{or} \quad D \leq \frac{N}{R}\ln\left[1 + \frac{1}{n(1-\lambda)}\right]. \tag{9.33}$$

Hadoop Fair Scheduler (HFS). The next objective of [541] is the development of a more complex Hadoop scheduler with several new capabilities:

1. Fair sharing at the level of users rather than jobs. This requires a two-level scheduling, the first level allocates task slots to pools of jobs using a fair sharing policy; at the second level each pool allocates its slots to jobs in the pool.
2. User controlled scheduling; the second level policy can be either FIFO or fair sharing of the slots in the pool.
3. Predictable turnaround time. Each pool has a guaranteed minimum share of slots. To accomplish this goal HFS defines a *minimum share timeout* and a *fair share timeout* and when the corresponding timeout occurs it kills buggy jobs or tasks taking a very long time. Instead of using a minimum skip count D use a *wait time* to determine how long a job waits to allocate a slot to its next ready-to-run task.

HFS creates a sorted list of jobs ordered according to its scheduling policy. Then scans down this list to identify the job allowed to schedule a task next and within each pool applies the pool's internal scheduling policy. Pools missing their minimum share are placed at the head of the sorted list and the other pools are sorted to achieve a weighted fair sharing. The pseudocode of the HFS scheduling algorithm maintains three variables for each job j initialized as $j.level = 0$, $j.wait = 0$, and $j.skipped = false$ when a heartbeat is received from node n:

```
HFS scheduling algorithm.

for each job $j$ with $j.skipped = true$
 increase $j.wait$ by the time since the last heartbeat and set $j.skipped = false$
  if $n$ has a free slot then
     sort jobs using hierarchical scheduling policy
     for $j$ in jobs do
        if $j$ has a node-local task $t$ on $n$ then
          set $j.wait = 0$ and $j.level = 0$
          return $t$ to $n$
       else
          if $j$ has a rack-local task $t$ on $n$ and $j.level > 2$ or $j.wait > W1 + 1$ then
            set $j.wait = 0$ and $j.level = 1$ return $t$ to $n$
         else
            if $j.level = 2$ and $j.level = 1$ and $j.wait > W2 + 1$
                    or $j.level = 0$ and $j.wait > W1 + W2 + 1$ then
                        set $j.wait = 0$ and $j.level = 2$ return any unlaunched task $t$ in $j$ to $n$
            else
                        set $j.skipped = true$
            end if
     end for
   end if
```

A job starts at locality level 0 and can only launch node-local tasks. After at least W1 seconds the job advances at level 1 and may launch rack-local tasks then after a further W2 seconds, it goes to level 2 and may launch off-rack tasks. If a job launches a local task with a locality higher than the level it is on, it goes back down to a previous level.

In summary, delay scheduling can be generalized to preferences other than locality and the only requirement is to have a sorted list of jobs according to some criteria. It can also be applied to resource types other than slots. Measurements reported in [541] show that near 100% locality can be achieved by relaxing fairness.

9.11 DATA-AWARE SCHEDULING

The analysis of delay scheduling emphasizes the important role of data locality for I/O-intensive application performance. This is the topic discussed in this section and addressed by a paper covering task scheduling of I/O-intensive applications [499].

There are many I/O-intensive applications translated into jobs described as a DAG (Direct Acyclic Graph) of tasks. Examples of such applications include MapReduce, approximate query processing applied to exploratory data analysis and interactive debugging, and machine learning applied to spam classification and machine translation.

Jobs running such applications consist of multiple sets of tasks running in every stage; later-stage tasks consume data generated by early-stage tasks. For some of these applications different subsets of tasks can be scheduled independently to optimize data locality, without affecting the correctness of the results. This is the case of an application using the gradient descent algorithm.

The gradient descent is a first-order iterative optimization algorithm. To find local minima, the algorithm takes steps proportional with the negative of the gradient of the function at each iteration. Figure 9.8 shows that such an application has multiple stages and that groups of tasks at each stage can be scheduled independently. Data-aware scheduling improves locality hence, response time and performance.

In this context *late binding* means correlating tasks with data dynamically, depending on the state of the cluster. *Data-aware scheduling* improves the locality of early-stage tasks and, whenever possible, the locality of the later-stage tasks of the job. Locality of all tasks is important because the job completion time is determined by the slowest task thus, a special attention should be paid to *straggler*[13] tasks.

Recall from Chapter 8 that communication latency increases, while the communication bandwidth decreases, with the "distance" between the server running a task and the one where the data is stored. An I/O-intensive task operates efficiently when its input data is already loaded in local memory and less efficiently when the data is stored on a local disk. The task efficiency decreases even further when the data resides on a different server of the same rack, and it is significantly lower when the data is on a server in a different rack.

A number of racks are interconnected by a cell switch thus, one of the scheduler's objective is to balance cross-rack traffic. Intermediate stages of a job often involve group communication among tasks running on servers in different racks, e.g., as one-to-many, many-to-one, and many-to-many data exchanges. A second important goal of data-aware scheduling is to reduce cross-rack communication through the placement of the producer and consumer tasks.

KMN, the scheduler discussed in [499], launches a number of additional tasks in the early stages thus, allows choices for the later stage tasks. The name of the system comes from its basic ideas,

[13]According to the Merriam Webster dictionary, "straggling" means to "walk or move in a slow and disorderly way," or "spread out from others of the same kind."

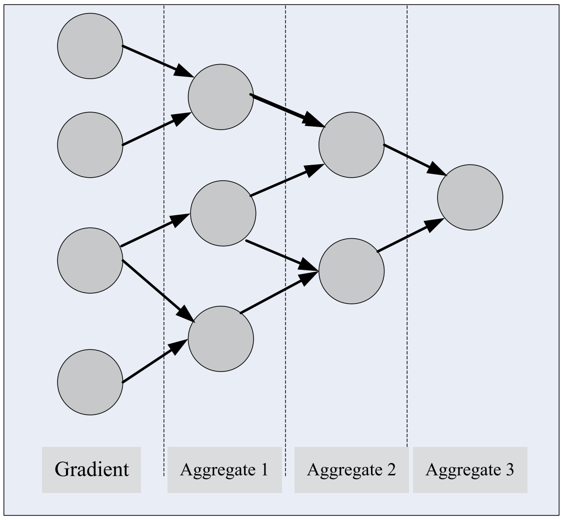


FIGURE 9.8

Stages of an application using the graduate descent algorithm. Tasks in the later stages of the computation use data produced by the early stage tasks. Of the four tasks in the *Gradient* stage the top group of two tasks can be scheduled independently of the group of the lower two tasks.

choose K out of N blocks of input data and schedule $M > K$ first-stage tasks on servers where these blocks reside. The "best" K out of $\binom{M}{K}$ choices increases the likelihood that upstream tasks outputs are distributed across racks and the next-stage tasks using these data as input are scheduled such that the cross-rack traffic is balanced. This heuristic is justified because selecting the best location of next stage tasks based on the output produced by earlier tasks is an NP-complete problem.

KMN is implemented in Scala[14] and it is built on top of Spark, the system for in-memory cluster computing discussed in Section 8.11. One of the novel ideas of the KMN scheduler is to choose K out of N input data blocks to improve locality of a cluster with S slots per server. If u denotes the utilization of a slot, then server utilization when all its slots are busy is u^S. The probability that one of the S tasks running on the server has a block of input data on the local disk is $p_t = 1 - u^S$.

When we choose K out of N the scheduler can choose $\binom{N}{K}$ input block combinations. The probability that K out of N tasks enjoy locality, $p_{K|N}$, is given by the binomial distribution assuming that the probability of success is p_t.

$$p_{K|N} = 1 - \sum_{i=0}^{K-1} \binom{N}{i} p_t^i (1 - p_t)^{N-i}, \tag{9.34}$$

[14]Scala is a general-purpose programming language with a strong static type system and support for functional programming. Scala code is compiled as Java byte code and runs on JVM (Java Virtual Machine). KMN consists of some 1 400 lines of Scala code.

or

$$p_{K|N} = 1 - \sum_{i=0}^{K-1} \binom{N}{i} \left(1 - u^S\right)^i u^{S(N-i)}. \tag{9.35}$$

It is easy to see that the probability of achieving locality is high even for very large utilization of the server slots, e.g., when $u = 90\%$.

The probability that all K blocks in one of the $f = \binom{N}{K}$ samples achieve locality is p_t^K and, as the samples are independent, the probability that at lest one of the samples achives locality is

$$p_{K|N}^{(1)} = (1 - p_t^K)^f. \tag{9.36}$$

This probability increases as f increases.

Selection of the best K outputs from the M upstream tasks using a round-robin strategy is described by the following pseudocode from [499]:

```
//Given: upstreamTasks - list with rack, index within rack for each task
//Given: K - number of tasks to pick
// Number of upstream tasks in each rack
upstreamRacksCount = map()
// Initialize
for task in upstreamTasks do
          upstreamRacksCount[task:rack] += 1
end for
// Sort the tasks in round-robin fashion
roundRobin = upstreamTasks.sort(CompareTasks)
chosenK = roundRobin[0 : K]
return chosenK
procedure COMPARETASKS(task1; task2)
        if task1:idx != task2:idx then
               // Sort first by index
        return task1:idx < task2:idx
        else
              // Then by number of outputs
              numRack1 = upstreamRacksCount[task1:rack]
              numRack2 = upstreamRacksCount[task2:rack]
              return numRack1 > numRack2
         end if
end procedure
```

A hash map with input the list of upstream tasks stores how many tasks should run on each rack. Then the tasks are sorted first by their index in the rack and then by the number of tasks in the rack.

Experiments conducted on an EC2 cluster with 100 servers show that when the KMN scheduler is used instead of the native *Spark* scheduler the average job completion time is reduced by 81%. This reduction is due to the 98% locality of input tasks and a 48% improvement in data transfer. The overhead of the KMN scheduler is small, it uses 5% additional resources.

9.12 APACHE CAPACITY SCHEDULER

Apache capacity scheduler [34] is a pluggable MapReduce scheduler for Hadoop. It supports multiple queues, job priorities, and guarantees to each queue a fraction of the capacity of the cluster. Other features of the scheduler are:

1. Free resources can be allocated to any queue beyond its guaranteed capacity. Excess allocated resources can be reclaimed and made available to another queue to meet its guaranteed capacity.
2. Excess resources taken from a queue will be restored to the queue within N minutes of the instance they are need.
3. Higher priority jobs in a queue have access to resources allocated to the queue, before jobs with lower priority have access.
4. Does not support preemption; once a job is running, it will not be preempted for a higher priority job.
5. Each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for them.
6. Supports memory-intensive jobs. A job can specify higher memory-requirements than the default, and the tasks of the job will only be run on TaskTrackers[15] that have enough memory to spare.

When a TaskTracker is free, the scheduler chooses the queue that needs to reclaim any resources the earliest and if no such queue exits it then chooses a queue whose ratio of the number of running slots to guaranteed capacity is the lowest. Once a queue is selected, the scheduler chooses a job in the queue. Jobs are sorted based on submission time and priority (if supported). Once a job is selected, the scheduler chooses a task to run. Periodically, the scheduler takes actions allowing queues to reclaim capacity as follows:

(a) A queue reclaims capacity when it has at least one task pending and a fraction of its guaranteed capacity is used by another queue; the scheduler determines the amount of resources to reclaim within the reclaim time for the queue.

(b) A queue does receive all resources it is allowed to reclaim and its reclaim time is about to expire; then the scheduler kills the tasks that started the latest.

The scheduler can be configured with several properties for each queue using the file *conf/capacity-scheduler.xml*. Queue properties can be defined by concatenating the string *mapred.capacity-scheduler.queue.*$\langle queue - name \rangle$ with the property name. The property name can be:

.guaranteed capacity – percentage of the number of slots guaranteed to be available for jobs in the queue i.

.reclaim-time-limit – the amount of time, in seconds, before resources distributed to other queues will be reclaimed.

.supports-priority – if true, priorities of jobs will be taken into account in scheduling decisions.

.user-limit-percent – if there is competition for resources each queue enforces a limit on the percentage of resources allocated to a user at any given time. If two users have submitted jobs to a queue, no single user can use more than 50% of the queue resources. If a third user submits a job, no single

[15]In Hadoop the JobTracker and TaskTracker daemons handle the processing of MapReduce jobs.

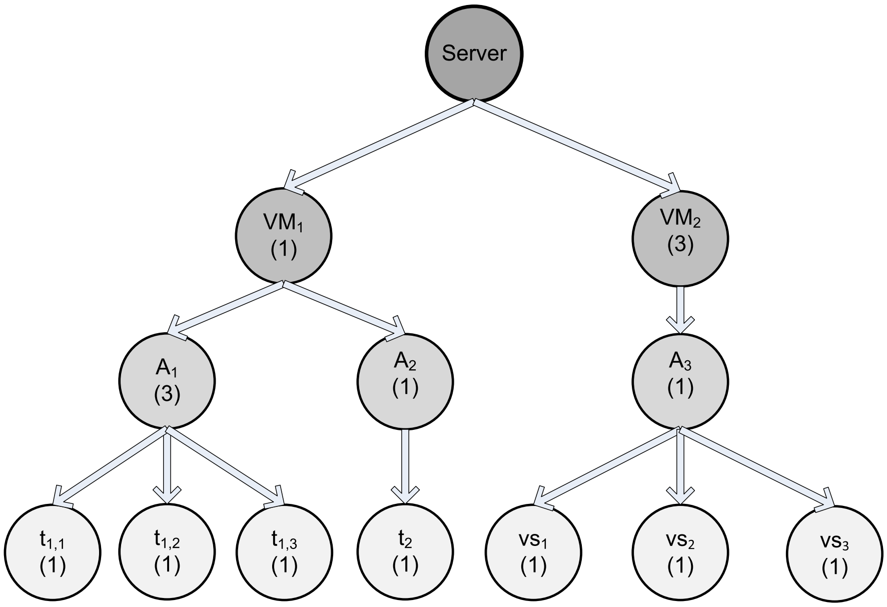


FIGURE 9.9

The SFQ tree for scheduling when two VMs VM_1 and VM_2 run on a powerful server. VM_1 runs two best-effort applications A_1, with three threads $t_{1,1}, t_{1,2}$, and $t_{1,3}$, and A_2 with a single thread t_2; VM_2 runs a video-streaming application A_3 with three threads vs_1, vs_2, and vs_3. The weights of VMs, applications, and individual threads are shown in parenthesis.

user can use more than 33% of the queue resources. With 4 or more users, no user can use more than 25% of the queue's resources. A value of 100 implies no user limits are imposed.

9.13 START-TIME FAIR QUEUING

A hierarchical CPU scheduler for multimedia operating systems was proposed in [200]. The basic idea of the *start-time fair queuing* (SFQ) algorithm is to organize the consumers of the CPU bandwidth in a tree structure. The root node is the processor and the leaves of this tree are the threads of each application.

A scheduler acts at each level of the hierarchy. The fraction of the processor bandwidth, B, allocated to the intermediate node i is

$$\frac{B_i}{B} = \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{9.37}$$

with w_j, $1 \leq j \leq n$, the weight of the n children of node i, see the example in Figure 9.9.

When a VM is not active, its bandwidth is reallocated to the other VMs active at the time. When one of the applications of a VM is not active, its allocation is transferred to the other applications running on the same VM. Similarly, if one of the threads of an application is not runnable then its allocation is transferred to the other threads of the applications.

Call $v_a(t)$ and $v_b(t)$ the virtual time of threads a and b, respectively, at real time t. The virtual time of the scheduler at time t is denoted by $v(t)$. Call q the time quantum of the scheduler, in milliseconds. The threads a and b have their time quantum, q_a and q_b, weighted by w_a and w_b, respectively; thus, in our example, the time quantum of the two threads, are q/w_a and q/w_b, respectively. The i-th activation of thread a will start at virtual time S_a^i and will finish at virtual time F_a^i. We call τ^j the real time of the j-th invocation of the scheduler.

An SFQ scheduler follows several rules:

1. Threads are serviced in the order of their virtual start up time; ties are broken arbitrarily.
2. The virtual startup time of the i-th activation of thread x is

$$S_x^i(t) = \max\left\{v(\tau^j),\, F_x^{(i-1)}(t)\right\} \quad \text{and} \quad S_x^0 = 0. \tag{9.38}$$

   The condition for thread i to be started is that thread $(i-1)$ has finished and that the scheduler is active.
3. The virtual finish time of the i-th activation of thread x is

$$F_x^i(t) = S_x^i(t) + \frac{q}{w_x}. \tag{9.39}$$

   A thread is stopped when its time quantum has expired; its time quantum is the time quantum of the scheduler divided by the weight of the thread.
4. The virtual time of all threads is initially zero, $v_x^0 = 0$. The virtual time $v(t)$ at real time t is computed as follows:

$$v(t) = \begin{cases} \text{Virtual start time of the thread in service at time } t, & \text{if CPU is busy} \\ \text{Maximum finish virtual time of any thread,} & \text{if CPU is idle} \end{cases} \tag{9.40}$$

In this description of the algorithm we have included the real time t to stress the dependence of all events in virtual time on the real time. To simplify the notation we'll use in our examples the real time as the index of the event, in other words at S_a^6 means the start up time of thread a at real time $t = 6$.

Example. The following example illustrates the application of the SFQ algorithm when there are two threads with the weights $w_a = 1$ and $w_b = 4$ and the time quantum is $q = 12$, see Figure 9.10.

Initially $S_a^0 = 0$, $S_b^0 = 0$, $v_a(0) = 0$, and $v_b(0) = 0$. Thread b blocks at time $t = 24$ and wakes up at time $t = 60$.

The scheduling decisions are made as follows:

1. $\underline{t = 0}$: we have a tie, $S_a^0 = S_b^0$ and arbitrarily thread b is chosen to run first; the virtual finish time of thread b is

$$F_b^0 = S_b^0 + q/w_b = 0 + 12/4 = 3. \tag{9.41}$$

2. $\underline{t = 3}$: both threads are runnable and thread b was in service, thus, $v(3) = S_b^0 = 0$; then

$$S_b^1 = \max\left\{v(3),\, F_b^0\right\} = \max\{0, 3\} = 3. \tag{9.42}$$

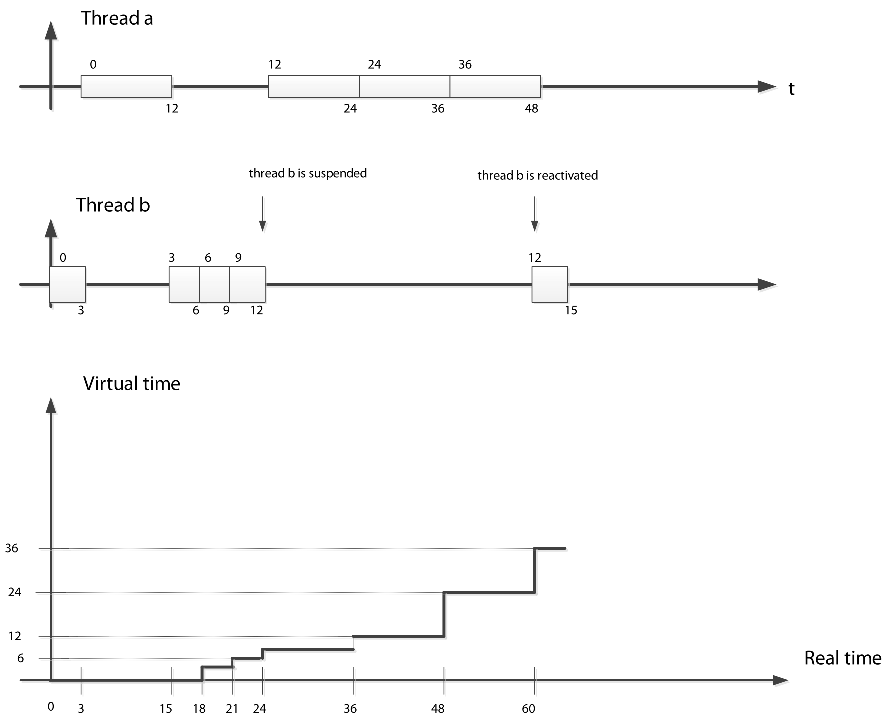


FIGURE 9.10

Top, the virtual startup time $S_a(t)$ and $S_b(t)$ and the virtual finish time $F_a(t)$ and $F_b(t)$ function of the real time t for each activation of threads a and b, respectively, are marked at the top and, respectively, at the bottom of the box representing a running thread. The virtual time of the scheduler $v(t)$ function of the real time is shown on the bottom graph.

But $S_a^0 < S_b^1$ thus thread a is selected to run. Its virtual finish time is

$$F_a^0 = S_a^0 + q/w_a = 0 + 12/1 = 12. \tag{9.43}$$

3. $\underline{t = 15}$: both threads are runnable and thread a was in service at this time thus,

$$v(15) = S_a^0 = 0 \tag{9.44}$$

and

$$S_a^1 = \max\left\{v(15), F_a^0\right\} = \max\{0, 12\} = 12. \tag{9.45}$$

As $S_b^1 = 3 < 12$, thread b is selected to run; the virtual finish time of thread b is now

$$F_b^1 = S_b^1 + q/w_b = 3 + 12/4 = 6. \tag{9.46}$$

4. $\underline{t = 18}$: both threads are runnable and thread b was in service at this time, thus,

$$v(18) = S_b^1 = 3 \tag{9.47}$$

and

$$S_b^2 = \max\left\{v(18), F_b^1\right\} = \max\{3, 6\} = 6. \tag{9.48}$$

As $S_b^2 < S_a^1 = 12$, thread b is selected to run again; its virtual finish time is

$$F_b^2 = S_b^2 + q/w_b = 6 + 12/4 = 9. \tag{9.49}$$

5. $\underline{t = 21}$: both threads are runnable and thread b was in service at this time, thus,

$$v(21) = S_b^2 = 6 \tag{9.50}$$

and

$$S_b^3 = \max\left\{v(21), F_b^2\right\} = \max\{6, 9\} = 9. \tag{9.51}$$

As $S_b^2 < S_a^1 = 12$, thread b is selected to run again; its virtual finish time is

$$F_b^3 = S_b^3 + q/w_b = 9 + 12/4 = 12. \tag{9.52}$$

6. $\underline{t = 24}$: Thread b was in service at this time, thus,

$$v(24) = S_b^3 = 9 \tag{9.53}$$

$$S_b^4 = \max\left\{v(24), F_b^3\right\} = \max\{9, 12\} = 12. \tag{9.54}$$

Thread b is suspended till $t = 60$, thus, the thread a is activated; its virtual finish time is

$$F_a^1 = S_a^1 + q/w_a = 12 + 12/1 = 24. \tag{9.55}$$

7. $\underline{t = 36}$: thread a was in service and it is the only runnable thread at this time, thus,

$$v(36) = S_a^1 = 12 \tag{9.56}$$

and

$$S_a^2 = \max\left\{v(36), F_a^2\right\} = \max\{12, 24\} = 24. \tag{9.57}$$

Then,

$$F_a^2 = S_a^2 + q/w_a = 24 + 12/1 = 36. \tag{9.58}$$

8. $\underline{t = 48}$: thread a was in service and it is the only runnable thread at this time thus,

$$v(48) = S_a^2 = 24 \tag{9.59}$$

and

$$S_a^3 = \max\left\{v(48), F_a^2\right\} = \max\{24, 36\} = 36. \tag{9.60}$$

Then,

$$F_a^3 = S_a^3 + q/w_a = 36 + 12/1 = 48. \tag{9.61}$$

9. $\underline{t = 60}$: thread a was in service at this time, thus,

$$v(60) = S_a^3 = 36 \tag{9.62}$$

and

$$S_a^4 = \max\left\{v(60), F_a^3\right\} = \max\{36, 48\} = 48. \tag{9.63}$$

But now thread b is runnable and $S_b^4 = 12$.
Thus, thread b is activated and

$$F_b^4 = S_b^4 + q/w_b = 12 + 12/4 = 15. \tag{9.64}$$

Several properties of the SFQ algorithm are proved in [200]. The algorithm allocates CPU fairly when the available bandwidth varies in time and provides throughput, as well as delay guarantees. The algorithm schedules the threads in the order of their virtual startup time, the shortest one first; the length of the time quantum is not required when a thread is scheduled, but only after the thread has finished its current allocation. The authors of [200] report that the overhead of the SFQ algorithms is comparable to that of the Solaris scheduling algorithm.

9.14 BORROWED VIRTUAL TIME

The objective of the *borrowed virtual time* (BVT) algorithm is to support low-latency dispatching of real-time applications, as well as a weighted sharing of the CPU among several classes of applications [155]. Like SFQ, the BVT algorithm supports scheduling a mixture of applications, some with hard, some with soft real-time constraints, and applications demanding only a best-effort.

Thread i has an *effective virtual time*, E_i, an *actual virtual time*, A_i, as well as a *virtual time warp*, W_i. The scheduler thread maintains its own *scheduler virtual time* (SVT) defined as the minimum actual virtual time A_j of any thread. The threads are dispatched in the order of their effective virtual time, E_i, a policy called the Earliest Virtual Time (EVT).

The virtual time warp allows a thread to acquire an earlier effective virtual time, in other words, to borrow virtual time from its future CPU allocation. The virtual warp time is enabled when the variable *warpBack* is set; in this case a latency-sensitive thread gains dispatching preference as

$$E_i \leftarrow \begin{cases} A_i & \text{if} \quad warpBack = OFF \\ A_i - W_i & \text{if} \quad warpBack = ON \end{cases} \tag{9.65}$$

The algorithm measures the time in *minimum charging units*, *mcu*, and uses a time quantum called *context switch allowance (C)* which measures the real time a thread is allowed to run when competing with other threads, measured in multiples of mcu; typical values for the two quantities are $mcu =$ 100 μsec and $C = 100$ msec. A thread is charged an integer number of mcu.

Context switches are triggered by traditional events, the running thread is blocked waiting for an event to occur, the time quantum expires, an interrupt occurs; context switching also occurs when a thread becomes runnable after sleeping. When the thread τ_i becomes runnable after sleeping, its actual virtual time is updated as follows

$$A_i \leftarrow \max\{A_i, SVT\}. \tag{9.66}$$

This policy prevents a thread that has been sleeping for a long time to claim control of the CPU for a longer period of time than it deserves.

If there are no interrupts threads are allowed to run for the same amount of virtual time. Individual threads have weights; a thread with a larger weight consumes its virtual time more slowly. In practice, each thread τ_i maintains a constant k_i and uses its weight w_i to compute the amount Δ used to advance its actual virtual time upon completion of a run

$$A_i \leftarrow A_i + \Delta. \tag{9.67}$$

Given two threads a and b

$$\Delta = \frac{k_a}{w_a} = \frac{k_b}{w_b}. \tag{9.68}$$

The BVT policy requires that every time the actual virtual time is updated, a context switch from the current running thread τ_i to a thread τ_j occurs if

$$A_j \leq A_i - \frac{C}{w_i}. \tag{9.69}$$

Example 1. The following example illustrates the application of the BVT algorithm for scheduling two threads a and b of best-effort applications. The first thread has a weight twice of the weight of the second, $w_a = 2w_b$; when $k_a = 180$ and $k_b = 90$, then $\Delta = 90$.

We consider periods of real time allocation of $C = 9$ mcu; the two threads a and b are allowed to run for $2C/3 = 6$ mcu and $C/3 = 3$ mcu, respectively.

Threads a and b are activated at times

$$\begin{array}{ll} a: & 0, 5, 5+9=14, 14+9=23, 23+9=32, 32+9=41, \dots \\ b: & 2, 2+9=11, 11+9=20, 20+9=29, 29+9=38, \dots \end{array} \tag{9.70}$$

The context switches occur at real times

$$2, 5, 11, 14, 20, 23, 29, 32, 38, 41, \dots \tag{9.71}$$

The time is expressed in units of mcu. The initial run is a shorter one, consists of only 3 mcu; a context switch occurs when a, which runs first, exceeds b by 2 mcu.

Table 9.4 shows the effective virtual time of the two threads at the time of each context switch. At that moment, its actual virtual time is incremented by an amount equal to Δ if the thread was allowed

Table 9.4 The real time of the context switch and the effective virtual time $E_a(t)$ and $E_b(t)$ at the time of a context switch. There is no time warp, thus, the effective virtual time is the same as the actual virtual time. At time $t = 0$, $E_a(0) = E_b(0) = 0$ and we choose thread a to run.

Context switch	Real time	Running thread	Effective virtual time of the running thread
1	$t = 2$	a	$E_a(2) = A_a(2) = A_a(0) + \Delta/3 = 30$
			b runs next as $E_b(2) = 0 < E_a(2) = 30$
2	$t = 5$	b	$E_b(5) = A_b(5) = A_b(0) + \Delta = 90$
			a runs next as $E_a(5) = 30 < E_b(5) = 90$
3	$t = 11$	a	$E_a(11) = A_a(11) = A_a(2) + \Delta = 120$
			b runs next as $E_b(11) = 90 < E_a(11) = 120$
4	$t = 14$	b	$E_b(14) = A_b(14) = A_b(5) + \Delta = 180$
			a runs next as $E_a(14) = 120 < E_b(14) = 180$
5	$t = 20$	a	$E_a(20) = A_a(20) = A_a(11) + \Delta = 210$
			b runs next as $E_b(20) = 180 < E_a(20) = 210$
6	$t = 23$	b	$E_b(23) = A_b(23) = A_b(14) + \Delta = 270$
			a runs next as $E_a(23) = 210 < E_b(23) = 270$
7	$t = 29$	a	$E_a(29) = A_a(29) = A_a(20) + \Delta = 300$
			b runs next as $E_b(29) = 270 < E_a(29) = 300$
8	$t = 32$	b	$E_b(32) = A_b(32) = A_b(23) + \Delta = 360$
			a runs next as $E_a(32) = 300 < E_b(32) = 360$
9	$t = 38$	a	$E_a(38) = A_a(38) = A_a(29) + \Delta = 390$
			b runs next as $E_b(11) = 360 < E_a(11) = 390$
10	$t = 41$	b	$E_b(41) = A_b(41) = A_b(32) + \Delta = 450$
			a runs next as $E_a(41) = 390 < E_b(41) = 450$

to run for its time allocation. The scheduler compares the effective virtual time of the threads and runs first the one with the minimum effective virtual time.

Figure 9.11 displays the effective virtual time and the real time of the threads a and b. When a thread is running, its effective virtual time increases as the real time increase; a running thread appears as a diagonal line. When a thread is runnable, but not running, its effective virtual time is constant; a runnable period is displayed as a horizontal line. We see that the two threads are allocated equal amounts of virtual time, but the thread a, with a larger weight, consumes its real time more slowly.

Example 2. Next we consider the previous example but, this time there is an additional thread, c, with real-time constraints, which wakes up at time $t = 9$ and then periodically at times $t = 18, 27, 36, \ldots$ for 3 units of time.

Table 9.5 summarizes the evolution of the system when the real-time application thread c competes with the two best-effort threads a and b. Context switches occur now at real times

$$t = 2, 5, 9, 12, 14, 18, 21, 23, 27, 30, 32, 36, 39, 41, \ldots \qquad (9.72)$$

The context switches at times

$$t = 9, 18, 27, 36, \ldots \qquad (9.73)$$

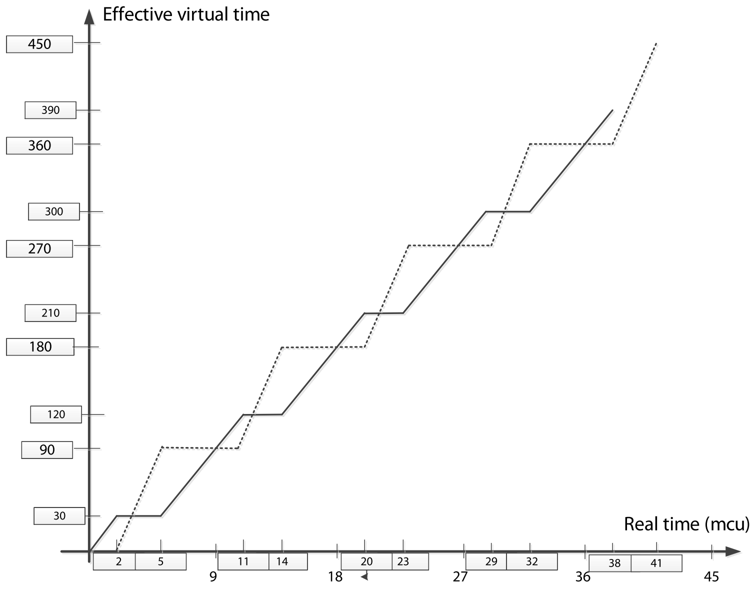


FIGURE 9.11

Example 1 – the effective virtual time and the real time of the threads a (solid line) and b (dotted line) with weights $w_a = 2w_b$ when the actual virtual time is incremented in steps of $\Delta = 90\ mcu$. The real time the two threads are allowed to use the CPU is proportional with their weights; the virtual times are equal but thread a consumes it more slowly. There is no time warp, the threads are dispatched based on their actual virtual time.

are triggered by the waking up of the thread c which preempts the currently running thread. At $t = 9$ the time warp $W_c = -60$ gives priority to thread c. Indeed

$$E_c(9) = A_c(9) - W_c = 0 - 60 = -60 \tag{9.74}$$

compared with $E_a(9) = 90$ and $E_b(9) = 90$. The same conditions occur every time the real-time thread wakes-up. The best-effort application threads have the same effective virtual time when the real-time application thread finishes and the scheduler chooses b to be dispatched first. We should also notice that the ratio of real times used by a and b is the same, as $w_a = 2w_b$.

Figure 9.12 shows the effective virtual times for the three threads a, b, and c. Every time when thread c wakes up it preempts the current running thread and it is immediately scheduled to run.

9.15 FURTHER READINGS

Cloud resource management poses new and extremely challenging problems so there should be no surprise that this is a very active area of research. A fair number of papers including [98], [436], and [33] are dedicated to different resource management policies. Several papers are concerned with QoS [171] and Service Level Agreements [194]. SLA-driven capacity management and SLA-based resource

Table 9.5 A real-time thread c with a time warp $W_c = -60$ is waking up periodically at times $t = 18, 27, 36, \ldots$ for 3 units of time and is competing with the two best-effort threads a and b. The real time and the effective virtual time of the three threads of each context switch are shown.

Context switch	Real time	Running thread	Effective virtual time of the running thread
1	$t = 2$	a	$E_a(2) = A_a(2) = A_a(0) + \Delta/3 = 0 + 90/3 = 30$
2	$t = 5$	b	$E_b^1 = A_b^1 = A_b^0 + \Delta = 0 + 90 = 90 \Rightarrow$ *a runs next*
3	$t = 9$	a	c wakes up
			$E_a^1 = A_a^1 + 2\Delta/3 = 30 + (-60) = 90$
			$[E_a(9), E_b(9), E_c(9)] = (90, 90, -60) \Rightarrow$ *c runs next*
4	$t = 12$	c	$SVT(12) = \min(90, 90)$
			$E_c^s(12) = SVT(12) + W_c = 90 + (-60) = 30$
			$E_c(12) = E_c^s(12) + \Delta/3 = 30 + 30 = 60 \Rightarrow$ *b runs next*
5	$t = 14$	b	$E_b^2 = A_b^2 = A_b^1 + 2\Delta/3 = 90 + 60 = 150 \Rightarrow$ *a runs next*
6	$t = 18$	a	c wakes up
			$E_a^3 = A_a^3 = A_a^2 + 2\Delta/3 = 90 + 60 = 150$
			$[E_a(18), E_b(18), E_c(18)] = (150, 150, 60) \Rightarrow$ *c runs next*
7	$t = 21$	c	$SVT = \min(150, 150)$
			$E_c^s(21) = SVT + W_c = 150 + (-60) = 90$
			$E_c(21) = E_c^s(21) + \Delta/3 = 90 + 30 = 120 \Rightarrow$ *b runs next*
8	$t = 23$	b	$E_b^3 = A_b^3 = A_b^2 + 2\Delta/3 = 150 + 60 = 210 \Rightarrow$ *a runs next*
9	$t = 27$	a	c wakes up
			$E_a^4 = A_a^4 = A_a^3 + 2\Delta/3 = 150 + 60 = 210$
			$[E_a(27), E_b(27), E_c(27)] = (210, 210, 120) \Rightarrow$ *c runs next*
10	$t = 30$	c	$SVT = \min(210, 210)$
			$E_c^s(30) = SVT + W_c = 210 + (-60) = 150$
			$E_c(30) = E_c^s(30) + \Delta/3 = 150 + 30 = 180 \Rightarrow$ *b runs next*
11	$t = 32$	b	$E_b^4 = A_b^4 = A_b^3 + 2\Delta/3 = 210 + 60 = 270 \Rightarrow$ *a runs next*
10	$t = 36$	a	c wakes up
			$E_a^5 = A_a^5 = A_a^4 + 2\Delta/3 = 210 + 60 = 270$
			$[E_a(36), E_b(36), E_c(36)] = (270, 270, 180) \Rightarrow$ *c runs next*
12	$t = 39$	c	$SVT = \min(270, 270)$
			$E_c^s(39) = SVT + W_c = 270 + (-60) = 210$
			$E_c(39) = E_c^s(39) + \Delta/3 = 210 + 30 = 240 \Rightarrow$ *b runs next*
13	$t = 41$	b	$E_b^5 = A_b^5 = A_b^4 + 2\Delta/3 = 270 + 60 = 330 \Rightarrow$ *a runs next*

allocation policies are covered in [5] and [35], respectively. [169] analyzes performance monitoring for SLAs. Dynamic request scheduling of applications subject to SLA requirements is presented in [72]. The QoS in clouds is analyzed in [72]. Semantic resource allocation using a multi-agent system is discussed in [162].

The autonomic computing era is presented in [182]. Energy-aware resource allocation in autonomic computing is covered in [36]. Policies for autonomic computing based on utility functions are analyzed in [266]. Coordination of multiple autonomic managers and power-performance trade-offs

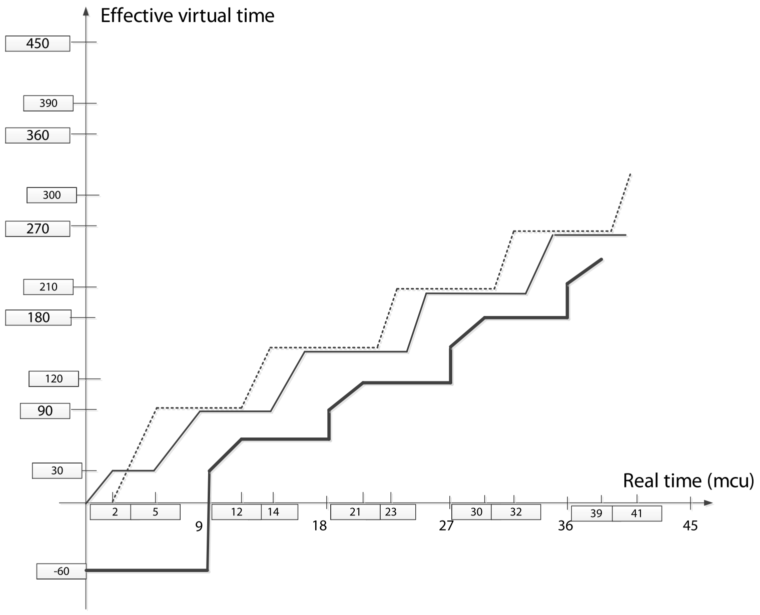


FIGURE 9.12

Example 2 – the effective virtual time and the real time of the threads a (solid line), b (dotted line), and the c with real-time constraints (thick solid line). c wakes up periodically at times $t = 9, 18, 27, 36, \ldots$, is active for 3 units of time and has a time warp of 60 mcu.

are dissected in [265]. Autonomic management of cloud services subject to availability guarantees is presented in [9]. The use of self-organizing agents for service composition in cloud computing is the subject of [214].

An authoritative reference on fault-tolerance is [43]; applications of control theory to resource allocation discussed in [157] and [93] cover resource multiplexing in data centers. Admission control policies are discussed in [211]. Optimal control problems are analyzed in [222] and system testing is covered in [227]. Verification of performance assertion on the cloud is the subject of [276].

Power and performance management are the subject of [285] and performance management for cluster based web services is covered in [386]. Autonomic management of heterogeneous workloads is discussed in [476] and application placement controllers is the topic of [479]. Application of pattern matching for forecasting on demand resources needs are discussed in [89]. Economic models for resource allocations are covered in [324,326], and [435].

Scheduling and resource allocation are also covered by numerous papers: a batch queuing system on clouds with Hadoop and HBase is presented in [547]; data flow driven scheduling for business applications is covered in [152]. Scalable thread scheduling is the topic of [524]. Reservation-based scheduling is discussed in [127]. Anti-caching in database management is the subject of [133]. Scheduling of real time services in cloud computing is presented in [309]. The OGF (Open Grid Forum) OCCI (Open Cloud Computing Interface) is involved in the definition of virtualization formats and APIs for IaaS.

Flexible memory exchange, bag of tasks scheduling and distributed low latency scheduling are covered in [375], [379] and [384], respectively. Reference [428] analyzes capacity management for pools of resources.

9.16 EXERCISES AND PROBLEMS

Problem 1. Analyze the benefits and the problems posed by the four approaches for the implementation of resource management policies: control theory, machine learning, utility-based, market-oriented.

Problem 2. Can optimal strategies for the five classes of policies, admission control, capacity allocation, load balancing, energy optimization, and QoS guarantees be actually implemented in a cloud? The term "optimal" is used in the sense of control theory. Support your answer with solid arguments. Optimal strategies for one may be in conflict with optimal strategies for one or more of the other classes. Identify and analyze such cases.

Problem 3. Analyze the relationship between the scale of a system and the policies and the mechanisms for resource management. Consider also the geographic scale of the system in your arguments.

Problem 4. What are the limitations of the control theoretic approach discussed in Section 9.4? Do the approaches discussed in Sections 9.5 and 9.6 remove or relax some of these limitations? Justify your answers.

Problem 5. Multiple controllers are probably necessary due to the scale of the cloud. Is it beneficial to have system and application controllers? Should the controllers be specialized; for example, some to monitor performance, others to monitor power consumption? Should all the functions we want to base the resource management policies on be integrated in a single controller and one such controller be assign to a given number of servers, or to a geographic region? Justify your answers.

Problem 6. In a scale-free network the degrees of the nodes have an exponential distribution. A scale-free network could be used as a virtual network infrastructure for cloud computing. *Controllers* represent a dedicated class of nodes tasked with resource management; in a scale-free network nodes with a high connectivity can be designated as controllers. Analyze the potential benefit of such a strategy.

Problem 7. Use the start-time fair queuing (SFQ) scheduling algorithm to compute the virtual startup and the virtual finish time for two threads a and b with weights $w_a = 1$ and $w_b = 5$ when the time quantum is $q = 15$ and thread b blocks at time $t = 24$ and wakes up at time $t = 60$. Plot the virtual time of the scheduler function of the real time.

Problem 8. Apply the borrowed virtual time (BVT) scheduling algorithm to the problem in Example 2 of Section 9.14 but with a time warp of $W_c = -30$.

Problem 9. In Section 9.2 we introduced the concept of energy-proportional systems and we saw that different system components have different dynamic ranges. Sketch a strategy to reduce the power consumption in a lightly-loaded cloud and discuss the steps for placing a computational server in a standby mode and then for bringing it back up to an active mode.

Problem 10. Overprovisioning is the reliance on extra capacity to satisfy the needs of a large community of users when the average-to-peak resource demand ratio is very high. Give an example of a large-scale system using overprovisioning and discuss if overprovisioning is sustainable in that case and what are the limitations of it. Is cloud elasticity based on overprovisioning sustainable? Give the arguments to support your answer.

CHAPTER

CLOUD RESOURCE VIRTUALIZATION 10

Three classes of fundamental abstractions are necessary to describe the operation of a computing system: interpreters, memory, and communications channels, [434]. Their respective physical realizations are: processors to transform information; primary and the secondary memory to store information; and communication systems allowing different systems to interact with one another. Processors, memory, and communication systems have different bandwidth, latency, reliability, and other physical characteristics. Software systems manage these resources and transform the physical implementations of the three abstractions into computer systems capable to process applications.

The traditional solution for a data center is to install a standard operating system (OS) on individual systems and to rely on conventional OS techniques to ensure resource sharing, application protection, and performance isolation. Cloud service providers, as well as cloud users, face multiple challenges in such a setup.

CSPs are stressed by system administration, accounting, security, and system resource management. The users are under pressure to develop and optimize the performance of their application for one system and, eventually, to start over again when the application is moved to another data center with different system software and libraries.

Resource virtualization, the technique analyzed in this chapter, is a ubiquitous alternative to the traditional data center operation. Virtualization, a basic tenet of cloud computing, simplifies some of the resource management tasks. For example, the state of a virtual machine running under a hypervisor can be saved and migrated to another server to balance the load. At the same time, virtualization allows users to operate in environments they are familiar with, rather than forcing them to work in idiosyncratic environments.

Resource sharing in a VM environment requires not only ample hardware support and, in particular, powerful processors and fast interconnects, but also architectural support for multi-level control as resource sharing occurs at multiple levels. Resources, such as CPU cycles, memory, secondary storage, and I/O and communication bandwidth, are shared among several VMs. The resources of one VM are shared among multiple processes or threads of an application.

This chapter starts with a discussion of virtualization principles and the motivation for virtualization. Section 10.1 is focused on performance and security isolation. Alternatives for the implementation of virtualization are analyzed in Section 10.2.

Two distinct approaches for processor virtualization, *full virtualization* and *paravirtualization* are presented in Section 10.3. Full virtualization is feasible when the hardware abstraction provided by the hypervisor is an exact replica of the physical hardware. In this case any operating system running on the hardware will run without modifications under the hypervisor. In contrast, paravirtualization requires modifications of the guest operating systems because the hardware abstraction provided by the hypervisor does not support all functions the hardware does.

Traditional processor architectures support two execution modes, kernel and user mode. In a virtualized environment the hypervisor controls resource sharing among the VMs. A guest OS manages resources allocated to the applications running under a VM. While a two-level scheduling for sharing CPU cycles can be easily implemented, sharing of other resources such as cache, memory, and I/O bandwidth is more intricate. In 2005 and 2006 the x86 processor architecture was extended to provide hardware support for virtualization, as discussed in Section 10.4. Nested virtualization allows hypervisors to run inside a VM complicating even further the virtualization landscape.

Several hypervisors are used nowadays. One of them, Xen, is analyzed in Section 10.5 and an optimization of its network performance is presented in Section 10.6. KVM, a virtualization infrastructure of the Linux kernel, is discussed in Section 10.7 and nested virtualization is analyzed in Section 10.8 followed by the presentation of a trusted kernel virtualization in Section 10.9.

High performance processors, e.g., Itanium,[1] have multiple functional units, but do not provide explicit support for virtualization, as discussed in Section 10.10 which covers Itanium paravirtualization. System functions, critical for the performance of a VM environment are cache and memory management, handling of privileged instructions, and I/O handling.

Cache misses are an important source of performance degradation in a VM environment as we shall see in Section 10.11. An overview of open source software platforms for virtualization is presented in Section 10.12. The potential risks of virtualization are the subject of Section 10.13 and virtualization software is discussed in Section 10.14.

10.1 PERFORMANCE AND SECURITY ISOLATION IN COMPUTER CLOUDS

To exhibit a predictable performance an application has to be isolated from other applications it shares resources with. Performance isolation is a critical condition for QoS guarantees in cloud computing where system resources are shared.

If the run-time behavior of an application is affected by other applications running concurrently and thus, competing for CPU cycles, cache, main memory, disk and network access, it is rather difficult to predict its completion time. Therefore, it is equally difficult, or in some instances impossible, to optimize application's performance. Performance unpredictability is a "deadly sin" for real-time operation and for embedded systems.

Several operating systems including Linux/RK [374], QLinux [475], and SILK [58] support some performance isolation. In spite of the efforts to support performance isolation, interactions among applications sharing the same physical system, often described as *QoS crosstalk*, still exist [482]. Accounting for all consumed resources and distributing the overhead of different system activities, including context switching and paging to individual users, is challenging.

Processor virtualization presents multiple copies of the same processor or of a core on multicore systems and the application code is executed directly by the hardware. Processor virtualization is different than *processor emulation* when the machine instructions of guest system are emulated by software running on the host system. Emulation is much slower than virtualization. For example, Microsoft's

[1]In the late 2000s Itanium was the fourth-most deployed microprocessor architecture for enterprise-class systems. The first three were Intel's x86-64, IBM's Power Architecture, and Sun's SPARC.

VirtualPC was designed to run on x86 processor architecture. VirtualPC was running on emulated x86 hardware until Apple adopted Intel chips.

Traditional operating systems multiplex multiple processes or threads, while virtualization supported by a hypervisor multiplexes full operating systems. There is a performance penalty due to hypervisor multiplexing. An OS is heavyweight and the overhead of context switching carried out by the hypervisor is larger.

A hypervisor executes directly on the hardware a subset of frequently-used machine instructions generated by the application and emulates privileged instructions including device I/O requests. The subset of instructions executed directly by the hardware includes arithmetic instructions, memory access and branching instructions.

Operating systems use the process abstraction not only for resource sharing, but also to support isolation. Unfortunately, this is not sufficient from a security perspective, once a process is compromised it is rather easy for an attacker to penetrate the entire system.

The software running on a VM has the constraints of its own dedicated hardware, it can only access virtual devices emulated by the software. This layer of software has the potential to provide a level of isolation nearly equivalent to the isolation presented by two different physical systems. Therefore, virtualization can be used to improve security in a cloud computing environment.

A hypervisor is a much simpler and better specified system than a traditional OS. For example, the Xen hypervisor discussed in Section 10.5 has approximately 60 000 lines of code while the Denali hypervisor [522] has only about half, 30 000 lines of code. The security vulnerability of hypervisors is considerably reduced, as the systems expose a much smaller number of privileged functions.

For example, Xen can be accessed through 28 hypercalls, while a standard Linux OS allows hundreds, e.g., Linux 2.6.11 allows 289 system calls. A traditional OS supports special devices, e.g., */dev/kmem*, and many privileged third party programs, e.g., *sendmail* and *sshd*, in addition to a plethora of system calls.

10.2 VIRTUAL MACHINES

A VM is an isolated environment with access to a subset of physical resources of the computer system. Each VM appears to be running on the bare hardware, giving the appearance of multiple instances of the same computer, though all are supported by a single physical system. The history of VMs can be traced back to the early 1960s.[2] In early 1970s IBM released its widely used VM 370 system followed in 1974 by the MVS (Multiple Virtual Storage) system.

There are two types of VMs, process and system, Figure 10.1A:

- A *process VM* is a virtual platform created for an individual process and destroyed once the process terminates. Virtually all operating systems provide a process VM for each one of the applications running, but the more interesting process VMs are those which support binaries compiled on a different instruction set.

[2]In July 1963 MIT announced Project MAC (Multiple Access Computer) and choose GE-645 as the mainframe for its Multics project. IBM was GE's competitor and realized that there is a demand for such systems and designed the CP-40 mainframe followed by CP-67, also called CP/CMS mainframes. CP was a program running on the mainframe used to create VMs running a single-user OS called CMS.

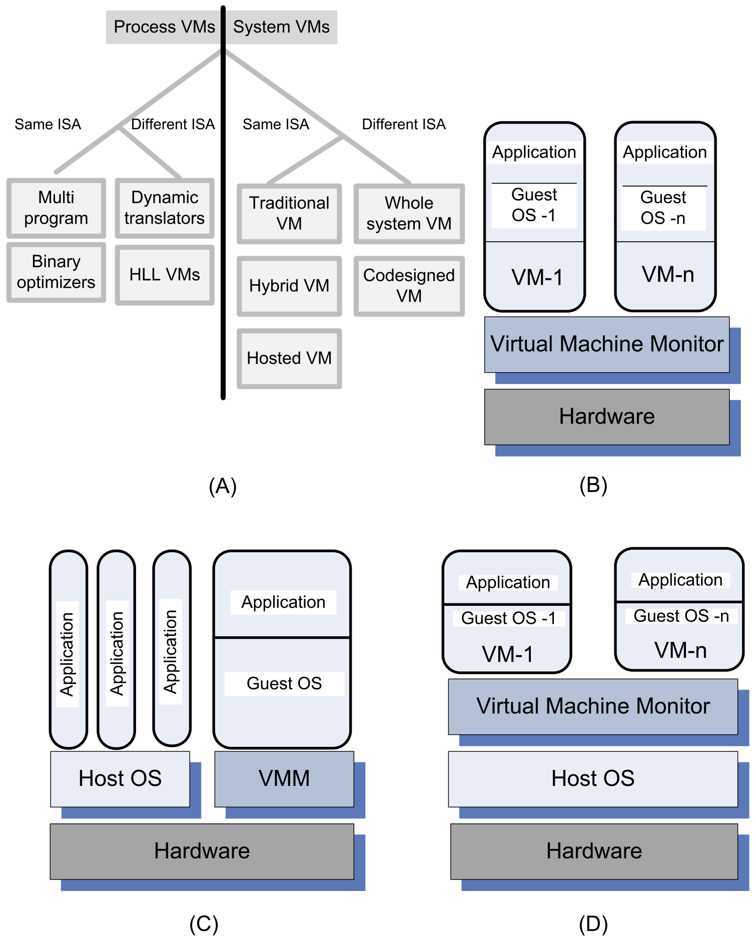


FIGURE 10.1

(A) A taxonomy of process and system VMs for the same and for different Instruction Set Architectures (ISAs). Traditional, Hybrid, and Hosted are three classes of VMs for systems with the same ISA. (B) Traditional VMs; the hypervisor supports multiple VMs and runs directly on the hardware. (C) Hybrid VM; the hypervisor shares the hardware with a host OS and supports multiple VMs. (D) Hosted VM; the hypervisor runs under a host OS.

- A *system VM* supports an OS together with many user processes. When the VM runs under the control of a normal OS and provides a platform-independent host for a single application we have an *application VM*, e.g., Java Virtual Machine (JVM).

A *system VM* provides a complete system; each VM can run its own OS, which in turn can run multiple applications. Systems such as Linux Vserver http://linux-vserver.org, OpenVZ (Open VirtualiZation) [378], FreeBSD Jails [419], and Solaris Zones [409] based on Linux, FreeBSD, and Solaris,

Table 10.1 A non-exhaustive inventory of system VMs. The host ISA refers to the instruction set of the hardware; the guest ISA refers to the instruction set supported by the VM. The VM could run under a host OS, directly on the hardware, or under a hypervisor. The guest OS is the OS running under the control of a VM which in turn may run under the control of the VM monitor.

Name	Host ISA	Guest ISA	Host OS	Guest OS	Company
Integrity VM	x86-64	x86-64	HP-Unix	Linux, Windows HP Unix	HP
Power VM	Power	Power	No host OS	Linux, AIX	IBM
z/VM	z-ISA	z-ISA	No host OS	Linux on z-ISA	IBM
Lynx Secure	x86	x86	No host OS	Linux, Windows	LinuxWorks
Hyper-V Server	x86-64	x86-64	Windows	Windows	Microsoft
Oracle VM	x86, x86-64	x86, x86-64	No host OS	Linux, Windows	Oracle
RTS Hypervisor	x86	x86	No host OS	Linux, Windows	Real Time Systems
SUN xVM	x86, SPARC	same as host	No host OS	Linux, Windows	SUN
VMware EX Server	x86, x86-64	x86, x86-64	No host OS	Linux, Windows Solaris, FreeBSD	VMware
VMware Fusion	x86, x86-64	x86, x86-64	MAC OS x86	Linux, Windows Solaris, FreeBSD	VMware
VMware Server	x86, x86-64	x86, x86-64	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Workstation	x86, x86-64	x86, x86-64	Linux, Windows	Linux, Windows Solaris, FreeBSD	VMware
VMware Player	x86, x86-64	x86, x86-64	Linux Windows	Linux, Windows Solaris, FreeBSD	VMware
Denali	x86	x86	Denali	ILVACO, NetBSD	University of Washington
Xen	x86, x86-64	x86, x86-64	Linux Solaris	Linux, Solaris NetBSD	University of Cambridge

respectively, implement *OS-level virtualization technologies*. Table 10.1 lists a subset of system VMs. A literature search reveals the existence of some 60 different VMs, many created by large software companies.

The OS-level virtualization allows a physical server to run multiple isolated OS instances subject to several constraints; the instances are known as containers, Virtual Private Servers, or Virtual Environments. For example, OpenVZ requires both the host and the guest OS to be Linux distributions. These systems claim performance advantages over the systems based on a hypervisor such as Xen or VMware. According to [378], there is only a 1% to 3% performance penalty for OpenVZ as compared to a standalone Linux server. OpenVZ is licensed under the GPL version 2.

Recall that a hypervisor allows several VMs to share a system. Several organizations of the software stack are possible:

- *Traditional – VM*, also called a "bare metal" hypervisor, a thin software layer that runs directly on the host machine hardware; its main advantage is performance, Figure 10.1B. Examples: VMWare ESX, ESXi Servers, Xen, OS370, and Denali.

- *Hybrid* – the hypervisor shares the hardware with an existing OS, Figure 10.1C. Example: VMWare Workstation.
- *Hosted* – the VM runs on top of an existing OS, Figure 10.1D; the main advantage of this approach is that the VM is easier to build and install. Another advantage of this solution is that the hypervisor could use several components of the host OS, such as the scheduler, the pager and the I/O drivers, rather than providing its own. A price to pay for this simplicity is the increased overhead and the associated performance penalty; indeed, the I/O operations, page faults, and scheduling requests from a guest OS are not handled directly by the hypervisor, instead they are passed to the host OS. Performance, as well as the challenges to support complete isolation of VMs make this solution less attractive for servers in a cloud computing environment. User-mode Linux is an example of a hosted VM.

As pointed out in [102] services provided by the VM "operate below the abstractions provided by the guest OS....... it is difficult to provide a service that checks file system integrity without the knowledge of on-disk structure." Hypervisors discussed in Section 8.4 manage resource sharing among the VMs running on a physical system.

10.3 FULL VIRTUALIZATION AND PARAVIRTUALIZATION

In 1974 Popek and Goldberg gave a set of sufficient conditions for a computer architecture to support virtualization and allow a hypervisor to operate efficiently. Their crisp description of these conditions in [406] is a major contribution to the field:

```
1. A program running under the hypervisor should exhibit
    a behavior essentially identical to that demonstrated
    when running on an equivalent machine directly.

2. The hypervisor should be in complete control of the
    virtualized resources.

3.  A statistically significant fraction of machine instructions
     must be executed without the intervention of the hypervisor.
```

One way to identify an architecture suitable for a VM is to distinguish *sensitive* machine instructions which require special precautions at execution time from machine instructions that can be executed without special precautions. In turn, sensitive instructions are:

- *Control sensitive*, instructions that attempt to change either the memory allocation, or operate in kernel mode.
- *Mode sensitive*, instructions whose behavior is different in kernel mode.

An equivalent formulation of the conditions for efficient virtualization can be based on this classification of machine instructions: *a hypervisor for a third or later generation computer can be constructed*

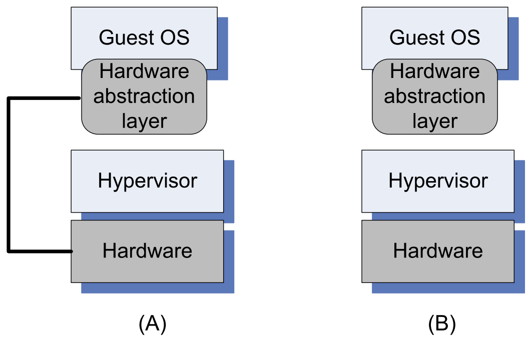


FIGURE 10.2

(A) Full virtualization requires the hardware abstraction layer of the guest OS to have some knowledge about the processor architecture. The guest OS runs unchanged thus, this virtualization mode is more efficient than paravirtualization. (B) Paravirtualization is used when the processor architecture is not easily virtualizable. The hardware abstraction layer of the guest OS does not have knowledge about the hardware. The guest OS is modified to run under the hypervisor and must be ported to individual hardware platforms.

if the set of sensitive instructions is a subset of the privileged instructions of that machine. To handle non-virtualizable instructions one could resort to two strategies:

- Binary translation. The hypervisor monitors the execution of guest operating systems; non-virtualizable instructions executed by a guest OS are replaced with other instructions.
- Paravirtualization. A guest OS is modified to use only instructions that can be virtualized.

There are two basic approaches to processor virtualization, see Figure 10.2: *full virtualization* when each VM runs on an exact copy of the actual hardware; and *paravirtualization* when each VM runs on a slightly modified copy of the actual hardware. Paravirtualization is often adopted for several reasons: (1) some aspects of the hardware cannot be virtualized; (2) has better performance; and (3) presents a simpler interface.

Full virtualization requires a virtualizable architecture. The hardware is fully exposed to the guest OS which runs unchanged and it is necessary to ensure that this execution mode is efficient. Systems such as the VMware EX Server support full virtualization on x86 architecture and have to address several problems, including the virtualization of the MMU. Privileged x86 instructions executed by a guest OS fail silently thus, traps must be inserted whenever privileged instructions are issued by a guest OS. The system must also maintain shadow copies of system control structures, such as page tables, and trap every event affecting the state of these control structures. Therefore, the overhead of many operations is substantial.

Computer architectures such as x86 are not easily virtualizable as we shall see in Section 10.4. Paravirtualization is the alternative, though it has its own problems. Paravirtualization requires modifications to a guest OS. Moreover, the code of the guest OS has to be ported to each individual hardware platform. Xen [53] and Denali [522] are based on paravirtualization.

Generally, the virtualization overhead negatively affects the performance of applications running under a VM. Sometimes, an application running under a VM could perform better than one running under a classical OS. This is the case of *cache isolation* when the cache is divided among VMs. In this case it is beneficial to run workloads competing for cache in two different VMs [453]. Often, the cache is not equally partitioned among processes running under a classical OS and one process may use the cache space better than the other. For example, in case of two processes, one write-intensive and the other read-intensive, the cache may be aggressively filled by the first process.

The I/O performance of applications running under a VM depends on factors such as, the disk partition used by the VM, the CPU utilization, the I/O performance of the competing VMs, and the I/O block size. The discrepancies between the optimal choice and the default ones on a Xen platform are between 8% and 35% [453].

10.4 HARDWARE SUPPORT FOR VIRTUALIZATION

In early 2000 it became obvious that hardware support for virtualization was necessary and Intel and AMD started working on the first generation virtualization extensions of the x86[3] architecture. In 2005 Intel released two Pentium 4 models supporting *VT-x* and in 2006 AMD announced Pacifica and then several Athlon 64 models.

The Virtual Machine Extension (VMX) was introduced by Intel in 2006 and AMD responded with the Secure Virtual Machine (SVM) instruction set extension. The *Virtual Machine Control Structure* (VMCS) of VMX tracks the host state and the guest VMs as control is transferred between them. Three types of data are stored in VMCS:

- *Guest state.* Holds virtualized CPU registers (e.g., control registers or segment registers) automatically loaded by the CPU when switching from kernel mode to guest mode on *VMEntry*.
- *Host state.* Data used by the CPU to restore register values when switching back from guest mode to kernel mode on *VMExit*.
- *Control data.* Data used by the hypervisor to inject events such as exceptions or interrupts into VMs and to specify which events should cause a *VMExit*; it is also used by the CPU to specify the *VMExit* reason.

VMCS is shadowed in hardware to overcome the performance penalties of nested hypervisors discussed in Section 10.8. This allows the guest hypervisor to access VMCS directly, without disrupting the root hypervisor in case of nested virtualization. VMCS shadow access is almost as fast as a non-nested hypervisor environment. VMX includes several instructions [250]:

1. *VMXON* – enter VMX operation;
2. *VMXOFF* – leave VMX operation;
3. *VMREAD* – read from the VMCS;

[3] x86-32, i386, x86 and IA-32 refer to the Intel CISC-based instruction architecture, now supplanted by x86-64 which supports vastly larger physical and virtual address spaces. The x86-64 specification is distinct from Itanium, initially known as IA-64 architecture.

4. *VMWRITE* – write to the VMCS;
5. *VMCLEAR* – clear VMCS;
6. *VMPTRLD* – load VMCS pointer;
7. *VMPTRST* – store VMCS pointer;
8. *VMLAUNCH/VMRESUME* – launch or resume a VM; and
9. *VMCALL* – call to the hypervisor.

A 2006 paper [356] analyzes the challenges to virtualizing Intel architectures and then presents VT-x and VT-i virtualization architectures for x86 and Itanium architectures, respectively. Software solutions at that time addressed some of the challenges, but hardware solution could improve not only performance but also security and, at the same time simplify the software systems. The problems faced by virtualization of the x86 architecture are:

- *Ring deprivileging.* This means that a hypervisor forces a guest VM including an OS and an application, to run at a privilege level greater than 0. Recall that the x86 architecture provides four protection rings, 0–3. Two solutions are then possible:
  1. The (0/1/3) mode when the hypervisor, the guest OS, and the application run at privilege levels 0, 1, and 3, respectively; this mode is not feasible for x86 processors in 64-bit mode, as we shall see shortly.
  2. The (0/3/3) mode when the hypervisor, a guest OS, and applications run at privilege levels 0, 3 and 3, respectively.
- *Ring aliasing.* Such problems are created where a guest OS is forced to run at a privilege level other than that it was originally designed for. For example, when the CS register[4] is *PUSH*ed, the current privilege level in the CR is also stored on the stack [356].
- *Address space compression.* A hypervisor uses parts of the guest address space to store several system data structures such as the interrupt-descriptor table and the global-descriptor table. Such data structures must be protected, but the guest software must have access to them.
- *Non-faulting access to privileged state.* Several instructions, LGDT, SIDT, SLDT, and LTR which load the registers GDTR, IDTR, LDTR, and TR, can only be executed by software running at privileged level 0 because these instructions point to data structures that control the CPU operation. Nevertheless, instructions that store these registers *fail silently* when executed at a privilege level other than 0. This implies that a guest OS executing one of these instructions does not realize that the instruction has failed.
- *Guest system calls.* Two instructions, SYSENTER and SYSEXIT support low-latency system calls. The first causes a transition to privilege level 0, while the second causes a transition from privilege level 0 and fails if executed at a level higher than 0. The hypervisor must then emulate every guest execution of either of these instructions and that has a negative impact on performance.
- *Interrupt virtualization.* In response to a physical interrupt the hypervisor generates a “virtual interrupt” and delivers it later to the target guest OS. But every OS has the ability to mask interrupts,[5] thus the virtual interrupt could only be delivered to the guest OS when the interrupt is not masked.

[4]The x86 architecture supports memory segmentation with a segment size of 64K. The CR (code-segment register) points to the code segment. *MOV, POP*, and *PUSH* instructions serve to load and store segment registers, including CR.
[5]The interrupt flag, IF in the EFLAGS register, is used to control interrupt masking.

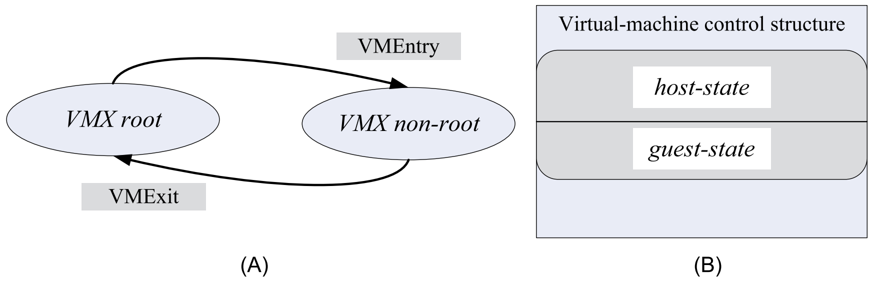


FIGURE 10.3

(A) The two modes of operation of VT-x, and the two operations to transit from one to another; (B) VMCS includes *host-state* and *guest-state* areas which control the *VM entry* and *VM exit* transitions.

Keeping track of all guest OS attempts to mask interrupts greatly complicates the hypervisor and increases the overhead.

- *Access to hidden state.* Elements of the system state, e.g., descriptor caches for segment registers, are hidden; there is no mechanism for saving and restoring the hidden components when there is a context switch from one VM to another.
- *Ring compression.* Paging and segmentation are the two mechanisms to protect hypervisor code from being overwritten by guest OS and applications. Systems running in 64-bit mode can only use paging, but paging does not distinguish between privilege levels 0, 1, and 2, thus the guest OS must run at privilege level 3, the so called (0/3/3) mode. Privilege levels 1 and 2 cannot be used thus, the name ring compression.
- *Frequent access to privileged resources increases hypervisor overhead.* The task-priority register (TPR) is frequently used by a guest OS; the hypervisor must protect the access to this register and trap all attempts to access it. That can cause a significant performance degradation.

Similar problems exist for the *Itanium* architecture discussed in Section 10.10.

A major architectural enhancement provided by the VT-x is the support for two modes of operation and a new data structure, VMCS, including *host-state* and *guest-state* areas, see Figure 10.3:

- *VMX root:* intended for hypervisor operations, and very close to the x86 without *VT-x*.
- *VMX non-root:* intended to support a VM.

When executing a *VMEntry* operation the processor state is loaded from the *guest-state* of the VM scheduled to run; then the control is transferred from the hypervisor to the VM. A *VMExit* saves the processor state in the *guest-state* area of the running VM; it loads the processor state from the *host-state* area, and finally transfers control to the hypervisor. All *VMExit* operations use a common entry point to the hypervisor.

Each *VMExit* operation saves in VMCS the reason for the exit and eventually some qualifications. Some of this information is stored as bitmaps. For example, the *exception bitmap* specifies which one of 32 possible exceptions caused the exit. The *I/O bitmap* contains one entry for each port in a 16-bit I/O space.

The VMCS area is referenced with a physical address and its layout is not fixed by the architecture, but can be optimized by a particular implementation. The VMCS includes control bits that facilitate the implementation of virtual interrupts. For example, *external-interrupt exiting*, when set, causes the execution of a *VM exit* operation; moreover the guest is not allowed to mask these interrupts. When the *interrupt window exiting* is set, a *VM exit* operation is triggered if the guest is ready to receive interrupts.

Processors based on two new virtualization architectures, VT-d[6] and VT-c have been developed. The first supports the I/O Memory Management Unit (I/O MMU) virtualization and the second the network virtualization.

Also known as *PCI pass-through* the I/O MMU virtualization gives VMs direct access to peripheral devices. VT-d supports:

- DMA address remapping, address translation for device DMA transfers.
- Interrupt remapping, isolation of device interrupts and VM routing.
- I/O device assignment, the devices can be assigned by an administrator to a VM in any configuration.
- Reliability features, it reports and records DMA and interrupt errors that may otherwise corrupt memory and impact VM isolation.

10.5 XEN – A HYPERVISOR BASED ON PARAVIRTUALIZATION

Xen is a hypervisor developed by the Computing Laboratory at the University of Cambridge, United Kingdom, in 2003. Since 2010 Xen has been a free software, developed by the community of users and licensed under the GNU General Public License (GPLv2). Several operating systems including Linux, Minix, NetBSD, FreeBSD, NetWare, and OZONE can operate as paravirtualized Xen guest operating systems running on x86, x86-64, Itanium, and ARM architectures.

The goal of the Cambridge group led by Ian Pratt was to design a hypervisor capable of scaling to about 100 VMs running standard applications and services without any modifications to the Application Binary Interface. Fully aware that the x86 architecture does not support efficiently full virtualization, the designers of Xen opted for paravirtualization.

We analyze next the original implementation of Xen for the x86 architecture discussed in [53]. The creators of Xen used the concept of *domain* (Dom) to refer to the ensemble of address spaces hosting a guest OS and address spaces for applications running under this guest OS. Each domain runs on a virtual x86 CPU. Dom0 is dedicated to the execution of Xen control functions and privileged instructions, and DomU is a user domain, Figure 10.4.

The most important aspects of Xen paravirtualization for virtual memory management, CPU multiplexing, and I/O device management are summarized in Table 10.2 [53]. Efficient management of the TLB (Translation Look-aside Buffer), a cache for page table entries, requires either the ability to identify the OS and the address space of every entry, or to allow software management of the TLB.

[6]The corresponding AMD architecture is called AMD-Vi.

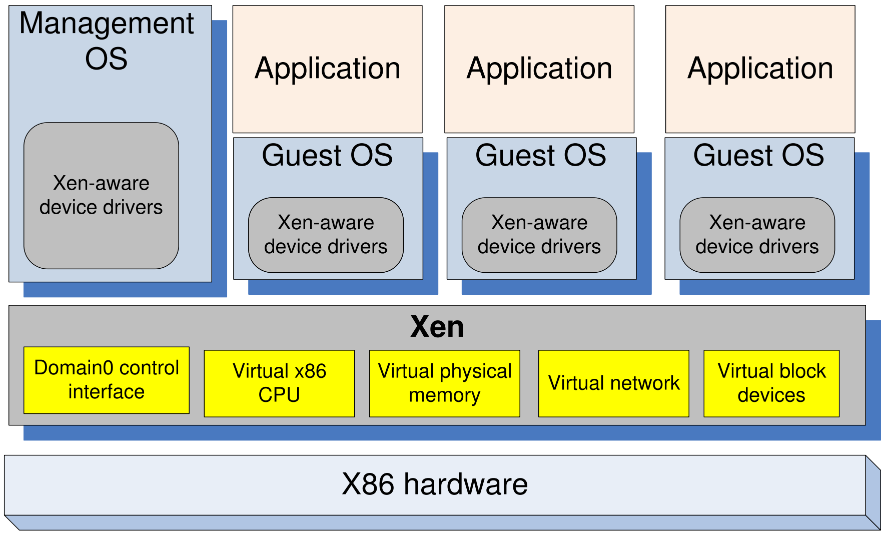


FIGURE 10.4

Xen for the x86 architecture. The management OS dedicated to the execution of Xen control functions and privileged instructions resides in Dom0. Guest operating systems and applications reside in DomU. A guest OS could be either XenoLinux, XenoBSD, or XenoXP in the original Xen implementation [53].

Table 10.2 Paravirtualization strategies for virtual memory management, CPU multiplexing, and I/O devices in the original x86 Xen implementation.

Function	Strategy
Paging	A domain may be allocated discontinuous pages. A guest OS has direct access to page tables and handles pages faults directly for efficiency; page table updates are batched for performance and validated by Xen for safety.
Memory	Memory is statically partitioned between domains to provide strong isolation. *XenoLinux* implements a *balloon driver* to adjust domain memory.
Protection	A guest OS runs at a lower priority level, in ring 1, while Xen runs in ring 0.
Exceptions	A guest OS must register with Xen a description table with the addresses of exception handlers previously validated.
System	To increase efficiency, a guest OS must install a “fast” handler.
Interrupts	A lightweight event system replaces hardware interrupts; synchronous system calls from a domain to Xen use *hypercalls* and notifications are delivered using the asynchronous event system.
Multiplexing	A guest OS may run multiple applications.
I/O devices	Data is transferred using asynchronous I/O rings.
Disk access	Only Dom0 has direct access to IDE and SCSI disks; all other domains access persistent storage through the Virtual Block Device (VBD) abstraction.

Unfortunately, the x86 architecture did not support either the tagging of TLB entries or the software management of the TLB. As a result, the address space switching when the hypervisor activates a different OS requires a complete TLB flush. Flushing the TLB has a negative impact on performance.

The solution adopted was to load Xen in a 64 MB segment at the top of each address space and to delegate the management of hardware page tables to the guest OS with minimal intervention from Xen. The 64 MB region occupied by Xen at the top of every address space is not accessible, or not re-mappable by the guest OS.

When a new address space is created, the guest OS allocates and initializes a page from its own memory, registers it with Xen, and relinquishes control of the write operations to the hypervisor. Thus, a guest OS could only map pages it owns. On the other hand, a guest OS has the ability to batch multiple page update requests to improve performance. A similar strategy is used for segmentation.

The x86 Intel architecture supports four protection rings or privilege levels; virtually all OS kernels run at level 0, the most privileged one, and applications at level 3. In Xen the hypervisor runs at level 0, the guest OS at level 1, and applications at level 3.

Applications make system calls using the so called *hypercalls* processed by Xen; privileged instructions issued by a guest OS are *paravirtualized* and must be validated by Xen. When a guest OS attempts to execute a privileged instruction directly, the instruction fails silently.

Memory is statically partitioned between domains to provide strong isolation. To adjust domain memory XenoLinux implements a *balloon driver* which passes pages between Xen and its own page allocator. For the sake of efficiency page faults are handled directly by the guest OS.

Xen schedules individual domains using the Borrowed Virtual Time scheduling algorithm discussed in Section 9.14. BVT is a work conserving[7] and low-latency wake-up scheduling algorithm. BVT uses a virtual-time warping mechanism to support low-latency dispatch to ensure timely execution whenever needed, for example, for timely delivery of TCP acknowledgments.

A guest OS must register with Xen a *description table* with the addresses of exception handlers for validation. Exception handlers are identical with the native x86 handlers; the only one that does not follow this rule is the page fault handler which uses an extended stack frame to retrieve the faulty address because the privileged register *CR2*, where this address is found, is not available to a guest OS.

Each guest OS can validate and then register a "fast" exception handler executed directly by the processor without the interference of Xen. A lightweight event system replaces hardware interrupts; notifications are delivered using this asynchronous event system. Each guest OS has a timer interface and is aware of "real" and "virtual" time.

XenStore is a Dom0 process supporting a system-wide registry and naming service. It is implemented as a hierarchical key-value storage. A *watch* function of the process informs listeners of changes of the key in the storage they have subscribed to. XenStore communicates with guest VMs via shared memory using Dom0 privileges, rather than grant tables.

Toolstack is another Dom0 component responsible for creating, destroying, and managing the resources and privileges of VMs. To create a new VM a user provides a configuration file describing memory and CPU allocations, as well as device configuration. Then the *Toolstack* parses this file and writes this information in the *XenStore*. Toolstack takes advantage of Dom0 privileges to map guest

[7]A work conserving scheduling algorithm does not allow the processor to be idle when there is work to be done.

memory, to load a kernel and virtual BIOS and to set up initial communication channels with the *XenStore* and with the virtual console when a new VM is created.

Xen defines abstractions for networking and I/O devices. *Split drivers* have a frontend in the DomU and the backend in Dom0; the two communicate via a ring in shared memory. Xen enforces access control for the shared memory and passes synchronization signals. Access Control Lists (ACLs) are stored in the form of *grant tables*, with permissions set by the owner of the memory.

Data for I/O and network operations moves vertically through the system, very efficiently, using a set of I/O rings, see Figure 10.5. A *ring* is a circular queue of descriptors allocated by a domain and accessible within Xen. Descriptors do not contain data, the data buffers are allocated off-band by the guest OS. Memory committed for I/O and network operations is supplied in a manner designed to avoid "crosstalk" and the I/O buffers holding the data are protected by preventing page faults of the corresponding page frames.

Each domain has one or more Virtual Network Interfaces (VNIs) which support the functionality of a network interface card. A VNI is attached to a Virtual Firewall-Router (VFR). Two rings of buffer descriptors, one for packet sending and one for packet receiving, are supported. To transmit a packet, a guest OS enqueues a buffer descriptor to the send ring, then Xen copies the descriptor and checks safety, and finally copies only the packet header, not the payload, and executes the matching rules.

The rules of the form $(< pattern >, < action >)$ require the *action* to be executed if the *pattern* is matched by the information in the packet header. The rules can be added or removed by Dom0; they ensure the demultiplexing of packets based on the destination IP address and port and, at the same time, prevent spoofing of the source IP address. Dom0 is the only one allowed to access directly the physical IDE or SCSI disks. Domains other than Dom0 access persistent storage through a Virtual Block Device (VBD) abstraction created and managed under the control of Dom0.

Xen includes a device emulator, QEMU, to support unmodified commodity operating systems. QEMU is a machine emulator, it runs unmodified OS images and emulates the ISA of the host it runs on. QEMU had several devices already emulated for the x86 architecture, including the chipset, network cards, and display adapters. QEMU emulates a DMA[8] and can map any page of a DomU memory. Each VM has its own instance of QEMU and runs it either as a Dom0 process, or as a process of the VM.

Xen, initially released in 2003, has undergone significant changes in 2005, when Intel released the *VT-x* processors. In 2006 Xen was adopted by Amazon for its EC2 service and in 2008 Xen running on Intel's *VT-d* passed the ACPI S3[9] test. Xen support for Dom0 and DomU was added to the Linux kernel in 2011.

[8]Direct Memory Access (DMA) is a specialized hardware allowing I/O subsystems to access the main memory without the intervention of the CPU. It can also be used for memory-to-memory copying and can offload expensive memory operations, such as scatter-gather operations, from the CPU to the dedicated DMA engine. Intel includes such engines on high-end servers and calles it I/O Acceleration Technology (I/OAT).

[9]Advanced Configuration and Power Interface (ACPI) specification is an open standard for device configuration and power management by the OS. It defines four Global "Gx" states and six Sleep "Sx" states, "S3" is referred to as Standby, Sleep, or Suspend to RAM.

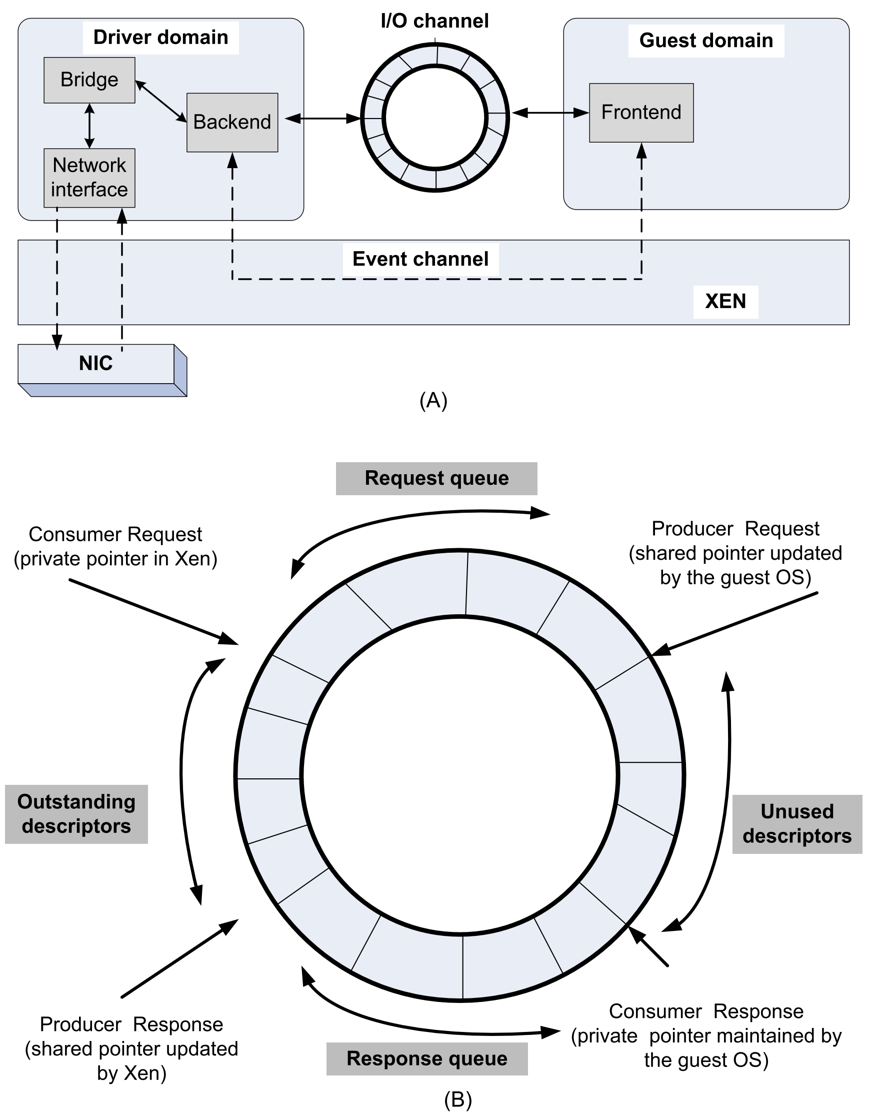

Driver domain
Bridge
Backend
Network interface
I/O channel
Guest domain
Frontend
Event channel
XEN
NIC
(A)
Request queue
Consumer Request (private pointer in Xen)
Producer Request (shared pointer updated by the guest OS)
Outstanding descriptors
Unused descriptors
Producer Response (shared pointer updated by Xen)
Consumer Response (private pointer maintained by the guest OS)
Response queue
(B)

FIGURE 10.5

Xen zero-copy semantics for data transfer using I/O rings. (A) The communication between a guest domain and the driver domain over an I/O and an event channel; NIC is the Network Interface Controller. (B) The circular ring of buffers.

In 2008 the PCI pass-through was incorporated for Xen running on *VT-d* architectures. The PCI[10] pass-through allows a PCI device, be it a disk controller, Network Interface Card (NIC),[11] graphic card, or Universal Serial Bus (USB) to be assigned to a VM. This avoids the overhead of copying and allows setting up of a *Driver Domain* to increase security and system reliability. A guest OS can exploit this facility to access the 3D acceleration capability of a graphics card. The BDF[12] of a device must be known for pass-through.

An analysis of VM performance for I/O-bound applications under Xen is reported in [411]. Two web servers, each running under a different VM, share the same server running Xen. The workload generator sends requests for files of fixed size ranging from 1 KB to 100 KB. When the file size increases from 1 KB to 10 KB, and to 100 KB the performance metrics change as follows: CPU utilization – 97.5%, 70.44%, and 44.4%; throughput – 1 900, 1 104, and 1 112 requests/sec; data rate – 2 018, 11 048, and 11 208 KBps; response time – 1.52, 2.36, and 2.08 msec. From the first group of results we see that for files 10 KB or larger the system is I/O bound. The second set of results shows that the throughput measured in requests/second decreases by less than 50% when the system becomes I/O bound, but the data rate increases by a factor of five over the same range. The response time increases only about 10% when the file size increases by two orders of magnitude.

The paravirtualization strategy in Xen is different from the one adopted by the group at the University of Washington, the creators of the Denali system [522]. Denali was designed to support a number of VMs running network services one or more orders of magnitude larger than Xen. The design of Denali did not target existing application binary interface and does not support some features of potential guest operating systems, for example, it does not support segmentation. Denali does not support application multiplexing, running multiple applications under a guest OS, while Xen does.

Finally, a few words regarding the complexity of porting commodity operating systems to Xen. It is reported that a total of about 3 000 lines, or 1.36% of the Linux code had to be modified. For Windows XP this figure is 4 620 lines, or about 0.04% of the Windows XP code [53].

10.6 OPTIMIZATION OF NETWORK VIRTUALIZATION IN XEN 2.0

A hypervisor introduces a significant network communication overhead. For example, it is reported that the CPU utilization of a VMware Workstation 2.0 system running Linux 2.2.17 was 5 to 6 times higher than that of the native system (Linux 2.2.17) while saturating a 100 Mbps network [471]. This means that the hypervisor executes a much larger number of instructions to saturate the network, 5 to 6 times larger, while handling the same amount of traffic as the native system.

Similar overheads are reported for other hypervisors and in particular for Xen 2.0 [340,341]. To understand the sources of the network overhead we examine the basic network architecture of Xen, see

[10]PCI stands for Peripheral Component Interconnect and describes a computer bus for attaching hardware devices to a computer. The PCI bus supports the functions found on a processor bus, but in a standardized format independent of any particular processor. The OS queries all PCI buses at startup time to identify the devices connected to the system and the memory space, I/O space, interrupt lines, and so on needed by each device present.

[11]A Network Interface Controller is the hardware component connecting a computer to a LAN.

[12]BDF stands for Bus.Device.Function and it is used to describe PCI devices.

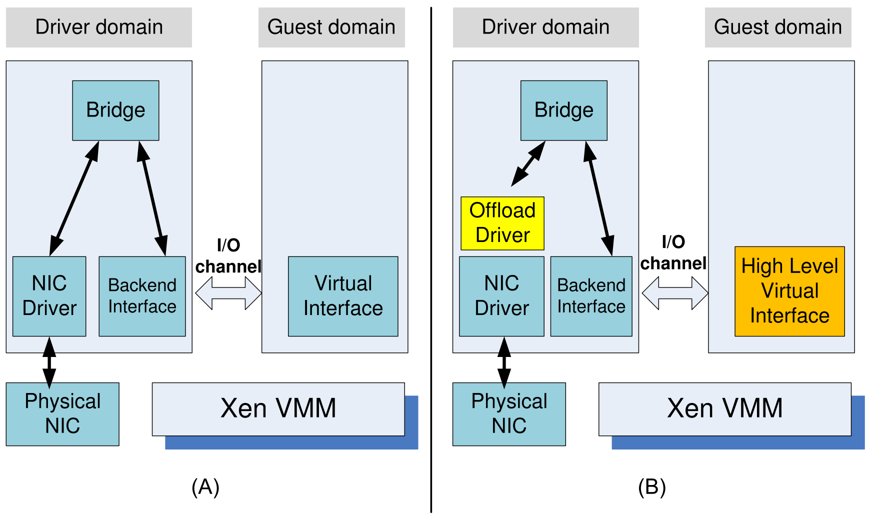


FIGURE 10.6

Xen network architecture: (A) The original architecture; (B) The optimized architecture.

Figure 10.6A. Recall that privileged operations, including I/O, are executed by Dom0 on behalf of a guest OS. In this context we shall refer to Dom0 as the *driver domain*.

The *driver domain* is called in to execute networking operations on behalf of the *guest domain*. It uses the native Linux driver for the NIC (Network Interface Controller), which in turn, communicates with the physical NIC, also called the network adapter. The *guest domain* communicates with the *driver domain* through an I/O channel, see Section 10.5. More precisely, the guest OS in the guest domain uses a virtual interface to send and receive data to/from the backend interface in the driver domain.

A *bridge* uses broadcast communication to identify the MAC address[13] of a destination system. Once identified, this address is added to a table. The bridge uses the link layer protocol to send the packet to the proper MAC address, rather than broadcast it when the next packet for the same destination arrives.

The bridge in the driver domain performs a multiplexing/demultiplexing function. Packets received from the NIC are demultiplexed and sent to the VMs running under the hypervisor. Similarly, packets arriving from multiple VMs have to be multiplexed into a single stream before being transmitted to the network adaptor. In addition to bridging, Xen supports IP routing based on network address translation.

Table 10.3 shows the ultimate effect of the longer processing chain for the Xen hypervisor, as well as, the effect of optimizations [341]. The receiving and sending rates from a guest domain are roughly 30% and 20%, respectively, of the corresponding rates of a native Linux application. Packet

[13]A Media Access Control (MAC) address is a unique identifier permanently assigned to a network interface by the manufacturer.

Table 10.3 A comparison of send and receive data rates for a native Linux system, the Xen driver domain, an original Xen guest domain, and an optimized Xen guest domain.

System	Receive data rate (Mbps)	Send data rate (Mbps)
Linux	2 508	3 760
Xen driver	1 728	3 760
Xen guest	820	750
optimized Xen guest	970	3 310

multiplexing/demultiplexing accounts for about 40% and 30% of the communication overhead for the incoming traffic and for the outgoing traffic, respectively.

The Xen network optimization discussed in [341] covers optimization of: (i) the virtual interface; (ii) the I/O channel; and (iii) the virtual memory. The effects of these optimizations are significant for the send data rate from the optimized Xen guest domain, an increase from 750 to 3 310 Mbps and rather modest for the receive data rate, 970 versus 820 Mbps.

We next examine each optimization area and start with the virtual interface. There is a trade-off between generality and the flexibility, on one hand, and the performance on the other hand. The original virtual network interface provides the guest domain with a simple low-level network interface abstraction supporting sending and receiving primitives.

The design supports a wide range of physical devices attached to the driver domain but does not take advantage of the capabilities of some physical NICs such as checksum offload, e.g., TSO,[14] and scatter/gather DMA support. These features are supported by the High Level Virtual Interface of the optimized system, Figure 10.6B.

The next target of the optimization effort is the communication between the guest domain and the driver domain. Rather than copying a data buffer holding a packet, each packet is allocated space in a new page and then the physical page containing the packet is re-mapped onto the target domain; for example, when a packet is received, the physical page is re-mapped to the guest domain. The optimization is based on the observation that there is no need to re-map the entire packet.

For example, when sending a packet, the network bridge needs only to know the MAC header of the packet. As a result of this, the optimized implementation is based on an "out-of-band" channel used by the guest domain to provide the bridge with the packet MAC header. This strategy contributed to a better than four times increase in the send data rate compared with the non-optimized version.

The third optimization covers virtual memory. The virtual memory in Xen 2.0 takes advantage of the *superpage* and *global page mapping* hardware features available on Pentium and Pentium Pro processors. A superpage increases the granularity of the dynamic address translation; a superpage entry covers 1 024 pages of physical memory and the address translation mechanism maps a set of contiguous pages to a set of contiguous physical pages. This helps reduce the number of TLB misses.

All pages of a superpage belong to the same guest OS. When new processes are created, the guest OS must allocate read-only pages for the page tables of the address spaces running under the guest

[14]TSO stands for TCP segmentation offload. This option enables the network adapter to compute the TCP checksum on *transmit* and *receive*, and to save the host CPU the overhead for computing the checksum; large packets have larger savings.

OS. This forces the system to use traditional page mapping, rather than the superpage mapping. The optimized version on network virtualization uses a special memory allocator to avoid this problem.

10.7 KERNEL-BASED VIRTUAL MACHINE

Kernel-based Virtual Machine (KVM) [286] is a virtualization infrastructure of the Linux kernel (see http://www.linux-kvm.org). KVM, developed in 2006, was released as part of the 2.6.20 Linux kernel in February 2007.

Running multiple guest operating systems on the x86 architecture was quite difficult before the introduction of VMX and SVM extensions of Intel architecture. These extensions allow the hypervisor to run within the privileged ring 1 and allow KVM to provides VMs with an execution environment nearly identical to the physical hardware. KVM executes guest VM's instructions directly on the host. Each guest OS is isolated, it runs in a different instance of the execution environment.

KVM does not run as a normal program inside Linux but relies on the Linux kernel infrastructure to run. Its organization is shown in Figure 10.7. As opposed to Xen, which runs on the physical hardware, KVM runs inside Linux as a driver handling the new virtualization instructions exposed by the hardware. A major advantage of this model is that KVM inherits all the new features of Linux in scheduling, memory management, power management, and so on. KVM has several components:

1. A generic host kernel module exposing the architecture-independent functionality.
2. An architecture-specific kernel module for the host system.
3. A user-space emulation of the VM hardware that the guest OS runs on.
4. A guest OS performance optimization addition.

kvm-userspace is a fork of the QEMU project; it short-circuits the emulation code to only allow x86-on-x86 and use the KVM API for running the guest OS on the host CPU. When the guest OS performs a privileged operation the CPU exits and KVM takes over. If KVM itself can service the request it then gives control back to the guest.

KVM exposes the */dev/kvm* interface allowing a user-space host to: (a) setup the guest VM address space; (b) feed the guest simulated I/O; and (c) map the video display of the host. The host supplies a firmware image used by the guest to bootstrap into the host OS.

10.8 NESTED VIRTUALIZATION

Nested virtualization describes a system organization when a *guest hypervisor* runs inside a VM which is itself running under a *host hypervisor*. Figure 10.8A illustrates an instance of nested virtualization where a KVM is the host hypervisor supporting resource sharing among three VMs. Two of the three VMs run guest hypervisors Xen and VMware's ESXi and the third VM runs Windows. There are two VM running under guest hypervisors, one runs Linux under Xen and another runs Windows under ESXi.

Nested virtualization is useful for experimenting with server setup or testing configurations. Nested virtualization allows IaaS users to run their own hypervisor as a VM. Nested virtualization can be also used for live migration of hypervisors together with their guests VM for load balancing, for hypervisor-level protection, and for supporting other security mechanisms. Another use of nested virtualization is to experiment with cloud interoperability alternatives.

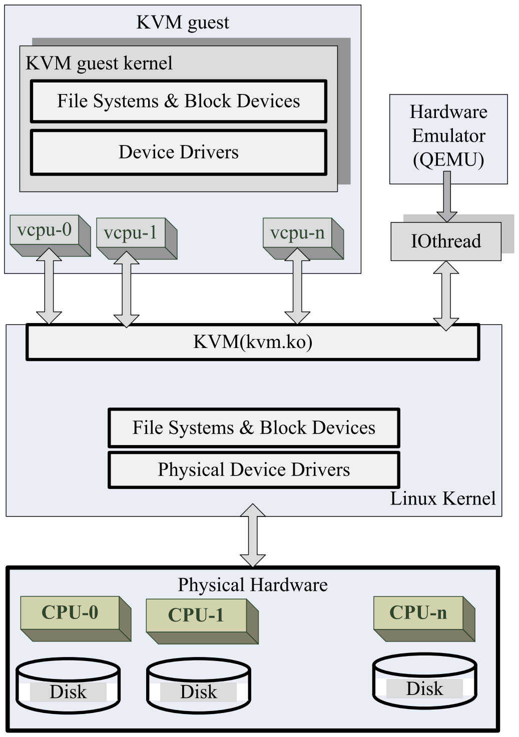


FIGURE 10.7

KVM organization. KVM runs inside Linus as a driver handling the new virtualization instructions exposed by hardware. The *IOthread* generates requests on the guest's behalf to the host; it also handles events.

x86 virtualization is based on the *trap and emulate* model. This model requires that every sensitive instruction executed by either a guest hypervisor or OS to be handled by the most privileged hypervisor. Nested virtualization incurres a substantial performance price, unless the switching between the levels of the virtualization stack is optimized. It is thus, not surprising that nested virtualization is not supported by many hypervisors and not all operating systems can nest successfully with all hypervisors.

Nested virtualization is supported differently by Intel and AMD processors. Consider for example the Intel version discussed in [128] and illustrated in Figure 10.8B. In this example KVM runs at level L_0 and controls the allocation of all resources. Xen runs at level L_1 and KVM uses $VMCS_{01}$ for the VM running Xen.

When Xen executes a *vmlaunch* operation to start a new VM (see Section 10.4), a new VM control structure, $VMCS_{12}$ is created. Then *vmlaunch* traps to L_0 and L_0 merges $VMCS_{01}$ with $VMCS_{12}$ and creates $VMCS_{02}$ to run Linux at level L_2. When an application running under Linux at level L_2 makes a system call, or when Linux itself executes a privileged instruction, L_2 traps and KVM decides whether to handle the trap itself or to forward it to Xen at level L_1 and, eventually, Xen resumes execution.

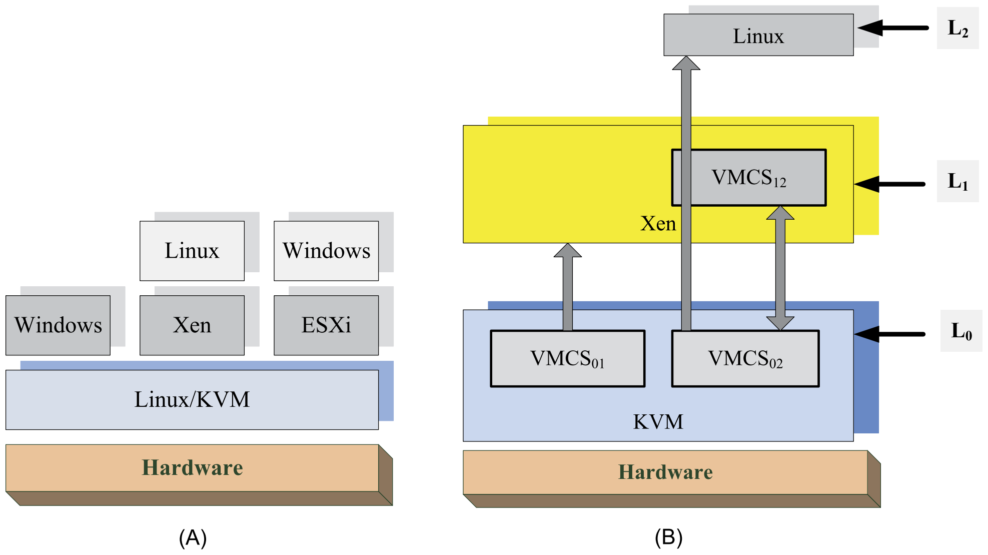


FIGURE 10.8

Nested virtualization [128]. (A) KVM allows three VM to run concurrently. Two VMs run hypervisors Xen and ESXi and the third runs Windows. A VM runs Linux under Xen and another VM runs Windows under ESXi. (B) Intel-supported nested virtualization. KVM runs at level L_0, Xen runs at level L_1 and KVM uses VMCS$_{01}$ for the VM running Xen.

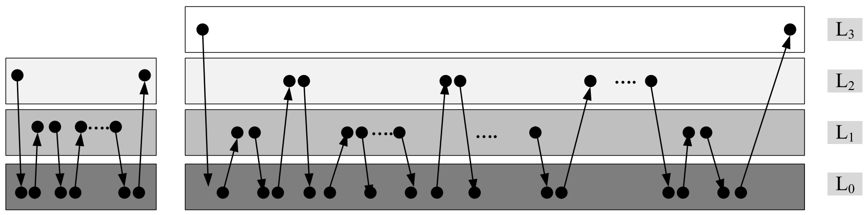


FIGURE 10.9

Nested virtualization with single-level hardware virtualization support. A trap is handled by the L_0 trap handler regardless of the hypervisor where a trap occurs. Nested traps for: (Left) Two-level, L_0, L_1, and L_2 nested hypervisors; and (Right) Three-level, L_0, L_1, L_2, and L_3 nested hypervisors.

Nested virtualization is limited by the hardware support. When the hardware supports *multi-level nested virtualization*, each hypervisor handles all traps caused by sensitive instructions of guest hypervisors running directly above of it. Multi-level nested virtualization is supported by the IBM System z architecture [383]. Intel and AMD processors support only *single-level nested virtualization*. This implies that the host hypervisor, the one running directly above the hardware and managing all system resources, handles all trapped instructions as shown in Figure 10.9.

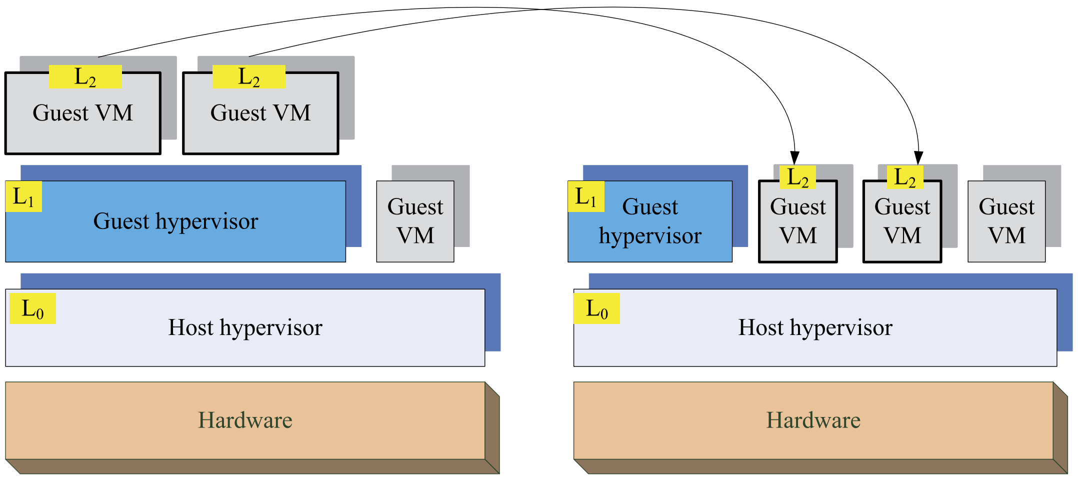


FIGURE 10.10

Multiple virtualization levels on the left are multiplexed into the singe hardware virtualization level on the right, as described in [61]. A VMX instruction used by a guest hypervisor running in guest mode at level L_i is trapped and translated by the host hypervisor at level L_0 running in kernel mode into one that can be used to a VM at level L_{i+1}.

An in-depth discussion of the intricacies of nested virtualization on x86 architecture can be found in a paper describing the Turtle project from IBM Israel [61]. A guest hypervisor cannot use the hardware virtualization support because the x86 provides only a single-level hardware virtualization support. The aim of the project was to show that a 6–8% overhead is feasible for "unmodified binary-only hypervisors executing non-trivial workloads."

Recall that VMX instructions can only be successfully executed in kernel mode. A guest hypervisors at level L_i operates in guest mode and whenever it executes a VMX instruction to launch a level L_{i+1} guest, the instruction is trapped and handled at level L_0. Trapping execution exceptions enables the host hypervisor at level L_0 running in kernel mode to emulate VMX instruction executed by guest hypervisors at level L_i. This mechanism supports a critical idea for increasing efficiency of nested virtualization, multiplexing multiple hypervisors, as shown in Figure 10.10.

As long as the host hypervisor at level L_0 emulates faithfully the VMX instruction set, a guest hypervisor at level L_1 cannot distinguish if it is running directly on the hardware or not. It follows that the guest hypervisor at level L_1 can lunch VMs using the standard mechanisms. The guest hypervisor at level L_1 does not run at the highest privileged level and the action of starting a VM is trapped and handled by the trap handler at level L_0. The specification of the new VM is then translated by the host hypervisor at level L_0 running in kernel mode into one that can be used to run L_2 directly on the hardware. This translation includes converting L_1 physical addresses to the physical address space of L_0.

The guest hypervisor can use the same technique to give another guest hypervisor at level L_2 the same illusion, namely that it is running directly on the hardware. The process can be extended,

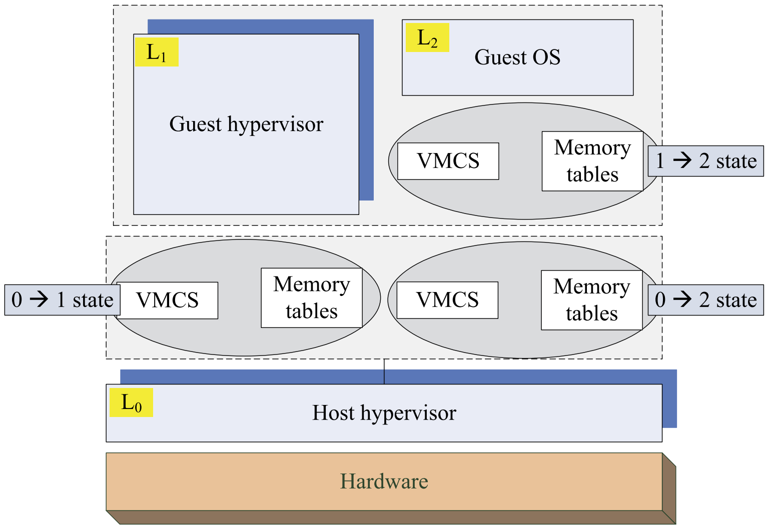


FIGURE 10.11

VMX extension for nested virtualization as described in [61].

a hypervisor at level L_i giving the illusion that the one at level L_{i+1} is running directly on the hardware.

The processor runs L_1 and L_2 maintained by L_0 using $VMCS_{0\rightarrow1}$ and $VMCS_{0\rightarrow2}$ environment specification, respectively. L_1 creates $VMCS_{1\rightarrow2}$ within its own virtualized environment and the processor uses it to emulate VMX for L_1, as illustrated by Figure 10.11. Switching from one level to another is emulated. For example, when an *VMExit* occurs while L_2 is running there are two possible paths:

- When an external interrupt, a non-maskable interrupt, or any trappable event specified in $VMCS_{0\rightarrow2}$ that was not specified in $VMCS_{1\rightarrow2}$ occurs, then L_0 handles the event and then L_2 execution is resumed.
- Trappable events specified in $VMCS_{1\rightarrow2}$ are handled by L_1. The host hypervisor at L_0 forwards the event to L_1 by copying $VMCS_{0\rightarrow2}$ fields updated by the processor to $VMCS_{1\rightarrow2}$ and then resuming L_1. This makes the L_1 hypervisor believe there was a VMExit directly from L_2 to L_1, handles the event and then resumes L_2 by executing *VMLAUNCH* or *VMRESUME*, emulated by L_0.

Another complication of nested virtualization is that the MMU must be virtualized to allow a guest hypervisor to translate guest virtual addresses to guest physical addresses. A multi-dimensional paging for multiplexing the three needed translation tables onto the two available in hardware is also described in [61].

10.9 A TRUSTED KERNEL-BASED VIRTUAL MACHINE FOR ARMV8

Advanced RISC Machine (ARM) processors are widely used in mobile devices such as smart phones, tables, and laptops. ARM processors are also used in embedded systems connected to the IoT. Such systems require an increased level of security therefore, it is not surprising that the latest generation of the ubiquitous ARM processors support the Trusted Execution Environment (TEE).

TEE functions are summarized at http://www.globalplatform.org/ as: "TEE's ability to offer isolated safe execution of authorized security software, known as trusted applications, enables it to provide end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity and data access rights." Trusted applications, running in TEE, their assets and data are isolated from the Rich Execution Environment where standard operating systems such as Linux run. TEE consists of several components:

1. A common abstraction layer, the Trusted Core Framework providing OS functions, such as memory management, entry points for trusted applications, panic and cancelation handling, and trusted application properties access.
2. Inter-process communication used by rich execution environment applications to request services from TEE.
3. API for accessing services such as Trusted Storage for Data and Keys, TEE Cryptographic Operations, Time, and TEE Arithmetica.

AArch64, the 64 bit ARM architecture, is compatible with AArch32, the 32 bit ARM architecture. The members of the AArch64 family including ARMv8 Cortex-Axx ($xx = \{35, 53, 57, 72, 73\}$) processors share a number of features:

- Support a new instruction set, A64, with the same instruction semantics as AArch32, but with fewer conditional instructions. A64 includes major functional enhancements:
  1. Has thirty two 128-bit wide registers.
  2. Advanced SIMD supports double-precision floating-point execution.
  3. Advanced SIMD supports full IEEE 754[15] execution.
- Provides instruction-level support for cryptography. Has two encode and two decode instructions for AES and SHA-1 and SHA-256 support.
- Has thirty one general purpose registers accessible at all times.
- Provides revised exception handling in the AArch64 state.
- Supports virtualization.
- Supports Trust Zone and Global Trust TEE.

The *ARM Trust Zone* (ATZ) splits an ARM-based system into the Secure World, a trusted subsystem responsible for the boot and the configuration of the entire system, and the Non-Secure World (NSW) intended for hosting operating systems such as Linux and Android, as well as user applications. CPU has banked registers for each World.

[15]IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines arithmetic formats, interchange formats, rounding rules, operations, and exception handling for floating point numbers, see "IEEE Standard for Floating-Point Arithmetic" IEEE Computer Society (August 29, 2008), doi:10.1109/IEEESTD.2008.4610935.

Security-specific configurations can only be performed in the Secure World mode while access to AMBA peripherals such as fingerprint readers, cryptographic engines, and others can be restricted to the Secure World. A secure context switch procedure routs interrupts, either to the Secure or to the Non-Secure World, depending upon the configuration and allows the two Worlds to communicate with one another.

ATZ is enabled by a set of hardware security extensions including:

1. A CPU with ARM Security Extensions (SE).
2. A compliant Memory Management Unit (MMU).
3. An AMBA system bus.[16]
4. Interrupt and cache controllers.

T-KVM, a KVM-based trusted hypervisor for ARMv8, combines Trust Zone with GlobalPlatform TEE and SELinux [391]. T-KVM implements: (a) a trusted boot; (b) support for Trusted Computing inside a Virtual Machine; (c) a zero copy shared memory mechanism for data sharing between the two Trust Zone Worlds and between the VM and the host; (d) a secure, ideally real-time, reliable, and error-free OS running in the Secure World.

The challenge of a secure boot is to eliminate vulnerabilities while the security mechanisms are not yet in place. The solution implemented in T-KVM is a four-stage boot process. A small program stored in the on-chip ROM, along with the public key needed for the attestation of the second stage loader, is activated in the first stage.

The second stage loads the microkernel in the Secure World zone and activates it. The third stage checks the integrity of the Linux kernel, a Non-Secure World binary, and of its loader and, finally, the fourth stage runs it. The failure of any check in this chain of events brings the system to a secure state stop. T-KVM boot sequence is shown in Figure 10.12A.

The main challenge for supporting Trusted Computing inside a VM is the virtualization of the TEE APIs. To allow the TEE Client API to execute directly in the Guest OS, a specific QEMU device implements the TEE control plane and sets up its data plane, see Figure 10.12B. Requests for service such as initialization/close session, command invocation, and notification of response are sent to the TEE Device which delivers them either to the trusted applications or to the client applications running on the guest OS. The data plane uses the shared memory. The TEE device notifies its driver upon receiving a response notification from the Trust Zone Secure World (TZSW) and the driver forwards the information to the Guest-Client application.

The zero-copy shared memory is based on the fact that trusted applications can read/write VMs shared memory because TZSW can access the entire NSW workspace. The TEE Device control plane extends the T-KVM shared memory mechanism, allowing it to send the shared memory address to the Secure World applications.

10.10 PARAVIRTUALIZATION OF ITANIUM ARCHITECTURE

We now analyze some of the findings of a Xen project at HP Laboratories [320]. This analysis will help us better understand the impact of the computer architecture on the ability to virtualize efficiently

[16]The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for connection and management of a large numbers of controllers and peripherals.

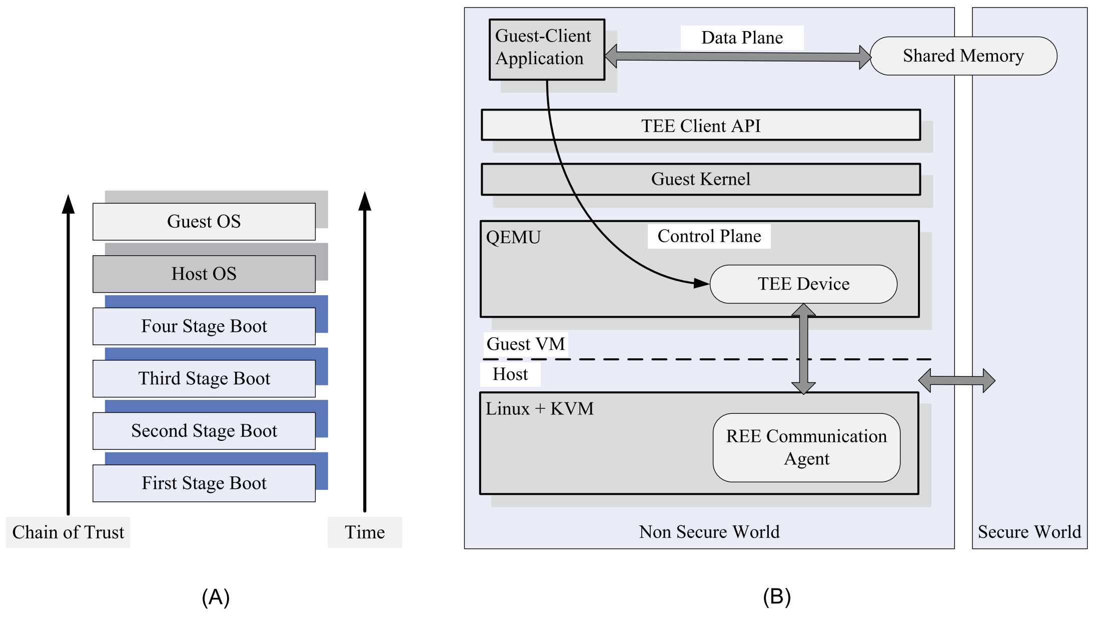


FIGURE 10.12

T-KVM. (A) The boot sequence; a trusted application runs the Non-Secure loader in stage three and in stage four a Non-Secure OS is booted. TEE attestation and SELinux permissions are enforced while the host OS is running. Guest Client applications (CSs) use the Secure TEE Services while the Guest OS is running. (B) Communication between the Non-Secure and the Secure Worlds as described in [391].

a given computer architecture. The goal of the project was to create a hypervisor for the Itanium family of IA64 Intel processors.

Itanium is a processor developed jointly by HP and Intel based on a new architecture, the Explicitly Parallel Instruction Computing. This architecture allows the processor to execute multiple instructions in each clock cycle and implements a form of Very Long Instruction Word (VLIW) architecture. In VLIW a single instruction word contains multiple instructions, see http://www.dig64.org/about/Itanium2_white_paper_public.pdf.

The design mandated that the hypervisor should be capable of supporting the execution of multiple operating systems in isolated protection domains with security and privacy enforced by the hardware. The hypervisor was also expected to support optimal server utilization and allow comprehensive measurement and monitoring for detailed performance analysis.

Virtualization of the IA64 Architecture. The discussion in Section 10.2 shows that to be fully virtualizable the ISA of a processor must conform to a set of requirements. Unfortunately, the IA64 architecture does not meet these requirements and that made the Xen project more challenging.

We first review the features of the Itanium processor important for virtualization and start with the observation that the hardware supports four *privilege rings*, PL0, PL1, PL2, and PL3. Privileged instructions can only be executed by the kernel running at level PL0, while applications run at level PL3 and can only execute non-privileged instructions; PL1 and PL2 rings are generally not used.

The hypervisor uses *ring compression* and runs itself at PL0 and PL1 while forcing a guest OS to run at PL2. A first problem called *privilege leaking* is that several non-privileged instructions allow an application to determine the Current Privilege Level (CPL). As a result, a guest OS may not accept to boot or run, or may itself attempt to make use of all four privilege rings.

Itanium was selected because of its multiple functional units and multi-threading support. The Itanium processor has 30 functional units: six general-purpose ALUs, two integer units, one shift unit, four data cache units, six multimedia units, two parallel shift units, one parallel multiply, one population count, three branch units, two 82-bit floating-point multiply-accumulate units, and two SIMD floating-point multiply-accumulate units. A 128-bit instruction word contains three instructions; the fetch mechanism can read up to two instruction words per clock from the L1 cache into the pipeline. Each unit can execute a particular subset of the instruction set.

The hardware supports 64-bit addressing. The processor has thirty two 64-bit general-purpose registers numbered R0 to R31 and ninety six automatically renumbered registers, R32 through R127, used by procedure calls. When a procedure is entered, the *alloc* instruction specifies the registers the procedure could access by setting the bits of a 7-bit field that controls the register usage. An illegal *read* operation from such a register out of range returns a zero value while an illegal *write* operation to it is trapped as an illegal instruction.

The Itanium processor supports isolation of the address spaces of different processes with eight privileged *region* registers; the *processor abstraction layer* firmware allows the caller to set the values in the region register. The hypervisor intercepts the privileged instruction issued by the guest OS to its processor abstraction layer and partitions the set of address spaces among the guests OS to ensure isolation. Each guest is limited to 2^{18} address spaces.

The hardware has an *IVA register* to maintain the address of the *interruption vector table*; the entries in this table control both the interrupt delivery and the interrupt state collection. Different types of interrupts activate the interrupt handlers pointed at from this table, provided that the particular interrupt is not disabled. Each guest OS maintains its own version of this vector table and has its own IVA register; the hypervisor uses the guest OS IVA register to give control to the guest interrupt handler when an interrupt occurs.

CPU virtualization. When a guest OS attempts to execute a privileged instruction the hypervisor traps and emulates the instruction. For example, when the guest OS uses the *rsm psr.i* instruction to turn off delivery of a certain type of interrupt, the hypervisor does not disable the interrupt, but records the fact that interrupts of that type should not be delivered to the guest OS and, in this case, the interrupt should be masked.

There is a slight complication because the Itanium does not have an Instruction Register (IR) and the hypervisor has to use state information to determine if an instruction is privileged. Another complication is caused by the *register stack engine* which operates concurrently with the processor and may attempt to access memory (load or store) and generate a page fault. Normally, the problem is solved by setting up a bit indicating that the fault is due to the register stack engine and, at the same time, the engine operations are disabled. The handling of this problem by the hypervisor is more intricate.

A number of *privileged-sensitive* instructions behave differently function of the privilege level. The hypervisor replaces each one of them with a privileged instruction during the dynamic transformation of the instruction stream. Among the instructions in this category are:

- *cover*, saves stack information into a privileged register; the hypervisor replaces it with a *break.b* instruction.
- *thash* and *ttag*, access data from privileged virtual memory control structures and have two registers as arguments. The hypervisor takes advantage of the fact that an illegal read returns a zero and an illegal write to a register in the range 32 to 127 is trapped and translates these instructions as: $thash\ Rx = Ry \rightarrow tpa\ Rx = R(y + 64)$ and $ttag\ Rx = Ry \rightarrow tak\ Rx = R(y + 64)$, $0 \leq y \leq 64$.
- *PSR.sp*, controls access to performance data from performance data registers by setting up a bit in the *Processor Status Register*.

Memory virtualization. The virtualization is guided by the realization that a hypervisor should not be involved in most of memory read and write operations to prevent a significant degradation of the performance. At the same time, the hypervisor should exercise a tight control and prevent a guest OS from acting maliciously. The Xen hypervisor does not allow a guest OS to access the memory directly, it inserts an additional layer of indirection called *metaphysical addressing* between virtual and real addressing.

A guest OS is placed in the metaphysical addressing mode. If the address is virtual, then the hypervisor first checks if the guest OS is allowed to access that address and, if so, the hypervisor provides the regular address translation. The hypervisor is not involved when the address is physical. The hardware distinguishes between virtual and real addresses using bits in the Processor Status Register.

10.11 A PERFORMANCE COMPARISON OF VIRTUAL MACHINES

There is well-documented evidence that hypervisors negatively affect the performance of applications [53,340,341]. The topic of this section is a quantitative analysis of VM performance. The performance of two virtualization techniques is compared with the performance of a plain vanilla Linux. The two VM systems, Xen and OpenVZ are based on paravirtualization and full virtualization, respectively [387].

OpenVZ, a system based on OS-level virtualization, uses a single patched Linux kernel. The guest operating systems in different containers may be different software distributions, but must use the same Linux kernel version that the host uses. An OpenVZ container emulates a separate physical server, it has its own files, users, process tree, IP address, shared memory, semaphores, and messages. Each container can have its own disk quotas.

OpenVZ's lack of virtualization flexibility is compensated by a lower overhead. OpenVZ memory allocation is more flexible than in hypervisors based on paravirtualization. The memory not used in one virtual environment can be used by other virtual environments. The system uses a common file system; each virtual environment is a directory of files isolated using *chroot*. To start a new VM one needs to copy the files from one directory to another, create a *config* file for the VM, and launch the VM.

OpenVZ has a two level scheduler: at the first level, a fair-share scheduler allocates CPU time slices to containers based on *cpuunits* values. The second level scheduler is a standard Linux scheduler deciding what process to run in that container. The I/O scheduler is also two-level; each container has an I/O priority and the scheduler distributes the available I/O bandwidth according to priorities.

The discussion in [387] is focused on user's perspective and the performance metrics analyzed are throughput and response time. The general question is whether consolidation of the applications and the servers is a good strategy for cloud computing. The specific questions examined are:

- How does performance scale up with the load?
- What is the impact on performance of a mixture of applications?
- What are the implications of the load assignment on individual servers?

There is substantial experimental evidence that the load placed on system resources by a single application varies significantly in time. A time series displaying CPU consumption of a single application clearly illustrates this fact and justifies CPU multiplexing among threads and/or processes. The concept of *application and server consolidation* is an extension of the idea of creating an aggregate load consisting of several applications and aggregating a set of servers to accommodate this load. The peak resource requirements of individual applications are unlikely to be synchronized therefore, the aggregate average resource utilization is expected to increase.

The application used for comparison in [387] is a two-tier system consisting of an Apache web server and a MySQL database server. A web application client starts a session as a user browses through different items in the database, requests information about individual items, and buys or sells items. Each session requires the creation of a new thread; thus, an increased load means an increased number of threads. To understand the potential discrepancies in performance among the three systems, a performance monitoring tool reports the counters that allow the estimation of: (i) the CPU time used by a binary; (ii) the number of L2-cache misses; and (iii) the number of instructions executed by a binary.

The experimental setup for three different experiments are shown in Figure 10.13. In the first group of experiments each one of the two tiers of the application, the web server and the database, run on a single server for the Linux, the OpenVZ, and the Xen systems.

When the workload increases from 500 to 800 threads, the throughput increases linearly with the workload. The response time increases only slightly for the base system and for the OpenVZ system, while it increases 600% for the Xen system. For 800 threads the response time of the Xen system is four times larger than the one for OpenVZ. The CPU consumption grows linearly with the load in all three systems. The DB consumption represents only 1–4% of the CPU consumption.

For a given workload the web-tier CPU consumption for the OpenVZ system is close to the base Linux system and it is about half of that for the Xen system. The performance analysis tool shows that the OpenVZ execution has two times more L2-cache misses than the base system, while Xen Dom0 has 2.5 times more and Xen application domain has 9 times more cache misses.

Recall that the base system and the OpenVZ run a Linux OS and the sources of cache misses can be compared directly, while Xen runs a modified Linux kernel. The procedures $hypervisor\_callback$, invoked when an event occurs, and $evtchn\_do\_upcall$, invoked to process an event, are responsible for 32% and 44%, respectively, of the L2-cache misses for the Xen-based system. The percentage of the instructions invoked by these two procedures are 40% and 8%, respectively.

Most of the L2-cache misses in OpenVZ and the base system occur in:

1. A procedure called $do\_anonimous\_pages$ used to allocate pages for a particular application with the percentage of cache misses 32% and 25%, respectively.
2. Two procedures, $\_copy\_to\_user\_ll$ and $\_copy\_from\_user\_ll$ used to copy data from user to system buffers and back. The percentage of cache misses are $(12 + 7)\%$ and $(10 + 1)\%$, respectively;

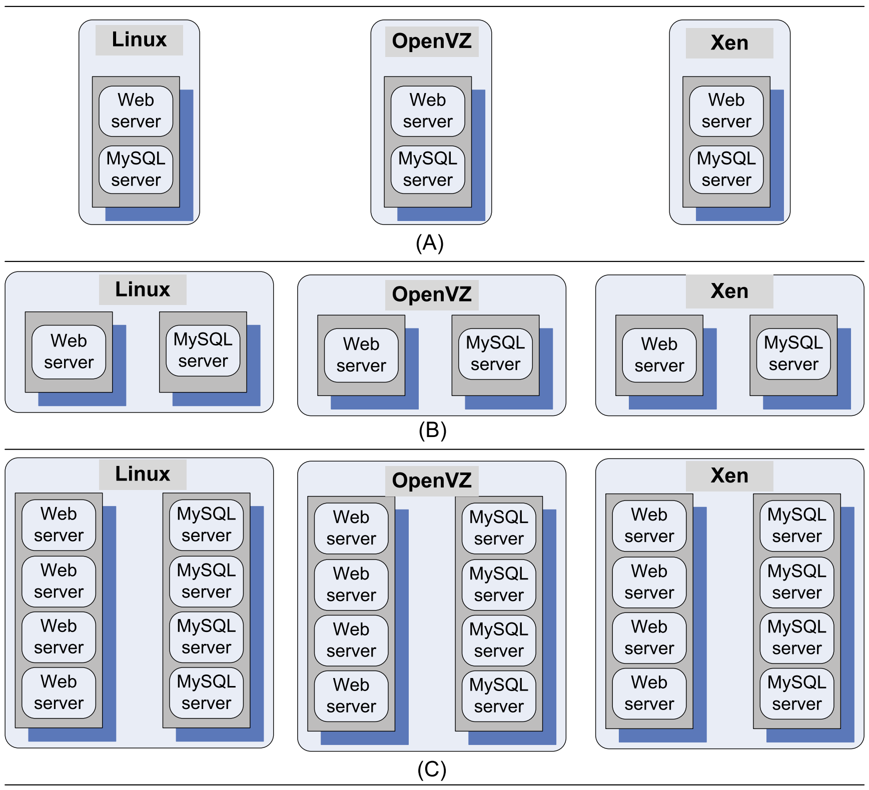


FIGURE 10.13

The setup for the performance comparison of a native Linux system with OpenVZ and Xen. The applications are a web server and a MySQL database server. (A) The first experiment - the web and the DB, share a single server; (B) The second experiment - the web and the DB run on two different servers; (C) The third experiment - the web and the DB run on two different servers and each has four instances.

the first figure refers to the copying from user to system buffers and the second to copying from system buffers to the user space.

In the second group of experiments each one of the three systems uses two servers, one for the web and the other for the DB application. When the load increases from 500 to 800 threads the throughput increases linearly with the workload. The response time of Xen increases only 114%, compared with 600% reported for the first experiments.

For the web application the CPU time of the base system, OpenVZ, and Xen Dom0 and DomU are similar. For the DB application the CPU time of OpenVZ is twice as large as that of the base system, while Dom0 and DomU require CPU times 1.1 and 2.5 times larger than the base system.

For web application the L2-cache misses relative to the base system is the same for OpenVZ and 1.5 larger for Dom0 of Xen and 3.5 times larger for the DomU. For DB application the L2-cache misses relative to the base system are 2 times larger for the OpenVZ and 3.5 larger for Dom0 of Xen and 7 times larger for DomU.

In the third group of experiments each one of the three systems uses two servers, one for the web and the other for the DB application, but runs four instances of the web and the DB application on the two servers. The throughput increases linearly with the workload for the range used in the previous two experiments, from 500 to 800 threads. The response time remains relatively constant for OpenVZ and increases 5 times for Xen.

The main conclusion drawn from these experiments is that the virtualization overhead of Xen is considerably higher than that of OpenVZ and that this is due primarily to L2-cache misses. Xen performance degradation is noticeable when the workload increases. Another important conclusion is that hosting multiple tiers of the same application on the same server is not optimal.

10.12 OPEN-SOURCE SOFTWARE PLATFORMS FOR PRIVATE CLOUDS

Private clouds provide a cost effective alternative for very large organizations. A private cloud has essentially the same structural components as a commercial one: the servers, the network, hypervisors running on individual systems, an archive containing disk images of VMs, a front-end for communication with the user, and a cloud control infrastructure. Open-source cloud computing platforms such as Eucalyptus [373], OpenNebula, and Nimbus can be used as a control infrastructure for a private cloud.

Schematically, a cloud infrastructure carries out the following steps to run an application:

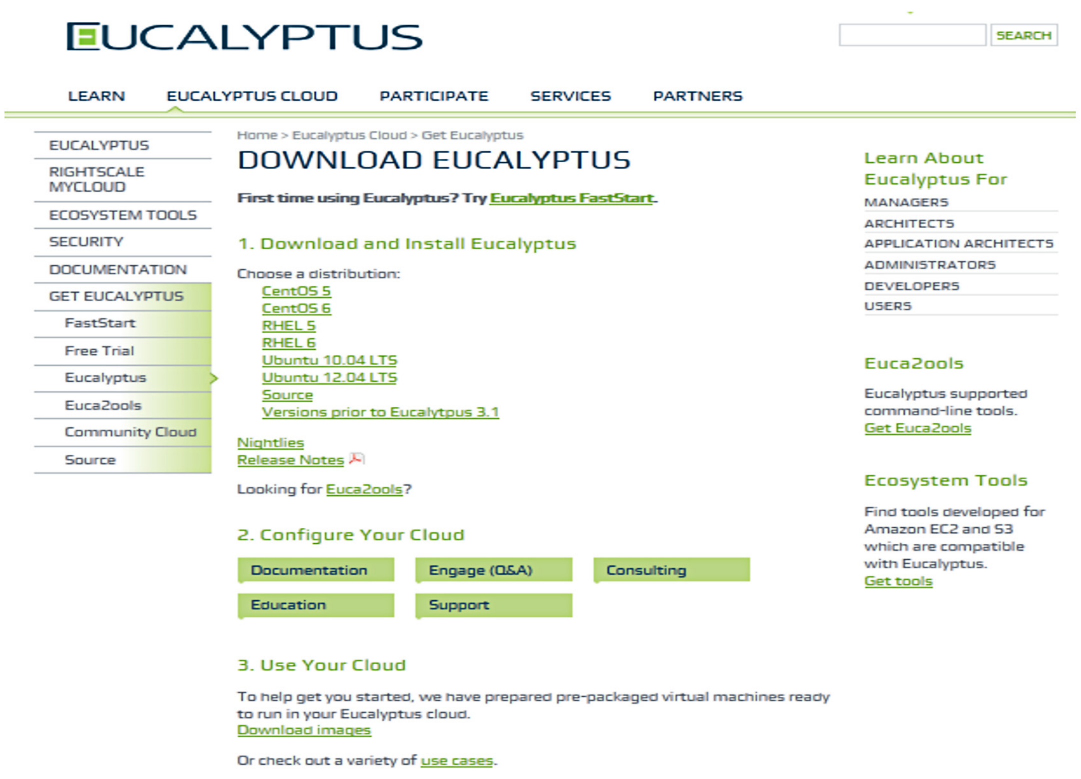
- Retrieves the user input from the front-end.
- Retrieves the disk image of a VM (Virtual Machine) from a repository.
- Locates a system and requests the hypervisor running on that system to set up a VM.
- Invokes the DHCP (see Section 5.1) and the IP bridging software to set up a MAC and IP address for the VM.

We discuss briefly the three open-source software systems, Eucalyptus, OpenNebula, and Nimbus.

Eucalyptus (http://www.eucalyptus.com/) can be regarded as an open-source counterpart of Amazon's EC2, see Figure 10.14. The system supports several operating systems including: CentOS 5 and 6, RHEL 5 and 6, Ubuntu 10.04 LTS and 12.04 LTS.

The components of the system are:

- Virtual Machine. Runs under several hypervisors including Xen, KVM, and VMware.
- Node Controller. Runs on every server/node designated to host a VM and controls the activities of the node. Reports to a cluster controller.
- Cluster Controller. Controls a number of servers. Interacts with the node controller on each server to schedule requests on that node. Cluster controllers are managed by cloud controller.
- Cloud Controller. Provides the cloud access to end-users, developers, and administrators. It is accessible through command line tools compatible with EC2 and through a web-based Dashboard. Manages cloud resources, makes high-level scheduling decisions and interacts with cluster controllers.



Home > Eucalyptus Cloud > Get Eucalyptus

DOWNLOAD EUCALYPTUS

First time using Eucalyptus? Try Eucalyptus FastStart.

1. Download and Install Eucalyptus

Choose a distribution:
CentOS 5
CentOS 6
RHEL 5
RHEL 6
Ubuntu 10.04 LTS
Ubuntu 12.04 LTS
Source
Versions prior to Eucalytpus 3.1

Nightlies
Release Notes

Looking for Euca2ools?

2. Configure Your Cloud

3. Use Your Cloud

To help get you started, we have prepared pre-packaged virtual machines ready to run in your Eucalyptus cloud.
Download images

Or check out a variety of use cases.

Learn About Eucalyptus For
MANAGERS
ARCHITECTS
APPLICATION ARCHITECTS
ADMINISTRATORS
DEVELOPERS
USERS

Euca2ools

Eucalyptus supported command-line tools.
Get Euca2ools

Ecosystem Tools

Find tools developed for Amazon EC2 and S3 which are compatible with Eucalyptus.
Get tools

FIGURE 10.14

Eucalyptus supports several distributions and is a well documented software for private clouds.

- Storage Controller. Provides persistent virtual hard drives to applications. It is the correspondent of EBS. Users can create snapshots from EBS volumes. Snapshots are stored in Walrus and shared across availability zones.
- Storage Service (Walrus). Provides persistent storage and, similarly to S3, allows users to store objects in buckets.

The system supports a strong separation between the user space and administrator space; users access the system via a web interface while administrators need root access. The system supports a decentralized resource management of multiple clusters with multiple cluster controllers, but a single head node for handling user interfaces. It implements a distributed storage system called Walrus, the analog of Amazon's S3 system. The procedure to construct a VM is based on the generic one described in [449]:

- The *euca2ools* front-end is used to request a VM.
- The VM disk image is transferred to a compute node.

Table 10.4 A side-by-side comparison of Eucalyptus, OpenNebula, and Nimbus.

	Eucalyptus	OpenNebula	Nimbus
Design	Emulate EC2	Customizable	Based on Globus
Cloud type	Private	Private	Public/Private
User population	Large	Small	Large
Applications	All	All	Scientific
Customizability	Administrators limited users	Administrators and users	All but image storage and credentials
Internal security	Strict	Loose	Strict
User access	User credentials	User credentials	x509 credentials
Network access	To cluster controller	–	To each compute node

- This disk image is modified for use by the hypervisor on the compute node.
- The compute node sets up network bridging to provide a virtual NIC with a virtual MAC address.
- The head node the DHCP is set up with the MAC/IP pair.
- The hypervisor activates the VM.
- The user can now *ssh* directly into the VM.

The system can support a large number of users in a corporate enterprise environment. Users are shielded from the complexity of disk configurations and can choose for their VM from a set of five configurations of available processors, memory and hard drive space setup by the system administrators.

Open-Nebula (http://www.opennebula.org/) is a private cloud with users actually logging into the head node to access cloud functions. The system is centralized and its default configuration uses the NFS filesystem. The procedure to construct a VM consists of several steps: (i) a user signs in to the head node using ssh; (ii) next, it uses the *onevm* command to request a VM; (iii) the VM template disk image is transformed to fit the correct size and configuration within the NFS directory on the head node; (iv) the *oned* daemon on the head node uses ssh to log into a compute node; (v) the compute node sets up network bridging to provide a virtual NIC with a virtual MAC; (vi) the files needed by the hypervisor are transferred to the compute node via the NFS; (vii) the hypervisor on the compute node starts the VM; (viii) the user is able to ssh directly to the VM on the compute node.

According to the analysis in [449], the system is best suited for an operation involving a small to medium size group of trusted and knowledgeable users who are able to configure this versatile system based on their needs.

Nimbus (http://www.nimbusproject.org/) is a cloud solution for scientific applications based on the Globus software. The system inherits from Globus the image storage, the credentials for user authentication, and the requirement that a running Nimbus process can ssh into all compute nodes. Customization in this system can only be done by the system administrators.

Table 10.4 summarizes the features of the three systems [449]. The conclusions of the comparative analysis are: Eucalyptus is best suited for a large corporation with its own private cloud as it ensures a degree of protection from user malice and mistakes; OpenNebula is best suited for a testing environment with a few servers; Nimbus is more adequate for a scientific community less interested in the technical internals of the system, but with broad customization requirements.

OpenStack is an open source project started in 2009 at NASA in collaboration with Rackspace (http://www.rackspace.com) to develop a scalable cloud OS for farms of servers using standard hardware. Though recently NASA has moved its cloud infrastructure to AWS, in addition to Rackspace, several other companies including HP, Cisco, IBM, and Red Hat have an interest in *OpenStack*. The current version of the system supports a wide range of features such as: APIs with rate limiting and authentication, live VM management to run, reboot, suspend, and terminate instances, role-based access control, and the ability to allocate, track, and limit resource utilization. The administrators and the users control their resources using an extensible web application called the *Dashboard*.

10.13 THE DARKER SIDE OF VIRTUALIZATION

Can virtualization empower the creators of malware[17] to carry out their mischievous activities with impunity and minimal danger of being detected? How difficult is it to implement such a system? What are the means to prevent this type of malware to be put in place? The answers to these questions are discussed in this section.

It is well understood that in a layered structure a defense mechanism at some layer can be disabled by malware running at a layer below it. Thus, the winner in the continuous struggle between the attackers and the defenders of a computing system is the one in control of the lowest layer of the software stack, the software controlling the hardware, the hypervisor in a virtualized cloud environment.

Recall that a hypervisor allows a guest OS to run on virtual hardware; the hypervisor offers to the guest operating systems a hardware abstraction and mediates its access to the physical hardware. We argued that a hypervisor is simpler and more compact than a traditional OS thus, it is more secure. What if the hypervisor itself is forced to run above another software layer thus, it is prevented to exercise direct control of the physical hardware?

A 2006 paper [272] argues that it is feasible to insert a "rogue hypervisor" between the physical hardware and an OS, as shown in Figure 10.15A. Such a rogue hypervisor is called a *Virtual-Machine Based Rootkit* (VMBR). The term *rootkit* refers to malware with a privileged access to a system. The name comes from *root*, the most privileged account on a Unix system, and *kit*, a set of software components.

It is also feasible to insert the VMBR between the physical hardware and a legitimate hypervisor, as in Figure 10.15B. As a VM running under a legitimate hypervisor sees a virtual hardware, the guest OS will not notice any change of the environment. The only trick is to present the legitimate hypervisor with a hardware abstraction, rather than allow it to run on the physical hardware.

Before we address the question how such an insertion is possible we should point out that in this approach the malware runs either inside a hypervisor or with the support of a hypervisor. A hypervisor

[17]Malware, an abbreviation for *malicious software*, is software designed specifically to circumvent the authorization mechanisms and gain access to a computer system, gather private information, block access to a system, or disrupt the normal operation of a system; computer viruses, worms, spyware, and Trojan horses are examples of malware.

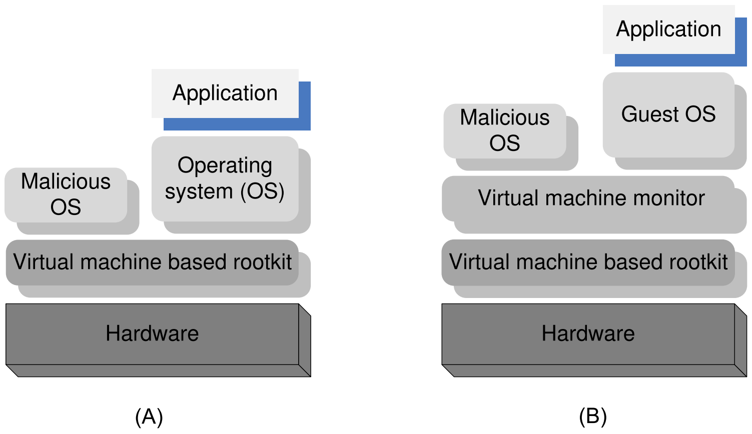


FIGURE 10.15

The insertion of a *Virtual-Machine Based Rootkit* (VMBR) as the lowest layer of the software stack running on the physical hardware; (A) below an OS; (B) below a legitimate hypervisor. The VMBR enables a malicious OS to run surreptitiously and makes it invisible to the genuine or the guest OS and to the application.

is a very potent engine for the malware, it prevents the software of the guest OS or the application to detect malicious activities. A VMBR can record key strokes, system state, data buffers sent to, or received from the network, data to be written to, or read from the disk with impunity; moreover, it can change any data at will.

The only way for a VMBR to take control of a system is to modify the boot sequence and to first load the malware and only then load the legitimate hypervisor, or the OS; this is only possible if the attacker has root privileges. Once the VMBR is loaded it must also store its image on the persistent storage.

The VMBR can enable a separate malicious OS to run surreptitiously and make this malicious OS invisible to the guest OS and to the application running under it. Under the protection of the VMBR the malicious OS could: (i) observe the data, the events, or the state of the target system; or (ii) run services such as spam relays or distributed denial-of-service attacks; or (iii) interfere with the application.

A proof-of-concept VMBRs to subvert Windows XP and Linux and several services based on these two platforms is described in [272]. We should stress that modifying the boot sequence is by no means an easy task and once an attacker has root privileges she is in total control of a system.

10.14 VIRTUALIZATION SOFTWARE

Several virtualization software packages including hypervisors, OS-level virtualization software, and desktop virtualization software are available. There are two types of hypervisors, native and hosted. The set of *native hypervisors* includes:

1. Red Hat Virtualization (RHV) – enterprise virtualization based on the KVM hypervisor.
2. Hyper-V or formerly Windows Server Virtualization – creates VMs on x86-64 systems running Windows.

3. z/VM – current version of IBM's VM operating systems.
4. VMware ESXi – enterprise-class, type-1 hypervisor from VMware.
5. Oracle VM Server for x86 – server virtualization from Oracle Corporation. Incorporates the free, open-source Xen. Supports Windows, Linux, and Solaris guests.
6. Adeos – the Adaptive Domain Environment for Operating Systems is a nanokernel hardware abstraction layer.
7. XtratuM – bare-metal hypervisor for embedded real-time systems. Available for the instruction sets x86, ARM Cortex-R4F processors, and others.

There are several *hosted independent hypervisors* including:

1. VMware Fusion – software hypervisor developed for Intel-based Macs to run Microsoft Windows, Linux, NetWare, or Solaris on VMs, along with the OS X OS. It is based on paravirtualization, hardware virtualization, and dynamic recompilation.
2. PearPC – architecture-independent PowerPC platform emulator for PowerPC operating systems, including pre-Intel versions of OS X, Darwin, and Linux.
3. Oracle VM VirtualBox – free and open-source hypervisor for x86 computers.
4. QEMU (Quick Emulator) – free and open-source hosted hypervisor.

Among the *hosted specialized hypervisors* we note:

1. coLinux – Cooperative Linux, allows Microsoft Windows and the Linux kernel to run simultaneously.
2. MoM – Mac-on-Mac is a port of Mac-on-Linux for Mac OS X.
3. Mac-on-Linux – open source VM for running the classic Mac OS or OS X on PowerPC computers running Linux.
4. bhyve – a type-1 hypervisor included in FreeBSD running FreeBSD 9+, OpenBSD, NetBSD, Linux and Windows desktop and Windows Server.
5. L^4Linux – a variant of Linux kernel running virtualized on L4. L4 is a microkernel and the L4Linux kernel runs a service.

10.15 HISTORY NOTES AND FURTHER READINGS

Virtual memory was the first application of virtualization concepts to commercial computers. Virtual memory allowed multiprogramming and eliminated the need for users to tailor their applications to the physical memory available on individual systems. Paging and segmentation are the two mechanisms supporting virtual memory. Paging was developed for the Atlas Computer built in 1959 at University of Manchester. Independently, the Burroughs Corporation developed B5000, the first commercial computer with virtual memory and released it in 1961; the virtual memory of B5000 used segmentation rather than paging.

In 1967 IBM introduced 360/67, the first IBM system with virtual memory expected to run on a new OS called TSS. Before TSS was released, an operating system called CP-67 was created; CP-67 gave the illusion of several standard IBM 360 systems without virtual memory. The first hypervisor supporting full virtualization was the CP-40 system and ran on a S/360-40 that was modified at the IBM Cambridge Scientific Center to support Dynamic Address Translation, a key feature that allowed virtualization. In CP-40, the hardware's supervisor state was virtualized as well, allowing multiple operating systems to run concurrently in separate VM contexts.

Virtualization was driven by the need to share a very expensive hardware among a large population of users and applications in the early age of computing. The VM/370 system, released in early 1970s for large IBM mainframes was very successful; it was based on a re-implementation of CP/CMS. In VM/370 a new VM was created for every user and this VM interacted with the applications. The hypervisor managed hardware resources and enforced the multiplexing of resources. Modern-day IBM mainframes, such as the zSeries line, retain backwards-compatibility with the 1960s-era IBM S/360 line.

The production of microprocessors coupled with advances in storage technology contributed to the rapid decrease of hardware costs and led to the introduction of personal computers at one end of the spectrum and of large mainframes and massively parallel systems at the other end of the spectrum. The hardware and the operating systems of the 1980s and 1990s gradually limited virtualization and focused instead on efficient multitasking, user interfaces, the support for networking and security problems brought in by interconnectivity.

The advancements in computer and communication hardware, the explosion of the Internet partially due to the success of the World Wide Web at the end of 1990s, renewed the interest in virtualization to support server security and isolation of services. In their review paper Rosenbloom and Grafinkel write [429]: "hypervisors give OS developers another opportunity to develop functionality no longer practical in today's complex and ossified operating systems, where innovation moves at a geologic pace."

Nested virtualization was first discussed in the early 1970s by Popek and Goldberg [196,406].

Further readings. The text of Saltzer and Kaashoek [434] is a very good introduction to virtualization principles. Virtual machines are dissected in a paper by Smith and Nair [455] and architectural principles for virtual computer systems are analyzed in [195,196].

An insightful discussion of hypervisors is provided by the paper of Rosenblum and Garfinkel [429]. Several papers [53,340,341] discuss in depth the Xen hypervisor and analyze its performance, while [529] is a code repository for Xen. The Denali system is presented in [522].

Modern systems such as Linux Vserver (http://linux-vserver.org/), OpenVZ (Open VirtualiZation) [378], FreeBSD Jails [419], and Solaris Zones [409] implement *OS-level virtualization technologies*. Reference [387] compares the performance of two virtualization techniques with a standard OS.

A 2001 paper [102] argues that virtualization allows new services to be added without modifying the OS. Such services are added below the OS level, but this process creates a semantic gap between the VMs and these services. Reflections on the design of hypervisors are the subject of [103] and a discussion of Xen is reported in [110]. The state of the art and the future of nested virtualization are the subject of [128]. An implementation of nested virtualization for KVM is discussed in [61]. [549] surveys security issues in virtual systems and [281] covers reliability in virtual infrastructures. Virtualization technologies in HPC are analyzed in [415] and [457] provides a critical view on virtualization. [516] reports on IBM virtualization strategies.

10.16 EXERCISES AND PROBLEMS

Problem 1. Identify the milestones in the evolution of operating systems during the half century from 1960 to 2010 and comment on the statement from [429] "Hypervisors give OS developers another opportunity to develop functionality no longer practical in today's complex and ossified operating systems, where innovation moves at a geologic pace."

Problem 2. Virtualization simplifies the use of resources, isolates users from one another, supports replication and mobility, but exacts a price in terms of performance and cost. Analyze each one of these aspects for: (i) memory virtualization, (ii) processor virtualization, and (iii) virtualization of a communication channel.

Problem 3. Virtualization of the processor combined with virtual memory management pose multiple challenges; analyze the interaction of interrupt handling and paging.

Problem 4. In Section 10.1 we stated that a hypervisor is a much simpler and better specified system than a traditional OS. The security vulnerability of hypervisors is considerably reduced, as the systems expose a much smaller number of privileged functions. Research the literature to gather arguments in support of these affirmations; compare the number of lines of code and of system calls for several operating systems including Linux, Solaris, FreeBSD, Unbuntu, AIX, and Windows with the corresponding figures for several system VMs in Table 10.1.

Problem 5. In Section 10.3 we state that a hypervisor for a processor with a given ISA can be constructed if the set of *sensitive instructions* is a subset of the privileged instructions of that processor. Identify the set of sensitive instructions for the x86 architecture and discuss the problem each one of these instruction poses.

Problem 6. Table 10.3 summarizes the effects of Xen network performance optimization reported in [341]. The send data rate of a guest domain is improved by a factor of more than 4, while the improvement of the receive data rate is very modest. Identify several possible reasons for this discrepancy.

Problem 7. In Section 10.5 we note that several operating systems including Linux, Minix, NetBSD, FreeBSD, NetWare, and OZONE can operate as paravirtualized Xen guest operating systems running on x86, x86-64, Itanium, and ARM architectures, while VMware EX Server supports full virtualization of x86 architecture. Analyze how VMware provides the functions discussed in Table 10.2 for Xen.

Problem 8. In 2012 Intel and HP announced that *Itanium* architecture will be discontinued. Review the architecture discussed in Section 10.10 and identify several possible reasons for this decision.

Problem 9. Read [387] and analyze the results of performance comparison discussed in Section 10.11.

SECTION

4

CHAPTER

CLOUD SECURITY 11

Security has been a concern since the early days of computing when a computer was isolated, and threats could only be posed by someone with access to the computer room. Once computers were able to communicate with one another the Pandora box of threats was wide opened. In an interconnected world, various embodiments of malware can migrate easily from one system to another, cross national borders, and infect systems all over the world.

Security of computer and communication systems takes on a new urgency as the society becomes increasingly more dependent on the information infrastructure. Even the critical infrastructure of a nation can be attacked by exploiting flaws in computer security and often human naivety. Malware, such as the Stuxnet virus, targets industrial systems controlled by software [104]. The concept of *cyberwarfare* meaning “actions by a nation-state to penetrate another nation’s computers or networks for the purposes of causing damage or disruption” [111], was recently included in the dictionary.

A computer cloud is a target-rich environment for malicious individuals and criminal organizations. It is, thus, no surprise that security is a major concern for existing users and for potential new users of cloud computing services. Some of the security risks faced by computer clouds are shared with other systems supporting network-centric computing and network-centric content, e.g., service-oriented architectures, grids, and web-based services.

Cloud computing is an entirely new approach to computing based on a new technology. It is therefore reasonable to expect that new methods to deal with some of the security threats will be developed, while other perceived threats will prove to be exaggerated [30]. Indeed, “early on in the life cycle of a technology, there are many concerns about how this technology will be used... they represent a barrier to the acceptance... over the time, however, the concerns fade, especially if the value proposition is strong enough” [245].

The breathtaking pace of developments in information science and technology has many side-effects. One of them is that standards, regulations, and laws governing the activities of organizations supporting the new computing services, and in particular utility computing, have yet to be adopted. As a result, many issues related to privacy, security, and trust in cloud computing are far from being settled. For example, there are no international regulations related to data security and privacy. Data stored on a computer cloud can freely cross national borders among the data centers of the CSP.

The chapter starts with a discussion of cloud users concerns related to security in Section 11.1 then, in Section 11.2 we elaborate on the security threats perceived by cloud users already mentioned in Section 2.11. Privacy and trust are covered in Sections 11.3 and 11.4. Encryption protects data in cloud storage, but data must be decrypted for processing as discussed in Section 11.5. Threats during processing originating from flaws in the hypervisors, rogue VMs, or a VMBR, discussed in Section 10.13, cannot be ignored.

The analysis of database service security in Section 11.6 is followed by a presentation of operating system security, VM security, and security of virtualization in Sections 11.7, 11.8, and 11.9, respectively. Sections 11.10 and 11.11 analyze the security risks posed by shared images and by a management OS. An overview of the Xoar hypervisor, a version of Xen which breaks the monolithic Design of the TCB, is discussed in Section 11.12 followed in Section 11.13 by a presentation of a trusted hypervisor and mobile device security in Section 11.14.

11.1 SECURITY, THE TOP CONCERN FOR CLOUD USERS

Some believe that moving to a computer cloud frees an organization from all concerns related to computer security and eliminates a wide range of threats to data integrity. They believe that cloud security is in the hands of experts, hence cloud users are better protected than when using their own computing resources. As we shall see throughout this chapter, these views are not entirely justified.

Outsourcing computing to a cloud generates major new security and privacy concerns. Moreover, the Service Level Agreements do not provide adequate legal protection for cloud computer users who are often left to deal with events beyond their control.

Some cloud users were accustomed to operate inside a secure perimeter protected by a corporate firewall. Now they have to extend their trust to the cloud service provider if they wish to benefit from the economical advantages of utility computing. The transition from a model when users have full control of all systems where their sensitive information is stored and processed is a difficult one. The reality is that virtually all surveys report that security is the top concern of cloud users.

Major user concerns are about the unauthorized access to confidential information and the data theft. Data is more vulnerable in storage, than while it is being processed. Data is kept in storage for extended periods of time, while during processing it is exposed to threats for relatively short time. Close attention should be paid to storage server security and to data in transit. There is also the risk of unauthorized access and data theft posed by rogue employees of a CSP. Cloud users are concerned about insider attacks because hiring and security screening policies of a CSP are totally opaque to the outsiders.

The next concerns regard the user control over the lifecycle of data. It is virtually impossible for a user to determine if data that should have been deleted was actually deleted. Even if deleted, there is no guarantee that the media was wiped out and the next user is not able to recover confidential data. This problem is exacerbated as the CSPs rely on seamless backups to prevent accidental data loss. Such backups are done without user knowledge or consent. During this exercise data records can be lost, accidentally deleted, or accessible to an attacker.

Lack of standardization is next on the list of concerns. Today there are no inter-operability standards as discussed in Section 2.7. Important questions do not have satisfactory answers, e.g.: What can be done when service provided by the CSP is interrupted? How to access critically needed data in case of a blackout? What if the CSP drastically raises its prices? What is the cost of moving to a different CSP? It is undeniable that auditing and compliance pose an entirely different set of challenges in cloud computing. These challenges are not yet resolved. A full audit trail on a cloud is an unfeasible proposition at this time.

Another, less analyzed user concern is that cloud computing is based on a new technology expected to evolve in the future. Case in point, autonomic computing is likely to enter the scene. When this happens self-organization, self-optimization, self-repair, and self-healing could generate additional security threats. In an autonomic system it will be even more difficult than at the present time to determine when an action occurred, what was the reason for that action, and how it created the opportunity for an attack or for data loss. It is still unclear how autonomic computing can be compliant with privacy and legal issues.

There is no doubt that multi-tenancy is the root cause of many user concerns. Nevertheless, multi-tenancy enables a higher server utilization, thus lower costs. The users have to learn to live with multi-tenancy, one of the pillars of utility computing. The threats caused by multi-tenancy differ from one cloud delivery model to another. For example, in case of SaaS private information such as name, address, phone numbers, possibly credit card numbers of many users are stored on one server; when the security of that server is compromised a large number of users are affected.

Users are also greatly concerned about the legal framework for enforcing cloud computing security. The cloud technology has moved much faster than cloud security and privacy legislation thus, users have legitimate concerns regarding the ability to defend their rights. The data centers of a CSP may be located in several countries and it is unclear what laws apply, the laws of the country where information is stored and processed, the laws of the countries the information crossed when sent by the user, or the laws of the user's country.

To make matter even more complicated, a CSP may outsource handling of personal and/or sensitive information. Existing laws stating that the CSP must exercise reasonable security may be difficult to implement in case when there is a chain of outsourcing to companies in different countries. Lastly, a CSP may be required by law to share private data with law enforcement agencies. For example, Microsoft was served a subpoena to provide emails exchanges by users of the Hotmail service.

The question is: What cloud users can and should do to minimize the security risks regarding the data handling by the CSP? First, a user should evaluate the security policies and the mechanisms the CSP has in place to enforce these policies. Then, the user should analyze the information that would be stored and processed on the cloud. Finally, the contractual obligations should be clearly spelled out.

The contract between a user and a CSP should [400] state clearly:

1. CSPs obligations to handle sensitive information and its obligation to comply with privacy laws.
2. CSP liabilities for mishandling sensitive information, e.g., data loss.
3. The rules governing ownership of the data.
4. Specify the geographical regions where information and backups can be stored.

To minimize the security risks a user may try to avoid processing sensitive data on a cloud. The Secure Data Connector from Google caries out an analysis of the data structures involved and allows access to data protected by a firewall. This solution is not feasible for several classes of applications, e.g., processing of medical or personnel records. The solution may not be feasible when the cloud processing workflow requires cloud access to the entire volume of user data. When the volume of sensitive data or the processing workflow requires sensitive data to be stored on the cloud then, whenever feasible, data should be encrypted [189] and [534].

11.2 CLOUD SECURITY RISKS

Some believe that it is easy, possibly too easy, to start using cloud services without the commitment to follow the ethics rules for cloud computing and without a proper understanding of the security risks. A cloud could be used to launch large-scale attacks against other components of the cyber infrastructure. A first question is: How the nefarious use of cloud resources can be prevented?

The next question is: What are the security risks faced by cloud users? There are multiple ways to look at the cloud security risks. A recent paper identifies three broad classes [109]: traditional security threats, threats related to system availability, and threats related to third-party data control.

Traditional threats are those experienced for some time by any system connected to the Internet, but with some cloud-specific twists. The impact of traditional threats is amplified due to the vast amount of cloud resources and the large user population that can be affected. The long list of cloud user concerns includes also the fuzzy bounds of responsibility between the providers of cloud services and users, as well as the difficulties to accurately identify the cause of a problem.

The traditional threats begin at the user site. The user must protect the infrastructure used to connect to the cloud and to interact with the application running on the cloud. This task is more difficult because some components of this infrastructure are outside the firewall protecting the user.

The next threat is related to authentication and authorization. Procedures in place for one individual do not extend to an enterprise, the cloud access of the members of an organization must be nuanced. Different individuals should be assigned distinct levels of privilege based on their role in the organization. It is also nontrivial to merge or adapt the internal policies and security metrics of an organization with the ones of the cloud.

Traditional attacks have already affected cloud service providers. The favorite means of attack are: distributed denial of service (DDDS) attacks which prevent legitimate users to access cloud services, phishing, SQL injection, or cross-site scripting. Phishing aims to gain information from a database by masquerading as a trustworthy entity. Such information could be names and credit card numbers, social security numbers, other personal information stored by online merchants or by other service providers.

SQL injection is typically used against a web site. An SQL command entered in a web form causes the contents of a database used by the web site to be either dumped to the attacker or altered. SQL injection can be used against other transaction processing systems and it is successful when the user input is not strongly typed or rigorously filtered. Cross-site scripting is the most popular form of attack against web sites; a browser permits the attacker to insert client-scripts into the web pages and thus, bypass the access controls at the web site.

Identifying the path followed by an attacker is more difficult in a cloud environment. Cloud servers host multiple VMs and multiple applications may run under one VM. Multi-tenancy, in conjunction with hypervisor vulnerabilities, could open new attack channels for malicious users. Traditional investigation methods based on digital forensics cannot be extended to a cloud where resources are shared among a large user population and traces of events related to a security incident are wiped out due to the high rate of write operations.

Availability of cloud services is another major concern. System failures, power outages, and other catastrophic events could shutdown cloud services for extended periods of time. Data lock-in discussed in Section 2.7 could prevent a large organization whose business model depends on these data to function properly, when such a rare event occurs.

Clouds can also be affected by phase transition phenomena and other effects specific to complex systems. Another critical aspect of availability is that the users cannot be assured that an application hosted on the cloud returns correct results.

Third-party control generates a spectrum of concerns caused by lack of transparency and limited user control. For example, a cloud provider may subcontract some resources from a third party whose level of trust is questionable. There are examples when subcontractors failed to maintain the customer data. There are also examples when the third party was not a subcontractor but a hardware supplier and the loss of data was caused by poor quality storage devices [109].

Storing proprietary data on the cloud is risky as cloud provider espionage poses real dangers. The terms of contractual obligations usually place all responsibilities for data security with the user. The Amazon Web Services customer agreement does not help user's confidence as it states "We ...will not be liable to you for any direct, indirect, incidental,.... damages.... nor... be responsible for any compensation, reimbursement, arising in connection with: (A) your inability to use the services... (B) the cost of procurement of substitute goods or services..or (D) any unauthorized access to, alteration of, or deletion, destruction, damage, loss or failure to store any of your content or other data."

It is very difficult for a cloud user to prove that data has been deleted by the service provider. The lack of transparency makes auditability a very difficult proposition for cloud computing. Auditing guidelines elaborated by the National Institute of Standards (NIST) such as the Federal Information Processing Standard (FIPS) and the Federal Information Security Management Act (FISMA) are mandatory for US Government agencies.

The 2010 Cloud Security Alliance (CSA) report. The report identifies seven top threats to cloud computing. These threats are: the abusive use of the cloud, APIs that are not fully secure, malicious insiders, shared technology, account hijacking, data loss or leakage, and unknown risk profile [123]. According to this report the IaaS delivery model can be affected by all threats. PaaS can be the affected by all, but the shared technology, while SaaS is affected by all, but abuse and shared technology.

Abusing the cloud refers to conducting nefarious activities from the cloud. For example, use multiple AWS instances or applications supported by IaaS to launch distributed denial of service attacks or to distribute spam and malware. *Shared technology* considers threats due to multi-tenant access supported by virtualization. Hypervisors can have flaws allowing a guest OS to affect the security of the platform shared with other VMs.

Insecure APIs may not protect the users during a range of activities starting with authentication and access control to monitoring and control of the application during runtime. The cloud service providers do not disclose their hiring standards and policies thus, the risks of *malicious insiders* cannot be ignored. The potential harm due to this particular form of attacks is high.

Data loss and data leakage are two risks with devastating consequences for an individual or an organization using cloud services. Maintaining copies of the data outside the cloud is often unfeasible due to the sheer volume of data. If the only copy of the data is stored on the cloud, then sensitive data is permanently lost when cloud data replication fails followed by a storage media failure. As some of the data often includes proprietary or sensitive data access to such information by third parties could have severe consequences.

Account or service hijacking is a significant threat and cloud users must be aware of and guard against all methods to steal credentials. Lastly, *unknown risk profile* refers to exposure to the ignorance or underestimation of the risks of cloud computing.

The 2011 CSA report. The report "Security Guidance for Critical Area of Focus in Cloud Computing V3.0," provides a comprehensive analysis of the risks and makes recommendations to minimize the risk in cloud computing [124].

An attempt to identify and classify the attacks in a cloud computing environment is discussed in [207]. The three actors involved in the model considered are: the user, the service, and the cloud infrastructure, and there are six types of attacks possible, see Figure 11.1. The user can be attacked

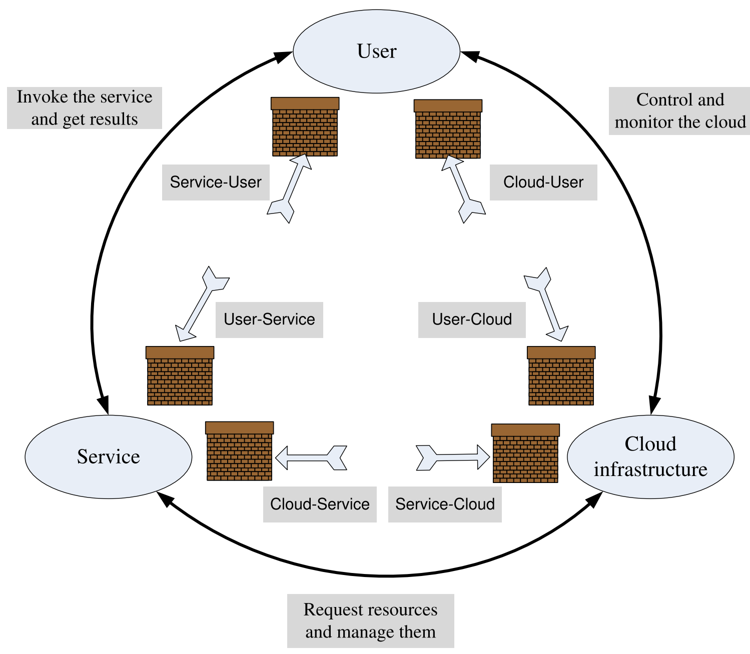


FIGURE 11.1

Surfaces of attacks in a cloud computing environment.

from two directions, the service and the cloud. Secure Sockets Layer (SSL) certificate spoofing, attacks on browser caches, or phishing attacks are example of attacks that originate at the service. The user can also be a victim of attacks that either truly originate or that spoof originating from the cloud infrastructure.

Buffer overflow, SQL injection, and privilege escalation are the common types of attacks from the service. The service can also be subject of attacks by the cloud infrastructure and this is probably the most serious line of attack. Limiting access to resources, privilege-related attacks, data distortion, injecting additional operations are only a few of the many possible lines of attacks originated at the cloud.

The cloud infrastructure can be attacked by a user which targets the cloud control system. These types of attacks are the same a user would direct toward any other cloud service. The cloud infrastructure may also be targeted by a service requesting an excessive amount of resources and causing the exhaustion of the resources.

Top twelve cloud security threats. The 2016 CSA report lists the top security threats [414]:

1. Data breaches. The most damaging breaches are for sensitive data including financial and health information, trade secrets, and intellectual property. The ultimate responsibility rests with the organizations maintaining data on the cloud and CSA recommends that organizations use multi-factor authentication and encryption to protect against data breaches. Multi-factor authentication such as one-time passwords, phone-based authentication, and smart card protection make it harder for attackers to use stolen credentials.

2. Compromised credentials and broken authentication. Such attacks are due to lax authentication, weak passwords, and poor key and/or certificate management.
3. Hacked interfaces and APIs. Cloud security and service availability can be compromised by a weak API. When third parties rely on APIs more services and credentials are exposed.
4. Exploited system vulnerabilities. Resource sharing and multi-tenancy create new attack surfaces but the cost to discover and repair vulnerabilities is small compared to the potential damage.
5. Account hijacking. All accounts should be monitored so that every transaction can be traced to the individual requesting it.
6. Malicious insiders. This threat can be difficult to detect and system administrator errors could sometimes be falsely diagnosed as threats. A good policy is to segregate duties and enforce activities such as logging, monitoring, and auditing administrator activities.

The other six threats are: advanced persistent threats (APTs), permanent data loss, inadequate diligence, cloud service abuse, DoS attacks, and shared technology.

An update of the "Cloud Controls Matrix" spells out the impact of control specifications on cloud architecture, cloud delivery models, and other aspects of the cloud ecosystem, according to https://cloudsecurityalliance.org/download/cloud-controls-matrix-v3-0-1/.

Cloud vulnerability incidents reported over a period of four years and data breach incidents in 2014 identified several other threats including: hardware failures, natural disasters, cloud-related malware, inadequate infrastructure design and planning, point-of-sale (POS) intrusions and payment card skimmers, crimeware and cyber-espionage, insider and privilege misuse, web app attacks, and physical theft/loss [480].

11.3 PRIVACY AND PRIVACY IMPACT ASSESSMENT

The term *privacy* refers to the right of an individual, a group of individuals, or an organization to keep information of personal nature or proprietary information from being disclosed. Many nations view privacy as a basic human right. The Universal Declaration of Human Rights, article 12, states: "No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honor and reputation. Everyone has the right to the protection of the law against such interference or attacks."

The U. S. Constitution contains no express right to privacy, however the Bill of Rights reflects the concern of the framers for protecting specific aspects of privacy.[1] In the United Kingdom privacy is guaranteed by the Data Protection Act. The European Court of Human Rights has developed many documents defining the right to privacy.

At the same time, the right to privacy is limited by laws. For example, the taxation laws require individuals to share information about personal income or earnings. Individual privacy may conflict with other basic human rights e.g., with freedom of speech. The privacy laws differ from country to

[1]The 1st Amendment covers the protection of beliefs, the 3rd Amendment privacy of homes, the 4th Amendment the privacy of person and possessions against unreasonable searches, the 5th Amendment the privilege against self-incrimination, thus the privacy of personal information, and, according to some Justices, the 9th Amendment that reads "The enumeration in the Constitution, of certain rights, shall not be construed to deny or disparage others retained by the people" can be viewed as a protection of privacy in ways not explicitly specified by the first eight amendments in the Bill of Rights.

country; laws in one country may require public disclosure of information considered private in other countries and cultures.

Digital age has confronted legislators with significant challenges related to privacy as new threats have emerged. For example, personal information voluntarily shared, but stolen from sites granted access to it or misused can lead to *identity theft*.

Some countries have been more aggressive in addressing the new privacy concerns than others. For example, European Union (EU) has very strict laws governing handling of personal data in the digital age. A sweeping new privacy right, the "right to be forgotten" is codified as part of a broad new proposed data protection regulation in EU. This right addresses the problem that it is hard to escape your past now when every photo, status update, and tweet lives forever on some web site.

Our discussion targets primarily public clouds where privacy has an entirely new dimension as data, often in an un-encrypted form, resides on servers owned by a CSP. Services based on individual preferences, location of individuals, membership in social networks, or other personal information present a special risk. The owner of the data cannot rely exclusively on the CSP to guarantee the privacy of the data.

Privacy concerns are different for the three cloud delivery models and also depend on the actual context. For example, consider the widely used Gmail; Gmail privacy policy reads (see http://www.google.com/policies/privacy/ accessed on October 6, 2012): "We collect information in two ways: information you give us... like your name, email address, telephone number or credit card; information we get from your use of our services such as:.. device information, ... log information,... location information,... unique application numbers,... local storage,... cookies and anonymous identifiers.... We will share personal information with companies, organizations or individuals outside of Google if we have a good-faith belief that access, use, preservation or disclosure of the information is reasonably necessary to: meet any applicable law, regulation, legal process or enforceable governmental request; ... protect against harm to the rights, property or safety of Google, our users or the public as required or permitted by law. We may share aggregated, non-personally identifiable information publicly and with our partners like publishers, advertisers or connected sites. For example, we may share information publicly to show trends about the general use of our services."

The main aspects of cloud privacy are: the lack of user control, potential unauthorized secondary use, data proliferation, and dynamic provisioning [400]. The lack of user control refers to the fact that user-centric data control is incompatible with cloud usage. Once data is stored on the servers of the CSP the user losses control on the exact location, and in some instances it could lose access to the data. For example, in case of the Gmail service the account owner has no control on where the data is stored or how long old Emails are stored on some backups of the servers.

A CSP may obtain revenues from unauthorized secondary usage of the information e.g., for targeted advertising. There are no technological means to prevent this use. Dynamic provisioning refers to threats due to outsourcing. A range of issues are very fuzzy; for example, how to identify the subcontractors of a CSP, what rights to the data they have, and what rights to data are transferable in case of bankruptcy or merger.

There is the need for legislation addressing the multiple aspects of privacy in the digital age. A document elaborated by the Federal Trading Commission for the US Congress states [172]: "Consumer-oriented commercial web sites that collect personal identifying information from or about consumers online would be required to comply with the four widely-accepted fair information practices:

1. Notice – web sites should be required to provide consumers clear and conspicuous notice of their information practices, including what information they collect, how they collect it (e.g., directly or through non-obvious means such as cookies), how they use it, how they provide Choice, Access, and Security to consumers, whether they disclose the information collected to other entities, and whether other entities are collecting information through the site.
2. Choice – web sites should be required to offer consumers choices as to how their personal identifying information is used beyond the use for which the information was provided, e.g., to consummate a transaction. Such choices would encompass both internal secondary uses (such as marketing back to consumers) and external secondary uses, such as disclosing data to other entities.
3. Access – web sites would be required to offer consumers reasonable access to the information a web site has collected about them, including a reasonable opportunity to review information and to correct inaccuracies or delete information.
4. Security – web sites would be required to take reasonable steps to protect the security of the information they collect from consumers. The Commission recognizes that the implementation of these practices may vary with the nature of the information collected and the uses to which it is put, as well as with technological developments. For this reason, the Commission recommends that any legislation be phrased in general terms and be technologically neutral. Thus, the definitions of fair information practices set forth in the statute should be broad enough to provide flexibility to the implementing agency in promulgating its rules or regulations."

There is the need for tools capable to identify privacy issues in information systems, the so called *Privacy Impact Assessment (PIA)*. As of mid 2017 there are no international standards for such a process, though different countries and organization require PIA reports. An example of an analysis is to assess the legal implications of the UK-US Safe Harbor process to allow US companies to comply with the European Directive 95/46/EC[2] on the protection of personal data.

Such an assessment forces a proactive attitude towards privacy. An ab initio approach for embedding privacy rules in new systems is preferable to painful changes that could affect the functionality of existing systems. A PIA tool that could be deployed as web-based service is proposed in [478]. The input to the tool includes: project information, an outline of project documents, privacy risks, and stakeholders. The tool will produce a PIA report consisting of a summary of findings, a risk summary, security, transparency, and cross-borders data flows.

The centerpiece of a PIA tool is a knowledge base (KB) created and maintained by domain experts. The users of the SaaS service providing access to the PIA tool must fill in a questionnaire. The system uses templates to generate additional questions necessary to fill in the PIA report. An expert system infers which rules are satisfied by the facts in the database as provided by the users and executes the rule with the highest priority.

11.4 TRUST

Trust, in the context of cloud computing, is intimately related to the general problem of trust in online activities. In this section we first discuss the traditional concept of trust and then the trust in online activities.

[2]See http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML.

Trust. According to the Merriam-Webster dictionary trust means "assured reliance on the character, ability, strength, or truth of someone or something." Trust is a complex phenomenon, it enables cooperative behavior, promotes adaptive organizational forms, reduces harmful conflict, decreases transaction costs, facilitates formulation of ad hoc work groups, and promotes effective responses to crisis [430].

Two conditions must exist for trust to develop. The first is *risk*, the perceived probability of loss. Indeed, trust would not be necessary if there is no risk involved, if there is a certainty that an action can succeed. The second is *interdependence*, the interests of one entity cannot be archived without reliance on other entities.

A trust relationship goes though three phases:

1. Building phase, when trust is formed.
2. Stability phase, when trust exists.
3. Dissolution phase, when trust declines.

There are different reasons and forms of trust. Utilitarian reasons could be based on the belief that the costly penalties for breach of trust exceed any potential benefits from opportunistic behavior. This is the essence of *deterrence-based* trust. Another reason is the belief that the action involving the other party is in the self-interest of that party. This is the so-called *calculus-based* trust. After a long sequence of interactions *relational trust* between entities can developed based on the accumulated experience of dependability and reliance on each other.

The common wisdom is that an entity must work very hard to build trust, but may lose trust very easily. A single violation of trust can lead to irreparable damages. *Persistent trust* is based on the long term behavior of an entity, while *dynamic trust* is based on a specific context, e.g., state of the system or the effect of technological developments.

Internet trust. Internet trust "obscures or lacks entirely the dimensions of character and personality, nature of relationship, and institutional character" of the traditional trust [360]. The missing identity, personal characteristics, and role definitions are elements we have to deal with in the context of online trust.

The Internet offers individuals the ability to obscure or conceal their identity. The resulting anonymity reduces the clues normally used in judgments of trust. Identity is critical for developing trust relations, it allows us to base our trust on the past history of interactions with an entity. Anonymity causes mistrust because identity is associated with accountability and in absence of identity accountability cannot be enforced.

The opacity extends immediately from identity to personal characteristics. It is impossible to infer if the entity or individual we transact with is who it pretends to be, as the transactions occur between entities separated in time and distance. Lastly, there are no guarantees that the entities we transact with fully understand the role they have assumed.

To remedy the loss of clues we need security mechanisms for access control, transparency of identity, and surveillance. The mechanisms for access control are designed to keep intruders and mischievous agents out. Identity transparency requires that the relation between a virtual agent and a physical person should be carefully checked through methods such as biometric identification. Digital signatures and digital certificates are used for identification. Surveillance could be based on *intrusion detection* or can be based on logging and auditing. The first is based on real-time monitoring, while the second relies on off-line sifting through audit records.

Credentials are used when the entity is not known; credentials are issued by a trusted authority and describe the qualities of the entity using the credential. A DDS (Doctor of Dental Surgery) diploma hanging on the wall of a dentist's office is a credential that the individual has been trained by an accredited university and hence capable to perform a set of procedures. A digital signature is a credential used in many distributed applications.

Policies and *reputation* are two ways of determining trust. Policies reveal the conditions to obtain trust, and the actions when some of the conditions are met. Policies require the verification of credentials. Reputation is a quality attributed to an entity based on a relatively long history of interactions or possibly observations of the entity. Recommendations are based on trust decisions made by others and filtered through the perspective of the entity assessing the trust.

In a computer science context "trust of a party A to a party B for a service X is the measurable belief of A that B behaves dependably for a specified period within a specified context (in relation to service X)," [376]. An assurance about the operation of particular hardware or software component leads to persistent social-based trust in that component.

A comprehensive discussion of trust in computer services in the semantic web can be found in [38]. In Section B.1 we discuss the concept of trust in the context of cognitive radio networks where multiple transmitters compete for communication channels. Then, in Section B.3 we present a cloud-based trust management service.

11.5 CLOUD DATA ENCRYPTION

The government, large corporations, and individual users ponder if it safe to store sensitive information on a public cloud. Encryption is the obvious solution to protect outsourced data and cloud service providers have been compelled to offer encryption services. For example, Amazon offers AWS Key Management Service (KMS) to create and control the encryption keys used by clients to encrypt their data. KMS is integrated with other AWS services including EBS, S3, RDS, Redshift, Elastic Transcoder, and WorkMail. AWS also offers Encryption SDK for developers.

The seminal RSA paper [424] and the survey of existing public-key crypto systems in [433] are some of the notable publications in the vast literature dedicated to cryptosystems. Several new research results in cryptography are important to data security in cloud computing. In 1999 Pascal Paillier proposed a trapdoor mechanism based on composite residuosity classes, i.e., factoring a hard-to-factor number $n = pq$ where p and q are two large prime numbers [388]. This solution exploits the homomorphic properties of composite residuosity classes to design distributed cryptographic protocols. A major breakthrough are the algorithms for Fully Homomorphic Encryption (FHE) proposed by Craig Gentry in his seminal 2009 dissertation at Stanford University [189,190]. In recent years, searchable symmetric encryption protocols have been reported in [86] and [168].

Homomorphic encryption. Sensitive data is safe while in storage, provided that it is encrypted with strong encryption. But encrypted data must be decrypted for processing and this opens a window of vulnerability. So a first question examined in this section is if it is feasible to operate on encrypted data. The *homomorphic encryption*, a long time dream of security experts, reflects the concept of homomorphism, a structure-preserving map $f(\cdot)$ between two algebraic structures of the same type see Figure 11.2.

When $f(\cdot)$ is a one-to-one mapping, call $f^{-1} : A' \to A$ the inverse of $f(\cdot)$. Then $a = f^{-1}(a'), b = f^{-1}(b'), c = f^{-1}(c')$. In this case we can carry out the composition operation $\diamond$ in the target domain

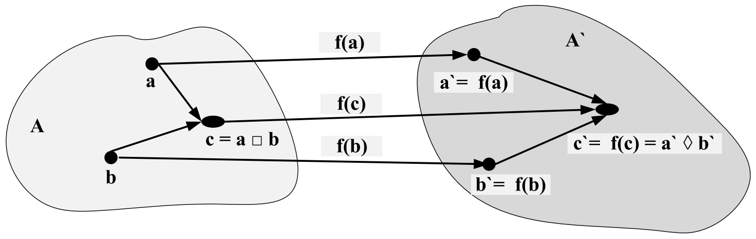


FIGURE 11.2

A homomorphism $f : A \to A'$ is a structure-preserving map between sets A and A' with the composition operations $\Box$ and $\diamond$, respectively. Let $a, b, c \in A$ with $c = a\Box b$ and $a', b', c' \in A'$ with $c' = a'\diamond b'$. Let $a' = f(a), b' = f(b), c' = f(c)$ be the results of the mapping $f(\cdot)$. If f is a homomorphism, then the composition operation $\diamond$ in the target domain A' produces the same result as mapping the result of the operation $\Box$ applied to the two elements in the original domain A: $f(a)\diamond f(b) = f(a\Box b)$.

and apply the inverse mapping to get the same result produced by the $\Box$ composition operation in the original domain, $f^{-1}(a)\diamond f^{-1}(b) = f(a\Box b)$, as shown in Figure 11.2.

In case of homomorphic encryption the mapping $f(\cdot)$ is a one-to-one transformation, the encryption procedure; its inverse, $f^{-1}(\cdot)$ is the decryption procedure and the composition operation can be any arithmetic and logic operation carried out with encrypted data. In this case we can carry arithmetic and/or logic operations with encrypted data and the decryption of the result of these operations is identical with the result of carrying out the same operations with the plaintext data. The window of vulnerability created when data is decrypted for processing disappears.

General computations with encrypted data are theoretically feasible using FHE algorithms. Unfortunately, the homomorphic encryption is not a practical solution at this time. Existing algorithms for homomorphic encryption increase the processing time with encrypted data by many orders of magnitude compared with processing of plaintext data. A recent implementation of FHE [218] requires about six minutes per batch; the processing time for a simple operation on encrypted data dropped to almost one second after improvements in other experiments [154].

Users send a variety of queries to many large databases stored on clouds. Such queries often involve logic and arithmetic functions so an important question is if it is feasible and practical to search encrypted databases. Application of widely used encryption techniques to database systems could lead to significant performance degradation. For example, if an entire column of a NoSQL database table contains sensitive information and it is encrypted, then a query predicate with a comparison operator requires a scan of the entire table to evaluate the query. This is due to the fact that existing encryption algorithms do not preserve order and database indices such as B-tree can no longer be used.

Order Preserving Encryption. OPE can be used for encryption of numeric data, it maps a range of numerical values into a much larger and sparse range of values [70]. Let a order-preserving function $f : \{1, ..., M\} \to \{1, ..., N\}$ with $N >> M$ be uniquely represented by a combination of M out of N ordered items. Given N balls in a bin, M black and $N - M$ white, we draw a ball at random without replacement at each step. The random variable X describing the total number of balls in our sample after we collect the k-th black ball follows the negative hypergeometric distribution (NHG). One can show that a order preserving $f(x)$ for a given point $x \in \{1, \ldots, M\}$ has a NHG distribution over a random choice of f.

To encrypt plaintext x the OPE encryption algorithm performs a binary search down to x. Given the secret key K the algorithm first assigns $Encrypt(K, M/2)$, then $Encrypt(K, M/4)$ if the index $m < M/2$ and $Encrypt(K, 3M/4)$ otherwise, and so on, until $Encrypt(K, x)$ is assigned. Each ciphertext assignment is made according to the output of the negative hypergeometric sampling algorithm. One can prove by strong induction on the size of the plaintext space that the resulting scheme induces a random order-preserving function from the plaintext to ciphertext space.

To allow efficient range queries on encrypted data, it is sufficient to have an order-preserving hash function family H (not necessarily invertible). The OPE algorithm would use a secret key $(K_{Encrypt}, K_H)$ where $K_{Encrypt}$ is a key for a normal (randomized) encryption scheme and K_H is a key for H. Then $Encrypt(K_{Encrypt}, x) \,||\, H(K_H, x)$ will be the encryption of x [70].

Searching encrypted databases is of particular interest [11]. Several types of searches are frequently conducted including: single-keyword, multi-keyword, fuzzy-keyword, ranked, authorized, and verifiable search. Searchable symmetric encryption (SSE) is used when an encrypted databases $\mathcal{E}$ is outsourced to a cloud or to a different organization. SSE hides information about the database and the queries.

The client only stores the cryptographic key. To search the database the client encrypts the query, sends it to the database server, receives the encrypted result of the query and decrypts it using the cryptographic key. The information leakage from these searches is confined to query patterns, while disclosure of explicit data and query plaintext values is prevented.

An SSE protocol supporting conjunctive search and general Boolean queries on symmetrically encrypted data was proposed in [86]. This SSE protocol scales to very large databases. It can be used for arbitrarily structured data including free text search with the moderate and well defined leakage to the outsourced server. Performance results of a prototype applied to encrypted search over the entire English Wikipedia are reported. The protocol was extended with support for range, substring, wildcard, and phrase queries [168].

The next question is if sensitive data stored on the servers of a private cloud is vulnerable. The threat posed by an outsider attacker is diminished if the private cloud is protected by an effective firewall. Nevertheless, there are dangers posed by an insider. If such an attacker has access to log files it can infer the location of database hot spots, copy data selectively, and use the data for a nefarious activity. To minimize the risks posed by an insider, a set of protections rings should be enforced to restrict the access of each member of the staff to a limited area of the data base.

11.6 SECURITY OF DATABASE SERVICES

Cloud users often delegate control of their data to the database services supported by virtually all CSP and are concerned with security aspects of DBaaS. The model used to evaluate DBaaS security includes several groups of entities: data owners, users of data, CSPs, and third party agents or Third Party Auditors (TPAs).

Data owners and DBaaS users fear compromised integrity and confidentiality, as well as data unavailability. Insufficient authorization, authentication and accounting mechanisms, inconsistent use of encryption keys and techniques, alteration or deletion of records without maintaining backup, and operational failures are the major causes of data loss in DBaaS.

Some data integrity and privacy issues are due to the absence of authentication, authorization and accounting controls, or poor key management for encryption and decryption. Confidentiality means that only authorized users should have access to the data. Unencrypted data is vulnerable to bugs, errors, and attacks from external entities affecting data confidentiality. Insider attacks are another concern for DBaaS users and data owners. Superusers have unlimited privileges and misuse of superuser privileges poses a considerable threat to confidential data such as medical records, sensitive business data, proprietary product data, and so on.

Malicious external attackers use spoofing, sniffing, man-in-the-middle attacks, side channeling and illegal transactions to launch DoS attacks. Another concern is illegal recovery of data from storage devices, a side effect of multi-tenancy. CSP often carry out sanitation operations after deleting data from physical devices, but sophisticated attackers can still recover information from storage devices, unless a thorough scrubbing operation is carried out. Data is also vulnerable during transfer from the data owner to the DBaaS through public networks. Encryption before data transmission can reduce the risks posed to the data in transit to the cloud.

Data provenance, the process of establishing the origin of data and its movement between databases, uses metadata to determine the data accuracy, but the security assessments are time-sensitive. Moreover, analyzing large provenance metadata graphs is computationally expensive.

Cloud users are not aware of the physical location of their data. This lack of transparency allows cloud service providers to optimize the use of resources but in case of security breaches it is next to impossible for users to identify compromised resources. DBaaS users do not have fine-grained control of the remote execution environment and cannot inspect the execution traces to detect the occurrence of illegal operations.

To increase availability, performance and to enhance reliability, cloud database services replicate data. Ensuring consistency among the replicas is challenging. Another critical function of DBaaS is to carry out timely backups of all sensitive and confidential data to facilitate quick recovery in case of disasters. Auditing and monitoring are important functions of a DBaaS but generate their own security risks when delegated to TPAs. Conventional methods for auditing and monitoring demand detailed knowledge of the network infrastructure and physical devices. Data privacy laws can be violated as consumers are unaware where the data is actually stored. Privacy laws in Europe and South America prohibit storing data outside the country of origin.

In summary, DBaaS data *availability* is affected by several threats including:

- Resource exhaustion caused by imprecise specification of user needs or incorrect evaluation of user specifications.
- Failures of the consistency management; multiple hardware and/or software failures lead to inconsistent views of user data.
- Failure of the monitoring and auditing system.

DBaaS data *confidentiality* is affected by insider and outsider attacks, access control issues, illegal data recovery from storage, network breaches, third-party access, inability to establish the provenance of the data.

11.7 OPERATING SYSTEM SECURITY

An operating system allows multiple applications to share the hardware resources of a physical system subject to a set of policies. A critical function of an OS is to protect applications against a wide range of malicious attacks such as unauthorized access to privileged information, tampering with executable code, and spoofing. Such attacks can target even single-user systems such as personal computers, tablets, or smart phones. Data brought in the system may contain malicious code; this could be the case of a Java applet, or of data imported by a browser from a malicious web site.

The *mandatory security* of an OS is considered to be [295]: "any security policy where the definition of the policy logic and the assignment of security attributes is tightly controlled by a system security policy administrator." Access control, authentication usage, and cryptographic usage policies are all elements of the mandatory OS security.

Access control policies specify how OS controls access to different system objects, authentication usage defines the authentication mechanisms used by the OS to authenticate a principal, and cryptographic usage policies specify the cryptographic mechanisms used to protect the data. A necessary but not sufficient condition for security is that the subsystems tasked to perform security-related functions are tamper-proof and cannot be bypassed. An OS should confine an application to a unique security domain.

Applications with special privileges performing security-related functions are called *trusted applications*. Such applications should only be allowed the lowest level of privileges required to perform their functions. For example, type enforcement is a mandatory security mechanism that can be used to restrict a trusted application to the lowest level of privileges.

Enforcing mandatory security through mechanisms left at user's discretion can lead to a breach of security, sometimes due to malicious intent, in other cases due to carelessness, or to lack of understanding. Discretionary mechanisms place the burden of security on individual users. Moreover, an application may change a carefully defined discretionary policy without the consent of the user, while a mandatory policy can only be changed by a system administrator.

Unfortunately, commercial operating systems do not support multi-layered security. They only distinguish between a completely privileged security domain and a completely unprivileged one. Some operating systems, e.g., Windows NT, allow a program to inherit all the privileges of the program invoking it, regardless of the level of trust in that program.

The existence of *trusted paths*, mechanisms supporting user interactions with trusted software is critical for system security. When such mechanisms do not exist malicious software can impersonate trusted software. Some systems allow servers to authenticate their clients and provide trusted paths for a few functions, such as login authentication and password changing.

A solution to the trusted path problem is to decompose a complex mechanism in several components with well-defined roles [295]. For example, the access control mechanism for the application space could consist of *enforcer* and *decider* components. To access a protected object the enforcer will gather the required information about the agent attempting the access, will pass this information to the decider together with the information about the object and the elements of the policy decision; finally, it will carry out the actions requested by the decider.

A trusted path mechanism is required to prevent malicious software invoked by an authorized application to tamper with the attributes of the object and/or with the policy rules. A trusted path is also required to prevent an impostor from impersonating the decider agent. A similar solution is proposed for the cryptography usage which should be decomposed into an analysis of the invocation mechanisms and an analysis of the cryptographic mechanism.

Another question is how an OS can protect itself and applications running under it from malicious mobile code attempting to gain access to data and other resources and compromise system confidentiality and/or integrity. Java Security Manager uses the type-safety attributes of Java to prevent unauthorized actions of an application running in a "sandbox." Yet, the Java Virtual Machine (JVM) accepts byte code in violation of language semantics; moreover, it cannot protect itself from tampering from other applications.

Even if these security problems could be eliminated, the security relies on the ability of the file system to preserve the integrity of the Java class code. Requiring digitally signed applets and accepting them only from trusted sources could fail due to the all-or-nothing security model. A solution to securing mobile communications could be to confine a browser to a distinct security domain.

Specialized *closed-box platforms* such as the ones on some cellular phones, game consoles, and Automatic Teller Machines (ATMs) could have embedded cryptographic keys that allow themselves to reveal their true identity to remote systems and authenticate the software running on them. Such facilities are not available to *open-box platforms*, the traditional hardware designed for commodity operating systems.

A highly secure operating system is necessary but not sufficient. Application-specific security is also necessary. Sometimes, security implemented above the operating system is better, e.g., electronic commerce requires a digital signature on each transaction.

We conclude that commodity operating systems offer low assurance. Indeed, an OS is a complex software system consisting of millions of lines of code and it is vulnerable to a wide range of malicious attacks. An OS poorly isolates one application from another; once an application is compromised, the entire physical platform and all applications running on it can be affected. The platform security level is thus reduced to the security level of the most vulnerable application running on the platform.

Operating systems provide only weak mechanisms for applications to authenticate one another and do not have a trusted path between users and applications. These shortcomings add to the challenges of providing security in a distribute computing environment. For example, a financial application cannot determine if a request comes from an authorized user or from a malicious program; in turn, a human user cannot distinguish a response from a malicious program impersonating the service from the response provided by the service.

11.8 VIRTUAL MACHINE SECURITY

The following discussion of VM security is restricted to the traditional system VM model in Figure 10.1B when a hypervisor controls the access to the hardware. The hybrid and the hosted VM models shown in Figures 10.1C and D, respectively, expose the entire system to the vulnerability of the host operating system thus, will not be analyzed.

Virtual security services are typically provided by the hypervisor as shown in Figure 11.3A; another alternative is to have a dedicated VM providing security service as in Figure 11.3B. A secure TCB (Trusted Computing Base) is a necessary condition for security in a VM environment. When the TCB is compromised then the security of the entire system is affected.

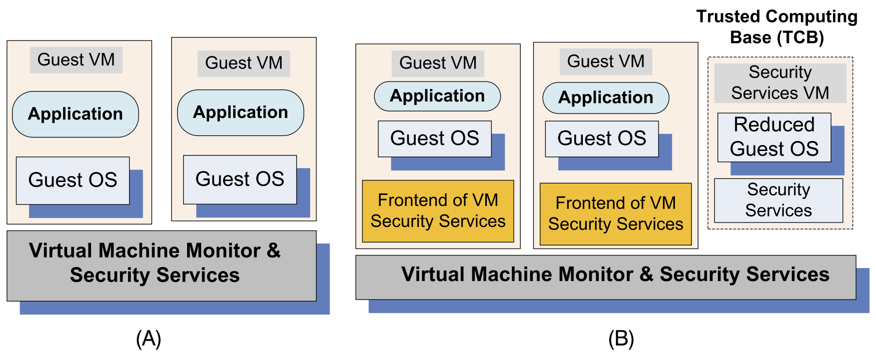

Trusted Computing Base (TCB)
Guest VM
Application
Guest OS
Virtual Machine Monitor & Security Services
Frontend of VM Security Services
Security Services VM
Reduced Guest OS
Security Services
(A)
(B)

FIGURE 11.3

(A) Virtual security services provided by the hypervisor/Virtual Machine Monitor; (B) A dedicated security VM.

The analysis of Xen and *vBlades* in Sections 10.5 and 10.10 shows that the VM technology provides a stricter isolation of VMs from one another than the isolation of processes in a traditional operating system. Indeed, a hypervisor controls the execution of privileged operations and can thus enforce memory isolation as well as disk and network access.

Hypervisors are considerably less complex and better structured than traditional operating systems thus, in a better position to respond to security attacks. A major challenge is that a hypervisor sees only raw data regarding the state of a guest OS while security services typically operate at a higher logical level, e.g., at the level of a file rather than a disk block.

A guest OS runs on simulated hardware and the hypervisor has access to the state of all VMs operating on the same hardware. The state of a guest VM can be saved, restored, cloned, and encrypted by the hypervisor. Replication can ensure not only reliability but also support security, while cloning could be used to recognize a malicious application by testing it on a cloned system and observing if it behaves normally.

We can also clone a running system and examine the effect of potentially dangerous applications. Another interesting possibility is to have the guest VMs files moved to a dedicated VM and thus, protect it from attacks [549]. This solution is possible because inter-VM communication is faster than communication between two physical machines.

Sophisticated attackers are able to fingerprint VMs and avoid VM honey pots designed to study the methods of attack. They can also attempt to access VM-logging files and thus, recover sensitive data; such files have to be very carefully protected to prevent unauthorized access to cryptographic keys and other sensitive data.

We expect to pay some price for the better security provided by virtualization. This price includes: (i) higher hardware costs because a virtual system requires more resources such as CPU cycles, memory, disk, and network bandwidth; (ii) the cost of developing hypervisors and modifying the host operating systems in case of paravirtualization; and (iii) the overhead of virtualization as the hypervisor is involved in privileged operations.

VM-based intrusion detection systems such as Livewire and Siren which exploit the three capabilities of a VM for intrusion detection, isolation, inspections, and interposition are surveyed in [549].

Resource isolation was examined in Section 8.10. Inspection means that the hypervisor has the ability to review the state of the guest VMs and interposition means that the hypervisor can trap and emulate the privileged instruction issued by the guest VMs. VM-based intrusion prevention systems such as, SVFS, NetTop, and IntroVirt, and surveys Terra, a VM-based trust computing platform are also discussed in [549]. Terra uses a *trusted hypervisor* to partition resources among VMs.

NIST security group distinguishes two groups of threats, hypervisor-based and VM-based.

There are several types of hypervisor-based threats:

1. Starvation of resources and denial of service for some VMs. Probable causes: (a) badly configured resource limits for some VMs; (b) a rogue VM with the capability to bypass resource limits set in hypervisor.
2. VM side-channel attacks: malicious attack on one or more VMs by a rogue VM under the same hypervisor. Probable causes: (a) lack of proper isolation of inter-VM traffic due to misconfiguration of the virtual network residing in the hypervisor; (b) limitation of packet inspection devices to handle high speed traffic, e.g., video traffic; (c) presence of VM instances built from insecure VM images, e.g., a VM image having a guest OS without the latest patches.
3. Buffer overflow attacks.

There are also several types of VM-based threats:

1. Deployment of rogue or insecure VM; unauthorized users may create insecure instances from images or may perform unauthorized administrative actions on existing VMs. Probable cause: improper configuration of access controls on VM administrative tasks such as instance creation, launching, suspension, re-activation and so on.
2. Presence of insecure and tampered VM images in the VM image repository. Probable causes: (a) lack of access control to the VM image repository; (b) lack of mechanisms to verify the integrity of the images, e.g., digitally signed image.

11.9 SECURITY OF VIRTUALIZATION

The complex relationship between virtualization and security has two distinct aspects: virtualization of security and security of virtualization [302]. In Chapter 10 we praised the virtues of virtualization. We also discussed two problems associated with virtual environments: (a) the negative effect on performance, due to the additional overhead; and (b) the need for more powerful systems to run multiple VMs. In this section we take a closer look at the security of virtualization.

The complete state of an operating system running under a VM is captured by the VM. The VM state can be saved in a file and then the file can be copied and shared. There are several useful implications of this important virtue of virtualization:

1. Supports the IaaS delivery model. An IaaS user selects an image matching the local application environment and then uploads and runs the application on the cloud using this image.
2. Increased reliability. An operating system with all the applications running under it can be replicated and switched to a hot standby in case of a system failure. Recall that a hot standby is a method to achieve redundancy. The primary and the backup systems, run simultaneously and have identical state information.
3. Straightforward mechanisms for implementing resource management policies. An OS and the applications running under it can be moved to another server to balance the load of a system. For

example, the load of lightly loaded servers can be moved to other servers and then lightly loaded servers can be switched off or placed in standby mode to reduce power consumption.

4. Improved intrusion detection. In a virtual environment a clone can look for known patterns in system activity and detect intrusion. The operator can switch a server to hot standby when suspicious events are detected.
5. Secure logging and intrusion protection. When implemented at the OS level intrusion detection can be disabled and logging can be modified by an intruder. When implemented at the hypervisor layer, the services cannot be disabled or modified. In addition, the hypervisor may be able to log only events of interest for a post-attack analysis.
6. More efficient and flexible software maintenance and testing. Virtualization allows the multitude of OS instances to share a small number of physical systems, instead of a large number of dedicated systems running under different operating systems, different versions of each OS, and different patches for each version.

Is there a price to pay for the benefits of virtualization outlined above? There is always the other side of a coin, so we should not be surprised that the answer to this question is a resounding Yes. In a 2005 paper [185] Garfinkel and Rosenblum argue that the serious implications of virtualization on system security cannot be ignored. This theme is revisited in 2008 by Price [410] who reaches similar conclusions.

A first group of undesirable effects of virtualization lead to a diminished ability of an organization to manage its systems and track their status. These undesirable effects are:

- The number of physical systems in the inventory of an organization is limited by cost, space, energy consumption, and human support. The explosion of the number of VMs is a fact of life; to create a VM one simply copies a file. The only limitation for the number of VMs is the amount of storage space available.
- There is also a qualitative side to the explosion of the number of VMs. Traditionally, organizations install and maintain the same version of system software. In a virtual environment such a uniformity cannot be enforced, the number of different operating systems, their versions, and the patch status of each version will be diverse and the diversity will tax the support team.
- One of the most critical problems posed by virtualization is related to the software lifecycle. The traditional assumption is that the software lifecycle is a straight line, hence the patch management is based on a monotonic forward progress. The virtual execution model *maps to a tree structure* rather than a line. Indeed, at any point in time multiple VM instances can be created and then each one of them can be updated, different patches installed, and so on. This problem has serious implication on security as we shall see shortly.

What are the direct implications of virtualization on security? A first question is how can the support team deal with the consequences of an attack in a virtual environment. Do we expect the infection with a computer virus or a worm to be less manageable in a virtual environment? The surprising answers to these questions is that an infection may last indefinitely.

Some of the infected VMs may be dormant at the time when the measures to clean up the systems are taken. Then, at a later time, the infected VMs wake up and infect other systems. The scenario can repeat itself and guarantee that infection will last indefinitely. This is in stark contrast with the manner an infection is treated in non-virtual environments. Once an infection is detected, the infected systems are quarantined and then cleaned up; the systems will then behave normally until the next infection occurs.

A more general observation is that in a traditional computing environment a steady state can be reached. In this steady state all systems are brought up to a "desirable" state, whereas "undesirable" states, states when some of the systems are either infected by a virus or display an undesirable pattern of behavior, are only transient. The desirable state is reached by installing the latest system software version and then applying the latest patches to all systems.

A virtual environment may never reach such a steady state due to the lack of control. In a non-virtual environment the security can be compromised when an infected laptop is connected to the network protected by a firewall, or when a virus is brought in on a removable media. But, unlike a virtual environment, the system can still reach a steady state.

A side effect of the ability to record the complete state of a VM in a file is the possibility to roll back a VM. This opens wide the door for a new type of vulnerability caused by events recorded in the memory of an attacker. Two such situations are discussed in [185]. The first is that one-time passwords are transmitted in clear and the protection is guaranteed only if the attacker does not have the possibility to access passwords used in previous sessions.

An attacker can replay rolled back versions and access past sniffed passwords if a system runs the S/KEY password system. S/KEY is a password system based on Leslie Lamport's scheme. It is used by several operating systems including, Linux, OpenBSD, and NetBSD. The real password of the user is combined with a short set of characters and a counter that is decremented at each use to form a single-use password. The second situation is related to the requirement of some cryptographic protocols and even non-cryptographic protocols regarding the "freshness" of the random number source used for session keys and nonces. This situation occurs when a VM is rolled back to a state when a random number has been generated but not yet used.

A nonce is a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. For example, nonces are used to calculate the MD5 of the password for HTTP digest access authentication. Each time the authentication challenge response code is presented, the nonces are different, thus replay attacks are virtually impossible. This guarantees that an online order to Amazon or other online store cannot be replayed.

Even non-cryptographic use of random numbers may be affected by the rollback scenario. For example, the initial sequence number for a new TCP connection must be "fresh." The door to TCP hijacking is left open when the initial sequence number is not fresh.

Another undesirable effect of virtual environment affects trust. Recall from Section 11.4 that *trust is conditioned by the ability to guarantee the identity of entities involved.* Each computer system in a network has a unique physical, or MAC address. The uniqueness of the MAC address guarantees that an infected or a malicious system can be identified and then shut down, or denied network access. This process breaks down for virtual systems when VMs are created dynamically. Often, a random MAC address is assigned to a newly created VM to avoid name collision. The other effect discussed at length in Section 11.10 is that popular VM images are shared by many users.

The ability to guarantee confidentiality of sensitive data is yet another pillar of security affected by virtualization. Virtualization undermines the basic principle that time-sensitive data stored on any system should be reduced to a minimum. First, the owner has very limited control on where sensitive data is stored, it could be spread across many servers and may be left on some of them indefinitely.

A hypervisor records the state of a VM to be able to roll it back; this process allows an attacker to access sensitive data the owner attempted to destroy.

11.10 SECURITY RISKS POSED BY SHARED IMAGES

Even if we assume that a cloud service provider is trustworthy there are other sources of concern many users either ignore, or underestimate the danger they pose. One of them, especially critical for the IaaS cloud delivery model, is image sharing. For example, an AWS user has the option to choose between Amazon Machine Images (AMIs) accessible through the Quick Start or the Community AMI menus of the EC2 service. The option of using one of these AMIs is especially tempting for a first time user, or for a less sophisticated one.

First, we review the process to create an AMI. We can start from a running system, from another AMI, or from the image of a VM and copy the contents of the file system to the S3, a process called *bundling*. The first of the three steps of bundling is to create an image, the second step is to compress and encrypt the image, and the last step is to split the image into several segments and then upload the segments to S3.

Two procedures, *ec2-bundle-image* and *ec2-bundle-volume*, are used for creation of an AMI. The first is used for images prepared as loopback files[3] when data is transferred to the image in blocks. To bundle a running system the creator of the image can use the second procedure when bundling works at the level of the file system and files are copied recursively to the image.

To use an image, a user has to specify the resources, provide the credentials for login, a firewall configuration, and specify the region, as discussed in Section 2.3. Once the image is instantiated, the user is informed about the public DNS and the VM is available. A Linux system can be accessed using *ssh* at port 22, while the Remote Desktop at port 3389 is used for Windows.

A recent paper reports on the results of an analysis carried over a period of several months, from November 2010 to May 2011 of over five thousand AMIs available through the public catalog at Amazon [49]. Many analyzed images allowed a user to *undelete* files, recover credentials, private keys, or other types of sensitive information with little effort, using standard tools. The results of this study were shared with the Amazon's Security Team which acted promptly to reduce the threats posed to AWS users.

The details of the testing methodology can be found in [49], here we only discuss the results of this analysis. The study was able to audit some 5 303 images out of the 8 448 Linux AMIs and 1 202 Windows AMIs at Amazon sites in the US, Europe and Asia. The audit covered software vulnerabilities and security and privacy risks.

The average duration of an audit was 77 minutes for a Windows image and 21 minutes for a Linux image; the average disk space used was about 1 GB and 2.7 GB, respectively. The entire file system of a Windows AMI was audited because most of the malware targets Windows systems. Only

[3]A *loopback file system* (LOFS) is a virtual file system providing an alternate path to an existing file system. When other file systems are mounted onto an LOFS file system, the original file system does not change. One useful purpose of LOFS is to take a CDROM image file, a file of type ".iso" and mount it on the file system and then access it without the need to record a CD-R. It is somewhat equivalent to the Linux *mount-o loop* option, but adds a level of abstraction; most commands that apply to a device can be used to handle the mapped file.

directories containing executables for Linux AMIs were scanned; this strategy and the considerably longer start-up time of Windows explain the time discrepancy of the audits for the two types of AMIs.

The *software vulnerability* audit revealed that 98% of the Windows AMIs (249 out of 253) and 58% (2 005 out of 3 432) Linux AMIs audited had critical vulnerabilities. The average number of vulnerabilities per AMI were 46 for Windows and 11 for Linux AMIs. Some of AMI images were rather old; 145, 38, and 2 Windows AMIs and 1 197, 364, and 106 Linux AMIs were older than two, three, and four years, respectively. The tool used to detect vulnerabilities, the Nessus system, available from http://www.tenable.com/productus/nessus, classifies the vulnerabilities based on their severity in four groups, at levels zero to three. The audit reported only vulnerabilities of the highest severity level, e.g., remote code execution.

Three types of *security risks* are analyzed: (1) backdoors and leftover credentials, (2) unsolicited connections, and (3) malware. An astounding finding is that about 22% of the scanned Linux AMIs contained credentials allowing an intruder to remotely login to the system. Some 100 passwords, 995 ssh keys, and 90 cases when both could be retrieved were identified.

To rent a Linux AMI a user must provide the public part of the her ssh key and this key is stored in the *authorized_keys* in the home directory. This opens a backdoor for a malicious creator of an AMI who does not remove her own public key from the image and can remotely login to any instance of this AMI. Another backdoor is opened when the ssh server allows password-based authentication and the malicious creator of an AMI does not remove her own password. This backdoor is even wider open as one can extract the password hashes and then crack the passwords using a tool such as John the Riper, see http://www.openwall.com/john.

Another threat is posed by the omission of the *cloud-init* script that should be invoked when the image is booted. This script provided by Amazon regenerates the host key an ssh server uses to identify itself; the public part of this key is used to authenticate the server. When this key is shared among several systems these systems become vulnerable to man-in-the middle attacks.

An attacker impersonates the agents at both ends of a communication channel in the *man-in-the middle* attack and makes them believe that they communicate through a secure channel. For example, if B sends her public key to A, but C is able to intercept it, such an attack proceeds as follows: C sends a forged message to A claiming to be from B, but instead includes C's public key. Then A encrypts her message with C's key, believing that she is using B's key, and sends the encrypted message to B. The intruder, C, intercepts, deciphers the message using her private key, possibly alters the message, and re-encrypts with the public key B originally sent to A. When B receives the newly encrypted message, she believes it came from A.

When this script does not run, an attacker can use the *NMap* tool[4] to match the ssh keys discovered in the AMI images with the keys obtained with *NMap*. The study reports that the authors were able to identify more than 2 100 instances following this procedure.

Unsolicited connections pose a serious threat to a system. Outgoing connections allow an outside entity to receive privileged information, e.g., the IP address of an instance and events recorded by a

[4]*NMap* is a security tool running on most operating systems including Linux, Microsoft Windows, Solaris, HP-UX, SGI-IRIX and BSD variants such as Mac OS X to map the network. Mapping the network means to discover hosts and services in a network.

syslog daemon to files in the *var/log* directory of a Linux system. Such information is available only to users with administrative privileges.

The audit detected two Linux instances with modified *syslog* daemon which forwarded to an outside agent information about events such as login and incoming requests to a web server. Some of the unsolicited connections are legitimate, for example, connections to a software update site. It is next to impossible to distinguish legitimate from malicious connections.

Malware including viruses, worms, spyware, and Trojans were identified using ClamAV, a software tool with a database of some 850 000 malware signatures, available from http://www.clamav.net. Two infected Windows AMIs were discovered, one with a *Trojan-Spy* (variant 50112) and a second one with a *Trojan-Agent* (variant 173287). The first Trojan carries out keylogging, and allows stealing data from the files system and monitoring processes; the AMI also included a tool to decrypt and recover passwords stored by the Firefox browser, called *Trojan.Firepass*.

The creator of a shared AMI assumes some privacy risks. Her private keys, IP addresses, browser history, shell history, and deleted files can be recovered from the published images. A malicious agent can recover the AWS API keys which are not password protected. Then the malicious agent can start AMIs and run cloud applications at no cost to herself, as the computing charges are passed on to the owner of the API key. The search can target files with names such as *pk-[0-9A-Z]*.pem* or *cert-[0-9A-Z]*.pem* used to store API keys.

Another avenue for a malicious agent is to recover ssh keys stored in files named *id_dsa* and *id_rsa*. Though ssh keys can be protected by a passphrase, the audit determined that the majority of ssh keys (54 out of 56) were not password protected. A *passphrase* is a sequence of words used to control access to a computer system. A passphrase is the analog of a password, but provides added security. For high security non-military applications NIST recommends an 80-bit strength passphrase. Therefore, a secure passphrase should consist of at least 58 characters including uppercase and alphanumeric characters. The entropy of written English is less than 1.1 bits per character.

Recovery of IP addresses of other systems owned by the same user requires access to the *lastlog* or the *lastb* databases. The audit found 187 AMIs with a total of more than 66 000 entries in their *lastb* databases. Nine AMIs contained Firefox browser history and allowed the auditor identify the domains contacted by the user.

612 AMIs contained at least one shell history file. The audit analyzed 869 history files named *~/.history*, *~/.bash_history*, and *~/.sh_history* containing some 160 000 lines of command history and identified 74 identification credentials. The users should be aware that, when the HTTP protocol is used to transfer information from a user to a web site, the GET requests are stored in the logs of the web server. Passwords and credit card numbers communicated via a GET request can be exploited by a malicious agent with access to such logs. When remote credentials, such as the DNS management password are available then a malicious agent can redirect traffic from its original destination to her own system.

Recovery of deleted files containing sensitive information poses another risk for the provider of an image. When the sectors on the disk containing sensitive information are actually overwritten by another file, recovery of sensitive information is much harder. To be safe, the creator of the image effort should use utilities such as *shred, scrub, zerofree* or *wipe* to make recovery of sensitive information next to impossible. If the image is created with the block-level tool discussed at the beginning of this section the image will contain blocks of the file system marked as free; such blocks may contain information from deleted files. The audit process was able to recover files from 98% of the AMIs using the *exundelete* utility. The number of files recovered from an AMI were as low as 6 and as high as 40 000.

We conclude that the users of published AMIs as well as the providers of images may be vulnerable to a wide range of security risks and must be fully aware of the dangers posed by image sharing.

11.11 SECURITY RISKS POSED BY A MANAGEMENT OS

We often hear that virtualization enhances security because a VM monitor or hypervisor is considerably smaller than an operating system. For example, the Xen hypervisor discussed in Section 10.5 has approximately 60 000 lines of code, one to two orders of magnitude fewer than a traditional operating system.[5]

A hypervisor supports a stronger isolation between the VMs running under it than the isolation between processes supported by a traditional operating system. Yet the hypervisor must rely on a management OS to create VMs and to transfer data in and out from a guest VM to storage devices and network interfaces.

A small hypervisor can be carefully analyzed, thus one could conclude that the security risks in a virtual environment are diminished. We have to be cautious with such sweeping statements. Indeed, the Trusted Computer Base (TCB)[6] of a cloud computing environment includes not only the hypervisor but also the management OS. The management OS supports administrative tools, live migration, device drivers, and device emulators.

For example, the TCB of an environment based on Xen includes not only the hardware and the hypervisor, but also the management operating system running in Dom0, see Figure 11.4. System vulnerabilities can be introduced by both software components, Xen and the management operating system. An analysis of Xen vulnerabilities reports that 21 of the 23 attacks were against service components of the control VM [116]; 11 attacks were attributed to problems in the guest OS caused by buffer overflow and 8 were denial of service attacks. Buffer overflow allows execution of arbitrary code in the privileged mode.

Dom0 manages the building of all user domains (DomU), a process consisting of several steps:

1. Allocate memory in the Dom0 address space and load the kernel of the guest OS from secondary storage.
2. Allocate memory for the new VM and use foreign mapping to load the kernel to the new VM. The foreign mapping mechanism of Xen is used by Dom0 to map arbitrary memory frames of a VM into its page tables.
3. Set up the initial page tables for the new VM.
4. Release the foreign mapping on the new VM memory, set up the virtual CPU registers, and launch the new VM.

[5]The number of lines of code of the *Linux* operating system evolved in time from 176 250 for *Linux 1.0.0*, released in March 1995, to 1 800 847 for *Linux 2.2.0*, released in January 1999, 3 377 902 for *Linux 2.4.0*, released in January 2001, and to 5 929 913 for *Linux 2.6.0*, released in December 2003.

[6]The TCB is defined as the totality of protection mechanisms within a computer system including hardware, firmware, and, software. The combination of all these elements is responsible for enforcing a security policy.

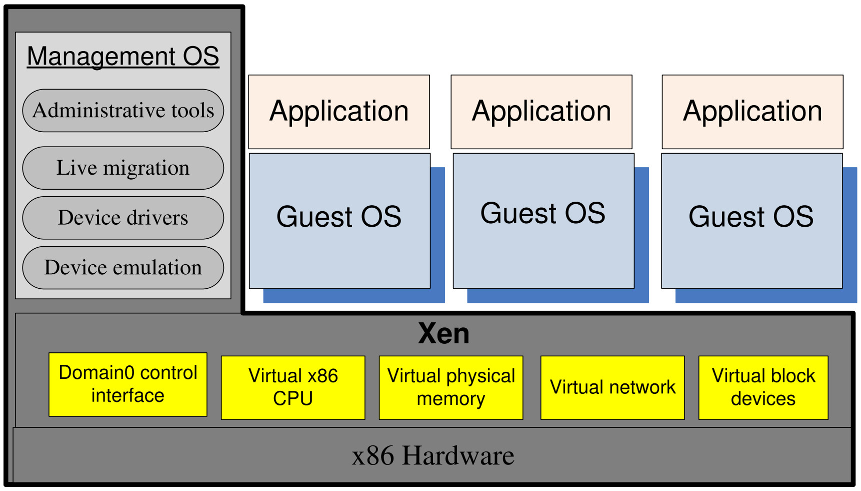


FIGURE 11.4

The trusted computing base of a Xen-based environment includes the hardware, Xen, and the management operating system running in Dom0. The management OS supports administrative tools, live migration, device drivers, and device emulators. A guest OS and applications running under it reside in a DomU.

A malicious Dom0 can play several nasty tricks at the time when it creates a DomU [302]:

- Refuse to carry out the steps necessary to start the new VM, an action that can be considered a *denial-of-service* attack.
- Modify the kernel of the guest OS in ways that will allow a third party to monitor and control the execution of applications running under the new VM.
- Undermine the integrity of the new VM by setting the wrong page tables and/or setup wrong virtual CPU registers.
- Refuse to release the foreign mapping and access the memory while the new VM is running.

We now turn our attention to the run-time interaction between Dom0 and a DomU. Recall that Dom0 exposes a set of abstract devices to the guest operating systems using *split drivers*; the frontend of such a driver is in the DomU and its backend in Dom0 and the two communicate via a ring in shared memory, see Section 10.5.

In the original implementation of Xen a service running in a DomU sends data to, or receives data from a client located outside the cloud using a network interface in Dom0; it transfers the data to I/O devices using a device driver in Dom0. Note that later implementations of Xen offer the pass-through option.

Therefore, we have to ensure that run-time communication through Dom0 is encrypted. Yet, Transport Layer Security (TLS) does not guarantee that Dom0 cannot extract cryptographic keys from the memory of the OS and applications running in DomU. A significant security weakness of Dom0 is that the entire state of the system is maintained by XenStore, see Section 10.5. A malicious VM can deny access to this critical element of the system to other VMs; it can also gain access to the memory of a DomU. This brings us to additional requirements for confidentiality and integrity imposed on Dom0.

Dom0 should be prohibited to use foreign mapping for sharing memory with a DomU, unless DomU initiates the procedure in response to a hypercall from Dom0. When this happens, Dom0 should be provided with an encrypted copy of the memory pages and of the virtual CPU registers. The entire process should be closely monitored by the hypervisor which, after the access, should check the integrity of the affected DomU.

A virtualization architecture that guarantees confidentiality, integrity, and availability for the TCB of a Xen-based system is presented in [302]. A secure environment when Dom0 cannot be trusted can only be ensured if the guest application is able to store, communicate and process data safely. The guest software should have access to a secure secondary storage on a remote storage server and to the network interfaces when communicating with the user. A secure run-time system is also needed.

To implement a secure run-time system we have to intercept and control the hypercalls used for communication between a Dom0 that cannot be trusted and a DomU we want to protect. Hypercalls issued by Dom0 that do not read from or write to the memory of a DomU or to its virtual registers should be allowed. Other hypercalls should be restricted either completely or during specific time window. For example, hypercalls used by Dom0 for debugging or for the control of the IOMMU should be prohibited. The Input/Output Memory Management Unit (IOMMU) connects the main memory with a DMA-capable I/O bus; it maps device-visible virtual addresses to physical memory addresses and provides memory protection from misbehaving devices.

We cannot restrict some of the hypercalls issued by Dom0, even though they can be harmful to the security of DomU. For example, foreign mapping and access to the virtual registers are needed to save and restore the state of DomU. We should check the integrity of DomU after the execution of such security-critical hypercalls.

New hypercalls are necessary to protect:

1. The privacy and integrity of the virtual CPU of a VM. When Dom0 wants to save the state of the VM the hypercall should be intercepted and the contents of the virtual CPU registers should be encrypted. The virtual CPU context should be decrypted and then an integrity check should be carried out when DomU is restored.
2. The privacy and integrity of the VM virtual memory. The *page table update* hypercall should be intercepted and the page should be encrypted so that Dome handles only encrypted pages of the VM. The hypervisor should calculate a hash of all the memory pages before they are saved by Dom0 to guarantee the integrity of the system. An address translation is necessary because a restored DomU may be allocated a different memory region [302].
3. The freshness of the virtual CPU and the memory of the VM. The solution is to add to the hash a version number.

As expected, the increased level of security and privacy leads to an increased overhead. Measurements reported in [302] show increases by a factor of: 1.7 to 2.3 for the domain build time, 1.3 to 1.5 for the domain save time, and 1.7 to 1.9 for the domain restore time.

11.12 XOAR – BREAKING THE MONOLITHIC DESIGN OF THE TCB

Xoar is a modified version of Xen designed to boost system security [116]. The security model of Xoar assumes that the system is professionally managed and that a privileged access to the system is granted only to system administrators. The model also assumes that administrators have neither financial incentives, nor the desire to violate the user's trust. Security threats come from a guest VM which could attempt to violate the data integrity or the confidentiality of another guest VM on the same platform, or to exploit the code of the guest. Another source of threats are bugs in initialization code of the management VM.

Xoar is based on microkernel[7] design principles. Xoar modularity makes exposure to risk explicit and allows the guests to configure the access to services based on their needs. Modularity allows the designers of Xoar to reduce the size of the permanent footprint of the system and increase the level of security of critical components. Ability to record a secure audit log is another critical function of a hypervisor facilitated by a modular design. The design goals of Xoar are:

- Maintain the functionality provided by Xen.
- Ensure transparency with existing management and VM interfaces.
- Tight control of privileges. Each component should only have the privileges required by its function.
- Minimize the interfaces of all components to reduce the possibility that a component can be used by an attacker.
- Eliminate sharing. Make sharing explicit, whenever it cannot be eliminated, to allow meaningful logging and auditing.
- Reduce the opportunity of an attack targeting a system component by limiting the time window when the component runs.

These design principles aim to break the monolithic TCB design of a Xen-based system. Inevitably, this strategy has an impact on performance, but the implementation should attempt to keep the modularization overhead to a minimum.

A close analysis shows that booting the system is a complex activity, but the fairly large modules used during booting are no longer needed once the system is up and running. In Section 10.5 we have seen that XenStore is a critical system component, as it maintains the state of the system thus, it is a prime candidate for hardening. The ToolStack is only used for management functions and can only be loaded upon request.

The Xoar system has four types of components: permanent, self-destructing, restarted upon request, and restarted on timer, see Figure 11.5:

1. Permanent components. XenStore-State maintains all information regarding the state of the system.
2. Components used to boot the system; they self-destruct before any user VM is started. The two components discover the hardware configuration of the server including the PCI drivers and then boot the system:

[7] A microkernel (μ-kernel) supports only the basic functionality of an OS kernel including low-level address space management, thread management, and inter-process communication. Traditional OS components such as device drivers, protocol stacks, and file systems are removed from the microkernel and run in user space.

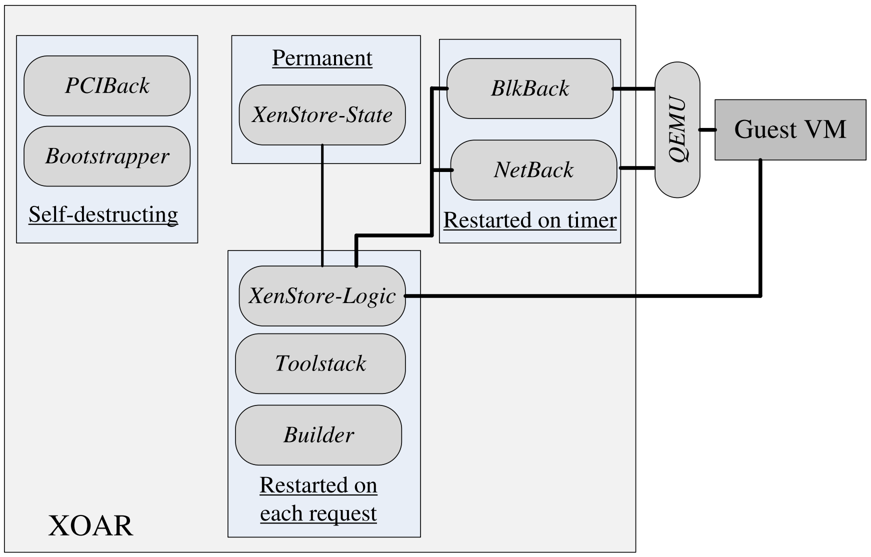


FIGURE 11.5

Xoar has nine classes of components of four types: permanent, self-destructing, restarted upon request, and restarted on timer. A guest VM is started using the Toolstack by the Builder and it is controlled by the XenStore-Logic. The devices used by the guest VM are emulated by the QEMU component.

- PCIBack – virtualizes access to PCI bus configuration.
- Bootstrapper – coordinates booting of the system.

3. Components restarted on each request:

   - XenStore-Logic
   - Toolstack – handles VM management requests, e.g., it requests the Builder to create a new guest VM in response to a user request.
   - Builder – initiates user VMs.

4. Components restarted on a timer: the two components export physical storage device drivers and the physical network driver to a guest VM.

   - BlkBack – exports physical storage device drivers using *udev*[8] rules.
   - NetBack – exports the physical network driver.

[8]*udev* is the device manager for the Linux kernel.

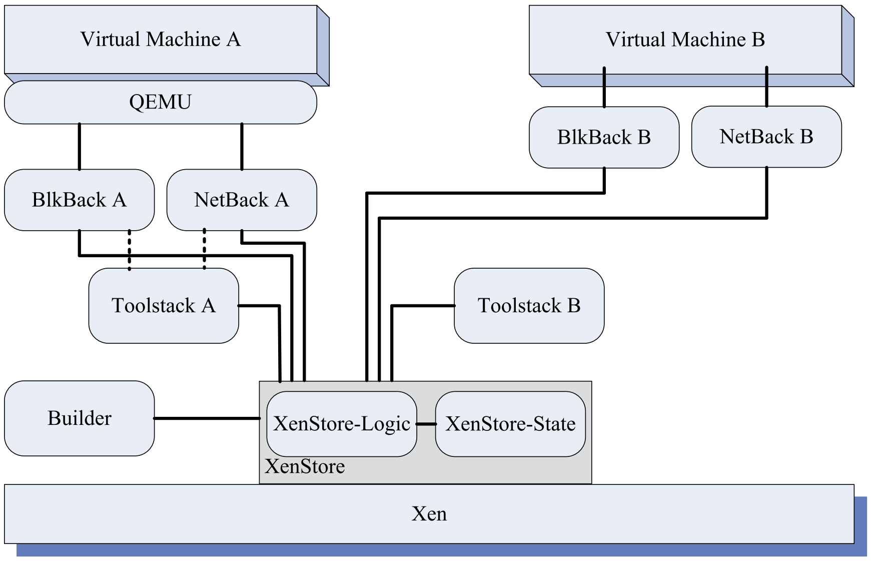


FIGURE 11.6

Component sharing between guest VM in Xoar. Two VM share only the *XenStore* components. Each one has a private version of the BlkBack, NetBack and Toolstack.

Another component, QEMU, is responsible for device emulation. Bootstrapper, PCIBack, and Builder are the most privileged components, but the first two are destroyed once Xoar is initialized. The Builder is very small, it consists of only 13 000 lines of code. XenStore is broken into two components, XenStore-Logic and XenStore-State. Access control checks are done by a small monitor module in XenStore-State. Guest VMs share only the Builder, XenStore-Logic, and XenStore-State, see Figure 11.6.

Users of Xoar are able to only share service VMs with guest VMs that they control; to do so they specify a tag on all of the devices of their hosted VMs. Auditing is more secure, whenever a VM is created, deleted, stopped, or restarted by Xoar the action is recorded in an append-only database on a different server accessible via a secure channel.

Rebooting provides the means to ensure that a VM is in a known good state. To reduce the overhead and the increased startup time demanded by a reboot, Xoar uses *snapshots* instead of rebooting. The service VM snapshots itself when it is ready to service a request. Similarly, snapshots of all components are taken immediately after their initialization and before they start interacting with other services or guest VMs. Snapshots are implemented using a copy-on-write mechanism[9] to preserve any page about to be modified.

[9]Copy-on-write (COW) is used by virtual memory operating systems to minimize the overhead of copying the virtual memory of a process when a process creates a copy of itself. Then the pages in memory that might be modified by the process or by its copy are marked as COW. When one process modifies the memory, the operating system's kernel intercepts the operation and copies the memory so that changes in one process's memory are not visible to the other.

11.13 A TRUSTED HYPERVISOR

After the discussion of Xoar we briefly analyze the design of a trusted hypervisor called *Terra* [184]. The novel ideas of this design are:

- A trusted hypervisor should support not only traditional operating systems, by exporting the hardware abstraction for open-box platforms, but also the abstractions for closed-box platforms discussed in Section 11.7. Note that the VM abstraction for a closed-box platform does not allow the contents of the system to be either manipulated or inspected by the platform owner.
- An application should be allowed to build its software stack based on its needs. Applications requiring a very high level of security, e.g., financial applications and electronic voting systems should run under a very thin OS supporting only the functionality required by the application and the ability to boot. At the other end of the spectrum are applications demanding low assurance, but a rich set of OS features. Such applications need a commodity operating system. *Information assurance* (IA) means to manage the risks related to the use, processing, storage, and transmission of information, as well as protecting the systems and processes used for those purposes. IA implies protection of the integrity, availability, authenticity, non-repudiation and confidentiality of the application data.
- Support additional capabilities to enhance system assurance:
  - Provide trusted paths from a user to an application. We have seen in Section 11.7 that such a path allows a human user to determine with certainty the identity of the VM it is interacting with and, at the same time, allows the VM to verify the identity of the human user.
  - Support attestation, the ability of an application running in a closed-box to gain trust from a remote party, by cryptographically identifying itself.
  - Provide air-tight isolation guarantees for the hypervisor by denying the platform administrator the root access.

The management VM is selected by the owner of the platform but makes a distinction between the *platform owner* and a *platform user*. The management VM formulates limits for the number of guest VMs running on the platform, denies access to the guest VM deemed unsuitable to run, grants access to I/O devices to running VMs and limits their CPU, memory, and disk usage.

Guest VMs expose a raw hardware interface including virtual network interfaces to virtual devices. The trusted hypervisor runs at the highest privilege level and it is secure even from the actions of the platform owner; it provides application developers with the semantics of a closed-box platform.

A significant challenge to the security of a trusted hypervisor comes from the device drivers used by different VMs running on the platform. Device drivers are large or very large software components, especially the drivers for high-end wireless cards and video cards. There is also a large variety of such drivers, many hastily written to accommodate new hardware features.

Typically, the device drivers are the lowest quality software components found in the kernel of an operating system thus, they pose the highest security risks. To protect a trusted hypervisor, the device drivers should not be allowed to access sensitive information and their memory access should be limited by different hardware protection mechanisms. Malicious I/O devices can use different hardware capabilities such as DMA to modify the kernel.

11.14 MOBILE DEVICES AND CLOUD SECURITY

Mobile devices are an integral part of the cloud ecosystem, mobile applications use cloud services to access and store data or to carry out a multitude of computational tasks. Security challenges for mobile devices common to all computer and communication systems include: (i) Confidentiality – ensure that transmitted and stored data cannot be read by unauthorized parties; (ii) Integrity – detect intentional or unintentional changes to transmitted and stored data; (iii) Availability – ensure that users can access cloud resources whenever needed; and (iv) Non-repudiation – the ability to ensure that a party to a contract cannot deny the sending of a message that they originated.

The technology stack of a mobile device consists of the hardware, the firmware, the operating system, and the applications. The separation between the firmware and the hardware of a mobile device is blurred. A baseband processor is used solely for telephony services involving data transfers over cellular networks operating outside the control of the mobile OS which runs on the application processor. Security-specific hardware and firmware store encryption keys, certificates, credentials, and other sensitive information on some mobile devices.

The nature of mobile devices places them at higher exposure to threats than stationary ones. Mobile devices are designed to easily install applications, to use third-party applications from application stores, and to communicate with computer clouds via often untrusted cellular and WiFi networks. Mobile devices interact frequently with other systems to exchange data and often use untrusted content.

Mobile devices often require a short authentication passcode and may not support strong storage encryption. Location services increase the risk of targeted attacks. Potential attackers are able to determine user's location, correlate the location with information from other sources on the individuals the user associates with, and infer other sensitive information.

Special precautions must then be taken due to exposure to the unique security threats affecting mobile devices, including:

1. Mobile malware.
2. Stolen data due to loss, theft, or disposal.
3. Unauthorized access.
4. Electronic eavesdropping.
5. Electronic tracking.
6. Access to data by third party applications.

Some of these threats can propagate to the cloud infrastructure a mobile device is connected to. For example, files stored on the mobile devices subject to ransomeware and encrypted by a malicious intruder can migrate to the backup stored on the cloud. The risks posed to the cloud infrastructure by mobile devices are centered around data leakage and compromise. Such security risks are due to a set of reasons including:

- Loss of the mobile device, lock screen protection, enabling smudge attacks and other causes leading to mobile access control. A smudge attack is a method to discern the password pattern of a touchscreen device such as a cell phone or tablet computer.
- Lack of confidentiality protection for data in transit in unsafe or untrusted WiFi or cellular networks.
- Unmatched firmware or software including operating system and application software bypassing the security architecture, e.g., rooted/jailbroken devices.
- Malicious mobile applications bypassing access control mechanisms.

- Misuse or misconfiguration of location services, such as GPS.
- Acceptance of fake mobility management profiles.

An in-depth discussion of the Enterprise Mobile Management (EMM) lists different EMM services including the Mobile Device Management (MDM) and the Mobile Application Management (MAM) and suggests a number of functional and security capabilities of the system [179]. Some of these policies and mechanisms should also be applied to mobile devices connected to computer cloud:

1. Use device encryption, application-level encryption, and remote wipe capabilities to protect storage.
2. Use Transport Layer Security (TLS) for all communication channels.
3. Isolate user-level applications from each other to prevent data leakage between applications using sandboxing.
4. Use device integrity checks for boot validation, verified application and OS updates.
5. Use auditing and logging.
6. Enforce authentication of the device owner.
7. Automatic, regular device integrity and compliance checks for threats and compliance.
8. Automated alerts for policy violations.

A system for Microsoft Outlook mobile application requires individuals who wish to participate in a managed scenario to download the Microsoft Community Portal application and input the required information including local authentication to the mobile OS via a lockscreen and the encryption capabilities provided by the mobile OS to protect data on the device. The cloud MDM portal discussed in [179] is available to administrators through a web interface.

11.15 FURTHER READINGS

The Cloud Security Alliance (CSA) is an organization with more than 100 corporate members. It aims to address all aspects of cloud security and serve as a cloud security standards incubator. The reports, available from the web site of the organization are periodically updated; the original report was published in 2009 [122] and subsequent reports followed [123] and [124]. An open security architecture is presented in [382].

A seminal paper [185] on the negative implications of virtualization on system security "When virtual is harder than real: security challenges in VM based computing environments" by Garfinkel and Rosenblum was published in 2005, followed by another one which reaches similar conclusions, [410]. Risk and trust are analyzed in [153], [252], and [317]. Cloud security is also discussed in [339], [468], and [474]. Managing cloud information leakage is the topic of [519].

A 2010 paper, [207] presents a taxonomy of attacks on computer clouds and [138] covers the management of security services lifecycle. Security issues vary depending on the cloud model as discussed in [377]. The privacy impact in cloud computing is the topic of [478]. A 2011 book [523] gives a comprehensive look at cloud security. Privacy and protection of personal data in the EU is discussed in a document available at http://ec.europa.eu/justice/policies/privacy.

The paper [40] analyzes the inadequacies of current risk controls for the cloud. Intercloud security is the theme of [63]. Secure collaborations are discussed in [66]. The paper [303] presents an approach for secure VM execution under untrusted management OS. The social impact of privacy in cloud computing is analyzed in [166]. An anonymous access control scheme is presented in [257].

An empirical study into the security exposure to hosts of hostile virtualized environments can be found at http://taviso.decsystem.org/virtsec.pdf. A model-based security testing approach for cloud computing is presented in [544]. Cold boot attacks on encryption keys are discussed in [217]. Cloud security concerns and mobile device security are covered in [370] and [371], respectively. [405] introduces an encryption system for query processing and [439] discusses a pragmatic security discipline.

11.16 EXERCISES AND PROBLEMS

Problem 1. Identify the main security threats for the SaaS cloud delivery model on a public cloud. Discuss the different aspects of these threats on a public cloud as compared to the threats posed to similar services provided by a traditional service-oriented architecture running on a private infrastructure.

Problem 2. Analyze how the six attack surfaces discussed in Section 11.2 and illustrated in Figure 11.1 apply to the SaaS, PaaS and IaaS cloud delivery models.

Problem 3. Analyze Amazon privacy policies and design a service level agreement you would sign on if you were to process confidential data using AWS.

Problem 4. Analyze the implications of the lack of trusted paths in commodity operating systems and give one or more examples showing the effects of this deficiency. Analyze the implications of the two-level security model of commodity operating systems.

Problem 5. Compare the benefits and the potential problems due to virtualization on public, private, and hybrid clouds.

Problem 6. Read [49] and discuss the measures taken by Amazon to address the problems posed by shared images available from AWS. Would it be useful to have a cloud service to analyze images and sign them before being listed and made available to the general public?

Problem 7. Analyze the risks posed by foreign mapping and the solution adopted by Xoar. What is the security risk posed by XenStore?

Problem 8. Read [116] and discuss the performance of the system. What obstacles to its adoption by the providers of IaaS services can you foresee?

Problem 9. Discuss the impact of international agreements regarding privacy laws on cloud computing.

Problem 10. Propagation of the malware in the Internet has similarities with the propagation of an infectious disease. Discuss the three models for the propagation of an infectious disease in a finite population, *SI, SIR*, and *SIS*. Justify the formulas describing the dynamics of the system for each model. Hint: read [76,161] and [267].

CHAPTER

BIG DATA, DATA STREAMING, AND THE MOBILE CLOUD 12

Advances in processor, storage, software, and networking technologies allow us to store and process massive amounts of data for the benefit of humanity, for profit, for entertainment, for nefarious schemes, or simply because we can. One can talk about "democratization of data," as scientists and decision makers, journalists and health care providers, artists and engineers, neophytes and domain experts attempt to extract knowledge from data.

This chapter covers three of the most exciting and demanding classes of cloud applications, Big Data, data streaming, and mobile cloud computing. Big Data is a reality, every day we generate 2.5 quintillion, 2.5×10^{18}, bytes of data.[1] This colossal volume of data is collected every day by devices ranging from inexpensive sensors in mobile phones to the detectors of the Large Hadron Collider, by online services offered by Google and Amazon, or by devices connected by the Internet of Things.

Big Data is a defining challenge for cloud computing; a significant amount of the data collected every day is stored and processed on computer clouds. Big Data and data streaming applications require low-latency, scalability, versatility, and a high degree of fault-tolerance. Achieving these qualities at scale is extremely challenging.

It makes sense to define a more comprehensive concept of scale for the applications discussed in this chapter. The "scale" in this context means millions of servers operating in concert in large data centers, a very large number of diverse applications, tens of millions of users, and a range of performance metrics reflecting the views of users on one hand and those of the CSPs, on the other hand. The scale has a *disruptive* effect, it changes how we design and engineer such systems and broadens the range of problems that can be solved and of applications that can only run on computer clouds.

The scale amplifies unanticipated benefits, as well as dreaded nightmares of system designers. Even a slight improvement of the individual server performance and/or of the algorithms for resource management could lead to huge cost savings and rave reviews. At the same time, the failure of one of the millions of hardware and software components can be amplified, can propagate throughout the entire system and have catastrophic consequences ultimately taking down the system.

Several important lessons when engineering large-scale systems for Big Data storage and processing are: (a) Prepare for the unexpected as low probability events occur and can cause major disruptions; (b) It is utterly unreasonable to assume that strict performance guarantees can be offered at scale; and (c) It is unfeasible to build fault-free systems beyond a certain level of system complexity; understanding this truth motivated the next best solution, the development of fault-tolerant system design principles.

[1]See the April 2015 report at http://www.vcloudnews.com/every-day-big-data-statistics-2-5-quintillion-bytes-of-data-created-daily/.

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00016-9

A realistic alternative for the applications discussed in this chapter is to develop tail-tolerant techniques for the distributions of performance metrics such as response time latency. This means to understand why a performance metric has a heavy tail distribution, detect the events leading to such an undesirable state as early as feasible, and take the necessary actions to limit its effects. Another design principle for a large-scale information retrieval system is that an approximate, but fast response, is preferable to a delayed best result.

The defining attributes of Big Data are analyzed in Section 12.1. The next sections discuss how Big Data is stored and processed. High capacity datastores and databases are necessary to store the very large volume of data. Scaling data warehouses and databases poses its own challenges. Mesa datastore and Spanner and F1 databases developed at Google are discussed in Section 12.2.

Many cloud applications process Big Data; data analytics is an important class of such applications. Bootstrapping techniques offer a low latency alternative for responding to queries of very large datasets. Such techniques and approximate query processing are analyzed in Sections 12.3 and 12.4, respectively. Finally, another class of Big Data applications combining mathematical modeling with simulation and measurements, the dynamic, data-driven applications are discussed in Section 12.5.

Computer clouds host many classes of data streaming applications, ranging from content delivery data streaming to applications consuming a continuous stream of events. Such applications are discussed in Sections 12.6, 12.8, and 12.7.

Scale changes everything in the realm of cloud computing. Scale makes it possible to add mission-critical applications demanding very high availability to the clouds, a topic discussed in Section 12.9. Scale amplifies variability often causing heavy-tail distributions of critical performance metrics including latency, discussed in Section 12.10.

Mobile devices such as smart phones, tablets, laptops, and wearable devices are ubiquitous and indispensable for life in a modern society. A great benefit of mobile devices is due to their symbiotic relationship with computer clouds. Mobile cloud users benefit from democratization of data processing. An individual with a mobile device has access to vast amounts of computing cycles and storage available on computer clouds.

Mobile cloud computing is positioned at the intersection of cloud computing, wireless networks, and mobile devices. Mobile devices are producers and consumers of cloud data. Sensores embedded in mobile devices generate data stored on the cloud and shared with others; applications running on mobile devices use cloud data and can trigger execution of cloud computations.

Section 12.11 is an introduction to mobile computing and its applications, while Section 12.12 covers energy efficiency of mobile computing. Section 12.13 analyzes the effects of latency and presents alternative mobile computing models including Cloudlets. The same theme continues with the discussion of the mobile edge clouds and Markov Decision Processes in Section 12.14.

12.1 BIG DATA

Some of the defining characteristics of Big Data are the three Vs, volume, velocity, and variety, as well as persistency. Volume is self-explanatory and velocity means that responses to queries and data analysis requests have to be provided very fast. Variety recognizes the wide range of data sources and data formats. Persistency means that data has a lasting value, it is not ephemeral.

Big Data covers a wide spectrum of data including user-generated content and machine-generated data. Some of the data is highly structured, as is the case of patient records in healthcare, insurance claims, or mortgage documents. Others are raw data from sensors, log files, or data generated by social media.

Big Data has affected the organization of database systems. The traditional relational databases are unable to satisfy some of these requirements and NoSQL databases proved to be better suited for many cloud applications. A database *schema* is a way to logically group objects such as tables, views, stored procedures etc. A schema can be viewed as a container of objects. One can assign a user login permission to a single schema so that the user can only access the objects they are authorized to.

For decades the database community used the *schema-on-write* approach. First, one defines a schema, then writes the data. When reading the data comes back according to the original schema. As an alternative, the *schema-on-read* loads the data as-is. Then, a user-defined filter is used to extract the data for processing. Schema-on-read has several advantages:

- Often data is a shared asset among individuals with differing roles and differing interests who want to get different insights from that data. Schema-on-read can present data in a schema that is best adapted to the queries being issued.
- When multiple datasets are consolidated it is not necessary to develop a super-schema that covers all of the datasets.

An insightful discussion of the state of research on databases [2] starts by acknowledging that "Big Data requirements will cause massive disruptions to the ways that we design, build, and deploy data management solutions." The report continues by identifying three major causes for these disruptions "...it has become much cheaper to generate a wide variety of data, due to inexpensive storage, sensors, smart devices, social software, multiplayer games, and the emerging IoT .... it has become much cheaper to process large amounts of data, due to advances in multicore CPUs, solid state storage, inexpensive cloud computing, and open source software .... not just database administrators and developers, but many more types of people have become intimately involved in the process of generating, processing, and consuming data...."

Big Data revolutionized computing. We have discussed in Chapter 7 specialized frameworks for Big Data processing such as MapReduce and Hadoop. The pleiad of system software components reviewed in Chapter 8 including Pig, Hive, Spark, and Impala are essential elements of a more effective infrastructure for processing unstructured or semi-structured data. The evidence of the effort to mold the cloud infrastructure to the requirements of Big Data is overwhelming. The success of these efforts is due to the fact that in spite of diversity Big Data workloads have a few common traits:

- Data is immutable, widely used storage systems for Big Data such as HDFS only allow *append* operations.
- Jobs such as MapReduce are deterministic therefore, fault-tolerance can be ensured by re-computations.
- The same operations are carried out on different segments of the data on different servers; replicating programs is less expensive than replicating data.

Table 12.1 Capacity and bandwidth of hard disks (HDD), solid-state disks (SDD), and memory of a modern server according to http://www.dell.com/us/business/p/servers. The network bandwidth is 1.25 GB/sec.

Media	Capacity (TB)	Bandwidth GB/sec
HDD (x12)	12–36	0.2–2
SDD (x4)	1–4	1–4
Memory	0.128–0.512	10–100

- It is feasible to identify a *working set*, a subset of data frequently used within a time window, and keep this working set in the memory of the servers of a large cluster. Systems such as Spark [542] and Tachyon [301] exploit this idea to dramatically improve performance.
- *Locality,* the proximity of the data to the location where it is processed, is critical for the performance. Supporting data locality is a main objective of scheduling algorithms including delay scheduling [541] and data-aware scheduling [499], as we have seen in Chapter 9.

The cloud hardware infrastructure also faces a number of challenges. Bridging the gap between the processor speed and the communication latency and bandwidth is a major challenge. This challenge is greatly amplified by response time constrains when Big Data is processed on the cloud. Addressing this challenge requires prompt adoption of faster networks such as InfiniBand and Myrinet, and of full bisection bandwidth networks between servers. Remote direct memory access capabilities can also help bridge this gap.

Non-volatile random-access memories, specialized processors such as GPUs and field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) contribute to the effort to develop a scalable infrastructure. The storage technology has dramatically improved, see Table 12.1.

Some of the enduring challenges posed by Big Data are:

1. Develop a scalable data infrastructures capable of responding promptly to timing constraints of an application.
2. Develop effective means of accommodating diversity in data management systems.
3. Support comprehensive end-to-end processing and knowledge extraction from data.
4. Develop convenient interfaces for a layman involved in data collection and analysis.

The next sections address the scalability challenges faced by the development of large data storage systems and by processing very large volumes of data.

12.2 DATA WAREHOUSES AND GOOGLE DATABASES FOR BIG DATA

Data warehouses and databases for cloud Big Data have been developed in recent years. A *data warehouse* is a central repository for the important data of an enterprise. A data warehouse is a core enterprise software component required by a range of applications related to the so-called *business intelligence*. Data from multiple operational systems are uploaded and used for predictive analysis, the detection of hidden patterns in the data. Data models developed using Statistical Learning Theory allow enterprises to optimize their operations and maximize their profits.

Scaling data repositories is very challenging due to the low-latency, versatility, availability, and fault-tolerance requirements. Google realized early on the need to develop a coherent storage architecture for the massive amounts of data required by their cloud services and by AdWords.[2] This storage architecture reflects the realization that "developers do not ask simple questions of the data, change their data access patterns frequently and use APIs that hide storage requests while expecting uniformity of performance, strong availability and consistent operations, and visibility into distributed storage requests" [173].

Two of the most popular Google data stores, BigTable and Megastore discussed in Sections 6.9 and 6.10, respectively, were designed and developed early on to support Google's cloud computing services. A major complaint about BigTable is the lack of support for cross-row transactions. Megastore supports schematized semi-relational tables and synchronous replication. At least 300 applications within Google use Megastore including Gmail, Picasa, Calendar, Android Market, and AppEngine.

From the experience with the BigTable storage system the developers of Google storage software stack learned that it is hard to share distributed storage. Another lesson is that distributed transactions are the only realistic option to guarantee the low latency requited for a high volume transaction processing system. They also learned that the end-user latency matters and that application complexity is reduced if an application is close to its data [173]. Another important lesson for developing a large-scale system is that making strong assumptions about applications is not advisable.

These lessons led to the development of Colossus, the successor of GFS as a next-generation cluster-level file system, of Mesa warehouse, and of Spanner and F1 databases discussed in this section. Colossus is built for real-time services and now supports virtually all web services, from Gmail, Google Docs, and YouTube, to Google Cloud Storage service offered to third-party developers. Colossus allows client-driven replication and encoding. Data is typically encoded using Reed–Solomon codes to reduce cost. The automatically shared metadata layer enables availability analysis.

Mesa – a scalable data warehouse. Mesa [212] is an example of a data warehouse designed to support measurement data for the multibillion advertising business of Google. The system is expected to support near-real-time data processing, be highly available, and scalable. The extremely high availability is ensured by geo-replication.[3]

Mesa is able to handle petabytes of data, respond to billions of queries accessing trillions of rows of data per day, and update millions of rows per second. The system complexity reflects the stringent requirements including the support for:

- Complex queries such as "How many ad clicks were there for a particular advertiser matching the keyword "fig" during the first week of December between 11:00 AM and 2:00 PM and displayed on google.com for users in a specific geographic location using a mobile device?" [212].
- Multi-dimensional data with two classes of attributes: *dimensional* attributes, called keys and *measure* attributes, called values.

[2]AdWords consists of hundreds of applications supporting Google's advertising services.

[3]The term *geo-replication* used in the title of the paper [212] means that the Mesa system runs at multiple sites concurrently. The term used in [213] for this strategy supporting high availability is *multi-homing*.

- Atomic updates, consistency and correctness. Multiple data views defined for different performance metrics are affected by a single user action and all must be consistent. The BigTable storage system does not support atomicity, while the Megastore system provides consistency across geo-replicated data, see Sections 6.9 and 6.10, respectively.
- Availability. Planned Mesa downtime is not allowed and unplanned downtime should never be experienced.
- Scalability. The system should accommodate a very large volume of data and a large user population.
- Near real-time performance. Users should be able to support live customer queries and reports and updates. It should allow queries to multiple data views from multiple data centers.
- Flexibility and ability to support new features.

Logical and physical data organization in Mesa. The system stores data using tables with very large key, K, and value, V, spaces. These spaces are represented by tuples of columns of items of identical data type, e.g., integers, strings, or floating point numbers. The data is horizontally partitioned and replicated.

The table structure and the *aggregation function* $F : V \times V \mapsto V$ are specified by a *table schema*. Function F is associative and often commutative. To maximize the throughput updates, each consisting of at most one aggregated value for every (table name, key) pair, are applied in batches.

The updates are applied by *version number* and are atomic, the next update can only be applied after the previous has finished. The time associated with a version is the time when the version was generated. A query has also a version number and a predicate P on key K.

A *delta* is a pre-aggregation of versioned data consisting of a set of rows corresponding to the set of keys for a range of versions $[V_1, V_2]$ with $V_1 \leq V_2$. Deltas can be aggregated by merging row keys and aggregating values accordingly e.g.,

$$[V_1, V_2] \; \& \; [V_2 + 1, V_3] \; \rightarrow \; [V_1, V_3]. \tag{12.1}$$

Mesa limits the time a version can be queried to reduce the amount of space. Older versions can be deleted and queries for such versions are rejected. For example, updates can be aggregated into base $B \geq 0$ with version $[0, B]$ and any updates with $[V_1, V_2]$ with $0 \leq V_1 \leq V_2 \leq B$ can be deleted.

Deltas are immutable and the rows in one are stored in sorted order in files of limited size. A row consists of multiple row blocks; each row block is transposed and compressed. Each table has one or more *table indexes* and each table index has a copy of the data sorted according to the index order.

Mesa instances. One Mesa instance runs at every site and consists of two subsystems, the *update/maintenance* and the *query* subsystem as shown in Figure 12.1. The pool of workers of the first subsystem operate on data stored in Colossus. The workers load updates, carry out table compaction, apply schema changes, and run table checksums under the supervision of controllers which determine the work to be done and manage the metadata stored on the BigTable.

To scale, the controller is *sharded* by table; recall that a shard is a horizontal partition of data in a data store and that each shard is held on separate physical storage device. The controller maintains separate queues of work for each type of worker and redistributes the workload of a slow worker to

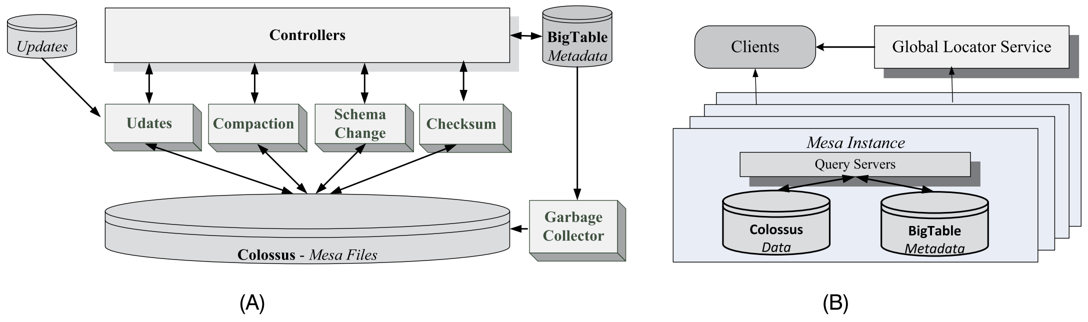


FIGURE 12.1

Mesa instance (A) The controller/worker subsystem. The controllers interact with four types of workers: update, compaction, schema change, and checksum. The data and the metadata are stored on Colossus and BigTable, respectively. (B) The query subsystem of a Mesa instance interacts with the clients and with the Global Location Service.

another worker in the same pool. The workers poll the controller for additional work when idle and notify the controller upon completion of a task. Work requiring global coordination, including schema change and checksums, is initiated by components outside the controller.

The query subsystem includes a pool of query servers which process client queries. A query server looks up the BigTable for the metadata, determines the files where data is stored, performs on-the-fly aggregation of the data, and converts the data from the internal format to the client protocol format. To reduce the access time and to optimize the system performance multiple queries acting upon the same tables are assigned to a group of servers.

Mesa instances are running at multiple sites to support a high level of availability. A *committer* coordinates updates at all sites. The committer assigns a new version number to batches of updates and publishes the metadata for the update to the *versions* database, a globally replicated and consistent data store using the Paxos algorithm. The controllers detect the availability of new updates by listening to the changes in the versions database and then assign the work to update workers.

Mesa reads 30 to 60 MB compressed data, adds some 3×10^5 new rows per second, and updates 3 to 6 million distinct rows. Each day it executes more than 500 million queries and returns $(1.7\text{–}3.2) \times 10^{12}$ rows. Updates arrive in batches about every five minutes, with median and 95th percentile commit times of 54 seconds and 211 seconds, respectively.

Spanner – a globally distributed database. Scaling traditional databases is not without major challenges. Spanner [119] is a distributed database replicating data on many sites across the globe. Some applications replicate their data across three to five data centers running Spanner in one geographic area. Other applications spread their data over a much larger area, e.g., F1 maintains five replicas of data around the US. The F1 system is presented later in this section.

Spanner has been used by many Google applications since its release in 2011. The first user of Spanner was the F1 system. The data model of each application using Spanner is layered on top of the directory-bucketed key-value mappings supported by the distributed database.

Spanner supports consistent backups, atomic schema updates, and consistent MapReduce execution and provides externally-consistent reads and writes, and globally-consistent reads across the database at a time stamp. This is possible because Spanner assigns globally-meaningful commit time stamps to transactions reflecting the serialization order, even though transactions may be distributed. Serialization order satisfies *external consistency*. This means that the commit time stamp of transaction T_1 is lower than that of transaction T_2 when T_1 commits before T_2 starts.

Spanner applications can control the number and the location of the data centers where the replicas reside, as well as the read and write latency. The read and write latencies are determined by how far the data is from its users and, respectively, how far are the replicas from one another. The system is organized in *zones*, units of administrative deployment and of physical isolation similar with AWS zones.

Spanner organization. In each zone a *zonemaster* is in charge of several thousand *spanservers*. The clients use *location proxies* to find the spanservers able to serve their data. A *placement driver* handles the migration of data across zones with latencies of minutes and the *universe master* maintains the state of all zones. A spanserver serves data to the clients. The spanserver implements a Paxos state machine per tablet, and manages 100–1 000 tablets. A *tablet* is a data structure implementing a bag of the mapping

$$(key : string, aimestamp : int64) \rightarrow string. \tag{12.2}$$

A great deal of attention is paid to replication and concurrency control. A set of replicas is called a *Paxos group*. The system administrators control the number and types of replicas and the geographic placement of them. Applications control the manner the data is replicated.

Each spanserver implements a *lock table* for concurrency control. Every replica has a leader. Every Paxos write is logged twice, once in the tablet's log and once in the Paxos log. The $(key, value)$ mapping state is stored in the tablet. The local spanserver software stack at each site includes a site *participant leader* communicating with its peers at other the sites. This leader controls a *transaction manager* and manages the lock table.

The system stores all tablets in Colossus. The implementation of the Paxos algorithm is optimized. The algorithm is pipelined to reduce latency. The leaders of the Paxos algorithm discussed in Section 3.12 are long lived, their life time is about 10 seconds.

Spanner *directories*, also called *buckets*, are sets of contiguous keys sharing a common prefix. A directory is the smallest data unit whose placement can be specified by an application. A background task called *moved* moves data directory-by-directory between the Paxos groups. This task is also used to add or remove replicas to/from Paxos groups.

Spanner transactions. The database supports read-write transactions, read-only transactions, and snapshot reads. A *read-write* transaction implements a standalone *write*; the concurrency control is pessimistic. A *read-only* transaction benefits from snapshot isolation,[4] it is lock-free, the *read* does not block incoming writes, it is executed at a system-chosen time stamp without locking. A *snapshot read*

[4]Snapshot isolation is a guarantee that all reads made in a transaction will see a consistent snapshot of the database.

Table 12.2 Spanner latency and throughput for write, read-only, and snapshot read. The mean and standard deviation over 10 runs as reported in [119].

	Latency in ms			Throughput in Kops/sec		
Replicas	**Write**	**Read-only**	**Snapshot read**	**Write**	**Read-only**	**Snapshot read**
1	14.4 ± 1.0	1.4 ± 0.1	1.3 ± 0.1	4.1 ± 0.5	10.9 ± 0.4	13.5 ± 0.1
3	13.9 ± 0.6	1.3 ± 0.1	1.2 ± 0.1	2.2 ± 0.5	13.8 ± 3.2	38.5 ± 0.3
5	14.4 ± 0.4	1.4 ± 0.05	1.3 ± 0.4	2.8 ± 0.3	25.3 ± 5.2	50.0 ± 1.1

is lock-free *read* in the past at a time stamp specified by the client or at a time stamp chosen by the system before a time stamp upper bound specified by the client.

The system supports atomic schema change transactions. The transaction is assigned a time stamp t in the future. The time stamp is registered during the prepare phase so that the schema changes on thousands of servers can complete with minimal disruption to other concurrent activity. Reads and writes, implicitly depending on the schema, are synchronize with any registered schema-change and may proceed if their time stamps precede time t, otherwise they must block until schema changes.

Some of the results of a micro-benchmark reported in [119] are shown in Table 12.2. The data was collected on timeshared systems with spanservers running in each zone on four-core AMD Barcelona 2200 MHz servers with 4 GB RAM. The two-phase commit scalability was also assessed: it increases from 17.0 ± 1.4 ms for one participant to 30.0 ± 3.7 for 10 participants, to 71.4 ± 7.6 for 100, and 150.5 ± 11.0 ms for 200 participants.

TrueTime. A two-phase locking is used for transactional reads and writes. An elaborate process for time stamp management based on the TrueTime is in place. The TrueTime API enables the system to support consistent backups, atomic schema updates, and other desirable features. This API represents time as a *TTinterval* with the starting and ending times of type *TTstamp*. A *TTinterval* has bounded time uncertainty. Three methods using t of type *TTstamp* as argument are supported

$TT.now()$ – returns a $TTinterval : [earliest, latest]$.

$TT.after(t)$ – returns *true* if time t has definitely past.

$TT.before(t)$ – returns *true* if time t has definitely not arrived.

TrueTime guarantees that for an invocation

$$tt = TT.now(), tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest \tag{12.3}$$

where e_{now} is the invocation event. Each data center has one *timemaster* and each server has its own *timeslave* daemon interacting with several time masters to reduce the probability of errors.

TrueTime guarantees correctness of concurrency control and supports externally consistent transactions, lock-free read-only transactions, and non-blocking reads in the past. It guarantees that a whole-database audit at time t sees the effects of every transaction committed up to time t. The results show that refining clock uncertainty associated with jitter or skew allows building distributed systems with much stronger time semantics.

F1 – scaling a traditional SQL database. A traditional database design shares the goals with the one discussed for data warehouses, i.e., scalability, availability, consistence, usability, and latency hiding. Google system developers realized that scaling up their sharded MySQL implementation supporting online transaction processing and online analytical processing systems was not feasible. Instead, a distributed SQL database called F1 was developed. F1 uses Spanner and stores its data on Colossus File System (CFS).

F1 database [451] was used since 2012 for the AdWords advertising ecosystem. F1 inherits Spanner's scalability, synchronous replication, strong consistency, and ordering properties and adds to them: (1) distributed SQL queries; (2) secondary indexes transactionally consistent; (3) asynchronous schema changes; (4) optimistic transactions, and (5) automatic change history recording and publishing. Users interract with F1 using a client library.

The organization of F1 is shown in Figure 12.2. F1 servers are co-located in each data center along with Spanner servers. Multiple Spanner instances run at each site along with multiple CFS instances. CFS is not a globally replicated service and Spanner instances communicate only with local CFS instances. Storing data locally reduces the latency. The system is scalable and its throughput can be increased by adding additional F1 and Spanner servers. The commit latencies are in the range 50–150 ms due to the synchronous data replication across multiple data centers.

F1 has a logical Relational Database Management System schema with some extensions including explicit table hierarchy and columns with Protocol Buffer data types. F1 stores each child table clustered with and interleaved within the rows from its parent table. Table columns contain structured data types based on the schema and binary encoding format of Google's open source Protocol Buffer library. F1 supports *non-blocking schema changes* in spite of several challenges including:

- The system scale.
- High availability and tight latency constraints.
- The requirement to continue queries and transaction processing while schema changes.
- It is impractical to require atomical schema updates for all servers as each F1 server has a copy of the schema in local memory for efficiency reasons.

The schema change algorithm requires that at most two different schema are active at any time; one can be the current schema, the other the next schema. A server cannot use a schema after its lease expires. A schema change is divided in multiple phases such that consecutive pairs of phases are mutually compatible.

F1 transactions. F1 supports ACID transactions discussed in Section 6.2 and required by systems such as financial systems and AdWords. The three types of F1 transactions built on top of Spanner transactions are:

1. Snapshot transactions – read-only transactions with snapshot semantics used by SQL queries and by MapReduce. Snapshot isolation is a guarantee that all reads made in a transaction see a consis-

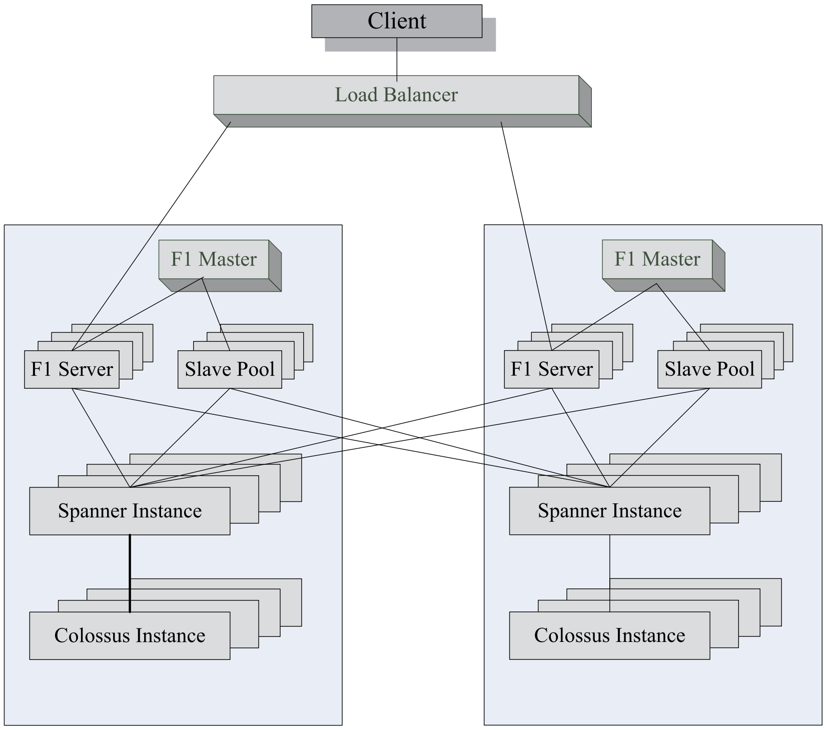


FIGURE 12.2

F1 architecture. A load balancer distributes the workload to F1 server instances at every site which, in turn, interact with Spanner instances at all sites where F1 is running. F1 data is stored on Colossus at the same site. The shared slave pool execute parts of distributed query plans on behalf of regular F1 servers. F1 master monitors the health of slave pool processes and communicates to F1 server the list of available slaves.

tent snapshot of the database. The transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.

2. Pessimistic transactions – map to the same Spanner transaction type.
3. Optimistic transactions – have an arbitrarily long lockless read phase followed by a short write phase and are the default transactions used by the clients.

Optimistic transactions have a number of benefits:

- Are long-lasting.
- Can be retried transparently by an F1 server.
- The state is kept on the client and is immune to server failure.
- The time stamp for a read can be used by a client for a speculative write that can only succeed if no other writes occurred after the read.

Optimistic transactions have two drawbacks, insertion phantoms and low throughput for high contention. Insertion phantoms occur when one transaction selects a set of rows, then another transaction inserts rows that meet the same criteria; then, different results are produced when the first transaction re-executes the query.

The default locking in F1 is at the row-level, though concurrency levels can be changed in the schema. By default tables are change-tracked, unless the schema shows that the some tables or columns have opted out. Every transaction creates one or more ChangeBatch Protocol Buffers, including the primary key and before and after values of changed columns for each updated row.

12.3 BOOTSTRAPPING TECHNIQUES FOR DATA ANALYTICS

The size of the datasets used for data analytics, as well as the complexity and the diversity of queries posed by impatient users, often without training in statistics, continually increase. In many instances, e.g., in case of exploratory queries, it is desirable to provide good enough, yet prompt answers, rather than perfect answers after a long delay.

This is only possible by limiting the search to a subset of data, but in such instances along with the answer the user expects an estimation of answer's quality. The quality of the answer is context-dependent and a general solution is far from trivial. Bootstrapping techniques discussed in this section can be applied to a broad range of applications for estimating the quality of such approximations. Given a set F and a random variable U, the bootstrapping methods are based on the *bootstrap substitution principle*

> To determine the probability distribution of $U \equiv u(Y, F)$ with $Y = \{Y_1, Y_2, \ldots, Y_n\}$ random samples of F, then F is replaced by a fitted model $\hat{F}$.

Thus, we make the approximation

$$Pr\{u(Y, F) \leq u \mid F\} \approx Pr\{u(Y^*, \hat{F}) \leq u\hat{F}\}. \tag{12.4}$$

The superscript $*$ distinguishes random variables and related quantities sampled from a probability model that has been fitted to the data. Sometimes, $u(Y, F) = T - \theta$ with T as the estimator of parameter $\theta \equiv t(F)$; more sophisticated cases involve transformations of T. Often $T = t(\tilde{F})$ and $\tilde{F}$ is an empirical distribution function of the data values.

While bootstrapping can produce reasonably accurate results, there are also instances when the results are unreliable. Some of such instances discussed in depth in [87] are:

1. Inconsistency of the bootstrap method when the model, the statistics, and the resampling fail to approximate the required properties, regardless of the sample size.
2. Incorrect resampling model in case of non-homogeneous data when the random variation of data is incorrectly modeled.

3. Non-linearity of statistics T as the good properties of the bootstrap are associated with an accurate linear approximation $T \approx \theta + n^{-1} \sum_i l(T_i)$ with $l(y)$ the influence function of $t(\cdot)$ at (y, F).

A key idea is to construct a proxy to the ground truth for a sample smaller than that of a fully observed dataset and compare the bootstrap's results with this proxy. The bootstrapping techniques discussed in this section are based on [273] which states "existing diagnostic methods target only specific bootstrap failure modes, are often brittle or difficult to apply, and generally lack substantive empirical evaluations... this paper presents a general bootstrap performance diagnostic which does not target any particular bootstrap failure mode but rather directly and automatically determines whether or not the bootstrap is performing satisfactorily."

The bootstrap method. Let P be an unknown distribution, $\theta(P)$ be some parameter of P, and $\mathcal{D}$ the set of n independent identically distributed (i.i.d.) sampled data points $\mathcal{D} = \{X_1, X_2, \ldots, X_n\}$ from P. Let $\mathbb{P} = n^{-1} \sum_{i=1}^{n} \delta_{X_i}$ be the empirical distribution of data. We wish to construct the estimate $\hat{\theta}(\mathcal{D})$ of $\theta(P)$ and then create $\xi(P, n)$, an assessment of the quality of $\hat{\theta}(\mathcal{D})$, consisting of a summary of the distribution Q_n of some quantity $u(\mathcal{D}, P)$.

Both P and Q_n are unknown thus, the estimate $\xi(P, n)$, called the *ground truth* in this discussion, cannot be computed directly, it can be approximated by $\xi(\mathbb{P}_n, n)$ using Monte Carlo procedure. The following steps are carried out repeatedly:

1. Form simulated datasets $\mathcal{D}^*$ of size n consisting of i.i.d. sampled points from $\mathbb{P}_n$;
2. Compute $u(\mathcal{D}^*, \mathbb{P}_n)$ for the simulated dataset $\mathcal{D}^*$;
3. Form the empirical distribution $\mathbb{Q}_n$ of the computed values of u;
4. Return the desired summary of this distribution.

The final bootstrap output will be a real-valued $\xi(\mathbb{Q}_n, n)$. The assessment $\xi(P, n)$ could compute:

1. The bias, the expectation of

$$u(\mathcal{D}, P) = \hat{\theta}(\mathcal{D}) - \theta(P). \tag{12.5}$$

2. A confidence interval based on the distribution of

$$u(\mathcal{D}, P) = n^{1/2}[\hat{\theta}(\mathcal{D}) - \theta(P)]. \tag{12.6}$$

3. Simply

$$u(\mathcal{D}, P) = \hat{\theta}(\mathcal{D}). \tag{12.7}$$

Given the estimator, the data generating distribution P, and n, the size of the data set, we want to determine if the output of the bootstrap procedure is *sufficiently close* to the ground truth. The formulation *sufficiently close* avoids a precise expression of accuracy giving the procedure a degree of generality and allowing its use for a range of applications with own accuracy requirements.

```
Pseudocode of the BPD Algorithm - Bootstrap Performance Diagnostic [273]

Input: $\mathcal{D}=\{X_1,\dots,X_n\}$: observed data
– $u$: quantity whose distribution is summarized to yield estimator quality assessments
– $\xi$: assessment of estimator quality
– $p$: number of disjoint subsamples used to compute ground truth approximations
– $b_1,\dots,b_k$: increasing sequence of subsample sizes for which ground truth approximations
–         are computed with $b_k \le \lfloor n/p \rfloor$ (e.g., $b_i = \lfloor n/(p2^{k-i}) \rfloor$ with $k=3$.
– $c_1 \ge 0$: tolerance for decreases in absolute relative deviation of mean bootstrap output
– $c_2 \ge 0$: tolerance for decreases in relative standard deviation of bootstrap output
– $c_3 \ge 0$, $\alpha \in [0,1]$: desired probability that bootstrap output at sample size $n$ has absolute
–   relative deviation from ground truth less than or equal to c3 (e.g., c3 = 0:5; = 0:95)
Output: true if the bootstrap is deemed to be performing satisfactorily, and false other-
wise
$\mathbb{P}_n \rightarrow n^{-1}\sum_{i=1}^{n}\delta_{X_i}$
for $i \leftarrow 1$ to $k$ do
–  $\mathcal{D}_{i1},\dots,\mathcal{D}_{ip} \rightarrow$ random disjoint subsets of $\mathcal{D}$, each containing $b_i$ data points
–  for $j \leftarrow 1$ to $p$ do
–       $u_{ij} \leftarrow u(\mathcal{D},\mathbb{P}_n)$
–       $\xi^*_{ij} \leftarrow bootstrap(\xi,u,b_i,\mathcal{D}_{ij})$
–  end
–  // Compute ground truth approximation for sample size $b_i$
–  $\mathbb{Q}_{b_i} \leftarrow \sum_{j=1}^{p}\delta_{u_{ij}}$
–  $\tilde{\xi}_i \leftarrow \xi(\mathbb{Q}_{b_i},b_i)$
–  // Compute absolute relative deviation of mean and relative standard deviation
–  // of bootstrap outputs for sample size $b_i$
–  $\Delta_i \leftarrow \mid \frac{mean\{\xi^*_{i1},\dots,\xi^*_{ip}\}-\tilde{\xi}_i}{\tilde{\xi}_i} \mid$     $\sigma_i \leftarrow \mid \frac{stddev\{\xi^*_{i1},\dots,\xi^*_{ip}\}-\tilde{\xi}_i}{\tilde{\xi}_i} \mid$
end
return true if all of the following hold, and false otherwise
```

$$\Delta_{i+1} < \Delta_i \quad \text{OR} \quad \Delta_{i+1} \le c_1, \forall i = 1,\dots,k \tag{12.8}$$

$$\sigma_{i+1} < \sigma_i \quad \text{OR} \quad \sigma_{i+1} \le c_2, \forall i = 1,\dots,k \tag{12.9}$$

$$\frac{\#\left(j \in \{1,\dots,p\}: \ \mid \frac{\tilde{\xi}^*_{kj}-\tilde{\xi}_k}{\tilde{\xi}_k} \mid \le c_3\right)}{p} \ge \alpha \tag{12.10}$$

In practice we can observe only one set with n data points rather than many independent sets of size n. The solution is to randomly sample $p \in \mathbb{N}$ disjoint subsets of the dataset $\mathcal{D}$, each of size $b \le \lfloor n/p \rfloor$. Then, to approximate the distribution of Q_b we use the set of values of u calculated for each subset. This distribution yields an approximation of the ground truth, $\xi(P,b)$ for datasets of size b. Then, to determine if the bootstrap performs as expected on sample size b, we run the bootstrap on each of the p subsets and compare the p bootstrap outputs to the ground truth.

Carrying out this procedure for a single sample size b is insufficient, the bootstrap performance may be acceptable for small sample size, but may get worse as the sample size increases or, conversely, be mediocre for small sizes, but improve as the sample size increases. It is thus, necessary to compare the distribution of bootstrap outputs for a range of sample sizes, $b_1,b_2,\dots,b_k$, $b_k \le \lfloor n/p \rfloor$. If the distribution of the bootstrap outputs converges monotonically for all smaller sample sizes $b_1, b_2, \dots, b_k$ we conclude that the bootstrap is performing satisfactorily for size n.

The convergence criteria are based on the relative deviation of the absolute value and the size of the standard deviation of the bootstrap output from the ground truth. The pseudocode of the Bootstrap Performance Diagnostic (BPD) algorithm illustrates these steps.

This algorithm generates a confidence interval with coverage $\alpha \in [0, 1]$.[5] A false positive, deciding that an approximation is satisfactory when it is not, is less desirable than a false negative, rejecting an approximation when in fact it is a satisfactory one. Equation (12.10) reflects this conservative approach. The absolute value of the deviation of a quantity γ from γ_0 is defined as $|\gamma - \gamma_0|/|\gamma_0|$ and an approximation is satisfactory if the output of the bootstrap run on a dataset of size n has an absolute value of the relative deviation from the ground truth of at most c_3 with a probability $\alpha \in [0, 1]$.

Choosing the sample sizes close together or choosing c_1 or c_2 too small will cause a larger number of false negatives. It is recommended to use an exponential distribution of sample sizes to ensure a meaningful comparisons of bootstrap performance for consecutive valuers b_i, b_{i+1} in the set $\{b_1, \dots, b_i, b_{i+1}, \dots, b_k\}$.

The process discussed in this section requires a substantial amount of data but this does not seem to be a problem in the age of Big Data. For example, according to [273] when $p = 100$ and $b_k = 1\,000$ then $n \geq 10^{15}$ and if b_k increases, $b_k = 10\,000$, $n \geq 10^{16}$. Processing such large data sets requires significant resources.

Simulation experiments for several distributions including *Normal(0; 1), Uniform(0; 10), StudentT(1.5), StudentT(3), Cauchy(0; 1), 0.95Normal(0; 1) + 0.05Cauchy(0; 1)*, and *0.99Normal(0; 1) + 0.01Cauchy(104; 1)* are reported in [273]. The results of these experiments show that the diagnostic performs well across a range of data generating distributions and estimators. Moreover, its performance improves with the size of the sample data sets.

In summary, given $\mathcal{S}$ a random sample from $\mathcal{D}$, the subsamples generated by disjoint partitions of $\mathcal{S}$ are also mutually independent random samples from $\mathcal{D}$. The procedure must be carried out using a sequence of samples of increasingly larger size, $b_1, \dots, b_i, \dots, b_k$. It is necessary to ensure that the error decreases while increasing the sample size and that the error is sufficiently small for the largest sample.

12.4 APPROXIMATE QUERY PROCESSING

The advantages of executing a query on a sample of data rather than on an entire dataset are quite perspicuous and, indeed, as early as 1970s this idea was applied to sampling relational databases. Different versions of this technique have been investigated since its first use. Approximate query processing and sampling-based approximate query processing have become popular enough to warrant the acronyms, AQP and S-AQP, respectively, along with the realization that approximate answers are most useful if accompanied by accuracy estimates.

[5]Confidence intervals offer a guarantee of the quality of a result. A procedure is said to generate confidence intervals with a specified coverage $\alpha \in [0, 1]$ if, on a proportion exactly α of the set of experiments, the procedure generates an interval that includes the answer. For example, a 95% confidence interval $[a, b]$ means that in 95% of the experiments the result will be in $[a, b]$.

Let θ be a query on a dataset $\mathcal{D}$ and the desired answer to it be $\theta(\mathcal{D})$. Simple random sampling with plug-in estimation is often used for approximate query processing. This method generates a sample with replacement $\mathcal{S} \subset \mathcal{D}$ of cardinality $n = |\mathcal{S}| \leq |\mathcal{D}|$ and produces a *sample estimate result* $\theta(\mathcal{S})$ instead of computing $\theta(\mathcal{D})$ with the *sampling error* $\epsilon = \theta(\mathcal{S} - \mathcal{D})$ and the *sampling error distribution* $Dist(\epsilon)$.

The emergence of Big Data processed on large clusters in the cloud and the requirement of near real-time response have increased the demand for high quality error estimates. Such error estimates can be reported to the users allowing them to judge the impact of errors on their specific applications and/or to application developers to decide if the sampling methods are adequate. These estimates can also be used to correlate errors with the sample size necessary for the accuracy-response time trade-offs.

Two methods for producing closed-form estimates for error bars[6] are based on the Central Limit Theorem (CLT)[7] and on Hoeffding bounds [215]. The derivation of Hoeffding bounds starts with the estimation of the mean

$$\mu = \frac{1}{m}\sum_{i=1}^{m} v(i) \tag{12.12}$$

with v a real-valued function defined on the set $\mathcal{S} = \{1, 2, \ldots, m\}$ and $m > 1$ a fixed integer. Let $L_1, L_2, \ldots, L_n$ and $L'_1, L'_2, \ldots, L'_n$ be random samples from the set $\mathcal{S}$ with and without replacement, respectively. Given $n > 1$ let $\bar{Y}_n$ and $\bar{Y}'_n$ be two estimators for μ and be defined as

$$\bar{Y}_n = \frac{1}{n}\sum_{i=1}^{n} v(L_i) \quad \text{and} \quad \bar{Y}'_n = \frac{1}{\min(n,m)} \sum_{i=1}^{\min(n,m)} v(L'_i). \tag{12.13}$$

These estimators are unbiased if $E[\bar{Y}_n] = E[\bar{Y}'_n]$. Hoeffding showed that for any $n \geq m$ and convex function f

$$E[f(\bar{Y}'_n)] \leq E[f(\bar{Y}_n)]. \tag{12.14}$$

It follows that when $f(x) = x^2 - \mu$

$$Var[f(\bar{Y}'_n)] \leq Var[f(\bar{Y}_n)]. \tag{12.15}$$

[6] An error bar is a line segment through a point on a graph, parallel to one of the axes, which represents the uncertainty or error of the corresponding coordinate of the point.

[7] Informally, CLT states that the sum of a large number of independent random variables has a normal distribution. More precisely, if $\{X_1, \ldots, X_n\}$ is a sequence of i.i.d. random variables with $E[X_i] = \mu$ and $Var[X_i] = \sigma^2 < \infty$ and $S_n = 1/n \sum_{i=1}^{n} X_i$ is the sample average then the random variables $\sqrt{n}(S_n - \mu)$ converge in distribution to a normal distribution, $N(0, \sigma^2)$ as n approaches infinity

$$\sqrt{n}\left[\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) - \mu\right] \xrightarrow{d} N(0, \sigma^2). \tag{12.11}$$

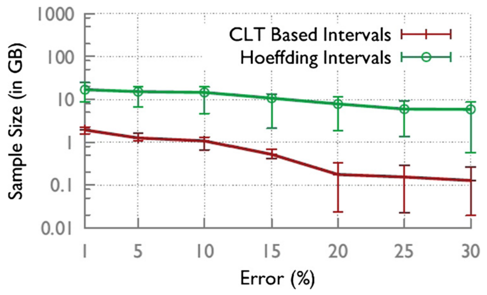


FIGURE 12.3

The sample size for CLT-based and Hoeffding-based estimation methods for the error bars [10].

These results are useful to bound the probability that the estimates deviate from μ more than a given amount. If the only information available before sampling is that

$$a \leq v(i) \leq b, \qquad 1 \leq i \leq m \tag{12.16}$$

Hoeffding established the following bounds for the estimation error

$$P\{|\bar{Y}_n - \mu| \geq t\} \leq 2e^{-2nt^2/(b-a)^2} \tag{12.17}$$

and

$$P\{|\bar{Y}'_n - \mu| \geq t\} \leq 2e^{-2n't^2/(b-a)^2} \tag{12.18}$$

for $t > 0$, $n \geq 1$ and

$$n' = \begin{cases} n & \text{if} \quad n < m; \\ +\infty & \text{if} \quad n \geq m. \end{cases} \tag{12.19}$$

The Hoeffding bounds overestimate the error and increase the computational effort. The sample sizes for a system using Hoeffding bounds are one to two orders of magnitude larger than for CLT-based or bootstrap-based methods but their accuracy is significantly higher. Experiments to estimate the sample sizes for achieving different levels of relative errors carried out on tens of terabytes of data are reported in [10] and reproduced in Figure 12.3.

Investigation of non-parametric bootstrap, closed-form estimation of variance, and large deviation bounds confirms that all techniques have different failure modes. A benchmark consisting of 100 different samples of some 10^6 rows reported by [10] shows that only queries with COUNT, SUM, AVG, and VARIANCE aggregates are amenable to closed-form error estimation. All aggregates are amenable to the bootstrap and 43.21% of the queries over one dataset and 62.79% of the queries over another can only be approximated using bootstrap-based error estimation methods. As expected, queries involving MAX and MIN are very sensitive to rare large or small values, respectively. In one dataset, these two functions involved 2.87% and 33.35% of all queries, respectively. Bootstrap error estimation fails for 86.17% of these queries.

The diagnostic algorithm discussed in Section 12.3 is computationally-intensive thus, impractical in many cases. A method to quickly determine if a technique will work well for a particular query based on symmetrically centered confidence intervals was proposed in [10]. The workflow for a large-scale distributed approximate query processing proposed has several steps:

1. Logical Plan (LP) – a query is compiled into an LP consisting of three procedure to compute: (1) an approximative answer $\theta(\mathcal{S})$; (2) the error $\bar{\xi}$; and (3) the diagnostic tests;
2. Physical Plan (PP) – initiates a DAG of tasks involving these procedures;
3. Data Storage Layer – distributes the samples to a set of servers and manages the cached data.

The system supports Poissonized resampling thus, allowing a straightforward implementation of the bootstrap error. For example, for a simple query of the form *SELECT foo(col_S) FROM $\mathcal{S}$* the bootstrap error is computed by using the BEC pseudocode shown below.

```
BEC: Bootstrap Error Computation for a SELECT query on 𝒮

SELECT foo(col_𝒮), ξ̂(resample_error) AS error
FROM (
.    SELECT  foo(col_𝒮)  AS  resample_answer
.    FROM   𝒮 TABLE_SAMPLE  POISSONIZED  (100)
.      UNION  ALL
.    SELECT  foo(col_𝒮)  AS  resample_answer
.    FROM   𝒮  TABLE_SAMPLE  POISSONIZED  (100)
.      UNION  ALL
....
.      UNION  ALL
.    SELECT  foo(col_𝒮)  AS  resample_answer
.    FROM  𝒮 TABLE_SAMPLE  POISSONIZED  (100)
)
```

An important source of inefficiency of the bootstrap method is the execution of the same query on different samples of the data. *Scan consolidation* can eliminate this source of inefficiency. As a first step of this process, the Logical Plan is optimized by extending the resampling operations. Each tuple in the sample $\mathcal{S}$ is extended with a set of 100 independent weights $w_1, w_2, \dots, w_{100}$ drawn from a Poisson distribution. The sample $\mathcal{S}$ is partitioned into multiple sets of $a = 50$, $b = 100$ and $c = 200$ MB, and three sets of weights, $D_{a1}, \dots, D_{a100}$, $D_{b1}, \dots, D_{b100}$ and $D_{c1}, \dots, D_{c100}$ are associated with each row to create 100 resamples of each set.

The logical plan is rewritten for a further optimization. After finding the longest set of consecutive operators that do not change the statistical properties of the set of columns that are being finally aggregated,[8] the custom Poissonized resampling operator is inserted right before the first non pass-through operator in the query graph. The subsequent aggregate operators are modified to compute a set of resample aggregates by appropriately scaling the corresponding aggregation column with the weights associated with every tuple. Further optimization of the cache management is used to improve the performance of the procedure discussed in [10].

[8] These so-called *pass-through operators* could be scans, filters, projections, and so on.

12.5 DYNAMIC DATA-DRIVEN APPLICATIONS

There are many applications combining mathematical modeling with simulation and measurements. Some of these applications involve Big Data. For example *large-scale-dynamic-data* refers to data captured by sensing instruments and control in engineered, natural, and societal systems. Some of the sensing instruments capture very large volumes of data; this is the case at the Large Hadron Collider at CERN, discussed in Section 4.12. A special case of such systems is discussed in this section, the Dynamic Data-Driven Application Systems (DDDAS). Such applications can benefit from dataflow architectures and programming models such as FreshBreeze developed at MIT by Professor Jack Dennis, [141,142,304].

Dynamic data-driven application systems. Efforts to analyze, understand, and predict the behavior of complex systems often require a dynamic feedback loop. Sometimes the simulation of a system uses measurement data to refine the system model through successive iterations, or to speedup the simulation. Alternatively, the accuracy of a measurement process can be iteratively improved by feeding the data to the system model and then using the results to control the measurement process. In all these cases the computational and the instrumentation have to work in concert. The mathematical modeling, the simulation algorithms, the control system, the sensors, and the computing infrastructure of a DDDAS system should support an optimal logical and physical dataflow through this feedback loop.

Some of the DDDAS applications discussed in http://www.1dddas.org/activities/2016-pi-meeting are: adaptive stream mining, modeling of nanoparticle self-assembly processes, dynamic integration of motion and neural data to capture human behavior, real-time assessment and control of electric microgrids, optimization, and health monitoring of large-scale structural systems.

The FreshBreeze multiprocessor architecture and FreshBreeze model of thread execution. A chip architecture guided by modular programming principles described in [141] is well suited for DDDAS applications as we shall see in the discussion of the Mahali project. The system is designed to satisfy the software design principles discussed in Section 4.7. One of the distinctive features of the architecture is to prohibit memory updates and use a cycle-free heap. Objects retrieved from memory are immutable and the allocation, release, and garbage collection of fixed-size chunks of memory are implemented by hardware mechanisms. This eliminates entirely the challenges posed by the cache coherence.

The organization of the system is depicted in Figure 12.4. The system consists of several multithreaded scalar processors (MTPs), a shared memory system (SMS) organized as a collection of fixed-size chunks of 1 024 bits, and an interconnection network. An MTP supports up to four execution threads involving integer and floating point operations. The MTPs communicate with Instruction Access Units (IAU) and Data Access Units (DAUs) to access chunks containing instructions and data, respectively. The access units have multiple slots of size equal to the size of memory chunks and maintain chunk usage data used by LRU algorithms for purging data to the SMS. The SMS performs automatically all memory updates including garbage collection.

An array is represented by a tree of chunks with element values at level 0. The number of tree levels is determined by the largest index value with a defined element, up to a maximum of eight levels. A collection of chunks containing pointers to other chunks forms a *heap*. The FreshBreeze execution model allows only the creation of *cycle-free* heaps. Many applications use also stream data types, unending series of values of uniform type. A Fresh Breeze stream is represented by a sequence of chunks, each chunk containing a group of stream elements. A chunk may include a reference to the

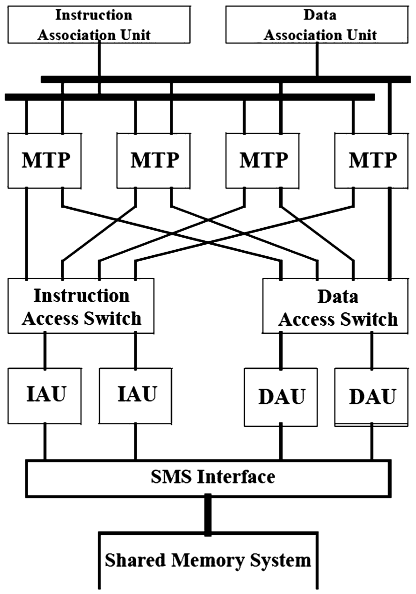


FIGURE 12.4

A FreshBreeze system consists of multithreaded processors (MTPs), shared memory system (SMS), instruction and data access units (IAU) and (DAUs), and instruction and data switches, [141].

chunk holding the next group of stream elements. As an alternative, an auxiliary chunk may contain "current" and "next" references.

The FreshBreeze execution model is discussed in [142]. A *master thread* spawns *slave threads* and initializes a *join point* providing the slave with a join ticket, similar to the return address of a method. Any slave thread may be a master for a group of slaves. The master does not continue after spawning the slaves and there is no interaction between the master and the slaves or among the slaves other than the contribution of each to the continuation thread. A program can generate an arbitrary hierarchy of concurrent threads corresponding to the available application parallelism.

A join point is a special entry in the local data segment of the master thread that provides space for a record of master thread status and for the result produced by the slave. Only one slave can be given a join ticket to the same join point to avoid race conditions. Several instructions are used to access a join point:

1. *Spawn* – sets a flag, stores a joint ticket in the slave's local data segment, and starts slave execution.
2. *EnterJoinResult* – allows the slave to save the result in the join point and then quit.
3. *ReadJoinValue* – returns the join value if it is available, or suspends the master if the join value has not yet been entered.

The operation of a thread is deterministic, any heap operation either reads data or creates private data. Operations at a join point are independent of the order of slave thread arrival.

The parallelization of a dot product of two vectors discussed next illustrates an application of the Fresh Breeze execution model [304]. First, the vectors are converted to tree-based memory chunks.

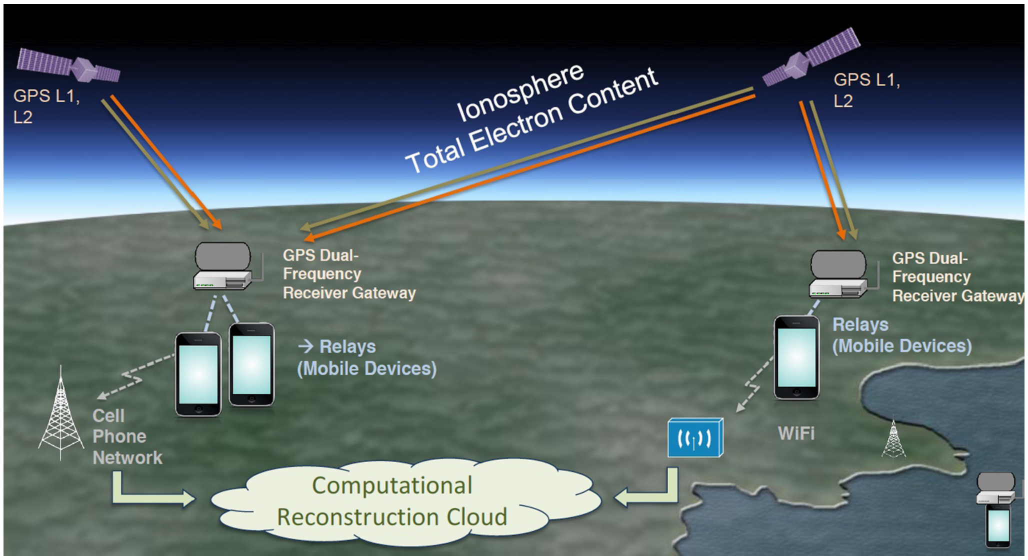

GPS L1, L2
Ionosphere Total Electron Content
GPS L1, L2
GPS Dual-Frequency Receiver Gateway
GPS Dual-Frequency Receiver Gateway
→ Relays (Mobile Devices)
Relays (Mobile Devices)
Cell Phone Network
WiFi
Computational Reconstruction Cloud

FIGURE 12.5

The Mahali system used entire ionosphere as sensor for ground-based and space-based phenomena and a multitude of mobile devices to gather the data and feed it to a cloud infrastructure https://mahali.mit.edu/.

The vectors are split into 16-element segments and organized as a tree structure. The leaf chunks hold the actual values, while the internal nodes of the tree store chunks holding handles of chunks that point to other chunks.

The *TraverseVector* thread takes the roots of two trees-of-chunks of vectors *A* and *B* as inputs and checks the depth of the tree to see if it is a leaf node. If not a leaf node then it recursively spawns *TraverseVector* threads taking the root handles of the next level as inputs. At the leaf level, a *Compute* thread is spawned to compute the dot product of 16 elements and return the result sum to the *Sync* chunk. The continuation *Reduce* thread adds all partial results in the *Sync* chunk filled by lower level *Compute/Reduce* threads. Then *Reduce* thread returns the handle of the *Sync* chunk to the upper level *Sync* chunk until reaching the root level *Sync* chunk as the final result.

Space weather monitoring. The Mahali[9] space weather monitoring project at MIT captures the quintessence of the DDDAS applications. The project uses multicore mobile devices, such as phones and tablets, to form a global space weather monitoring network. Multi-frequency GPS sensor data is processed in a cloud to reconstruct the structure of the space environment, and its dynamic changes see Figure 12.5.

The core ideas of the project are: (1) leverage entire ionosphere as a sensor for ground-based and space-based phenomena; and (2) take advantage of mobile technology as a game changer for observatories.

[9]"Kila Mahali" means "everywhere" in the Swahili language, see https://mahali.mit.edu/.

12.6 DATA STREAMING

Data streaming is the transfer of data at a steady high-speed rate, with low and well-controlled latency. In data streaming the volume of the data is very high and decisions have to be made in real-time. High-definition television (HDTV) is a ubiquitous application of data streaming.

Cloud data streaming services support content distribution from many organizations, e.g., AWS hosts Netflix and Google cloud hosts YouTube. Clouds support a variety of other data streaming applications in addition to hosting content providers. For example, AWS supports several streaming data platforms including Apache Kafka, Apache Flume, Apache Spark Streaming, and Apache Storm.

Data streaming versus batch processing. According to AWS: "Streaming data ..... is generated continuously by thousands of data sources, which typically send in the data records simultaneously, and in small sizes (order of Kilobytes). Streaming data includes a wide variety of data such as log files generated by customers using your mobile or web applications, ecommerce purchases, in-game player activity, information from social networks, financial trading floors, or geospatial services, and telemetry from connected devices or instrumentation in data centers. This data needs to be processed sequentially and incrementally on a record-by-record basis or over sliding time windows, and used for a wide variety of analytics including correlations, aggregations, filtering, and sampling."

There are important differences between streaming and batch data processing:

1. Streaming processes either individual records or micro batches, rather than large data batches.
2. Streaming processes only the most recent data or data over a rolling time window, rather than the entire, or a large segment of a data set.
3. Streaming requires latency of milliseconds, rather than minute or hours.
4. Streaming provides simple response functions, aggregates, and rolling metrics, rather than carrying out complex analytics.
5. It can be hard to reason about the global state in data streaming because different nodes may be processing data that arrived at different times, while in batch processing the system state is well defined and can be used to checkpoint and later restart the computation.

These differences suggest that the data streaming programming models and the APIs will be different from the ones for batch processing discussed in Chapters 7 and 8. It is therefore necessary to develop new data processing models for cloud data streaming services. Such models should be simple, yet effective.

Spark Streaming. Spark Streaming along with a data streaming model, called D-Streams, are discussed in [543]. The D-Streams model offers a high-level functional programming API, strong consistency, and efficient fault recovery. This model, prototyped as the Spark Streaming achieves fault tolerance by replication; there are two processing nodes, the upstream backup and a downstream node.

To address the latency concerns the Spark Streaming system relays on a storage abstraction, the *Resilient Distributed Dataset* (RDD) to rebuild lost data without replication. A *parallel recovery* mechanism involving all cluster nodes work in concert to reconstruct lost data. The system divides the time in short intervals, stores the input data received during each interval in RDD and then processes the data via deterministic parallel computations that may involve MapReduce and other frameworks. A D-Stream is a collection of RDDs.

Spark Streaming provides an API similar to DryadLINQ in the Scala language and supports stateless operations acting independently in each time interval, as well as aggregation over time window.

To recover in case of node failures D-Streams and RDDs track their lineage using a dependency graph. The system supports:

- Stateless transformations available in batch frameworks such as *map, reduce, groupBy*, and *join* and provides two operators:
  1. Stateless and statefull transform operators; the first class of operators act independently on each time interval and the second share data among intervals.
  2. Output operators that save data, e.g., store RDDs on HDFS.
- New statefull operations:
  1. Windowing; a window groups records from a range of past intervals into one RDD.
  2. Incremental aggregation over a window.
  3. Time-skewed joins when a stream is combined with RDDs.

Spark Streaming uses an inefficient upstream backup approach, does not support finer checkpointing as it creates checkpoints every minute, and depends on application idempotency and system slack for recovery.

Zeitgeist and MillWheel. Google's Zeitgeist,[10] a system used to track trends in web queries, is a typical application of data streaming. The system builds a historical model of each query and continually identifies queries that spike or dip. The Zeitgeist processing pipeline includes a window counter with query searches as input, a model calculator and a spike/dip detector, and an anomaly notification engine. The system buckets records arriving in one-second intervals and then compares the actual traffic for each time bucket to the expected traffic predicted by the model. An example of Zeitgeist aggregation of one-second buckets of (key, value, time stamp) query triplets given in [12] is shown in Figure 12.6.

The MillWheel framework developed at Google for building fault-tolerant and scalable data streaming systems supports persistent state [12] and addresses some of the limitations of Spark Streaming and other data streaming systems. The Zeitgeist system provided the initial requirements for the MillWheel stream processing services. Some of these requirements are:

- Immediate availability of data for services processing data.
- Exposure of persistent state abstraction to user applications.
- Graceful handling of out-of-order data.
- Exactly-once delivery of records.
- Constant latency as the system scales up.
- Monotonically increasing low watermarking of data time stamps.

MillWheel users express the logic of their data streaming applications as a directed graph where the stream of data records is delivered along the graph edges. A node or an edge in the arbitrary topology can fail at any time without affecting the correctness of the result. Record delivery is idempotent.[11]

[10]Zeitgeist is a German word literary translated as *time mind* and used with the sense of *the spirit of the time*. This concept, attributed to Georg Wilhelm Friedrich Hegel, an important figure of the German idealism philosophy school, reflects the dominant ideas in the society at a particular time.

[11]An idempotent operation can be carried out repeatedly without affecting the results.

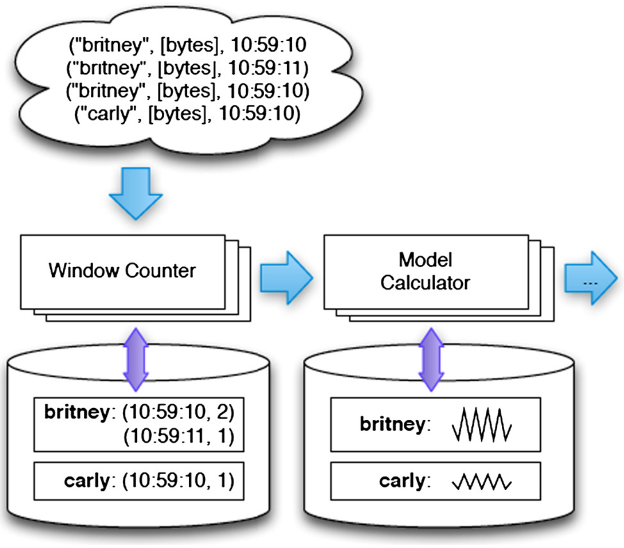

("britney", [bytes], 10:59:10
("britney", [bytes], 10:59:11)
("britney", [bytes], 10:59:10)
("carly", [bytes], 10:59:10)
Window Counter
Model Calculator
...
britney: (10:59:10, 2)
(10:59:11, 1)
carly: (10:59:10, 1)
britney:
carly:

FIGURE 12.6

Zeitgeist aggregation of one-second buckets of (key, value, time stamp) query triplets [12].

The data structures used by MillWheel as input and output are (key, value, time stamp) triplets. The *key* and the *value* can be any string. The *time stamp* is typically close to the wall time of the event, though it can be assigned an arbitrary value by MillWheel. Input data trigger computations invoking either user-defined actions, or MillWheel primitives.

The key extraction function uses a user-assigned key for the aggregation and the comparison among records. For applications such as Zeitgeist the key could be the text of the query. MillWheel uses the concept of *low water mark* to bound the time stamp of future records. Waiting for the low water mark allows computations to reflect a more complete picture of the data. *Timers* are programmatic hooks that trigger at a specific wall time or at a low watermark value for a particular key.

A user-defined computation subscribes to input streams and publishes output streams. The system guarantees delivery of both streams. Computation is run in the context of a specific key. The processing steps for an incoming record are:

- Check for duplication and discard a previously seen record.
- Run user code and identify pending changes to the timers, state, and production.
- Commit pending changes to backing store.
- Acknowledge senders.
- Send pending downstream productions.

There are two types of productions, strong and weak. Strong productions support handling of inputs that are not necessarily ordered or deterministic. Checkpointing of *strong productions* is done before delivery as a single atomic write; the state modification is done at the same time with the atomic write.

Weak productions are generated when the application semantics allows the user to disable the strong production option.

The computations are pipelined and may run on different hosts of the MillWheel cluster. To record the persistent state the system uses either BigTable or other databases supporting single-row updates. A replicated master balances the load and distributes it based on lexicographic analysis of the record keys.

Measurements conducted on the system report low latency and scalability, well within the targets set for stream processing at Google. A median record delay of 3.6 milliseconds and a 95 percentile latency of 30 milliseconds are reported for an experiment on 200 CPUs.

MillWheel is not suitable for monolithic computations whose checkpointing would interfere with the dynamic load balancing necessary to ensure low latency.

Caching strategies for data streaming. Several replacement policies are used by caching routers of a Content-Centric Network (CCN) discussed in Section 5.12 and used for data streaming. LRU (Least Recently Used) policy keeps in cache the most recently used content. LFU (Least Frequently Used) uses the cache request history to keep in cache the highly used content.

There are obvious limitations of both LRU and LFU. The former ignores the popularity of an item, the later ignores the history, e.g., items highly used in the past are kept in cache in spite of a new patterns of access. It should be no surprise that combinations of the two, the so called Least Recently/Frequently Used replacement strategies support trade-offs by specifying a weight that decays exponentially over time for each request.

The sLRFU (streaming Least Recently/Frequently Used) introduced in [425] aims to maximize cache. The novelties introduced by sLRFU are:

1. Partitions a cache of size C into an LFU-managed area using a sliding window of size k and an LRU-managed area of size $C - k$.
2. Estimates the top-k most popular data item in a sliding window of requests, keeps the frequently used ones in cache, and discards the old ones. k is set dynamically based on bounds given by the streaming algorithm.
3. For every request

    - Increases the reference counter for the data item referenced by the request and decreases the counter for the request falling out of the window.
    - If the data requested is not in cache it recovers the data and delivers it.
    - Using the last N requests, possibly rearranges the top-k most popular elements in cache (adding and/or removing content from the list), and complements the available spaces in cache with the $C - k$ most recently requested elements that are not already in cache.

Simulation results reported in [425] show that sLRFU has a hit rate of 70% compared with a baseline LRU of 65%.

12.7 A DATAFLOW MODEL FOR DATA STREAMING

There should be no surprise that a more sophisticated programming model for data streaming has been developed at Google. Alphabet, the holding company consisting of Google's core businesses and future-looking and finance stuff, earns billions from advertising embedded in data streaming. That's why Goggle is interested to support an effective computational model for event-time correlations. The Dataflow SDK model described in [13] is based on MillWheel [12] and FlumeJava [92].

Instead of the widely used terminology distinguishing batch processing from streaming, the two types of data processing are identified by a characterization of the input data: *bounded* for batch processing, and *unbounded* for streaming. The implications of this dichotomy are clear and reflect what the execution engine is expected to do. In the bounded case the engine processes a data set of known contents and size. In the unbounded case the engine processes a dynamic data set where one never knows if the set is complete, as new records are continually added and old ones are retracted.

The Dataflow SDK motivation for developing a new model is that grooming unbounded data sets to look as bounded ones does not allow an optimal balance between correctness, latency, and cost. It is thus necessary to refresh, simplify, and made flexible the programming model for unbounded datasets. The shortcomings of previous models are perfectly well illustrated by Google applications. For example, assume that a streaming video provider wants to bill advertisers placing their adds along with the video stream for the amount of advertising watched. This requires the ability to identify who and for how long watched a video stream and each add.

Some of the existing systems do not provide exactly-once semantics, do not scale well, are not fault-tolerant, have high latency, while others, such as MillWheel or Spark Streaming, do not have a high-level programming support for event-time correlations. The model introduced in [13] "Allows for the calculation of event-time ordered results, windowed by features of the data themselves, over an unbounded, unordered data source, with correctness, latency, and cost tunable across a broad spectrum of combinations."

It also "Decomposes pipeline implementation across four related dimensions, providing clarity, composability, and flexibility: *What* results are being computed. *Where* in event time they are being computed. *When* in processing time they are materialized. *How* earlier results relate to later refinements." The design of the system was guided by several principles:

- Never rely on the notion of completeness.
- Encourage clarity of implementation.
- Support data analysis in the context it was collected.

The new model uses *windowing* to split a dataset into groups of events to be processed together, an essential concept for unbounded data processing. Windowing is almost always time-based and the windows can be static/fixed or sliding. A *sliding* window has a size and a sliding period, e.g., a 2-hour window starting every 10 minutes. *Sessions* are sets of windows related by a data attribute, e.g., key for key-value datasets.

An *event* causes the change of the system state, for example, the arrival of a new record in case of data streaming. Typically the *time skew*, the difference between the event time and the event processing time, varies during processing. The distinction between the event time and the processing time of the event is illustrated in Figure 12.7.

The Dataflow SDK uses the *ParDo* and *GroupByKey* operations of FlumeJava discussed in Section 3.14 and defines a new operation *GroupByKeyAndWindow*. The entities flowing through the pipeline are four-tuples $(key, value, eventtime, window)$. Several operations involving windows are de-

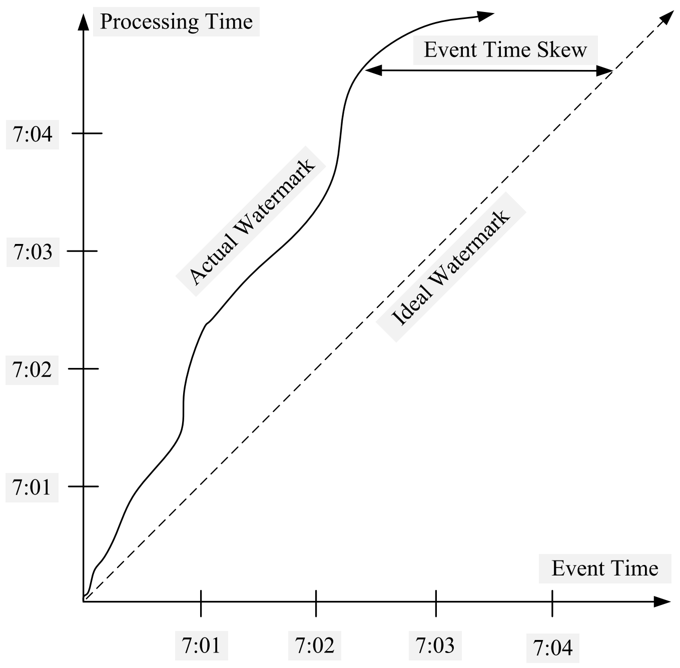


FIGURE 12.7

The event time skew. The *event time* is the wall clock time when the event occurred; this time never changes. The *event processing time* is the time when the event was observed during the processing pipeline; this time changes as the event flows through the processing pipeline. For example, an event occurring at 7:01 is processed at 7:02.

fined. *Window assignment* replicates an object in every window, the $(key, value)$ pairs are duplicated in windows overlapping the time stamp of an event.

Window merging is a more complex operation involving the six steps in Figure 12.8 where the window size is 30 minutes and there are four events, three with key k_1 and one with key k_2. In Step 2 the four-tuples (*key*; *value*; *event time*; *window*) are transformed as three-tuples (*key*; *value*; *window*). The *GroupByKey* combines the three events with k_1 in Step 3. Overlapping windows $[13:02, 13:32]$ and $[13:20, 13:50]$ for k_1 are merged as $[13:02, 13:50]$ in Step 4 of the operation. Then in Step 5 the two k_1 events with values v_1 and v_4 in the same window $[13:02, 13:50]$ are grouped together. Finally, in Step 6 time stamps are added.

Watermarks used in MillWheel to trigger processing of the events in a window cannot ensure correctness by themselves, sometimes late data is missed. At the same time, a watermark may delay pipeline processing. The Dataflow system uses *triggers* to determine the time when the results of groupings are emitted as panes.[12]

[12]A pane is a well defined area within a window for the display of, or interaction with, a part of that window's application or output.

$$\begin{array}{c}(k_1, v_1, 13{:}02, [0, \infty)),\\ (k_2, v_2, 13{:}14, [0, \infty)),\\ (k_1, v_3, 13{:}57, [0, \infty)),\\ (k_1, v_4, 13{:}20, [0, \infty))\end{array}$$

$$\Big\downarrow \; \begin{array}{l} AssignWindows(\\ \quad Sessions(30m)) \end{array}$$

$$\begin{array}{c}(k_1, v_1, 13{:}02, [13{:}02, 13{:}32)),\\ (k_2, v_2, 13{:}14, [13{:}14, 13{:}44)),\\ (k_1, v_3, 13{:}57, [13{:}57, 14{:}27)),\\ (k_1, v_4, 13{:}20, [13{:}20, 13{:}50))\end{array}$$

$$\downarrow \; DropTimestamps$$

$$\begin{array}{c}(k_1, v_1, [13{:}02, 13{:}32)),\\ (k_2, v_2, [13{:}14, 13{:}44)),\\ (k_1, v_3, [13{:}57, 14{:}27)),\\ (k_1, v_4, [13{:}20, 13{:}50))\end{array}$$

$$\downarrow \; GroupByKey$$

$$\begin{array}{c}(k_1, [(v_1, [13{:}02, 13{:}32)),\\ (v_3, [13{:}57, 14{:}27)),\\ (v_4, [13{:}20, 13{:}50))]),\\ (k_2, [(v_2, [13{:}14, 13{:}44))])\end{array}$$

$$\Big\downarrow \; \begin{array}{l} MergeWindows(\\ \quad Sessions(30m)) \end{array}$$

$$\begin{array}{c}(k_1, [(v_1, [\mathbf{13{:}02, 13{:}50})),\\ (v_3, [13{:}57, 14{:}27)),\\ (v_4, [\mathbf{13{:}02, 13{:}50}))]),\\ (k_2, [(v_2, [13{:}14, 13{:}44))])\end{array}$$

$$\downarrow \; GroupAlsoByWindow$$

$$\begin{array}{c}(k_1, [([\mathbf{v_1, v_4}], [13{:}02, 13{:}50)),\\ ([\mathbf{v_3}], [13{:}57, 14{:}27))]),\\ (k_2, [([\mathbf{v_2}], [13{:}14, 13{:}44))])\end{array}$$

$$\downarrow \; ExpandToElements$$

$$\begin{array}{c}(k_1, [v_1, v_4], \mathbf{13{:}50}, [13{:}02, 13{:}50)),\\ (k_1, [v_3], \mathbf{14{:}27}, [13{:}57, 14{:}27)),\\ (k_2, [v_2], \mathbf{13{:}44}, [13{:}14, 13{:}44))\end{array}$$

FIGURE 12.8

Window merging in Dataflow SDK involves six steps [13]. (1) AssignWindow; (2) Drop the time stamps; (3) GroupByKey; (4) MergeWindows – based on windowing strategy; (5) GroupAlsoByWindow – for each key group values by window; and (6) ExpandToElements – expand per-key, per-window groups of values into (*key*; *value*; *event time*; *window*) tuples, with new per-window time stamps.

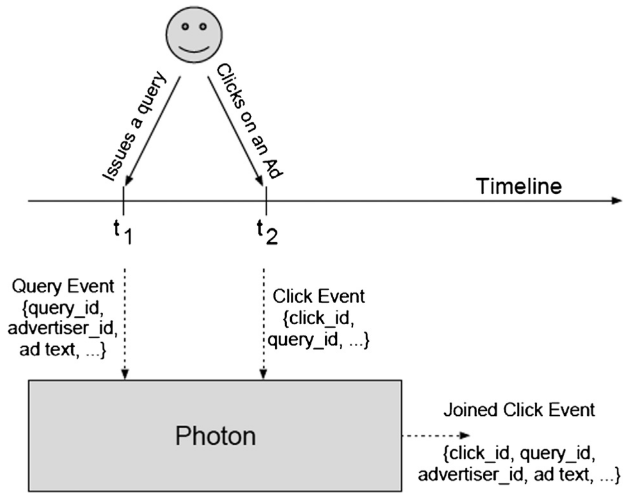

Issues a query
Clicks on an Ad
Timeline
t1
t2
Query Event {query_id, advertiser_id, ad text, ...}
Click Event {click_id, query_id, ...}
Photon
Joined Click Event
{click_id, query_id, advertiser_id, ad text, ...}

FIGURE 12.9

Primary and secondary stream events in Photon are query stream and ad click stream events. At time t_1 a web server responding to a query also serves an ad. The user clicks on the ad at time t_2 and then the query event and the click event are combined into a joined click event [29].

The system has several refinement mechanisms to control how several panes are related to none another. Window contents are discarded once triggered, provided that later pipelines stages expect the values of the triggers to be independent. The window contents are also discarded when the user expresses no interest in later events. The contents of a window can be saved in persistent storage when needed for refining the system state.

12.8 JOINING MULTIPLE DATA STREAMS

Applications such as advertising, IP network management, and telephone fraud detection require the ability to correlate in real-time events occurring in separate high-speed data streams. For example, Google search engines deliver ads along with answers to queries. The Photon system [29] developed for Google Advertising System produces joint logs for the query data stream and the ad click data stream. The joint logs are used for billing the advertisers. A typical application of Photon is illustrated in Figure 12.9.

Photon processes millions of events per minute with an average end-to-end latency of less than 10 seconds. The system joins 99.9999% events within a few seconds, and 100% events within a few hours.

The Proton designers had to address a set of challenges including:

- Each click on an ad should be processed once and only once. This means: at-most-once semantics at any point of time, near-exact semantics in real-time, and exactly-once semantics eventually. If

a click is missed Google loses money; if one click is processed multiple times the advertisers are overcharged.
- Automated data center-level fault-tolerance. Manual recovery takes too long. Photon instances in multiple data centers will attempt to join the same input event, but must coordinate their output to guarantee that each input event is joined at-most-once.
- Latency constraints. Low latency is required by advertisers as it helps optimize their marketing campaigns. Load balancers redirect a user request to the closest running server, where it is processed without interacting with any other server.
- Scalability. The event rates are very high now and are expected to increase in the future, therefore to satisfy latency constraints the system has to scale up.
- Combine ordered and unordered streams. While the query stream events are ordered by time stamps, the events in the click stream may occur at any time and are not sorted. Indeed, the user may click on the ad long after the results of the query are displayed.
- Delays due to the system scale. The servers generating query and click events are distributed over the entire world. The volume of query logs is much greater than that of the click logs thus, the query logs can be delayed relative to the click logs.

Photon organization and operation. The *IdRegistry* stores the critical state shared among running Photon instances all over the world. This critical state includes the *eventId* of all events joined during the last N days. Each instance checks whether the *eventId* already exists in the *IdRegistry* before writing a joined event; if so, it skips processing the event, otherwise adds the event to *IdRegistry*.

The *IdRegistry* is implemented using the Paxos protocol discussed in Section 3.12. An in-memory key-value store based on PaxosDB is consistently replicated across multiple data centers to ensures availability in case of the failure of one or more data centers. All operations are carried out as read-modify-write transactions to ensure that writing to the *IdRegistry* is carried out if and only if the event is not already in. An *eventId* uniquely identifies the server, the process on that server that generated the event, and the time the event was generated

$$EventId = (ServerIP, ProcessId, Timestamp). \tag{12.20}$$

Events with different Ids can be processed independently of each other. This allows the *EventId space* of the *IdRegistry* to be partitioned into disjoint shards. *EventId*'s from separate shards are managed by separate *IdRegistry* servers.

Identical Photon pipelines run at multiple data centers around the world. A Photon pipeline has three major components shown in Figure 12.10:
1. The *Dispatcher* – reads the stream of clicks and feeds them to the joiner.
2. The *EventStore* – supports efficient query lookup.
3. The *Joiner* – generates joined output logs.

The joining process involves several steps:
1. The *Dispatcher* monitors the logs and when new events are detected it looks-up the *IdRegistry* to determine if the *clickId* is already recorded.

   - If already recorded, skip processing the click.
   - Else, send the event to the joiner and wait for a reply. To guarantee at-least-once semantics the dispatcher resends it to the joiner until it gets a positive acknowledgment.

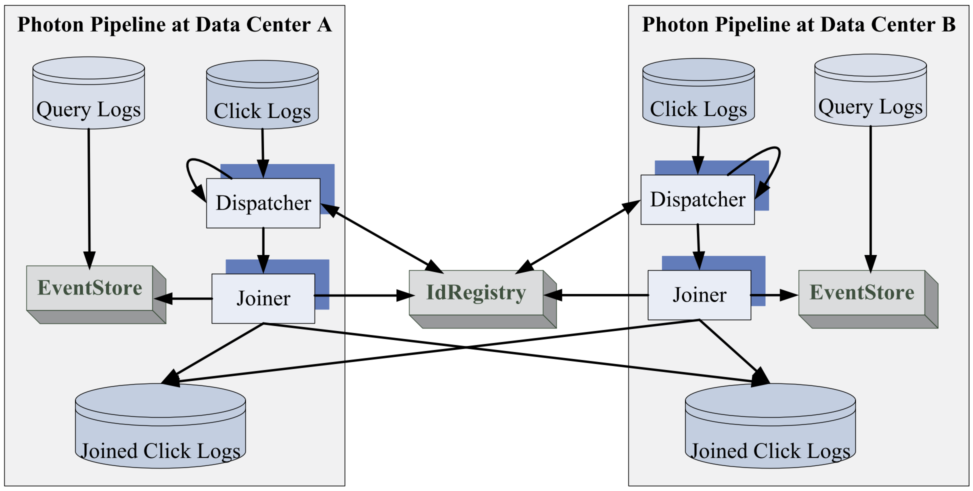


FIGURE 12.10

The organization of the Photon pipeline. At each site the pipeline includes the *Dispatcher*, the *EventStore* and the *Joiner*. These components operate on the query logs, the click logs, and the joined click logs. The *IdRegistry* stores the critical state shared among running Photon instances all over the world.

2. The *Joiner* extracts *queryId* and carries out an *EventStore* lookup to locate the corresponding query. If the query is

   - Found - the joiner attempts to register the *clickId* in the *IdRegistry*:
     - if *clickId* is in the *IdRegistry* the *Joiner* assumes that the join has already been done.
     - if not, the *clickId* is recorded in the *IdRegistry* and the event is recorded in the joint event log.
   - Not found - the *Joiner* sends a failure response to the dispatcher; this causes a retry.

In the US there are five replicas of the *IdRegistry* in data centers in three geographical regions up to 100 ms apart in round-trip latency. The other components in the pipelines are deployed in two geographically distant regions on the East and West coast. According to [29]: "during the peak periods, Photon can scale up to millions of events per minute,... each day, Photon consumes terabytes of foreign logs (e.g. clicks), and tens of terabytes of primary logs (e.g. queries), ... more than a thousand *IdRegistry* shards run in each data center, ... each data center employs thousands of *Dispatchers* and *Joiners*, and hundreds of *CacheEventStore* and *LogsEventStore* workers."

12.9 SYSTEM AVAILABILITY AT SCALE

As the cloud user population grows and the scale of the data center infrastructure expands the concerns regarding system availability are amplified. Very high system availability and correctness are critical for data streaming processing systems. Such systems are particularly vulnerable because the likelihood of missed events is very high when the resources required for event processing suddenly become unavailable.

When we think about availability, a 99% figure of merit seems quite high. There is another way to look at this figure; a 99% availability translates to 22 hours of downtime per quarter and this is not acceptable for mission-critical systems. Goggle aims to achieve 99.9999–99.99999% availability for their data streaming systems [213]. How can this be achieved? An answer to this question is the topic of this section.

Multiple failure scenarios are possible. Individual servers may fail, all servers in one rack may be affected by a power failure, and a power failure may force the entire data center infrastructure to shutdown, though such events are rare. The interconnection network may fail and partition the network, the public networks connecting a data center with the Internet or the private networks connecting the data center with other data centers of the same CSP may fail.

The workloads can be migrated to functioning systems in case of partial failures affecting a subset of resources. The users may experience slower communication and extended execution times in such cases. When partial failures affect data streaming some events are missed. Often, it takes some time before the cause of a partial failure is found and corrective measures are taken.

Shutdown of servers, networks, or the entire data center may be planned for hardware or software updates or maintenance. Such events are announced in advance, and in such cases the shutdown is graceful and without data loss or other unpleasant consequences for the cloud users. The unplanned shutdown are the ones the CSPs are concerned about. The most consequential are the data center-level failures.

While current systems manage quite well the failure of individual servers, data center-level failures, though seldom occurring, have catastrophic consequences especially for *single-hommed* software systems, the ones running *only* at the site affected by the failure. Less affected are the *failover-based* software systems. Such systems only run at one site, but checkpoints are created periodically and sent to backup data centers. When the primary data center fails the last checkpoints of the critical systems are used to restart processing at the backup site. Data streaming is affected in this case as events are likely to be missed.

Checkpointing can be done asynchronously or synchronously. In the first case, all events are processed exactly once during the restart if the events in progress are drained before checkpointing, and only when the shutdown was planned. Synchronous checkpointing can, in principle, capture all events even in case of unplanned shutdowns, but their processing is considerably more complex. For planned shutdowns they require each pipeline to generate a checkpoint and block until the checkpoint is replicated.

Virtually all CSPs have multiple data centers distributed across several regions and a basic assumption for handling data center-level failures is that the probability of a simultaneous failure of data centers in different regions is extremely low. Critical software systems are *multi-homed* and the workload is dynamically distributed among these centers and, when one of them fails, its share of the workload is reassigned to the ones still running.

The design of multi-homed systems is not without its own challenges. First, the global state of the system must be available to all sites. The size of the metadata reflecting the global state should be minimized. Communication delays of tens of milliseconds and limited global communication bandwidth do not make determination of the global state an easy task, even when a Paxos-bases commit is used to update the global state.

Second, a data streaming pipeline is implemented as a network with multiple processing nodes. Checkpointing has to capture the state of all nodes, as well as metadata such as the state of the input and output queues of pending and completed node work. Clustering groups nodes together and recording only the global state of the cluster helps. Clustering increases the checkpointing efficiency and shrinks the checkpoint size.

Handling the same input events at several sites can be optimized so that the checkpoint includes only one copy of the event. This is actually done for the events recorded by logs in the system discussed next. Such primary events may require database lookup and if the database data does not change, then the state of the primary event should not record database information.

Lastly, guaranteeing exactly once semantics for multi-homed systems is nontrivial because pipelines may fail while producing output; multiple sites may process the same events concurrently. Updating global state can be atomic when the global state is stored in shared memory. Idempotence can help achieving the desired semantics. When multiple sites update the same record the desired semantics is guaranteed. If idempotence is not achievable a two-phase commit can be used to write the results produced by the streaming system to the output.

Data streaming services collect data generated by user interactions and expect consistency. Data streaming consistency means that when the state at time t includes an event e any future state, at time $t + \tau$, must also include event e; an observer should see consistent outcomes if more than one are generated.

The concern for data center-level failures forced Google to run multiple copies of critical systems at different data centers. Several large-scale Google systems run hot in multiple data centers, including the F1 database [451] discussed in Section 12.2 and the Photon system [29] discussed in Section 12.8.

An infrastructure supporting adds management at Google is discussed in [213]. The strategy for supporting availability and consistency for streaming system is to log the events caused by user interactions in multiple data centers. Local logs are then collected by a *log collection service* and are sent to several locations designated as *log data centers*. Consistency of these logs is critical and requires an *exactly once* event processing semantics.

Once a system is developed for a single site it is very hard to adapt to multi-site operations, especially when consistency among sites is a strong requirement. An ab initio design as a multi-home is the optimal solution for mission critical systems. Running a system at multiple sites transfers the burden to solve the very hard problem caused by data center-level failures to the infrastructure. This burden falls to the users in traditional systems designed to run at one site only. Indeed, the users have to take periodic checkpoints and transfer them to backup sites.

Multi-homing is challenging for system designers and adds to the resource costs. Additional resources are needed to process the workload normally processed by the failing data center and to catch up after delays. The spare capacity has to be reserved ahead of time and should be ready to accept the additional workload.

12.10 THE SCALE AND THE LATENCY

The scale of clouds has altered our ideas on how to compute efficiently. The scale has also generated new challenges in virtually all areas of computer science and computer engineering, from computer

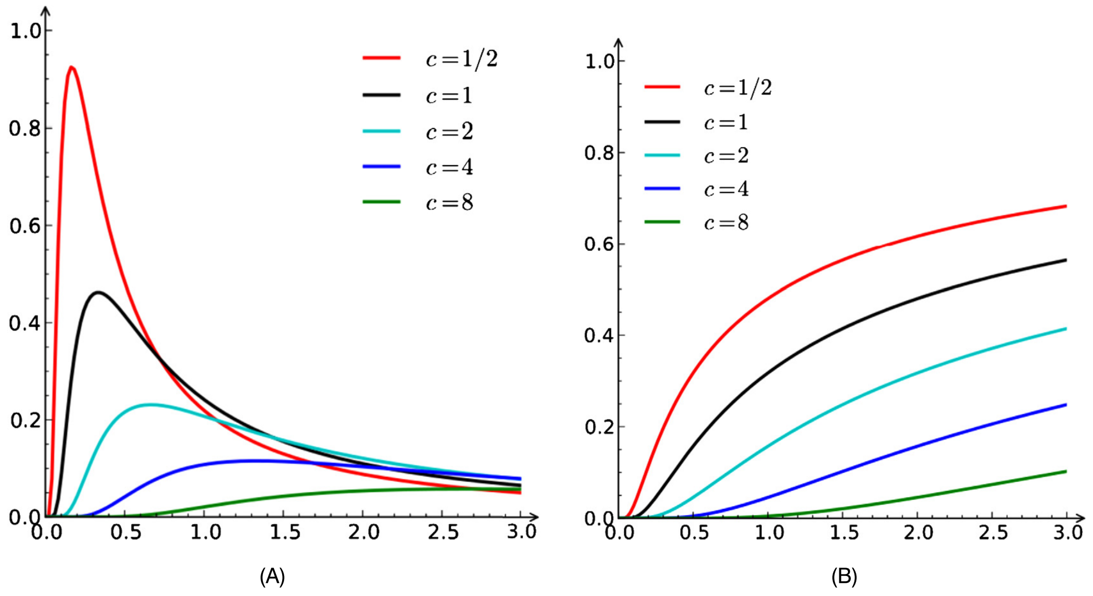


FIGURE 12.11

(A) The probability density function of the unshifted Lévy distribution, $\mu = 0$; (B) The cumulative distribution function.

architecture to security, via software engineering and machine learning. A fair number of models and algorithms for parallel processing have been adapted to the cloud environment.

Frameworks for Big Data processing, resource management for large-scale systems, scheduling algorithms for latency-critical workloads discussed in Chapters 7, 8, 9, 10, and 12 offer only a limited window into the new computing landscape. Several critically important ideas for the future of cloud computing are discussed in this section.

Latency, the time elapsed from the instant when an action is initiated and its completion, has always been an important measure of quality of service for interactive applications, but its relevance and impact has been substantially amplified in the age of online searches and web-based electronic commerce. There are quite a few latency-critical applications of cloud computing now and their number is likely to explode when a large number of cyber-physical systems of the IoT will invade our working and living space.

The latency has three major components, communication time, waiting time, and service time. Only the later two are discussed in this section because the communication speed is not controlled by the cloud service provider and in practice it has little impact on latency. A heavy tail distribution of latency due to the last two components is undesirable, it affects the user's experience and, ultimately, the ability of the service provider to compete.

Heavy-tail distributions. A random variable X with the cumulative distribution function $F_X(x)$ is said to have a heavy-right tail if

$$\lim_{x \to \infty} e^{\lambda x} Pr\,[X > x] = \infty, \quad \forall\, \lambda > 0. \tag{12.21}$$

This definition can also be expressed in terms of the tail distribution function

$$\bar{F}_X(x) \equiv Pr\,[X > x] \tag{12.22}$$

and implies that $MF(t)$, the moment generating function of $F_X(x)$, is infinite for $t > 0$. The Lévy distribution is an example of a heavy-tail distribution. Its probability density function (PDF) over the domain $x \geq \mu$ and cumulative distribution function (CDF) are, respectively,

$$f_X(x;\mu,c) = \sqrt{\frac{c}{2\pi}} \times \frac{e^{-\frac{c}{2(x-\mu)}}}{(x-\mu)^{3/2}} \quad \text{and} \quad F_X(x;\mu,c) = erfc\left(\sqrt{\frac{c}{2(x-\mu)}}\right) \tag{12.23}$$

with μ the location parameter, c the scale parameter, and $erfc(y)$ the complementary error function. Figure 12.11 displays the PDF and the CDF of the Lévy distribution. In probability theory and statistics, a scale parameter is a special kind of numerical parameter of a parametric family of probability distributions. The larger the scale parameter, the more spread out is the distribution function.

Query processing at Google. The analysis of query processing at Google gives us some insight into the reasons why latency has a heavy-tail distribution and how this can be managed. A query fans-out into thousands of requests to a large number of servers where cached data resides. Some of these servers contain images, others videos, web data, blogs, books, news, and many other bits of data that could possibly answer the query.

The individual-leaf-request finishing times measured from the root node to the leafs of the query fan-out tree show the effect of the heavy-tail distribution [131]. As the limit x in the latency cumulative distribution function increases from 50 to 95 and then to 99 percentile, the latency increases:

- When a single randomly selected leaf node of the tree is observed, the latency increases from 1 to 5, and then to 10 milliseconds, respectively.
- When we request that 95% of all leafs finish execution, the latency increases from 12 to 32, and then to 70 milliseconds, respectively.
- When we request that all nodes finish execution, the latency increases from 40 to 87, and then to 140 milliseconds, respectively.

These measurements show that latency increases with the number of leafs of the fan-out tree; it also increases as we wait for more leafs to finish. The extent of the latency tail is significant, the difference between 95 and 99 percentile is dramatic, it doubles or nearly doubles for the three cases examined, 10 versus 5, 70 versus 32, and 140 versus 87, mimicking the law of diminishing returns.

Google is able to address the problems posed by the heavy-tail latency distribution [131]: "The system updates the results of the query interactively as the user types predicting the most likely query based on the prefix typed so far, performing the search and showing the results within a few tens of milliseconds." How this is done is discussed in this section.

A brief analysis of the factors affecting the variability of the query response time helps understanding how this variability can be limited. Resource sharing is unavoidable and contention for system resources translates into longer response time for the losers. There are actors working behind the scene carrying out system management and maintenance functions. For example, daemons are themselves users of resources and competitors of production workloads; they can occasionally cause an increase of the response time. Data migration, data-replication, load balancing, garbage collection, log compaction are all activities competing for system resources.

Multiple layers of queuing in the hierarchical infrastructure increase the waiting time and this can be addressed by priority queuing and techniques discussed next. Optimization of resource utilization and energy saving mechanisms contribute to an increased latency. For example, servers are switched to a power-saving mode when the workload dips. The latency can increase in this case because it takes some time before servers wakeup and are able to absorb a spike in demand. Dynamic frequency and voltage scaling, a mechanism to adapt the power consumption to the workload intensity, could also contribute to latency variability.

Several techniques can reduce the effects of component-level variability on the heavy-tail query latency distribution. These techniques are:

1. Define different service classes and use priority queuing to minimize waiting for latency-critical workloads.
2. Keep server queues short allowing requests from latency-critical workloads to benefit from their high priority. For example, Google storage servers keep few operations outstanding, instead of maintaining a queue. Thus, high-priority requests are not delayed when earlier requests from low priority tasks are served.
3. Reduce head-of-line blocking. When a long-running task cannot be preempted, other tasks waiting for the same resource are blocked, a phenomenon called *head-of-line blocking*.[13] The solution is to split a long-running task into a number of shorter tasks and/or to use time-slicing to allocate the resource to task for brief periods of time. For example, Google's Web search system uses time-slicing to prevent computationally expensive queries to add to the latency.
4. Limit the disruption caused by background activities. Some of the behind-the-scene activities such as garbage collection or log compaction require multiple resources. Their continuous running in the background could lead to increased latency of many high-priority queries over extended periods of time. It is more beneficial to allow such background activities to proceed in synchronous bursts of concurrent utilization of several servers thus, affect only the latency-critical tasks over the time of the burst.

A very important realization is that the heavy-tail distribution of latency cannot be eliminated, the only alternative is to develop tail-tolerant techniques for masking long latencies. Two types of tail-tolerant techniques are used at Google: (i) *within request, short-term*, acting in milliseconds; and (ii) *cross-request, long-term*, acting at a scale of seconds.

The former technique works well for replicated data, e.g., for distributed files systems with data striped and replicated across multiple storage servers and for read-only datasets. For example, the spelling-correction service benefits from this mechanisms as the model is updated once a day and handles a very high rate of requests, in the hundreds of thousands per second. Cross-request techniques aim to increase parallelism and prevent imbalance either by creating micro-partitions, by replicating the items likely to cause imbalance, or by latency-induced probation.

Hedged and *tied* requests are short-term tail-tolerant techniques. In both cases the client issues multiple replicas of the request to increase the chance of a prompt reply. Hedged requests are separated

[13]For an intuitive scenario of head-of-line blocking imagine a slow-moving truck on a one-lane, winding mountain road. It is very likely that a large number of cars will be stuck behind the truck.

Table 12.3 Read latency in ms with requests tied 1 ms [131] for a latency-critical application. Columns 2 & 3 - mostly idle system. Columns 4 &5 - system running a background job in addition to the latency-critical application.

Limit (%)	Not hedged	Hedged-tied	Not hedged	Hedged-tied
50	19	16	24	19
90	38	29	54	38
99	67	42	108	67
99.9	98	61	159	108

by a short time interval; the client issues the requests, accepts the first answer, and then notifies the other servers canceling the request. The client should wait a fraction, say 90% to 95%, of the average response time before sending the replica of the request to limit the additional workload and to avoid duplication.

The requests are tied when each replica includes the address of the other servers. In this case the servers receiving the request communicate with one another; the first server able to start processing the request sends a canceling message to the other servers at the time it starts execution. If the input queue of all recipients of the request is empty then all can start processing at about the same time and canceling messages crisscross the network unable to prevent work duplication. This undesirable situation is prevented if the client waits a time equal to twice the average message delay, before sending a replica of the request.

Hedged requests are very effective. For example, a Google benchmark reads the key value of 1 000 key-value pairs stored in a large BigTable distributed over 100 servers. Hedged requests sent 10 milliseconds after the original ones reduced the 99 percentile latency dramatically, from 1 700 to 74 milliseconds while sending only 2% more requests. Another Google benchmark shows the effect of the queries tied one millisecond to an idle cluster and the other query on a cluster running a batch job in the background. The BigTable data is not cached and each file chunk has three replicas on different storage servers. Table 12.3 shows read latencies with no hedged and with hedged tied requests.

The results in Table 12.3 show first that hedged and tied requests work well not only when the system is lightly loaded, but also at higher system loads. This also implies that the overhead of this mechanisms is low and adds only a minute workload to the system. These results also show the extent of the heavier tail; the difference in latency between 99% and 99.9% is significant for both the lightly and the heavily loaded system.

Micro-partitions, replication of items likely to cause imbalance, and latency-induced probation are long-term tail-tolerant techniques. Perfect load balancing in a large-scale system is practically unachievable for many reasons. These reasons include the difference in server performance, the dynamic nature of the workloads, and the impossibility of having an accurate picture of the global system state. A fairly intuitive approach is a fine-grain partitioning of resources on every server, thus the name micro-partitions.

These “virtual servers” are more nimble and able to process fine-grained units of work in shorter time. Error-recovery is also less painful, as less work is lost in case of errors. For a long time immovable data replication has been a method of choice for improved performance and it is also widely used at Google. Intermediate servers in a hierarchically-organized system can identify slow-responding servers and avoid sending them latency-critical tasks, while continuing to monitor their behavior.

12.11 MOBILE COMPUTING AND APPLICATIONS

According to http://www.mobilecloudcomputingforum.com/ "...mobile cloud computing at its simplest, refers to an infrastructure where both the data storage and data processing happen outside of the mobile device. Mobile cloud applications move the computing power and data storage away from mobile phones and into the cloud, bringing applications and mobile computing to not just smartphone users but a much broader range of mobile subscribers."

Mobile devices are in a symbiotic relationship with computer clouds. Mobile devices are producers as well as consumers of data stored on the cloud. They also take advantage of cloud resources for computationally-intensive tasks. Mobile devices benefit from this symbiotic relationship; their reliability is improved as data and the applications are backed up on the cloud.

Clouds extend the utility of mobile devices providing access to vast amount of information stored on their servers and also extend their limited physical resources:

- The processing power and the storage capacity – an ubiquitous applications of mobile devices is to capture still images or videos, upload to a computer cloud, and make them available via cloud services such as Flickr, Youtube, or Facebook.
- The battery life – migrating mobile game components to a cloud can save 27% of the energy consumption for computer games and 45% for chess game [126].

The integration of mobile devices in the cloud ecosystem has major benefits [442]:

- Mobile devices capture images and videos rich in content, rather than simple scalar data such as temperature. Such information can be used to understand the extent of catastrophic events such as earthquakes or forest fires.
- There is power in numbers – cloud sourcing amplifies the power of sensing and can be used for applications related to security, rapid service discovery, and so on.
- Support near-real time data consistency, important in disaster relief scenarios. For example, the aftershocks of an earthquake often trigger major structural changes of the buildings. Images and/or videos taken before and after the event help asses the extend on the damages and identify buildings to be immediately evacuated.
- Enable opportunistic information gathering. For example, anti-lock braking devices on cars transmit their GPS coordinates on each activation, enabling maintenance crews to identify the slick spots on roads.
- Computer vision algorithms running on clouds can use images captured by mobile devices to locate lost children in crows, estimate the size of crowds.

Applications of mobile cloud computing. Many mobile applications consist of a light-weight front-end component running on the mobile device and a back-end data-intensive and computationally-intensive component running on a cloud. There are countless examples of such ubiquitous applications of mobile cloud computing in areas as diverse as healthcare, learning, electronic commerce, mobile gaming, mobile location services, and searching.

An important application of mobile healthcare is patient record storage and retrieval and medical image sharing using computer clouds and mobile devices. Other applications in this area are [149]:

health monitoring services where patients are monitored using wireless services; emergency management involves coordination of emergency vehicles; and wearable system for monitoring the vital signs of outpatients after a surgery.

There are many educational tools helping students learn and understand subjects as diverse as anatomy, computer science and engineering, or arts. Though some progress has been made, the potential of mobile learning is not fully exploited at this time. Classroom tools using AI algorithms to identify relevant questions posed by the students during lectures and allowing the instructor to interact with the students in classes with large enrollments are yet to be developed.

Mobile gaming has the potential of generating large earnings for CSPs. The engine requiring large computing resource can be offloaded to a cloud, while gamers only interact with the screen interface running on their mobile devices. For example, the Maui system [126] partitions the application codes at runtime based on the costs of network communication and on the CPU power and the energy consumption of the mobile device, to maximize energy savings. Browsers running on mobile devices are often used for keyword-, voice- or tag-based searching of large debases available on computer clouds.

All applications of mobile cloud computing face challenges related to communication and computing. Low bandwidth, service availability, and network heterogeneity pose serious communication problems. Security, efficiency of data access, and offloading overhead are top concerns on the computing side.

Mobile devices are exposed to a range of threats including viruses, worms, Trojan horses, and ransomware. Such threats are amplified as mobile devices have limited power resources and it is impractical to continually run virus detection software. Moreover, once files located on the mobile device are infected, the infection propagates to the cloud when the files are automatically uploaded. At the same time the integration of GPS is a source of concern for privacy as the number of location-based services (LBS) increases. Cloud data security, integrity, authentication along with digital rights management are also sources of concern.

Enhancing the efficiency of data access requires a delicate balance between local and remote operations and the amount of data exchanged between the mobile device and the cloud. The number of I/O operations executed on the cloud in response to a mobile device request should be minimized to reduce the access time and the cost. The memory and storage capacity of the mobile device should be used to increase the speed of data access, reduce latency, and improve energy efficiency of mobile devices.

The future of mobile cloud computing. As the technology advances mobile devices will be equipped with more advanced functional units, including high-resolution cameras, barometers, light sensors, and so on. Augmented reality and mobile gaming are emerging as important mobile cloud computing applications. Augmented reality could be ubiquitous with new mobile devices and fast access to computer clouds. Composition of real-time traffic maps from collective traffic data sensing, monitoring environmental pollution, traffic and pollution management in smart cities are only a few of the potential future applications of mobile cloud computing. A recent survey [513] analyzes applications, the solutions to the challenges they pose, and the future solutions:

- Code and computation offloading – currently based on static partitioning and dynamic profiling is expected to be automated in the future.
- Task-oriented mobile services – currently provided by Mobile-Data-as-a-Service, Mobile-Computing-as-a-Service, Mobile-Multimedia-as-a-Service and Location-based Services is expected to be replaced by human-centric mobile services.

- Elasticity and scalability – components of the resource allocation and scheduling are expected to be validated by algorithms using valid traffic VM migration models.
- Cloud service pricing – currently based on auctions and bidding is expected to be replaced by empirical validation and optimization algorithms.

Unquestionably, inclusion of mobile devices into the cloud ecosystem has countless benefits. It opens new avenues for learning, increased productivity, and entertainment, but can also have less desirable effects, it can overwhelm us with information. Herb Simon reflected on the effects of information overload [452] "What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention and a need to allocate that attention efficiently among the overabundance of information sources that might consume it."

12.12 ENERGY EFFICIENCY OF MOBILE COMPUTING

Mobile computing enjoys the benefits of virtually infinite resources available on demand on computer clouds. Utility computing offers considerable economic advantages to mobile cloud users. To enjoy these benefits mobile system need network access to these islands of plentiful computing and storage resources.

Cloud users transfer data to/from the cloud, but there is a significant dissimilarity between cloud users connected to computer clouds via landlines and using stationary devices and users of mobile devices connected via cellular or wireless local area networks. The first are concerned with the time and the cost of transferring massive amounts of data, while for mobile cloud users the key issue is the energy consumption for communication.

The battery technology is lagging behind the needs of modern mobile devices. The amount of energy stored in a battery is growing only about 5% annually. Increasing the battery size is not an option, as devices tend to be increasingly lighter. Moreover, the power of small mobile devices without active cooling is limited to about three watts [372].

An analysis of the critical factors regarding mobile devices every consumption reported in [342] is discussed in this section. As expected, the computing-to-communication ratio is the critical factor for balancing local processing and computation offloading. The analysis shows that not only the amount of transferred data, but also the traffic pattern is important. Indeed, sending a larger amount of data in a single burst consumes less energy than sending a sequence of small packets.

The analysis uses several parameters: E_{cloud} and E_{local}, the energy for a computation on the cloud and locally, respectively; D – the amount of data to be transferred; C – the amount of computation expressed as CPU cycles. C_{eff} and D_{eff} are the efficiencies of computing and device specific data transfer, respectively. C_{eff} is measured in CPU cycles per Joule and represents the amount of computations that can be carried out with a given energy.

Dynamic voltage and frequency scaling affects only slightly the power and performance of the CPU thus, C_{eff}, during execution. For example, the N810 Nokia processor requires 0.8 W and has $C_{eff} = 480$ cycles/J when running at 400 MHz and needs only 0.3 W and has a $C_{eff} = 510$ cycles/J at 165 MHz. This is also true for the more performant N900 Nokia processor; when running at 600 MHz the power required and the C_{eff} are 0.9 W and 650 cycles/J, respectively, while at 250 MHz the same parameters are 0.4 W and 700 cycles/J, respectively.

D_{eff} is measured in bytes per Joule and represents the amount of data that can be transferred with a given energy. D_{eff} is affected by the traffic pattern. The time the network interface is active affects the energy consumption for communication. Typically, the power consumption for activation and deactivation of a cellular network interface is larger than that of a wireless interface. The larger the clock rate, the larger the power consumption and D_{eff}. For example, for N810 the power consumption and D_{eff} at 400 MHz are 1.5 W and 390 KB/J, respectively, and decrease to 1.1 W and 310 KB/J, respectively, at 165 MHz.

It makes sense to offload a computation to a cloud if

$$E_{cloud} < E_{local} \tag{12.24}$$

with

$$E_{cloud} = \frac{D}{D_{eff}} \quad \text{and} \quad E_{local} = \frac{C}{C_{eff}}. \tag{12.25}$$

The condition expressed by Equation (12.24) becomes:

$$\frac{C}{D} > \frac{C_{eff}}{D_{eff}}. \tag{12.26}$$

According to [342] a rule of thumb is that “offloading computation is beneficial when the workload needs to perform more than 1 000 cycles of computation for each byte of data.”

The CPU cycles per byte ratios measured on a system with an ARM Cortex-A8 core running at 720 MHz for several applications are: 330 for gzip ACII compression, 1 300 for x264 VBR encoding, 1 900 for CBR encoding, 2 100 for htm12text on wikipedia.org and 5 900 for htm12text on en.wikipedia.org.

Several conclusions were drawn from the experiments reported in [342]:

- The energy consumption of a mobile device is affected by the end-to-end chain involved in each transaction thus, the server-side resource management is important.
- Higher performance typically contributes to better energy efficiency.
- Simple models able to guide design decisions for increasing energy efficiency should be developed.
- Automated decisions on whether a computation should be uploaded to a cloud to maximize energy efficiency of mobile devices should be built in the middleware of mobile cloud computing applications.
- Latencies associated with wireless communication are critical for interactive workloads.

It is unlikely to see dramatic improvements in the energy storage capacity of batteries for mobile devices in the immediate future. The need for increasingly more sophisticated software for energy optimization should motivate the research in this challenging area as new mobile cloud computing data and computation-intensive applications are developed every day. At the same time, the other critical limitations of mobile cloud computing applications, communication speed and effectiveness, are improving as WLAN speed is steadily increasing along with the mobile device antenna efficiency.

12.13 ALTERNATIVE MOBILE CLOUD COMPUTING MODELS

In Section 12.11 we have seen that mobile device access to the large concentration of resources in cloud computing data centers is supported by wide area networks (WANs), cellular networks, and wireless networks. In this section we examine the limitations due to the communication latency and, to some extent, of the bandwidth and discuss alternative mobile cloud computing architectures.

The effects of latency. An important observation is that communication latency through WANs and wireless networks is unlikely to improve. It is not hard to see that network security, energy efficiency, network manageability, along with bandwidth are the main sources of concern for networking companies and for networking research.

Unfortunately, virtually all methods to address these concerns have a negative side effect, *they increase the end-to-end communication latency*. Indeed, the techniques to increase energy efficiency discussed in Section 12.12 include reduction of the time the network interface is active, delaying data transmission until large data blocks of data can be sent, and turning on the transceivers of mobile devices for short periods of time to receive and to acknowledge data packets buffered at a base station.

The speed of the fastest wireless LAN (802.11n) and of the wireless Internet HSPDA (High-Speed Downlink Packet Access) technologies are 400 Mbps and 2 Mbps, respectively, and the corresponding transmission delays for a 4 Mbyte Jpeg image are 80 versus 16 msec, respectively. The range of Internet latencies is 30–300 msec. For example, the mean Berkeley to Trondheim–Norway and Berkeley to Canberra–Australia latencies are 197 and 174 msec, respectively, while Pittsburg to Hong Kong and Pittsburg to Seattle are 223 and 83.9 msec, respectively, as measured in 2008–2009 [441]. The current Internet latencies between selected pairs of points on the globe can be found at https://www.internetweathermap.com/map.

The latency effect differs from application to application. For example, the subjective effects of the latency, L, for a particular application, GNU Manipulation Program (GIMP) for Virtual Network Computing communication software, the graphical desktop sharing system that uses the Remote Frame Buffer protocol are: crisp when $L < 150$ msec; noticeable to annoying when $150 < L < 1\,000$ msec; annoying when $1 < L < 2$ sec; unacceptable when $2 < L < 5$ sec; and unusable when $L > 5$ sec [441].

Cloudlets. A possible solution to reduce end-to-end latency mimics the model supporting wireless communication where access points are scattered throughout campuses of large organizations and in the cities. Micro data centers or "clouds in a box," called *cloudlets*, are placed in the proximity of regions with a high concentrations of mobile systems. A cloudlet could be a cluster of multicore processors with a fast interconnect and a high-bandwidth wireless LAN.

The resources available on such cloudlets pale in comparison with the ones on a cloud. Cloudlets only assist mobile devices for data and computationally-intensive tasks, they are not permanent repositories of data. A mobile device could connect to a cloudlet and upload code to the cloudlet.

There are obvious benefits, as well as problems, with this solution proposed in [441]. The main benefit is the short end-to-end communication delay for mobile devices in the proximity of a cloudlet. On the other hand, the cost and the maintenance of cloudlets able to support a wide range of mobile users are of concern. It is safe to assume that hardware costs will decline as the processor technology evolves and that the processor bandwidth and reliability will increase. The software maintenance effort can be reduced if self-management techniques are developed. A range of business and technical challenges including cloudlet sizing must be addressed before this solution gets sufficient traction.

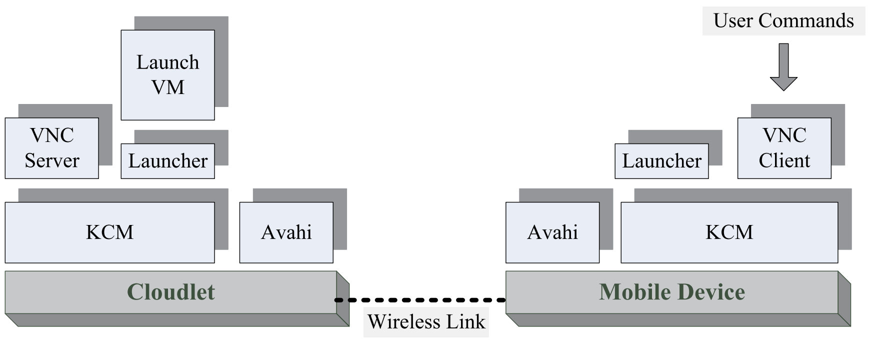


FIGURE 12.12

Kimberly organization. Kimberley Control Manager (KCM) runs on both the cloudlet and the mobile device. The two communicate over a wireless link using the Virtual Network Computing (VNC) communication software and Avahi [441].

The solution for software organization proposed in [441] is to have a permanent cloudlet host software environment and a transient guest software environment. A transient customization configures the system for running an application and, once the run is complete, a cleanup phase clears the way for running the next one. A VM encapsulates the transient guest environment.

The VM can be created on the mobile device, runs locally until the VM needs additional resources, then stopped, its state is saved in a file, and the file is sent to a cloudlet in the vicinity of the mobile device where the VM is restarted. The problem with this straightforward solution is that the latency can be increased when the footprint of the VM migrated to the cloudlet is substantial.

Another solution is the *dynamic VM synthesis* when the mobile device delivers to the cloudlet only a small overlay. This solution assumes that the cloudlet already has the base VM used to derive the small overlay, therefore the cloudlet environment can start execution immediately without the need to contact a cloud or other cloudlets. The assumption that a relatively small set of base VMs will suffice for a large range of applications may prove to be overly optimistic.

Kimberlay – a proof of concept cloudlet system. The cloudlet infrastructure consists of a desktop running Ubuntu Linux and the mobile device is a Nokia N810 tablet running Maemo 4.0 Linux [441]. The cloudlet uses a hosted hypervisor for Linux called VirtualBox and a tool with three components, *baseVM, install-script* and *resume-script* to create the VM overlays for any OS compatible with the components of the tool.

First, the *baseVM* is launched, then the *install-script* in the guest OS is executed, and finally the *resume-script* in the guest OS launches the application. The VM encapsulates the application, the so called *launchVM* can be activated without the need to reboot, and finally this VM is compressed and encrypted.

Kimberley organization depicted in Figure 12.12 shows the software components running on the cloudlet and the mobile device. Communication through the wireless link is supported by Virtual

Network Computing (VNC) communication software and Avahi,[14] a free zero-configuration[15] for automatic networking implementation supporting multicast DNS/DNS-SD service discovery.

KCM abstracts the service discovery including browsing and publishing in Linux using Avahi. A secure TCP tunnel using SSL is established between KCM instances. After the establishment of the secure TCP tunnel the authentication using the Simple Authentication and Security Layer (SASL) framework is carried out. Then the KCM on the cloudlet fetches the VM overlay from the mobile device KCM, decrypts and decompresses the VM overlay, and applies the overlay to the base VM.

12.14 MOBILE EDGE CLOUD AND MARKOV DECISION PROCESSES

Follow-me cloud [477] and *mobile edge-cloud* are variations on the theme of cloudlets discussed in Section 12.13. The two systems aim to reduce the end-to-end latency for cloud access. The mobile edge-cloud concept allows mobile devices to carry out computational-intensive tasks on stationary servers located in the small data centers distributed across the network and connected directly to base stations at the edge of the network.

The new twist is to support *dynamic service placement*, in other words, to allow computations initiated by a mobile device in motion to migrate from one mobile edge-cloud server to another following the movement of the mobile device [491,514]. Optimal service migration policies pose very challenging problems due to: (i) the uncertainty of the mobile device movements; and (ii) the likely non-linearity of migration and communication costs. One method to address these challenges is to formulate the migration problem in terms of Markov decision process discussed next.

Markov decision processes. Markov decision processes are *discrete-time* stochastic control processes used for a variety of optimization problems where the outcome is partially random and partially under the control of the decision maker. Markov decision processes extend Markov chains with choice and motivations; *actions* allow choices, and *rewards* provide motivation for actions.

The state of the process in slot t is s_t and the decision maker may choose any action $a_t \in \mathcal{A}(s_t)$ available in that state. As a result of action a_t the system moves to a new state s' and provides the reward $\mathcal{R}_{a_t}(s_t, s')$. *The next state s' depends only on the current state s_t and the action taken.* The probability that the system moves to state s' is given by the transition function $p_{a_t}(s_t, s')$.

A Markov decision process is a 5-tuple: $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot,\cdot), \mathcal{R}(\cdot,\cdot), \gamma)$ with

- $\mathcal{S}$ – a finite set of system states;
- $\mathcal{A}$ – a finite set of actions; $\mathcal{A}_{s_t} \in \mathcal{A}$ is the finite set of actions available in state $s_t \in \mathcal{S}$ in time slot t.
- $\mathcal{P}$ – the set of transition probabilities; $p_{a_t}(s_t, s') = Pr(s(t+1) = s' | s_t, a_t)$ – the probability that action a_t in state s_t in time slot t will lead to state s' in time slot $t+1$.
- $\mathcal{R}_{a_t}(s_t, s')$ – the immediate reward after the transition from state s_t to state s'.

[14]Avahi is the Malagasy and scientific Latin name of the woolly lemur primates indigenous to Madagascar.

[15]Zero-configuration networking (zeroconf) creates a TCP/IP computer network based on automatic assignment of numeric network addresses for networked devices, automatic distribution and resolution of computer hostnames, and automatic location of network services. It is used to interconnect computers or network peripherals.

- $\gamma \in [0, 1]$ is a discount factor representing the difference in importance between present and future rewards.

The goal is to optimize a policy π maximizing a cumulative function of the random rewards, e.g., the expected discounted sum of rewards over an infinite time horizon is

$$\sum_{t=0}^{\infty} \gamma^t \mathcal{R}_{a_t}(s_t, s_{t+1}). \tag{12.27}$$

To calculate the optimal policy given $\mathcal{P}$ and $\mathcal{R}$, the state transitions and, respectively, the rewards, one needs two arrays $\mathcal{V}(s)$ and $\pi(s)$ indexed by state, the value and policies, respectively. The first array contains values and the second actions:

$$\pi(s) = \arg\max_{\alpha} \left\{ \sum_{s'} \mathcal{P}_\alpha(s, s') \left(\mathcal{R}_\alpha(s, s') + \gamma \mathcal{V}(s')\right) \right\} \tag{12.28}$$

$$\mathcal{V}(s) = \sum_{s'} \mathcal{P}_{\pi(s)}(s, s') \left(\mathcal{R}_{\pi(s)}(s, s') + \gamma \mathcal{V}(s')\right) \tag{12.29}$$

In the value induction proposed by Bellman the calculation of $\pi(s)$ is substituted in the calculation of $\mathcal{V}(s)$:

$$\mathcal{V}_{i+1}(s) = \max_{\alpha} \left\{ \sum_{s'} \mathcal{P}_\alpha(s, s') \left(\mathcal{R}_\alpha(s, s') + \gamma V_i(s')\right) \right\}. \tag{12.30}$$

A Markov decision process can be solved by linear programming or by dynamic programming.

Migration decisions and cost in mobile cloud edge. The solution proposed in [491,514] assumes that:

1. The user's location in each slot is the same and changes from one slot to the next are according to the Markovian model discussed in this section, see Figure 12.13.
2. The set of all possible locations, $\mathcal{L}$, can be represented as a 2D-vector and the distance between two locations $l_1, l_2 \in \mathcal{L}$ is given by $|| \, l_1 - l_2 \, ||$.
3. The migration time is very small and will be neglected.

The following notations are used in [514]:

- $u(t)$ – users's location in slot t.
- $h(t)$ – service location in slot t.
- $d(t)$ – the distance between the user and the service; $d(t) = \|u(t) - h(t)\|$ in slot t.
- N – the maximum allowed distance between the user and the service, $N = \max d(t)$.
- $s(t) = (u(t), h(t))$ – the initial system state at the beginning of slot t. The initial state is $s(0) = s_0$.
- $c_m(x)$ – the migration cost is a non-decreasing function of x, the distance between the two edge cloud servers, $x = \|h(t) - h'(t)\|$.
- $c_d(x)$ – the transmission cost is a non-decreasing function of x, the distance between the edge cloud server and the user, x. with $x = \|u(t) - h'(t)\|$. Initially, $c_d(0) = 0$.

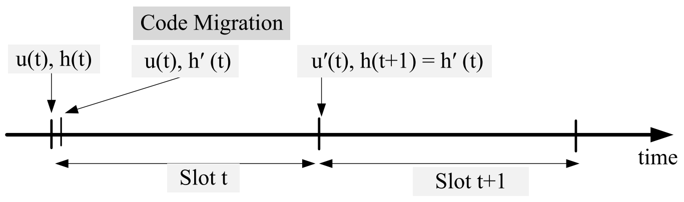


FIGURE 12.13

Edge cloud time slots. At the beginning of slot t the user and service locations are $u(t)$ and $h(t)$, respectively. The service migrates to $h'(t)$, as soon as slot t starts and during the slot t the system operates with $u(t)$ and $h'(t)$. At the beginning of slot $t+1$ the service location is $h(t+1) = h'(t)$.

- π – the policy used for control decision based on $s(t)$.
- $a_\pi(s(t))$ – the control action taken under policy π when the system is in state $s(t)$.
- $\mathcal{C}_{a_\pi}$ – the sum of migration and transmission costs incurred by a control $a_\pi(s(t))$ in slot t. $\mathcal{C}_{a_\pi} = c_m(\|h(t) - h'(t)\|) + c_d(\|u(t) - h'(t)\|)$.
- γ – discount factor, $0 < \gamma < 1$.
- $\mathcal{V}_\pi$ – expected discount under policy π.

The Markov decision process controller makes a decision at the beginning of each slot. The decision could be:

1. Do not migrate the service; then the cost is $c_m(x) = 0$.
2. Migrate the service from location $h(t)$ to location $h'(t) \in \mathcal{L}$; $c_m(x) > 0$.

Given a policy π, its long-term expected discounted sum cost is

$$\mathcal{V}_\pi(s_0) = \lim_{t\to\infty} \mathbb{E}\left\{\sum_{\tau=0}^{t} \gamma^\tau \mathcal{C}_{a_\pi}(s(\tau)) \mid s(0) = s_0\right\}. \tag{12.31}$$

An optimal control policy is one that minimizes $\mathcal{V}_\pi(s_0)$ starting from any initial state

$$\mathcal{V}^*(s_0) = \min_\pi \mathcal{V}_\pi(s_0), \quad \forall s_0. \tag{12.32}$$

A stationary policy is given by Bellman's equation written as

$$\mathcal{V}^*(s_0) = \min_\alpha \left\{\mathcal{C}_\alpha(s_0) + \gamma \sum_{s_1 \in \mathcal{L}\times\mathcal{L}} \mathcal{P}_\alpha(s_0, s_1)\mathcal{V}^*(s_1)\right\} \tag{12.33}$$

with $\mathcal{P}_\alpha(s_0, s_1)$ the transition probability from state $s'(0) = \alpha(s_0)$ to $s(1) = s_1$. The intermediate state $s'(t)$ has no randomness when $s(t)$ and $\alpha(\cdot)$ are given.

A proposition reflecting the intuition that it is not optimal to migrate the service to a location that is farther away from the current location of the mobile device has the following intuitive implication that simplifies the search for optimal policy [514]:

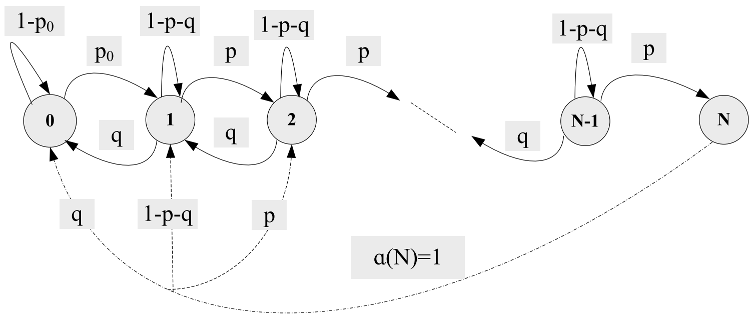


FIGURE 12.14

Markov decision process state transitions assuming a 1-D mobility model for the edge cloud. The mobile device moves one step to the left or right with probability r_1 and stays in the same location with probability $1-2r_1$ thus, $p=q=r_1$ and $p_0=2r_1$. Migration occurs in slot t if and only if $d(t) \geq N$. The actions in slot t with distance $d(t)$ are $\alpha(d(t)) = \alpha(N)$ for $d(t) > N$. This implies that we need only to study the states where $d(t) \in [0, N]$. After an action $\alpha(N)$ the system moves to states 0, 1, and 2 with probabilities $q, 1-p-q$ and p, respectively.

Proposition. *If $c_m(x)$ and $c_d(x)$ are constants and*

$$c_m(0) < c_m(x) \quad \textit{and} \quad c_d(0) < c_d(x) \quad \textit{for} \quad x > 0 \tag{12.34}$$

then the current user location is not optimal.

Figure 12.14 shows the system transition model when the transition probabilities are p_0, p and q assuming a uniform 1-D mobility model. In this model $u(t), h(t)$ and $h'(t)$ are scalars. h' is the new service location chosen such that

$$\|h(t) - h'(t)\| = \|d(t) - d'(t)\| \quad \text{and} \quad \|u(t) - h'(t)\| = d'(t). \tag{12.35}$$

The migration occurs along the shortest path connecting $u(t), h(t)$ and $h'(t)$.

An operating procedure is discussed in [514] and workload scheduling for edge clouds is presented in [491].

12.15 FURTHER READINGS

Several references including [2,301,541,542] discuss the defining characteristics of Big Data applications. Insights into Google's storage architecture for Big Data can be found in [173]. Several Google systems including Mesa, Spanner, and F1 are presented in [212], [119], and [451], respectively.

Data analytics is the subject of [536] and [106] analyzes interactive analytical processing in Big Data systems. [203] covers in-memory performance of Big Data. Continuous pipelines are discussed in [143]. The Starfish system for Big Data analytics is covered in [230] and [249] analyzes enterprise use of Big Data. Several papers including [87] and [273] cover bootstrapping techniques and [10] analyzes approximate query processing. Hoeffding bounds are discussed in [215]. Several references such as [141,142,304] cover Dynamic Data-Driven Application Systems (DDDAS) and the FreshBreeze system.

The Spark Streaming system is discussed in [543] and [12] covers the MillWheel framework developed at Google for building fault-tolerant and scalable data streaming systems. [425] presents caching strategies for data streaming. [29] covers Google's Photon system and system scalability and performance of large-scale system is the topic of [213]. The problems posed by the heavy tail distribution of latency are analyzed in [131].

Mobile devices and applications are covered in the literature including [126,149,441,442,513]. Energy efficiency of mobile computing is analyzed in [342]. The use of mobile devices for space weather monitoring is discussed in [390]. The Follow-me cloud and edge cloud computing are presented in [477] and [491,514].

12.16 EXERCISES AND PROBLEMS

Problem 1. Read [178] and analyze the benefits and the problems related to *dataspaces*. Research the literature for potential application of dataspaces to data management of data-intensive applications in science and engineering.

Problem 2. Discuss the possible solution for stabilizing cloud services mentioned in [177] inspired by BGP routing [204,498].

Problem 3. Discussing the bootstrap method presented in Section 12.3 reference [273] states: "Ideally, we might approximate $\xi(P,n)$ for a given value of n by observing many independent datasets, each of size n. For each dataset, we would compute the corresponding value of u, and the resulting collection of u values would approximate the distribution Q_n, which would in turn yield a direct approximation of the ground truth value $\xi(P,n)$. Furthermore, we could approximate the distribution of bootstrap outputs by simply running the bootstrap on each dataset of size n. Unfortunately, however, in practice we only observe a single set of n data points, rendering this approach an unachievable ideal. To surmount this difficulty, our diagnostic, the BPD Algorithm, executes this ideal procedure for dataset sizes smaller than n. That is, for a given $p \in \mathbb{N}$ and $b \leq \lceil n/p \rceil$ we randomly sample p disjoint subsets of the observed dataset $\mathcal{D}$, each of size b. For each subset, we compute the value of u; the resulting collection of u values approximates the distribution Q_b, in turn yielding a direct approximation of $\xi(P,b)$ the ground truth value for the smaller dataset size b. Additionally, we run the bootstrap on each of the p subsets of size b, and comparing the distribution of the resulting p bootstrap outputs to our ground truth approximation, we can determine whether or not the bootstrap performs acceptably well at sample size b."
(1) Estimate the bootstrap speedup function of n, p, b; ignore the time to compute Δ_i, σ_i. (2) Examine the simulation results in [273] and discuss the impact of the sample size.

Problem 4. Read [10] and discuss the merits and the shortcomings of the three techniques for estimating the sample distribution using only a single sample: non-parametric bootstrap, close-form estimation, and large deviation bounds.

Problem 5. What is a key extraction function and what role does it play in MillWheel? Give an example of such a key extraction function in Zeitgeist.

Problem 6. In systems with very high fan-out a request may exercise an untested code path, causing crashes or extremely long delays on thousands of servers simultaneously. How can this problem be prevented?

Problem 7. Variability in the latency distribution of individual components is magnified at the service level. For example, consider a system where each server typically responds in 10 ms but with a 99th-percentile latency of one second. If a user request must collect responses from 100 such servers in parallel then 63% of user requests will take more than one second [131]. Assume that there are 1 000 servers and the user request needs responses from 1 000 servers running in parallel. What is the probability that response latency will be than one second? What if instead of 1 000 individual requests the user request needs data from 2 000 servers running in parallel?

Problem 8. Research the power consumption of processors used in mobile devices and their energy efficiency. Rank the components of a mobile device in terms of power consumption. Establish a set of guidelines to minimize the power consumption of mobile applications.

Problem 9. What is the main benefit of Algorithm 1 for finding the optimal policy of a Markov Decision Process in [513] compared with the standard approaches?

CHAPTER

ADVANCED TOPICS 13

Cloud functionality is evolving as new services and more diverse and powerful instance types are released every year. For example, Amazon recently introduced Lambda, a service when applications are triggered by user-defined conditions and events. In late 2016 AWS added P2, a powerful, scalable instance with GPU-based parallel compute capabilities. Google has been adding to the software stack for coarse-grained parallelism based on MapReduce. IBM's efforts target computational intelligence displayed by the success of Watson in healthcare and data analytics. Microsoft attempts to extend the range of commercial applications supported by its growing cloud infrastructure.

This chapter aims of providing a glimpse at the challenges cloud computing practitioners and cloud research community will face in the next future. Section 13.1 overviews these challenges. Real-time applications related to IoT, smart cities, the smart grid, and others will undoubtedly join the broad spectrum of cloud applications. Sections 13.2 and 13.3 discuss some of the challenges posed by real-time scheduling.

Cloud infrastructure will most likely continue to scale up for accommodating the demands of an increasingly larger cloud user community. A fair question is whether current cloud management systems can sustain this expansion. Self-organization and self-management alternatives come to mind, but the very slow progress made by the autonomic computing initiative is likely to dampen the enthusiasm of those believing in self-management. Nevertheless, we discuss emergence and self-organization in Section 13.4 and market-based self-organization and combinatorial auctions in Section 13.5.

13.1 A GLIMPSE AT THE FUTURE

Cloud computing will continue to have a profound influence on the large number of individuals and institutions who are now empowered to process huge amounts of data. Cloud research community will most likely be faced with new and challenging problems. Computer clouds operate in an environment characterized by *variability of everything* and by *conflicting requirements*. Such disruptive qualities ultimately demand a new thinking in system design.

Variability is a defining characteristics of a computer cloud. The physical infrastructure consists of servers with different architecture and performance and it is frequently updated as solid state technologies and processor architecture evolves. New software orchestrating a coherent system view is developed every month and the depth of the software stack is continually increasing. If cloud computing will continue to be successful, it is very likely that new applications will emerge. The size of the cloud user population and the diversity of their requirements will grow.

Conflicting requirements in system design are not a novelty, but the depth and the breadth of such conflicts is unprecedented and qualitatively different due to scale of the cloud infrastructure. The many contradictory requirements in the design of computer clouds have to be carefully balanced. For example, resource sharing is a basic design principle, yet strict performance and security isolation of cloud application are also critical. To deliver cheap computing cycles the cloud infrastructure should always run at full capacity, while sufficient resources should be kept in reserve to respond to large workload spikes. The system as a whole should present itself as flawless, indefectible, though the failure rate of the cheap, off-the-shelf system components can be fairly high. Performance guarantees should be provided, while a mix of workloads with very different requirements will continue to dynamically share system resources.

Unquestionably, computer clouds will continue to evolve but how? A parallel between clouds and the Internet is unavoidable. Initially, Arpanet, the precursor of the Internet, was a best-effort data network designed to transfer data files from one location to another. It was a best-effort network doing its best to transport data packets without providing end-to-end delivery guarantees. Support for communication with real-time delivery constraints was not foreseen. The Internet's success forced changes. The Internet of today supports low-latency and high-bandwidth data streaming. The traffic is shaped to guarantee that routers have enough resources to transmit a continuous stream of data with low jitter.

How will computer clouds simultaneously provide QoS guarantees, increase resource utilization, support elasticity, and be more secure? Cloud evolution poses fundamental questions that deserve further research. A first question is whether applications with special requirements such as real-time constraints or applications exhibiting fine-grained parallelism, could migrate to the cloud. Such a migration requires changes in software, in particular in resource management and scheduling components of the software stack. At the same time the hardware and, in particular, the cloud interconnection networks have to offer lower latency and higher bandwidth as we have seen in Section 7.10.

Another question is how to reduce cost by increasing resource utilization, without affecting the QoS promises of cloud computing. It is self-evident that cloud elasticity cannot be supported without some form of overprovisioning, and overprovisioning implies lower average resource utilization. The conclusion is that alternative means to reduce cost should be considered.

A solution practiced by AWS and others is to combine a reservation system with spot allocations. Spot allocations are designed to consume excess resources if and when such resources are available. Cloud users with a good understanding of the resources needed and the time required by their applications should use the reservation system. They will benefit from QoS guarantees and pay more for cloud services. The other cloud users should compete for lower cost spot allocations.

An alternative is to use machine learning and profile cloud users and cloud applications. This solution requires large databases of historic data and data analytics to predict the resource needs and the time required by an application. Once this information is available a virtual private cloud that best fits the profile of the application and user's choices should be configured. Another alternative is to support cloud self-organization and self-management [261,328]. Market-based resource allocation could be at the heart of such an approach in spite of the problems it posses [450], including auctions as suggested in [329,330].

Homogeneity of the cloud computing infrastructure was one of the design tenants early on. The obvious advantages of infrastructure homogeneity are: simplification of resource management, lowering hardware and software maintenance costs. Also, acquiring large volumes of identical hardware components lowers the infrastructure cost.

In the last few years CSPs realized why heterogeneity is clamored by cloud users. As a result, today's clouds have different types of processors and co-processors such as GPUs. In addition to hard disk drives the cloud infrastructure now includes solid state disks. In the future the cloud infrastructure may include data flow engines. It is also likely that islands of systems communicating through Infini-

Band, Myrinet, or other high performance networks will be part of the cloud computing landscape. Such islands in the clouds are necessary to allow the fine-grained parallel scientific and engineering applications to perform well on computer clouds.

It is necessary to address the question how to accommodate cloud heterogeneity, while preventing a dramatic increase of both infrastructure cost and complexity of resource management policies and mechanisms implementing these policies. Market mechanisms proved to be successful in dealing with a diverse set of goods and a large consumer population could provide an answer to this question important for the future of computer clouds.

13.2 CLOUD SCHEDULING SUBJECT TO DEADLINES

Often, a service level agreement specifies the time when the results of computations done on the cloud should be available. This motivates us to examine cloud scheduling subject to deadlines, a topic drawing from a vast body of literature devoted to real-time applications.

Task characterization and deadlines. Real-time applications involve periodic or aperiodic tasks with deadlines. A task is characterized by a tuple (A_i, σ_i, D_i), where A_i is the arrival time, $\sigma_i > 0$ is the data size of the task, and D_i is the *relative deadline*. Instances of a *periodic task*, Π_i^q, with period q are identical, $\Pi_i^q \equiv \Pi^q$, and arrive at times $A_0, A_1, \ldots, A_i, \ldots$, with $A_{i+1} - A_i = q$.

The deadlines satisfy the constraint $D_i \leq A_{i+1}$ and generally the data size is the same, $\sigma_i = \sigma$. The individual instances of *aperiodic tasks*, Π_i, are different, their arrival times A_i are generally uncorrelated, and the amount of data σ_i is different for different instances. The *absolute deadline* for the aperiodic task Π_i is $(A_i + D_i)$.

We distinguish *hard deadlines* from *soft deadlines*. In the first case, if the task is not completed by the deadline other tasks which depend on it may be affected and there are penalties. A hard deadline is strict and expressed precisely as milliseconds, or possibly seconds.

Soft deadlines are more of a guideline and, in general, there are no penalties; soft deadlines can be missed by fractions of the units used to express them, e.g., minutes if the deadline is expressed in hours, or hours if the deadlines is expressed in days. The scheduling of tasks on a cloud is generally subject to soft deadlines though, occasionally, applications with hard deadlines may be encountered.

The model of the system. We consider only aperiodic tasks with arbitrarily divisible workloads. The application runs on a partition of a cloud, a virtual cloud with a *head node* called S_0 and *n worker nodes* $S_1, S_2, \ldots, S_n$. The system is homogeneous, all workers are identical, and the communication time from the head node to every worker node is the same. The head node distributes the workload to worker nodes and this distribution is done sequentially. In this context there are two important problems:

1. The order of execution of the tasks Π_i.
2. The workload partitioning and the task mapping to worker nodes.

Scheduling policies. The most common scheduling policies used to determine the order of execution of the tasks are:

- FIFO – First-In-First-Out, the tasks are scheduled for execution in order of their arrival.

Table 13.1 The parameters used for scheduling with deadlines.

Name	Description
Π_i	the aperiodic tasks with arbitrary divisible load of an application $\mathcal{A}$
A_i	arrival time of task Π_i
D_i	the relative deadline of task Π_i
σ_i	the workload allocated to task Π_i
S_0	head node of the virtual cloud allocated to $\mathcal{A}$
S_i	worker nodes $1 \leq i \leq n$ of the virtual cloud allocated to $\mathcal{A}$
σ	total workload for application $\mathcal{A}$
n	number of nodes of the virtual cloud allocated to application $\mathcal{A}$
n^{min}	minimum number of nodes of the virtual cloud allocated to application $\mathcal{A}$
$\mathcal{E}(n, \sigma)$	the execution time required by n worker nodes to process the workload σ
τ	time for transferring a unit of workload from the head node S_0 to worker S_i
ρ	time for processing a unit of workload
α	the load distribution vector $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$
$\alpha_i \times \sigma$	the fraction of the workload allocated to worker node S_i
Γ_i	time to transfer the data to worker S_i, $\Gamma_i = \alpha_i \times \sigma \times \tau,\ 1 \leq i \leq n$
Δ_i	time the worker S_i needs to process a unit of data, $\Delta_i = \alpha_i \times \sigma \times \rho,\ 1 \leq i \leq n$
t_0	start time of the application $\mathcal{A}$
A	arrival time of the application $\mathcal{A}$
D	deadline of application $\mathcal{A}$
$C(n)$	completion time of application $\mathcal{A}$

- EDF – Earliest Deadline First, the task with the earliest deadline is scheduled first.
- MWF – Maximum Workload Derivative First.

The *workload derivative* $DC_i(n^{min})$ of a task Π_i when n^{min} nodes are assigned to the application is defined as

$$DC_i(n^{min}) = W_i(n_i^{min} + 1) - W_i(n_i^{min}), \tag{13.1}$$

with $W_i(n)$ the workload allocated to task Π_i when n nodes of the cloud are available. The MWF policy requires that:

1. The tasks are scheduled in the order of their derivatives, the one with the highest derivative DC_i first.
2. The number n of nodes assigned to the application is kept to a minimum, n_i^{min}.

Two workload partitioning, the optimal partitioning and the equal partitioning, and task mappings to worker nodes are discussed next. In our discussion we use the derivations and some of the notations in [307]. These notations are summarized in Table 13.1.

Optimal Partitioning Rule (OPR). The workload is partitioned to ensure the earliest possible completion time. Optimality of OPR scheduling requires all tasks to finish execution at the same time.

The head node, S_0, distributes data sequentially to individual worker nodes. The workload assigned to worker node S_i is $\alpha_i\sigma$. The time for delivering input data to worker node S_i is $\Gamma_i = (\alpha_i \times \sigma) \times \tau$,

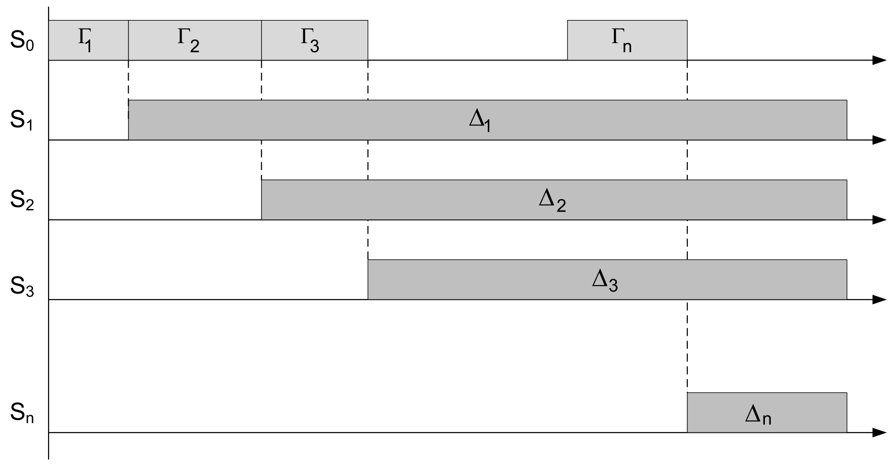


FIGURE 13.1

OPR timing diagram. The algorithm requires worker nodes to complete execution at the same time.

where $1 \leq i \leq n$. Worker node S_i starts processing the data as soon as the transfer is complete. The processing time of worker node S_i is $\Delta_i = (\alpha_i \times \sigma) \times \rho,\ 1 \leq i \leq n$.

The timing diagram in Figure 13.1 allows us to determine the execution time $\mathcal{E}(n, \sigma)$ for the OPR as

$$\begin{aligned}
\mathcal{E}(1,\sigma) &= \Gamma_1 + \Delta_1 \\
\mathcal{E}(2,\sigma) &= \Gamma_1 + \Gamma_2 + \Delta_2 \\
\mathcal{E}(3,\sigma) &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \Delta_3 \\
&\vdots \\
\mathcal{E}(n,\sigma) &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \ldots + \Gamma_n + \Delta_n.
\end{aligned} \tag{13.2}$$

We substitute the expressions of $\Gamma_i, \Delta_i,\ 1 \leq i \leq n$, and rewrite these equations as

$$\begin{aligned}
\mathcal{E}(1,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho \\
\mathcal{E}(2,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_2 \times \sigma \times \rho \\
\mathcal{E}(3,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \alpha_3 \times \sigma \times \rho \\
&\vdots \\
\mathcal{E}(n,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \ldots + \alpha_n \times \sigma \times \tau + \alpha_n \times \sigma \times \rho.
\end{aligned} \tag{13.3}$$

From the first two equations we find the relation between α_1 and α_2 as

$$\alpha_1 = \frac{\alpha_2}{\beta} \quad \text{with} \quad \beta = \frac{\rho}{\tau + \rho}, \quad 0 \leq \beta \leq 1. \tag{13.4}$$

This implies that $\alpha_2 = \beta \times \alpha_1$; it is easy to see that in the general case

$$\alpha_i = \beta \times \alpha_{i-1} = \beta^{i-1} \times \alpha_1. \tag{13.5}$$

But α_i are the components of the load distribution vector thus,

$$\sum_{i=1}^{n} \alpha_i = 1. \tag{13.6}$$

Next, we substitute the values of α_i and obtain the expression for α_1:

$$\alpha_1 + \beta \times \alpha_1 + \beta^2 \times \alpha_1 + \beta^3 \times \alpha_1 \ldots \beta^{n-1} \times \alpha_1 = 1 \quad \text{or} \quad \alpha_1 = \frac{1-\beta}{1-\beta^n}. \tag{13.7}$$

We have now determined the load distribution vector and we can now determine the execution time as

$$\mathcal{E}(n,\sigma) = \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho = \frac{1-\beta}{1-\beta^n}\sigma(\tau + \rho). \tag{13.8}$$

Call $C^{\mathcal{A}}(n)$ the completion time of an application $\mathcal{A} = (A, \sigma, D)$ which starts processing at time t_0 and runs on n worker nodes; then

$$C^{\mathcal{A}}(n) = t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau + \rho). \tag{13.9}$$

The application meets its deadline if and only if

$$C^{\mathcal{A}}(n) \leq A + D, \tag{13.10}$$

or

$$t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau + \rho) \leq A + D. \tag{13.11}$$

But, $0 < \beta < 1$ thus, $1 - \beta^n > 0$ and it follows that

$$(1-\beta)\sigma(\tau + \rho) \leq (1-\beta^n)(A + D - t_0). \tag{13.12}$$

The application can meet its deadline only if $(A + D - t_0) > 0$ and under this condition this inequality becomes

$$\beta^n \leq \gamma \quad \text{with} \quad \gamma = 1 - \frac{\sigma \times \tau}{A + D - t_0}. \tag{13.13}$$

If $\gamma \leq 0$ there is not enough time even for data distribution and the application should be rejected. When $\gamma > 0$ then $n \geq \frac{\ln \gamma}{\ln \beta}$. Thus, the minimum number of nodes for the OPR strategy is

$$n^{min} = \lceil \frac{\ln \gamma}{\ln \beta} \rceil. \tag{13.14}$$

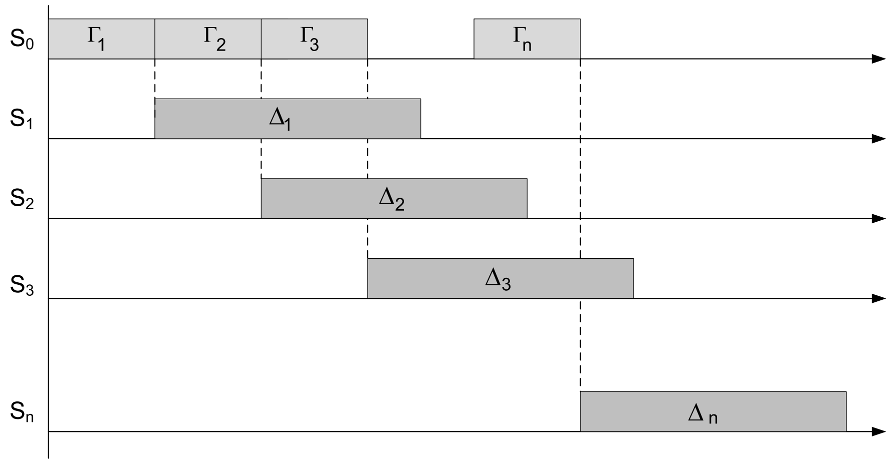


FIGURE 13.2

EPR timing diagram.

Equal Partitioning Rule. EPR assigns an equal workload to individual worker nodes, $\alpha_i = 1/n$. The workload allocated to worker node S_i is σ/n. The head node, S_0, distributes sequentially the data to individual worker nodes. The time to deliver the input data to S_i is $\Gamma_i = (\sigma/n) \times \tau,\ 1 \leq i \leq n$. Worker node S_i starts processing the data as soon as the transfer is complete. The processing time for node S_i is $\Delta_i = (\sigma/n) \times \rho,\ 1 \leq i \leq n$.

From the diagram in Figure 13.2 we see that

$$\mathcal{E}(n,\sigma) = \sum_{i=1}^{n} \Gamma_i + \Delta_n = n \times \frac{\sigma}{n} \times \tau + \frac{\sigma}{n} \times \rho = \sigma \times \tau + \frac{\sigma}{n} \times \rho. \tag{13.15}$$

The condition for meeting the deadline, $C^{\mathcal{A}}(n) \leq A + D$, leads to

$$t_0 + \sigma \times \tau + \frac{\sigma}{n} \times \rho \leq A + D \quad \text{or} \quad n \geq \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau}. \tag{13.16}$$

Thus,

$$n^{min} = \lceil \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau} \rceil. \tag{13.17}$$

The pseudo code for a general schedulability test for FIFO, EDF, and MWF scheduling policies, for two node allocation policies, MN (minimum number of nodes) and AN (all nodes), and for OPR and EPR partitioning rules is given in [307]. The same reference reports on a simulation study for ten algorithms.

The generic format of the names of the algorithms is *Sp-No-Pa* with Sp=FIFO/EDF/MWF, No=MN/AN, and Pa=OPR/EPR. For example, MWF-MN-OPR uses MWF scheduling, minimum number of nodes, and OPR partitioning. The relative performance of the algorithms depends on the relations between the unit cost of communication τ and the unit cost of computing ρ.

Table 13.2 The parameters used for Hadoop scheduling with deadlines.

Name	Description
Q	the query $Q = (A, \sigma, D)$
A	arrival time of query Q
D	deadline of query Q
Π_m^i	a map task, $1 \le i \le u$
Π_r^j	a reduce task, $1 \le j \le v$
J	the job to perform the query $Q = (A, \sigma, D)$, $J = (\Pi_m^1, \Pi_m^2, \ldots, \Pi_m^u, \Pi_r^1, \Pi_r^2, \ldots, \Pi_r^v)$
τ	cost for transferring a data unit
ρ_m	map task time for processing a unit data
ρ_r	reduce task time for processing a unit data
n_m	number of map slots
n_r	number of reduce slots
n_m^{min}	minimum number of slots for the map task
n	total number of slots, $n = n_m + n_r$
t_m^0	start time of the map task
t_r^{max}	maximum value for the start time of the reduce task
α	map distribution vector; the EPR strategy is used and , $\alpha_i = 1/u$
ϕ	filter ratio, the fraction of the input produced as output by the map process

13.3 SCHEDULING MAPREDUCE APPLICATIONS SUBJECT TO DEADLINES

We now discuss scheduling of MapReduce applications on the cloud subject to deadlines. Several options for scheduling Apache Hadoop, an open source implementation of the MapReduce algorithm are: FIFO, the Fair Scheduler [541], the Capacity Scheduler, and the Dynamic Proportional Scheduler [438].

A recent paper [264] applies the deadline scheduling framework discussed in Section 13.2 to Hadoop tasks. Table 13.2 summarizes the notations used for this analysis. The term *slots* is equivalent with *nodes* and means the number of instances.

We make two assumptions for our initial derivation:

- The system is homogeneous, ρ_m and ρ_r, the cost of processing a unit data by the Map and the Reduce tasks, respectively, are the same for all servers.
- Load equipartition.

Under these conditions the duration of the job J with input of size σ is

$$\mathcal{E}(n_m, n_r, \sigma) = \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right]. \tag{13.18}$$

Thus, the condition that the query $Q = (A, \sigma, D)$ with arrival time A meets the deadline D can be expressed as

$$t_m^0 + \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right] \leq A + D. \tag{13.19}$$

It follows immediately that the maximum value for the startup time of the *reduce* task is

$$t_r^{max} = A + D - \sigma \phi \left(\frac{\rho_r}{n_r} + \tau \right). \tag{13.20}$$

We now plug the expression of the maximum value for the startup time of the *reduce* task in the condition to meet the deadline

$$t_m^0 + \sigma \frac{\rho_m}{n_m} \leq t_r^{max}. \tag{13.21}$$

It follows immediately that n_m^{min}, the minimum number of slots for the *map* task, satisfies the condition

$$n_m^{min} \geq \frac{\sigma \rho_m}{t_r^{max} - t_m^0}, \quad \text{thus,} \quad n_m^{min} = \lceil \frac{\sigma \rho_m}{t_r^{max} - t_m^0} \rceil. \tag{13.22}$$

The assumption of homogeneity of the servers can be relaxed and assume that individual servers have different costs for processing a unit workload $\rho_m^i \neq \rho_m^j$ and $\rho_t^i \neq \rho_t^j$. In this case we can use the minimum values $\rho_m = \min \rho_m^i$ and $\rho_r = \min \rho_r^i$ in the expression we derived.

A Constraints Scheduler based on this analysis and an evaluation of the effectiveness of this scheduler are presented in [264].

13.4 EMERGENCE AND SELF-ORGANIZATION

Computer clouds are complex systems and should be analyzed in the context of the environment they operate in. The more diverse the environment, the more challenging is the cloud resource management. Cloud self-management and self-organization [328] offer a glimpse of hope.

Two most important concepts for understanding complex systems are *emergence* and *self-organization*. *Emergence* lacks a clear and widely accepted definition, but it is generally understood as *a property of a system that is not predictable from the properties of individual system components.* There is a continuum of emergence spanning multiple scales of organization. Halley and Winkler argue that simple emergence occurs in systems at, or near thermodynamic equilibrium while complex emergence occurs only in non-linear systems driven far from equilibrium by the input of matter or energy [219].

Physical phenomena which do not manifest themselves at microscopic scales, but occur at macroscopic scale are manifestations of emergence. For example, temperature is a manifestation of the microscopic behavior of large ensembles of particles. For such systems at equilibrium the temperature is proportional to the average kinetic energy per degree of freedom. This is not true for ensembles of a small number of particles. Even the laws of classical mechanics can be viewed as limiting cases of quantum mechanics applied to large masses.

Emergence could be critical for complex systems such as financial systems, the air-traffic control system, and the power grid. The May 6, 2010 event when Dow Jones Industrial Average dropped 600 points in a short period of time is a manifestation of emergence. The cause of this failure of the trading systems is attributed to interactions of trading systems developed independently and owned by organizations which work together, but their actions are motivated by self-interest.

A recent paper [460] points out that dynamic coalitions of software-intensive systems used for financial activities pose serious challenges because there is no central authority and there are no means to control the behavior of the individual trading systems. The failures of the power grid (for example, the Northeast blackout of 2003) can also be attributed to emergence. Indeed, during the first few hours of this event the cause of the failure could not be identified due to the large number of independent systems involved. It was established only later that multiple causes, including the deregulation of the electricity market and the inadequacy of the transmission lines of the power grid, contributed to this failure.

Informally, self-organization means synergetic activities of elements when no single element acts as a coordinator and the global patterns of behavior are distributed [188,445]. The intuitive meaning of self-organization is captured by the observation of Alan Turing [490]: "global order can arise from local interactions."

Self-organization is prevalent in nature; for example, in chemistry this process is responsible for molecular self-assembly, for self-assembly of monolayers, for the formation of liquid and colloidal crystals, and in many other instances. Spontaneous folding of proteins and other biomacromolecules, the formation of lipid bilayer membranes, the flocking behavior of different species, the creation of structures by social animals, are all manifestation of self-organization of biological systems.

Inspired by biological systems, self-organization was proposed for the organization of different types of computing and communication systems [240,325], including sensor networks, for space exploration [236], or even for economical systems [282].

The generic attributes of complex systems exhibiting self-organization are summarized in Table 13.3. Non-linearity of physical systems used to build computing and communication systems has countless manifestations and consequences. For example, when the clock rate of a microprocessor doubles, the power dissipation increases $4 - 8$ $(2^2 - 2^3)$ times, depending of the solid state technology used. This means that the heat removal system of much faster microprocessors has to use a different technology when we double the speed.

This non-linearity is ultimately the reason why in the last years we have seen the clock rate of general-purpose microprocessors increasing only slightly[1]. Nevertheless the number of transistors used

[1]In 1975, the Intel 8080 had a clock rate of 2 MHz; the HP PA-7100, a RISC microprocessor released in 1992, and the Intel P5 Pentium, released in 1995, had a 100 MHz clock rate; in 2002 Intel Pentium 4 had a clock rate of 3 GHz.

Table 13.3 Attributes associated with self-organization and complexity.

Simple systems; no self-organization	Complex systems; self-organization
Mostly linear	Non-linear
Close to equilibrium	Far from equilibrium
Tractable at component level	Intractable at component level
One or few scales of organization	Many scales of organization
Similar patterns at different scales	Different patterns at different scales
Do not require a long history	Require a long history
Simple emergence	Complex emergence
Unaffected by phase transitions	Affected by phase transitions
Limited scalability	Scale-free

to build multi-core chips has increased as postulated by Moore's law. This example illustrates also the so called *incommensurate scaling,* another attribute of complex systems. Incommensurate scaling means that when the size of the system, or when one of its important attributes such as speed increases, different system components are subject to different scaling rules.

The fact that computing and communication systems operate far from equilibrium is clearly illustrated by the traffic carried out by the Internet; there are patterns of traffic specific to the time of the day but, there is no steady-state. The many scales of the organization and the fact that there are different patterns at different scales is also clear in the Internet which is a collection of networks where, in turn, each network is also a collection of smaller networks, each one with its own specific traffic patterns.

The concept of *phase transition* comes from thermodynamics and describes the transformation, often discontinuous, of a system from one phase/state to another, as a result of a change in the environment. Examples of phase transitions are: *freezing*, transition from liquid to solid and its reverse, *melting*; *deposition* transition from gas to solid and its reverse, *sublimation*; *ionization*, transition from gas to plasma and its reverse, *recombination*.

Phase transitions can occur in computing and communication systems due to avalanche phenomena, when the process designed to eliminate the cause of an undesirable behavior leads to a further deterioration of the systems state. A typical example is thrashing due to competition among several memory-intensive processes which lead to excessive page faults.

Another example is an acute congestion which can cause a total collapse of a network; the routers start dropping packets and, unless congestion avoidance and congestion control means are in place and operate effectively, the load increases as senders retransmit packets and the congestion increases. To prevent such phenomena some form of *negative feedback* has to be built into the system.

A defining attribute of *self-organization* is scalability, the ability of the system to grow without affecting its global function(s). Complex systems encountered in nature, or man-made, exhibit an intriguing property, they enjoy a *scale-free organization* [51,52]. This property reflects one of the few attributes of self-organization that can be precisely quantified.

The scale-free organization can be best explained in terms of the network model of the system, a random graph [71] with vertices representing the entities and the links representing the relationships among them. In a scale-free organization the probability $P(m)$ that a vertex interacts with m other vertices decays as a power law

$$P(m) \approx m^{-\gamma} \tag{13.23}$$

with γ a real number, regardless of the type and function of the system, the identity of its constituents and the relationships between them.

Empirical data available for social networks, power grids, the web, or the citation of scientific papers, confirm this trend. As an example of a social network, consider the collaborative graph of movie actors where links are present if two actors were ever cast in the same movie; in this case $\gamma \approx 2.3$. The power grid of the Western US has some 5 000 vertices representing power generating stations and in this case $\gamma \approx 4$.

The exponent of the World Wide Web scale-free network is $\gamma \approx 2.1$. This means that the probability that m pages point to one page is $P(m) \approx m^{-2.1}$ [52]. Recent studies indicate that $\gamma \approx 3$ for the citation of scientific papers. The larger the network, the closer a power law with $\gamma \approx 3$ approximates the distribution [51].

13.5 RESOURCE BUNDLING; COMBINATORIAL AUCTIONS FOR CLOUD RESOURCES

Resources in a cloud are allocated in *bundles*; users get maximum benefit from a specific combination of resources. Indeed, along with CPU cycles, an application needs specific amounts of main memory, disk space, network bandwidth, and so on. Resource bundling complicates traditional resource allocation models; it has generated an interest in economic models and, in particular, in auction algorithms. In the context of cloud computing, an auction is the allocation of resources to the highest bidder.

Combinatorial auctions. Auctions in which participants can bid on combinations of items or *packages* are called *combinatorial auctions* [120]. Such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the *Simultaneous Clock Auction* [41] and the *Clock Proxy Auction* [42]. The algorithm discussed in this section and introduced in [465] is called *Ascending Clock Auction, (ASCA)*. In all these algorithms the current price for each resource is represented by a "clock" seen by all participants at the auction.

We consider a strategy when prices and allocation are set as a result of an auction; in this auction, users provide bids for desirable bundles and the price they are willing to pay. We assume a population of U users, $u = \{1, 2, \ldots, U\}$, and R resources, $r = \{1, 2, \ldots, R\}$. The bid of user u is $\mathcal{B}_u = \{\mathcal{Q}_u, \pi_u\}$ with $\mathcal{Q}_u = (q_u^1, q_u^2, q_u^3, \ldots)$ an R-component vector.

Each element of this vector, q_u^i, represents a bundle of resources user u would accept and, in return, pay the total price π_u. Each vector component q_u^i is either a positive quantity and encodes the quantity of a resource desired, or if negative, it encodes the quantity of the resource offered. A user expresses her desires as an *indifference set* $\mathcal{I} = (q_u^1 \text{ XOR } q_u^2 \text{ XOR } q_u^3 \text{ XOR } \ldots)$.

The final auction prices for individual resources are given by the vector $p = (p^1, p^2, \ldots, p^R)$ and the amounts of resources allocated to user u are $x_u = (x_u^1, x_u^2, \ldots, x_u^R)$. Thus, the expression $[(x_u)^T p\,]$ represents the total price paid by user u for the bundle of resources if the bid is successful at time T. The scalar $[\min_{q \in \mathcal{Q}_u}(q^T p)]$ is the final price established through the bidding process.

Table 13.4 The constraints for a combinatorial auction algorithm.

$x_u \in \{0 \cup \mathcal{Q}_u\},\ \forall u$	–a user gets all resources or nothing
$\sum_u x_u \leq 0$	–final allocation leads to a net surplus of resources
$\pi_u \geq (x_u)^T p,\ \forall u \in \mathcal{W}$	–auction winners are willing to pay the final price
$(x_u)^T p = \min_{q \in \mathcal{Q}_u}(q^T p),\ \forall u \in \mathcal{W}$	–winners get the cheapest bundle in $\mathcal{I}$
$\pi_u < \min_{q \in \mathcal{Q}_u}(q^T p),\ \forall u \in \mathcal{L}$	–the bids of the losers are below the final price
$p \geq 0$	–prices must be non-negative

The bidding process aims to optimize an *objective function* $f(x, p)$. This function could be tailored to measure the net value of all resources traded, or it can measure the *total surplus*, the difference between the maximum amount the users are willing to pay minus the amount they pay. Other optimization function could be considered for a specific system, e.g., the minimization of energy consumption, or of the security risks.

Pricing and allocation algorithms. A pricing and allocation algorithm partitions the set of users in two disjoint sets, winners and losers, denoted as $\mathcal{W}$ and $\mathcal{L}$, respectively; the algorithm should:

1. Be computationally tractable. Traditional combinatorial auction algorithms such as Vickey-Clarke-Groves (VLG) fail this criteria, they are not computationally tractable.
2. Scale well. Given the scale of the system and the number of requests for service, scalability is a necessary condition.
3. Be objective; partitioning in winners and losers should only be based on the price π_u of a user's bid; if the price exceeds the threshold then the user is a winner, otherwise the user is a loser.
4. Be fair. Make sure that the prices are *uniform*, all winners within a given resource pool pay the same price.
5. Indicate clearly at the end of the auction the unit prices for each resource pool.
6. Indicate clearly to all participants the relationship between the supply and the demand in the system.

The function to be maximized is

$$\max_{x,p} f(x, p). \tag{13.24}$$

The constraints in Table 13.4 correspond to our intuition: (a) the first one states that a user either gets one of the bundles it has opted for or nothing, no partial allocation is acceptable; (b) the second one expresses the fact that the system awards only available resources, only offered resources can be allocated; (c) the third one is that the bid of the winners exceeds the final price; (d) the fourth one states that the winners get the least expensive bundles in their indifference set; (e) the fifth one states that losers bid below the final price; (f) finally, the last one states that all prices are positive numbers.

Ascending Clock Auction algorithm (ASCA). Informally, the participants at the auction based on the ASCA algorithm [465] specify the resource and the quantities of that resource offered or desired at the price listed for that time slot. Then the *excess vector*

$$z(t) = \sum_u x_u(t) \tag{13.25}$$

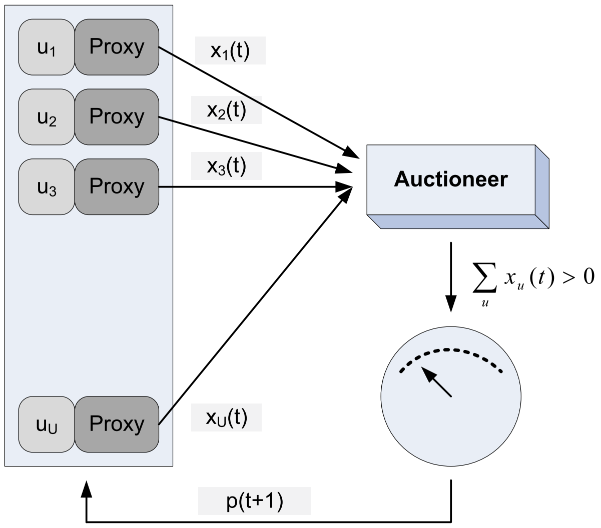


FIGURE 13.3

The schematics of the ASCA algorithm; to allow for a single round auction users are represented by proxies which place the bids $x_u(t)$. The auctioneer determines if there is an excess demand and, in that case, it raises the price of resources for which the demand exceeds the supply and requests new bids.

is computed. If all its components are negative, then the auction stops; negative components mean that the demand does not exceed the offer. If the demand is larger than the offer, $z(t) \geq 0$, then the auctioneer increases the price for items with a positive excess demand and solicits bids at the new price.

The algorithm satisfies conditions 1 through 6 in this section. All users discover the price at the same time and pay or receive a "fair" payment relative to uniform resource prices, the computation is tractable, and the execution time is linear in the number of participants at the auction and the number of resources. The computation is robust, generates plausible results, regardless of the initial parameters of the system.

There is a slight complication as the algorithm involves user bidding in multiple rounds. To address this problem the user proxies automatically adjust their demands on behalf of the actual bidders, as shown in Figure 13.3. These proxies can be modeled as functions which compute the "best bundle" from each $\mathcal{Q}_u$ set given the current price

$$\mathcal{Q}_u = \begin{cases} \hat{q}_u & \text{if } \hat{q}_u^T p \leq \pi_u \quad \text{with } \hat{q}_u \in \arg\min(q_u^T p) \\ 0 & \text{otherwise} \end{cases} \tag{13.26}$$

In this algorithm $g(x(t), p(t))$ is the function for setting the price increase. This function can be correlated with the excess demand $z(t)$ as in $g(x(t), p(t)) = \alpha z(t)^+$ (the notation x^+ means $\max(x, 0)$) with α a positive number. An alternative is to ensure that the price does not increase by an amount larger than δ; in that case $g(x(t), p(t)) = \min(\alpha z(t)^{+}, \delta e)$ with $e = (1, 1, \ldots, 1)$ is an R-dimensional vector and minimization is done componentwise.

The input to the ASCA algorithm: U users, R resources, $\bar{p}$ the starting price, and the update increment function, $g : (x, p) \mapsto \mathbb{R}^R$. The pseudo code of the algorithm is:

```
Pseudo code for the ASCA algorithm.

set $t = 0$, $p(0) = \bar{p}$
    loop
       collect bids $x_u(t) = \mathcal{G}_u(p(t))$   $\forall u$
       calculate excess demand $z(t) = \sum_u x_u(t)$
       if $z(t) < 0$ then
          break
       else
          update prices $p(t+1) = p(t) + g(x(t), p(t))$
          $t \leftarrow t + 1$
       end if
    end loop
```

The convergence of the optimization problem is guaranteed *only if* all participants at the auction are either providers of resources or consumers of resources, but not both providers and consumers at the same time. Nevertheless, the clock algorithm only finds a feasible solution, it does not guarantee its optimality.

The authors of [465] have implemented the algorithm and allowed internal use of it within Google; their preliminary experiments show that the system led to substantial improvements. One of the most interesting side effects of the new resource allocation policy is that the users were encouraged to make their applications more flexible and mobile to take advantage of the flexibility of the system controlled by the ASCA algorithm.

Auctioning algorithms are very appealing because they support resource bundling and do not require a model of the system. At the same time, a practical implementation of such algorithms is challenging. First, requests for service arrive at random times while in an auction all participants must react to a bid at the same time. Periodic auctions must then be organized but this adds to the delay of the response time. Second, there is an incompatibility between cloud elasticity which guarantees that the demand for resources of an existing application will be satisfied immediately and the idea of periodic auctions.

13.6 CLOUD INTEROPERABILITY AND SUPER CLOUDS

Vendor lock-in is a concern, therefore cloud interoperability is a topic of great interest for the cloud community [91,313,332]. This section addresses several questions: What are the challenges? What is realistic to expect now? What could be done in the future for cloud interoperability? It makes only sense to discuss interoperability of PaaS and IaaS cloud delivery models, the expectation that SaaS services offered by one CSP will be offered by others e.g., that Google's Gmail will be supported by Amazon is not realistic.

A workload can migrate from one server to another server in the same data center or among data centeres of the same CSP. Migrating a workload to a different CSP is not feasible at this time. To use multiple clouds data must be replicated and application binaries must be created for all targeted clouds. This is a costly proposition thus, unfeasible in practice.

There is already a significant body of work on cloud standardization carried out at NIST, but it may take some time before the standards are adopted. First, cloud computing is a fast-changing field and early standardization would hinder progress and slowdown or stifle innovation. It is also likely that the CSPs will resist the standardization efforts.

CSPs are adamant to share information about their internal specifications of the software stack, their policies, the mechanisms implementing these policies, and data formats. Each CSP is confident that such information gives it an advantage over the competition. There are also technical reasons why cloud interoperability poses a fair number challenges, some insurmountable due to the current limitations of computing and communication technologies.

So why a complex system such as the Internet is so successful while the development of an Intercloud, a worldwide organization allowing CSPs to share load is so challenging? The Internet is a network of networks and its architecture is based on two simple ideas:

- Every communicating entity must be identified by an address thus, a host at the periphery of the Internet, or a router at its core must have one or more IP address;
- Data sent should be able to reach its destination in this maze of networks, therefore each network should route packets using the same protocol, the IP.

The function of a digital network, regardless of its physical substrate used for communication, is to transport bits of data regardless of what their provenance, music, voice, images, data collected by a sensor, text, or any other conceivable type of information. To make matters even easier these bits can be packaged together in blocks of small, large, medium, or blocks of any desirable size and can be repackaged whenever the need arises. All that matters is to deliver these bits from a source to a destination in a finite amount of time or in a very short time if the application so requires.

Things cannot be more different for an Intercloud. First of all, most applications running on clouds need a large volume of data as input to produce results. Transferring say 1 TB of data over a 10 Gbps network takes 8×10^5 seconds, slightly less than a day. Increasing the network speed by an order of magnitude will still require a few hours to transfer this relatively modest volume of data for most cloud applications. Often, we need to transfer more than 1 TB and it is unlikely that Internet speeds of 100 Gbps will be available to connect data centers to one another.

What we expect from an Intercloud is different from what is expected from the Internet where the only function required is to transport data and where all routers, regardless of their architecture, run software implementing the IP protocol. On the other hand, the spectrum of computations done on a cloud is extremely broad and the heterogeneity of the cloud infrastructure cannot be ignored. Clouds use processors with different architectures, different configurations of cache, memory, and secondary storage, support different operating systems, and use different hypervisors.

The architecture of the server matters; one can only execute code on a server with the same ISA as the one the code was compiled for. The operating system running on a server matters because user code makes system calls to carry out privileged operations and a binary running under one OS cannot be migrated to another OS. The hypervisor running on the server matters because each hypervisor supports only a set of operating systems.

Fortunately, there are VMs and containers so there is a glimpse of hope. A VM including the OS and the application can be migrated to a system with similar architecture and the same hypervisor. Nested virtualization discussed in Section 10.8 allows a hypervisor to run another hypervisor and this idea discussed later in the section adds to the degrees of freedom for VM migration.

Container technologies, such as Docker and LXC, are incredibly useful but one cannot move a Docker container from one host to another. What can be done to preserve data that an application has created inside the container is to commit the changes in the container to an image using Docker *commit*, move the image to a new host, and then start a new container with Docker *run*.

Moreover, a Docker container is intended to run a single application. There are Docker containers to run applications such as MySQL. A new back-end Docker engine, the *libcontainer*, can run any application. LXC containers run an application under an instance of Linux. A Windows-based container runs an application as an instance of Windows.

13.7 IN SEARCH FOR BLOOMS AMID A FLURRY OF CHALLENGES

A cursory look at the cloud computing literature reveals the extraordinary attention given to this emerging field of computer science. Areas such as computer architecture, concurrency, data management and databases, resource management, scheduling, and mobile computing have bloomed in response to the need of finding efficient solutions to the challenges brought about by cloud computing. Even somewhat ossified areas such as operating systems have been brought back to life by problems posed by virtualization and containerization.

Several areas critical for the future of cloud computing, including communication and security, still demand special attention. Increasing the bandwidth and lowering the communication latency will make cloud computing more attractive for real-time applications and for integrations of services related to IoT. Optimization of communication protocols could lower the latency up to the limits imposed by the laws of physics. Ultimately, communication latency depends on the distance between the producer and the consumer of data.

Computer clouds and mobile devices are in a symbiotic relationship with one another and effective communication to/from clouds and inside the cloud infrastructure has to keep pace with advances in processor and storage technology. Faster cloud interconnects are also necessary to accommodate data-intensive and communication-intensive applications in need of a large number of servers working in concert. Applications in computational sciences and engineering exhibiting fine-grained parallelism would greatly benefit from lower latency.

Data security and privacy are major concerns not properly addressed by existing SLAs. Though sensitive information has been leaked or stolen from large data centers, many cloud users are unaware of the potential dangers they are exposed to when entrusting their data to a third party and trusting the protection guaranteed by SLAs.

Strong encryption protects data in storage, but processing encrypted data is only feasible for some types of queries. Most applications only operate with plaintext data thus, encrypted data has to be decrypted before processing. This creates a window of vulnerability that can be exploited by insider attacks. Hybrid clouds offer an alternative to protect sensitive information. In this case effective mechanisms to hide sensitive information stored on the public cloud and revealed only on the private side of the cloud must be conceived.

Virtualization, in spite of its benefits, creates significant complications for software maintenance. A checkpointed virtual machine containing an older version of an operating system without the current security patches may be activated at a later time, opening a window of vulnerability that could affect the entire cloud infrastructure.

Response times plagued by a heavy tail distribution cannot be tolerated by most interactive or real-time applications, but eliminating the tail of the latency at the scale of clouds is an enduring challenge. Another enduring challenge is the reduction of energy consumption and, implicitly, increasing the average server utilization. Elasticity without overprovisioning, requires accurate knowledge of resource consumption.

Resource reservations can help, but reservations place an additional burden on cloud users expected to know well the needs of their applications. Moreover, accurately predicting resource consumption is possible if and only if the system enforces strict performance isolation, yet another major headache for systems based on multi-tenancy.

Even skeptics cautioning about the dangers inherent to systems "too big to fail" have to recognize that the cloud ecosystem plays an important role in the modern society, that it has democratized computing, the same way the web has completely changed the manner we access and use information. The Internet will continue to morph, the web will evolve to a semantic web or Web 3.0. It is thus, fair to expect that computer clouds will continue to change under the pressure from consumers of cloud services and from new technologies.

It is hard to predict how the cloud ecosystem will look five or ten years from now, but it should be perfectly clear that the disruptive qualities of computer clouds ultimately demand a new thinking in system design. The design of large scale systems requires an ab initio preparation for the unexpected, as low probability events occur and can cause major disruptions.

We have seen that the separation of control and routing planes in the Internet is partially responsible for the rapid assimilation of new communication technologies. Only a holistic approach could lead to a similar separation of concerns for computer clouds and allow computing technology to evolve at its lightning pace.

There is a glimmer of hope that machine learning, data analytics, and market-based resource management will play a transformative role in cloud computing [47,328]. As more data are collected after the execution of all instances of an application it may be possible to construct the application profile, optimize its execution, and, ultimately, optimize the overall system performance. It may also be possible to identify conditions leading to phase transitions and prevent their occurrence often leading to data center shutdown.

In this maze of challenges and uncertainties there is one prediction few could argue against: the interest in cloud computing, as well as the need for individuals well trained in this field will continue to grow for the foreseeable future. The incredible pace of developments in cloud computing poses its own challenges and demands the grasp of fundamental concepts in many areas of computer science and computer engineering, as well as curiosity and desire to continually learn.

APPENDIX

CLOUD APPLICATION DEVELOPMENT

A

In the previous chapters our discussion was focused on research issues in cloud computing; now we examine computer clouds from the perspective of an application developer. This chapter presents a few recipes useful to assemble a cloud computing environment on a local system and to use basic cloud functions.

It is fair to assume that the population of application developers and cloud users is, and will continue to be, very diverse. Some cloud users have developed and run parallel applications on clusters or other types of systems for many years and expect an easy transition to the cloud. Others, are less experienced, but willing to learn and expect a smooth learning curve. Many view cloud computing as an opportunity to develop new businesses with minimum investment in computing equipment and human resources.

The questions we address are: How easy is it to use the cloud? How knowledgeable should an application developer be about networking and security? How easy is it to port an existing application to the cloud? How easy is it to develop a new cloud application?

The answers to these questions are different for the three cloud delivery models, SaaS, PaaS, and IaaS; the level of difficulty increases as we move towards the base of the cloud service pyramid as shown in Figure A.1. Recall that SaaS applications are designed for the end-users and are accessed over the web; in this case the user must be familiar with the API of a particular application. PaaS provides a set of tools and services designed to facilitate application coding and deploying, while IaaS provides the hardware and the software for servers, storage, networks, including operating systems and

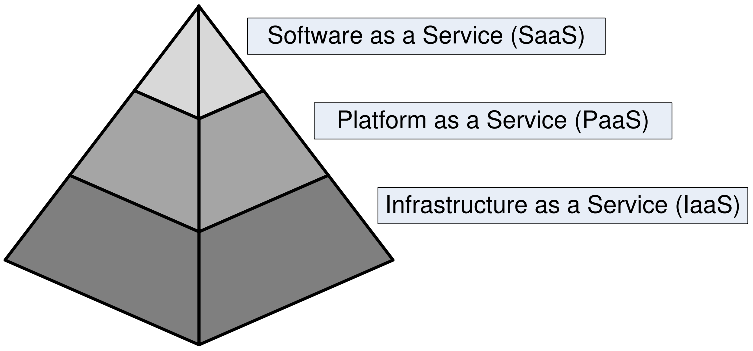

Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)

FIGURE A.1

A pyramid model of cloud computing paradigms; the infrastructure provides the basic resources, the platform adds an environment to facilitate the use of these resources, while software allows direct access to services.

Cloud Computing. DOI: 10.1016/B978-0-12-812810-7.00025-X

storage management software. We restrict our discussion to the IaaS cloud computing model and we concentrate on popular services offered by AWS.

Though the AWS services are well documented, the environment they provide for cloud computing requires some effort to benefit from the full spectrum of services offered. In this section we report on lessons learned from the experience of a group of students with a strong background in programming, networking, and operating systems; each one of them was asked to develop a cloud application for a problem of interest in their own research area. First, we discuss several issues related to cloud security, a major stumbling block for many cloud users; then we present a few recipes for the development of cloud applications, and finally we analyze several cloud applications developed by individuals in this group over a period of less than three months.

A.1 AWS EC2 INSTANCES

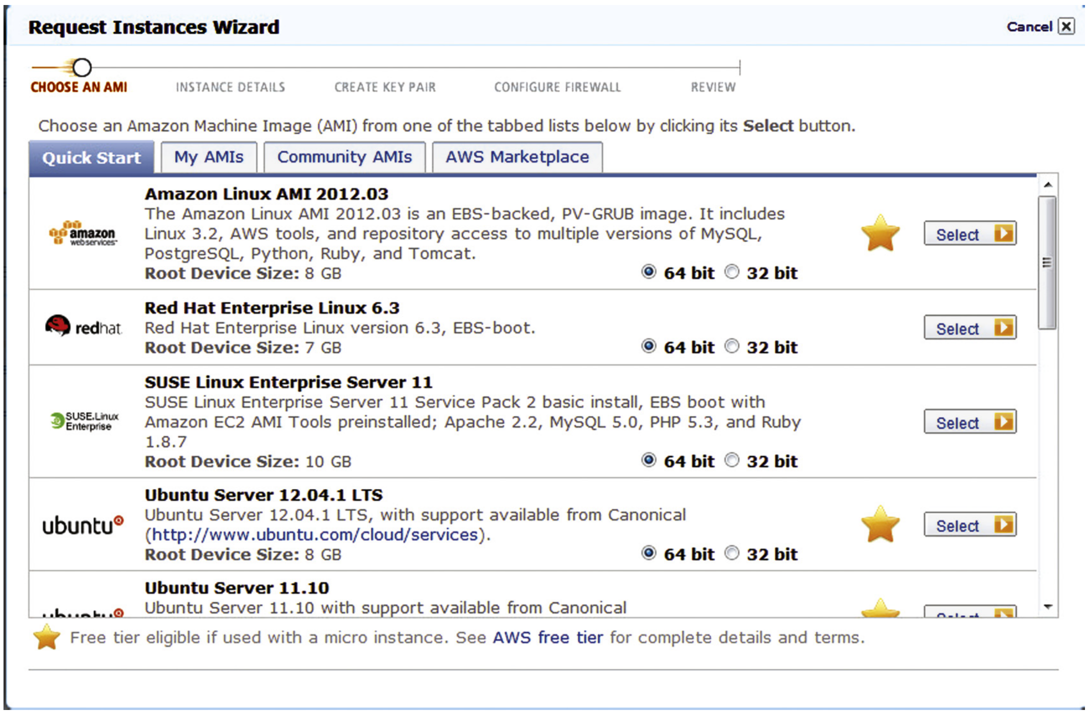
In spite of the wealth of information available from the providers of cloud services, the learning curve of an application developer is still relatively steep. The examples discussed in this chapter are designed to help overcome some of the hurdles faced when someone first attempts to use the AWS. Due to space limitations we have chosen to cover only a few of the very large number of combinations of services, operating systems, and programming environments supported by AWS.

Amazon Web Services are grouped in several categories: computing and networking, storage and content delivery, deployment and management, databases, and application services. In Sections 2.3 and 2.4 we mentioned that new services are continually added to AWS; the look and feel of the web pages changes in time. The screen shots reflect the state of the system at the time of the writing of the first edition of the book, second half of 2012. To access AWS one must first create an account at http://aws.amazon.com/. Once the account is created the Amazon Management Console (AMC) allows the user to select one of the service, e.g., EC2 and then start an instance.

Recall that an EC2 instance is a virtual server started in a region and availability zone selected by the user. Instances are grouped into a few classes and each class has a specific amount of resources such as CPU cycles, main memory, secondary storage, communication, and I/O bandwidth available to it. Several operating systems are supported by AWS including: Amazon Linux, Red Hat Enterprize Linux 6.3, SUSE Linux Enterprize Server 11, Ubuntu Server 12.04.1, as well as several version of Microsoft Windows, see Figure A.2.

The next step is to create an AMI (Amazon Machine Image) on one of the platforms supported by AWS and start an instance using the *RunInstance* API. An AMI is a unit of deployment, an environment including all information necessary to set up and boot an instance. If an application needs more than 20 instances then a special form must be filled out. The local instance persists in storage only for the duration of an instance. The data persist when an instance is started using the Amazon EBS (Elastic Block Storage) and then the instance can be restarted at a later time.

Once an instance is created the user can perform several actions; for example, connect to the instance, launch more instances identical to the current one, or create an EBS AMI. The user can also terminate, reboot, or stop the instance, see Figure A.3. The *Network & Security* panel allows the creation of *Security Groups, Elastic IP addresses, Placement Groups, Load Balancers* and *Key Pairs* (see the discussion in Section A.3), while the EBS panel allows the specification of volumes and the creation of snapshots.



Choose an Amazon Machine Image (AMI) from one of the tabbed lists below by clicking its **Select** button.

Amazon Linux AMI 2012.03
The Amazon Linux AMI 2012.03 is an EBS-backed, PV-GRUB image. It includes Linux 3.2, AWS tools, and repository access to multiple versions of MySQL, PostgreSQL, Python, Ruby, and Tomcat.
Root Device Size: 8 GB [✓ **64 bit**] **32 bit**

Red Hat Enterprise Linux 6.3
Red Hat Enterprise Linux version 6.3, EBS-boot.
Root Device Size: 7 GB [✓ **64 bit**] **32 bit**

SUSE Linux Enterprise Server 11
SUSE Linux Enterprise Server 11 Service Pack 2 basic install, EBS boot with Amazon EC2 AMI Tools preinstalled; Apache 2.2, MySQL 5.0, PHP 5.3, and Ruby 1.8.7
Root Device Size: 10 GB [✓ **64 bit**] **32 bit**

Ubuntu Server 12.04.1 LTS
Ubuntu Server 12.04.1 LTS, with support available from Canonical (http://www.ubuntu.com/cloud/services).
Root Device Size: 8 GB [✓ **64 bit**] **32 bit**

Ubuntu Server 11.10
Ubuntu Server 11.10 with support available from Canonical

Free tier eligible if used with a micro instance. See AWS free tier for complete details and terms.

FIGURE A.2

The Instance menu allows the user to select from existing AMIs.

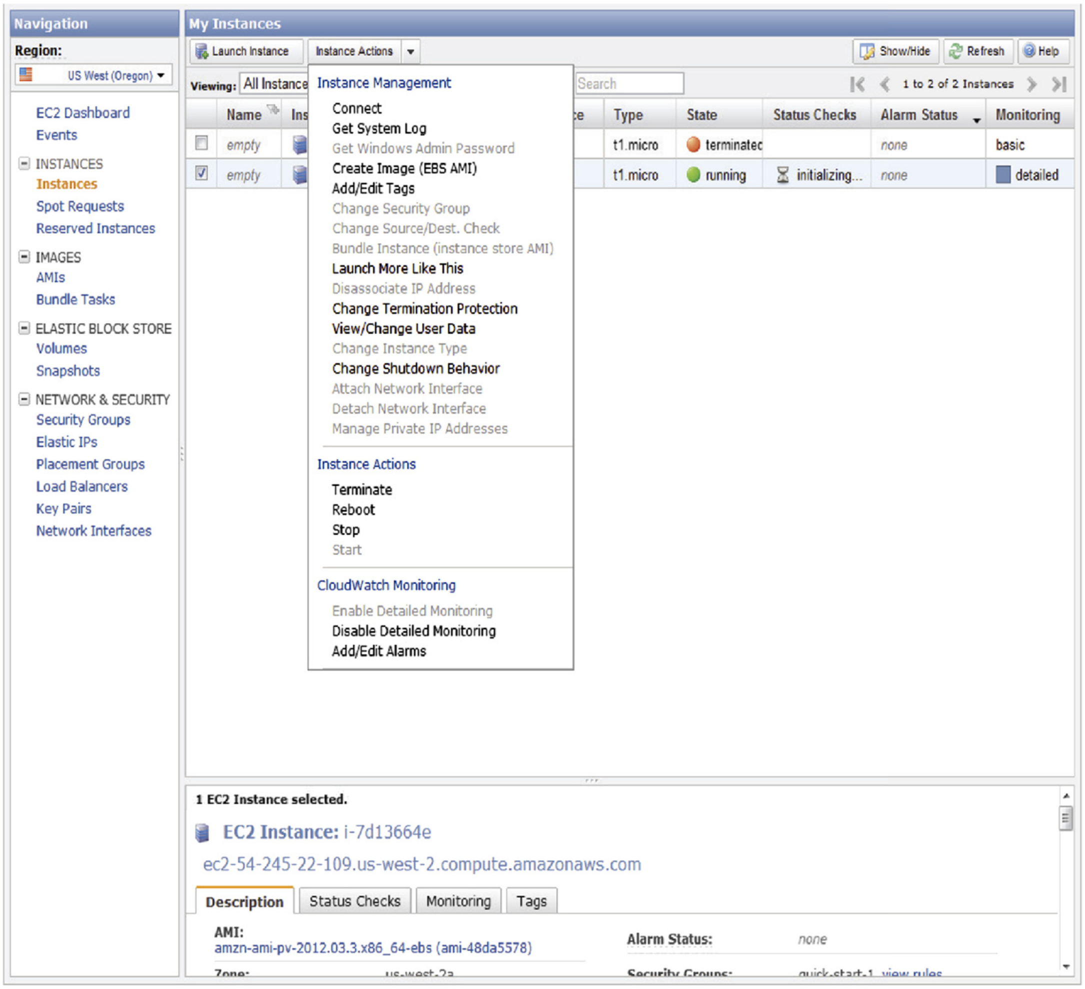
A.2 CONNECTING CLIENTS TO CLOUD INSTANCES THROUGH FIREWALLS

A firewall is a software system based on a set of rules for filtering network traffic; its function is to protect a computer in a local area network from unauthorized access. The first generation of firewalls, deployed in the late 1980s, carried out *packet filtering*; they discarded individual packets which did not match a set of acceptances rules. Such firewalls operated below the transport layer, and discarded packets based on the information in the headers of physical, data link, and transport layer protocols.

The second generation of firewalls operate at the transport layer and maintain the state of all connections passing through them. Unfortunately, this traffic filtering solution opened the possibility of *denial of service attacks*; a denial of service (DOS) attack targets a widely used network service and forces the operating system of the host to fill the connection tables with illegitimate entries. DOS attacks prevent legitimate access to the service.

The third generation of firewalls "understand" widely-used application layer protocols such as FTP, HTTP, TELNET, SSH, and DNS. These firewalls examine the header of application layer protocols and support *intrusion detection systems*.

Firewalls screen incoming traffic and, sometimes, filter outgoing traffic as well. A first filter encountered by the incoming traffic in a typical network is a firewall provided by the operating system of the



Name	Type	State	Status Checks	Alarm Status	Monitoring
empty	t1.micro	terminated		none	basic
empty	t1.micro	running	initializing...	none	detailed

1 EC2 Instance selected.

EC2 Instance: i-7d13664e

ec2-54-245-22-109.us-west-2.compute.amazonaws.com

AMI: amzn-ami-pv-2012.03.3.x86_64-ebs (ami-48da5578)

Alarm Status: none

Zone: us-west-2a

Security Groups: quick-start-1 view rules

FIGURE A.3

The *Instance Action* pull-down menu of the *Instances* panel of the *AWS Management Console* allows the user to interact with an instance, e.g., *Connect, Create an EBS AMI Image*, and so on.

router; the second filter is a firewall provided by the operating system running on the local computer, see Figure A.4.

Typically, the local area network (LAN) of an organization is connected to the Internet via a router; a router firewall often hides the true address of hosts in the local network using the network address translation (NAT) mechanism. The hosts behind a firewall are assigned addresses in a "private address range" and the router uses the NAT tables to filter the incoming traffic and translate external IP addresses to private ones. The mapping between the pair *(external address, external port)* and the

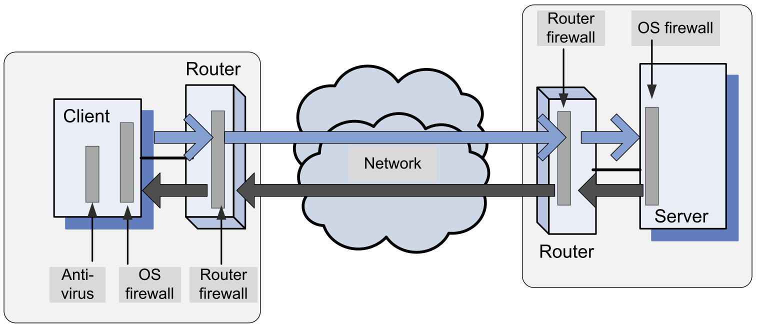


FIGURE A.4

Firewalls screen incoming and sometimes outgoing traffic. The first obstacle encountered by the inbound or outbound traffic is a router firewall, the next one is the firewall provided by the host operating system; sometimes, the antivirus software provides a third line of defense.

Table A.1 Firewall rule setting. The columns indicate if a feature is supported or not by an operating system: the second column – a single rule can be issued to accept/reject a default policy; the third and fourth columns – filtering based on IP destination and source address, respectively; the fifth and sixth columns – filtering based on TCP/UDP destination and source ports, respectively; the seventh and eights columns – filtering based on Ethernet MAC destination and source address, respectively; the ninth and tenth columns – inbound (ingress) and outbound (egress) firewalls, respectively.

Operating system	Def rule	IP dest addr	IP src addr	TCP/ UDP dest port	TCP/ UDP src port	Ether MAC dest	Ether MAC src	In-bound fwall	Out-bound fwall
Linux iptables	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OpenBSD	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Windows XP	No	No	Yes	Partial	No	No	No	Yes	No
Cisco Acces List	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Juniper Networks	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

(internal address, internal port) tuple carried by the network address translation function of the router firewall is called a *pinhole*.

If one tests a client-server application with the client and the server in the same local area network, the packets do not cross a router; once a client from a different LAN attempts to use the service, the packets may be discarded by the firewall of the router. The application may no longer work if the router is not properly configured.

The firewall support in several operating systems is discussed next. Table A.1 summarizes the options supported by different operating systems running on a host or on a router.

A *rule* specifies a filtering option at: (i) the network layer, when filtering is based on the destination/source IP address; (ii) the transport layer, when filtering is based on destination/source port number; (iii) the MAC layer, when filtering is based on the destination/source MAC address.

In Linux or Unix systems the firewall can be configured only by someone with a *root* access using the *sudo* command. The firewall is controlled by a kernel data structure, the *iptables*. The *iptables* command is used to set up, maintain, and inspect the tables of the *IPv4* packet filter rules in the Linux kernel. Several tables may be defined; each table contains a number of built-in chains and may also contain user-defined chains.

A *chain* is a list of rules which can match a set of packets: the *INPUT* rule controls all incoming connections; the *FORWARD* rule controls all packets passing through this host; and the *OUTPUT* rule controls all outgoing connections from the host. A *rule* specifies what to do with a packet that matches: *Accept* – let the packet pass; *Drop* – discharge the packet; *Queue* – pass the packet to the user space; *Return* – stop traversing this chain and resume processing at the head of the next chain. For complete information on the *iptables* see http://linux.die.net/man/8/iptables.

To get the status of the firewall, specify the L (List) action of the *iptables* command

```
 sudo iptables -L
```

As a result of this command the status of the *INPUT, FORWARD,* and *OUTPUT* chains will be displayed.

To change the default behavior for the entire chain, specify the action P (Policy), the chain name, and target name; e.g., to allow all outgoing traffic to pass unfiltered use

```
sudo iptables -P OUTPUT ACCEPT
```

To add a new security rule specify: the action, A (add), the chain, the transport protocol, *tcp* or *udp*, and the target ports as in

```
sudo iptables -A INPUT -p -tcp -dport ssh -j ACCEPT
sudo iptables -A OUTPUT -p -udp -dport 4321 -j ACCEPT
sudo iptables -A FORWARD -p -tcp -dport 80 -j DROP
```

To delete a specific security rule from a chain, set the action D (Delete) and specify the chain name and the rule number for that chain; the top rule in a chain has number 1:

```
sudo iptables -D INPUT 1
sudo iptables -D OUTPUT 1
sudo iptables -D FORWARD 1
```

By default the Linux virtual machines on Amazon's EC2 accept all incoming connections.

The ability to access that virtual machine will be permanently lost when a user accesses an EC2 virtual machine using ssh and then issues the following command

```
sudo iptables -P INPUT DROP.
```

The access to a Windows firewall is provided by a GUI accessed as follows:

```
Control Panel -> System & Security -> Windows Firewall -> Advanced Settings
```

The default behavior for incoming and/or outgoing connections can be displayed and changed from the window *Windows Firewall with Advanced Security on Local Computer*.

The access to the Windows XP firewall is provided by a GUI accessed by selecting *Windows Firewall* in the *Control Panel*. If the status is *ON*, incoming traffic is blocked by default, and a list of Exceptions (as noted on the *Exceptions* tab) define the connections allowed. The user can only define exceptions for: *tcp* on a given port, *udp* on a given port, and a specific program. *Windows XP* does not provide any control over outgoing connections.

Antivirus software running on a local host may provide an additional line of defense. For example, the Avast antivirus software (see www.avast.com) supports several real-time shields. The *Avast network shield* monitors all incoming traffic; it also blocks access to known malicious websites. The *Avast web shield* scans the HTTP traffic and monitors all web browsing activities. The antivirus also provides statistics related to its monitoring activities.

A.3 SECURITY RULES FOR APPLICATION- AND TRANSPORT-LAYER PROTOCOLS IN EC2

A client must know the IP address of a virtual machine in the cloud, to be able to connect to it. Domain Name Service (DNS) is used to map human-friendly names of computer systems to IP addresses in the Internet or in private networks. DNS is a hierarchical distributed database and plays a role reminiscent of an Internet phone book.

In 2010 Amazon announced a DNS service called Route 53 to route users to AWS services and to infrastructure outside of AWS. A network of DNS servers scattered across the globe, which enables customers to gain reliable access to AWS and place strict controls over who can manage their DNS system by allowing integration with AWS Identity and Access Management (IAM).

For several reasons, including security and the ability of the infrastructure to scale up, the IP addresses of instances visible to the outside world are mapped internally to private IP addresses. A virtual machine running under Amazon's EC2 has several IP addresses:

1. *EC2 Private IP Address:* The internal address of an instance; it is only used for routing within the EC2 cloud.
2. *EC2 Public IP Address:* Network traffic originating outside the AWS network must use either the public IP address or the elastic IP address of the instance. The public IP address is translated using the Network Address Translation (NAT) to the private IP address when an instance is launched and it is valid until the instance is terminated. Traffic to the public address is forwarded to the private IP address of the instance.
3. *EC2 Elastic IP Address:* The IP address allocated to an account and used by traffic originated from outside AWS. NAT is used to map an elastic IP address to the private IP address. Elastic IP addresses allow the cloud user to mask instance or availability zone failures by programmatically re-mapping a public IP addresses to any instance associated with the user's account. This allows fast recovery after a system failure; for example, rather than waiting for a cloud maintenance team to reconfigure or replace the failing host, or waiting for DNS to propagate the new public IP to all of the customers of a web service hosted by EC2, the web service provider can re-map the elastic IP address to a replacement instance. Amazon charges a fee for unallocated Elastic IP addresses.

To control access to a user VMs AWS uses *security groups*. A VM instance belongs to one, and only one, security group which can only be defined before the instance is launched. Once an instance is running, the security group the instance belongs to cannot be changed. However, more than one instance can belong to a single security group.

Security group rules control inbound traffic to the instance and have no effect on outbound traffic from the instance. The inbound traffic to an instance, either from outside the cloud or from other instances running on the cloud, is blocked, unless a rule stating otherwise is added to the security group of the instance. For example, assume a client running on instance A in the security group Σ_A is to connect to a server on instance B listening on TCP port P, where B is in security group Σ_B. A new rule must be added to security group Σ_B to allow connections to port P; to accept responses from server B a new rule must be added to security group Σ_A.

The following steps allow the user to add a security rule:

1. Sign in to the AWS Management Console at http://aws.amazon.com using your Email address and password and select EC2 service.
2. Use the *EC2 Request Instance Wizard* to specify the instance type, whether it should be monitored, and specify a key/value pair for the instance to help organize and search, see Figure A.6.
3. Provide a name for the key pair, then on the left hand side panel choose *Security Groups* under *Network & Security*, select the desired security group and click on the *Inbound* tab to enter the desired rule, see Figure A.5.

To allocate an elastic IP address use the *Elastic IPs* tab of the *Network & Security* left hand side panel.

On Linux or Unix systems the port numbers below 1024 can only be assigned by the *root*. The plain ASCII file called *services* maps friendly textual names for Internet services to their assigned port numbers and protocol types as in the following example:

```
netstat 15/tcp
ftp     21/udp
ssh     22/tcp
telnet  23/tcp
http    80/tcp
```

A.4 HOW TO LAUNCH AN EC2 LINUX INSTANCE AND CONNECT TO IT

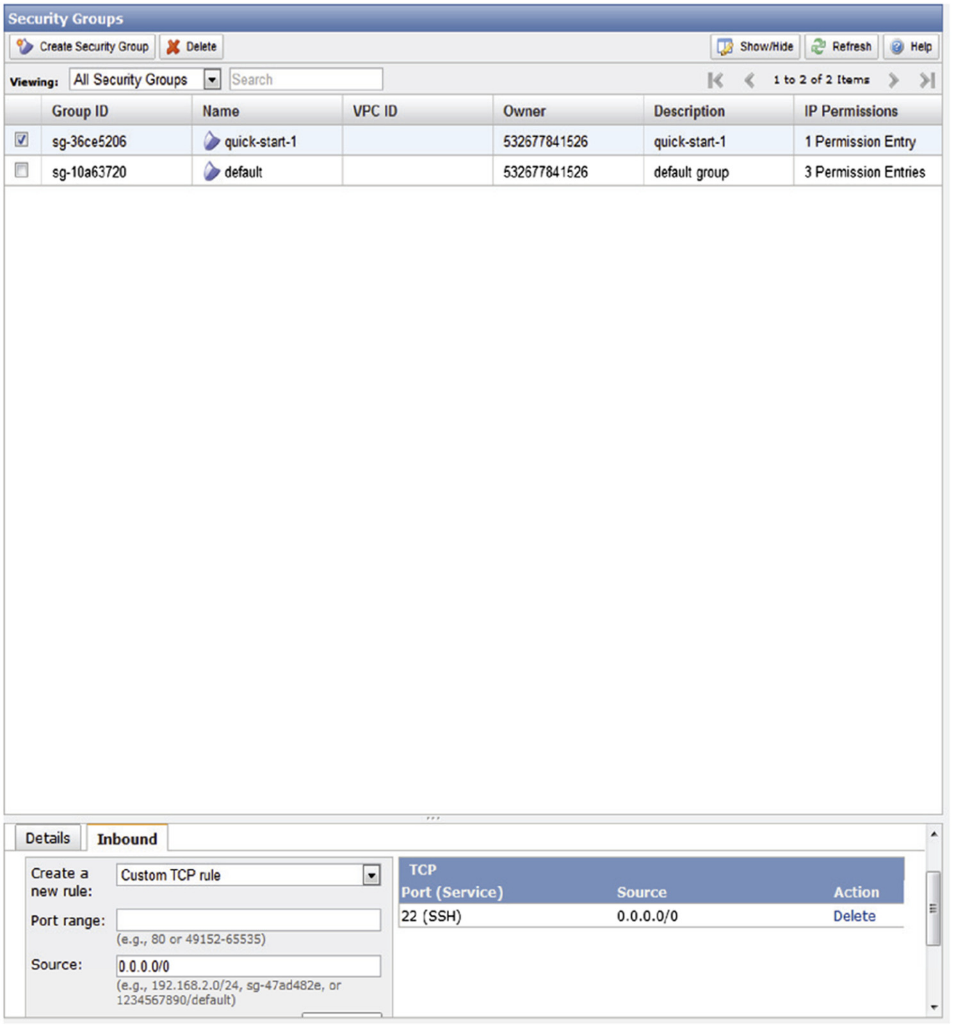
This section gives a step-by-step process to launch an EC2 Linux instance from a Linux platform.

A. Launch an instance

1. From the *AWS management console*, select EC2 and, once signed in, go to *Launch Instance Tab*.
2. To determine the processor architecture when you want to match the instance with the hardware enter the command

   ```
   uname -m
   ```

   and choose an appropriate Amazon Linux AMI by pressing *Select*.
3. Choose *Instance Details* to control the number, size, and other settings for instances.
4. To learn how the system works, press *Continue* to select the default settings.



Viewing: All Security Groups | Search | 1 to 2 of 2 Items

	Group ID	Name	VPC ID	Owner	Description	IP Permissions
✓	sg-36ce5206	quick-start-1		532677841526	quick-start-1	1 Permission Entry
☐	sg-10a63720	default		532677841526	default group	3 Permission Entries

Create a new rule: Custom TCP rule

Port range:
(e.g., 80 or 49152-65535)

Source: 0.0.0.0/0
(e.g., 192.168.2.0/24, sg-47ad482e, or 1234567890/default)

TCP

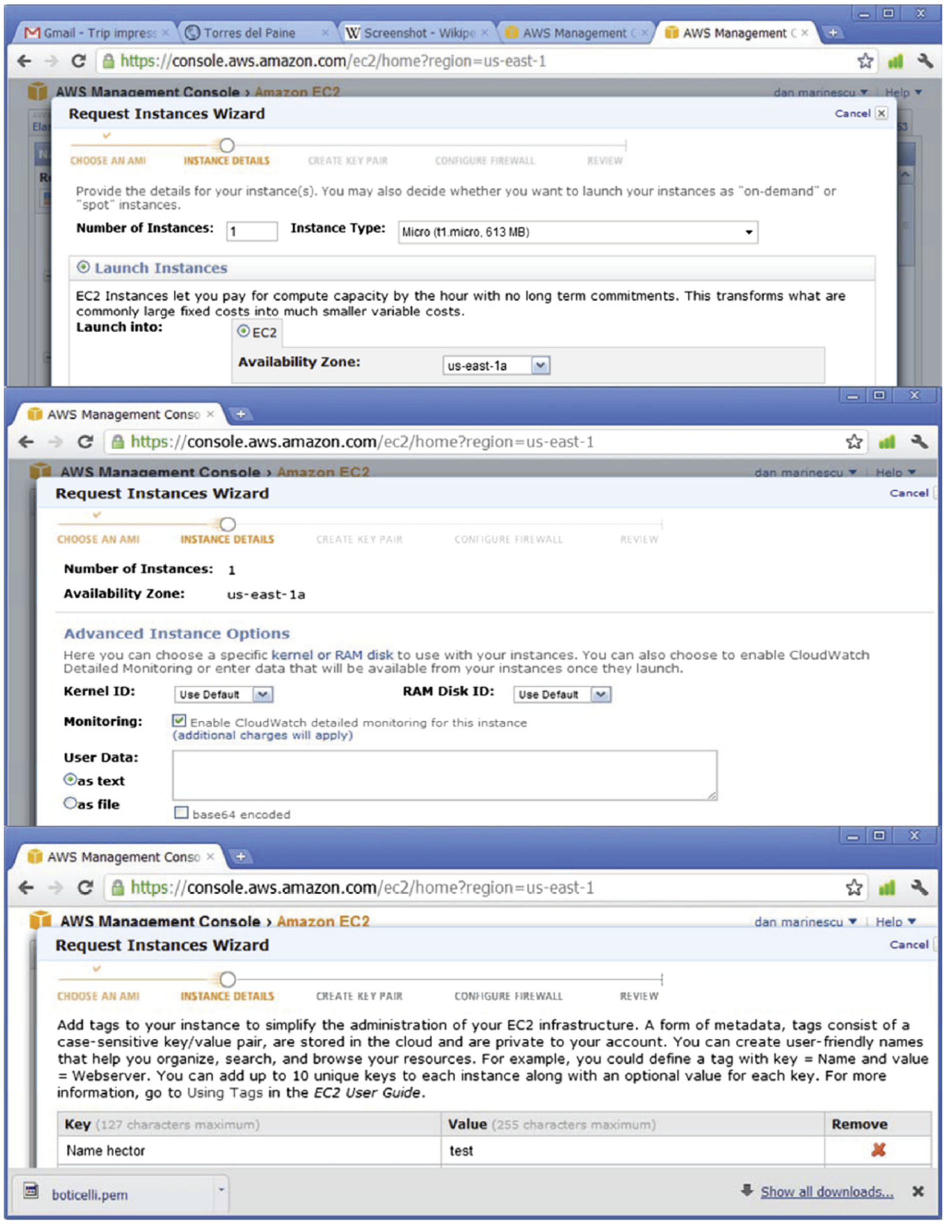
Port (Service)	Source	Action
22 (SSH)	0.0.0.0/0	Delete

FIGURE A.5

AWS security. Choose *Security Groups* under *Network & Security*, select the desired security group and click on the *Inbound* tab to enter the desired rule.

5. Define the instances security, as discussed in Section A.3: in the *Create Key Pair* page enter a name for the pair and then press *Create and Download Key Pair*.
6. The key pair file downloaded in the previous step is a *.pem* file and it must be hidden to prevent unauthorized access; if the file is in the directory *awcdir/dada.pem* enter the commands

```
cd awcdir
chmod 400 dada.pem
```



Gmail - Trip impress | Torres del Paine | Screenshot - Wikipe | AWS Management C | AWS Management C

https://console.aws.amazon.com/ec2/home?region=us-east-1

AWS Management Console › Amazon EC2 — dan marinescu ▾ | Help ▾

Request Instances Wizard — Cancel

CHOOSE AN AMI — INSTANCE DETAILS — CREATE KEY PAIR — CONFIGURE FIREWALL — REVIEW

Provide the details for your instance(s). You may also decide whether you want to launch your instances as "on-demand" or "spot" instances.

Number of Instances: 1 **Instance Type:** Micro (t1.micro, 613 MB)

Launch Instances

EC2 Instances let you pay for compute capacity by the hour with no long term commitments. This transforms what are commonly large fixed costs into much smaller variable costs.

Launch into: EC2

Availability Zone: us-east-1a

AWS Management Conso

https://console.aws.amazon.com/ec2/home?region=us-east-1

AWS Management Console › Amazon EC2 — dan marinescu ▾ | Help ▾

Request Instances Wizard — Cancel

CHOOSE AN AMI — INSTANCE DETAILS — CREATE KEY PAIR — CONFIGURE FIREWALL — REVIEW

Number of Instances: 1

Availability Zone: us-east-1a

Advanced Instance Options

Here you can choose a specific kernel or RAM disk to use with your instances. You can also choose to enable CloudWatch Detailed Monitoring or enter data that will be available from your instances once they launch.

Kernel ID: Use Default **RAM Disk ID:** Use Default

Monitoring: [✓] Enable CloudWatch detailed monitoring for this instance (additional charges will apply)

User Data: [✓ as text] as file

[ ] base64 encoded

AWS Management Conso

https://console.aws.amazon.com/ec2/home?region=us-east-1

AWS Management Console › Amazon EC2 — dan marinescu ▾ | Help ▾

Request Instances Wizard — Cancel

CHOOSE AN AMI — INSTANCE DETAILS — CREATE KEY PAIR — CONFIGURE FIREWALL — REVIEW

Add tags to your instance to simplify the administration of your EC2 infrastructure. A form of metadata, tags consist of a case-sensitive key/value pair, are stored in the cloud and are private to your account. You can create user-friendly names that help you organize, search, and browse your resources. For example, you could define a tag with key = Name and value = Webserver. You can add up to 10 unique keys to each instance along with an optional value for each key. For more information, go to Using Tags in the *EC2 User Guide*.

Key (127 characters maximum)	**Value** (255 characters maximum)	**Remove**
Name hector	test	✗

boticelli.pem — Show all downloads...

FIGURE A.6

EC2 Request Instances Wizard is used to: (A) specify the number and type of instances and the zone; (B) specify the kernelId, the RAM diskId, and enable the *CloudWatch* service to monitor the EC2 instance; (C) add tags to the instance; a tag is stored in the cloud and consists of a case-sensitive key/value pair private to the account.

7. Configure the firewall; go to the page *Configure Firewall*, select the option *Create a New Security Group* and provide a *Group Name*. Normally one uses *ssh* to communicate with the instance; the default port for communication is port 8080 and one can change the port and other rules by creating a new rule.
8. Press *Continue* and examine the review page which gives a summary of the instance.
9. Press *Launch* and examine the confirmation page and then press *Close* to end the examination of the confirmation page.
10. Press the *Instances* tab on the navigation pane to view the instance.
11. Look for your *Public DNS* name. As by default some details of the instance are hidden, click on the *Show/Hide* tab on the top of the console and select *Public DNS*.
12. Record the *Public DNS* as *PublicDNSname*; it is needed to connect to the instance from the Linux terminal.
13. Use the *ElasticIP* panel to assign an elastic IP address if a permanent IP address is required.

B. Connect to the instance using ssh and the tcp transport protocol.

1. Add a rule to the *iptables* to allow ssh traffic using the tcp protocol. Without this step either an*access denied* or a *permission denied* error message appears when trying to connect to the instance.

```
sudo iptables -A iptables  -p  -tcp  -dport  ssh  -j  ACCEPT
```

2. Enter the Linux command

```
ssh -i abc.pem ec2-user@PublicDNSname
```

If you get the prompt *You want to continue connecting?* respond *Yes*; a warning that the DNS name was added to the list of known hosts will appear.

3. An icon of the *Amazon Linux AMI* will be displayed.

C. Gain root access to the instance

By default the user does not have *root* access to the instance thus, cannot install any software. Once connected to the EC2 instance use the following command to gain *root* privileges

```
sudo -i
```

Then use *yum* install commands to install software, e.g., *gcc* to compile C programs on the cloud.

D. Run the service *ServiceName*

If the instance runs under *Linux* or *Unix* the service is terminated when the *ssh* connection is closed; to avoid the early termination use the command

```
nohup ServiceName
```

To run the service in the background and redirect *stdout* and *stderr* to files *p.out* and *p.err*, respectively, execute the command

```
nohup ServiceName > p.out 2 > p.err &
```

A.5 HOW TO USE S3 IN JAVA

The Java API for Amazon Web Services is provided by the AWS SDK. A software development kit (SDK) is a set of software tools for the creation of applications in a specific software environment. Java Development Kit (JDK) is an SDK for Java developers available from Oracle.

JDK includes a set of programming tools such as: *javac*, the Java compiler which converts Java source code into Java bytecode; *java*, the loader for Java applications, it can interpret the class files generated by the Java compiler; *javadoc* the documentation generator; *jar*, the archiver for class libraries; *jdb*, the debugger; *JConsole*, the monitoring and management console; *jstat*, JVM statistics monitoring; *jps*, JVM process status tool; *jinfo*, the utility to get configuration information from a running Java process; *jrunscript*, the command-line script shell for Java; *appletviewer* tool to debug Java applets without a web browser; and *idlj*, the IDL-to-Java compiler. The *Java Runtime Environment* is also a component of the JDK consisting of a Java Virtual Machine (JVM) and libraries.

Create an S3 client. S3 access is handled by the class *AmazonS3Client* instantiated with the account credentials of the AWS user

```
AmazonS3Client s3 = new AmazonS3Client(
new BasicAWSCredentials("your_access_key", "your_secret_key"));
```

The access and the secret keys can be found on the user's AWS account home page as mentioned in Section A.3.

Buckets. An *S3 bucket* is analogous to a file folder or directory and it is used to store *S3 Objects*. Bucket names must be *globally unique* hence, it is advisable to check first if the name exists

```
s3.doesBucketExist("bucket_name");
```

This function returns "true" if the name exists and "false" otherwise. Buckets can be created and deleted either directly from the AWS Management Console or programmatically as follows:

```
s3.createBucket("bucket_name");
s3.deleteBucket("bucket_name");
```

S3 objects. An *S3 object* stores the actual data and it is indexed by a key string. A single key points to only one S3 object in one bucket. Key names do not have to be globally unique, but if an existing key is assigned to a new object, then the original object indexed by the key is lost. To upload an object in a bucket one can use the *AWS Management Console,* or programmatically a file $local\_file\_name$ can be uploaded from the local machine to the bucket $bucket\_name$ under the key key using

```
File f = new File("local_file_name");
s3.putObject("bucket_name", "key", f);
```

A versioning feature for the objects in S3 was made available recently; it allows to preserve, retrieve, and restore every version of an S3 object. To avoid problems when uploading large files, e.g., the drop of the connection, use the *.initiateMultipartUpload()* with an API described at the *AmazonS3Client*. To access this object with key key from the bucket $bucket\_name$ use:

```
S3Object myFile = s3.getObject("bucket_name", "key");
```

To read this file, you must use the S3Object's *InputStream*:

```
InputStream in = myFile.getObjectContent();
```

The *InputStream* can be accessed using *Scanner, BufferedReader* or any other method supported. Amazon recommends closing the stream as early as possible, as the content is not buffered and it is streamed directly from the S3; an open *InputStream* means an open connection to S3. For example, the following code will read an entire object and print the contents to the screen:

```
AmazonS3Client s3 = new AmazonS3Client(
    new BasicAWSCredentials("access_key", "secret_key"));
    InputStream input = s3.getObject("bucket_name", "key")
       .getObjectContent();
    Scanner in = new Scanner(input);
    while (in.hasNextLine())
        {
         System.out.println(in.nextLine());
        }
in.close();
input.close();
```

Batch Upload/Download. Batch upload requires repeated calls of *s3.putObject()* while iterating over local files.

To view the keys of all objects in a specific bucket use

```
 ObjectListing listing = s3.listObjects("bucket_name");
```

Object Listing supports several useful methods including *getObjectSummaries(). S3ObjectSummary* encapsulates most of an S3 object properties (excluding the actual data), including the key to access the object directly,

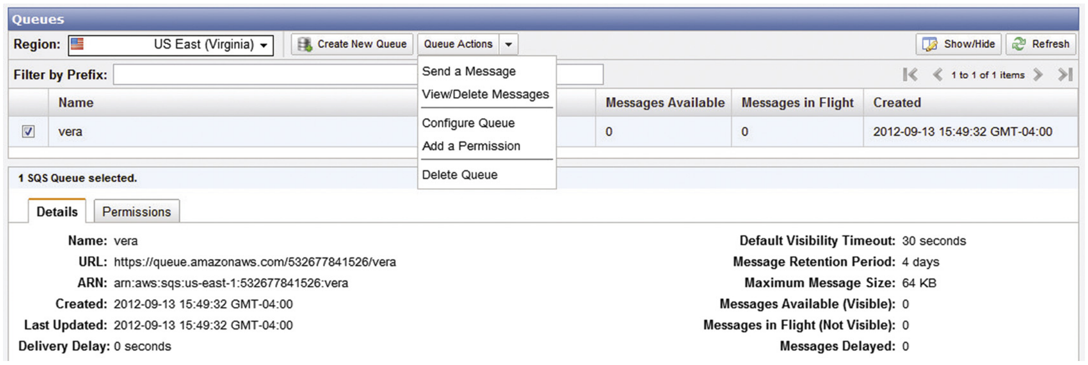
```
List<S3ObjectSummary> summaries = listing.getObjectSummaries();
```

For example, the following code will create a list of all keys used in a particular bucket and all of the keys will be available in string form in *List < String > allKeys*:

```
AmazonS3Client s3 = new AmazonS3Client(
    new BasicAWSCredentials("access_key", "secret_key"));
    List<String> allKeys = new ArrayList<String>();
    ObjectListing listing = s3.listObjects("bucket_name");
    for (S3ObjectSummary summary:listing.getObjectSummaries())
      {
        allKeys.add(summary.getKey());
      }
```

Note that if the bucket contains a very large number of objects then *s3.listObjects()* will return a truncated list. Use the following command to test if the list is truncated one could use *listing.isTruncated()*; to get the next batch of objects

```
 s3.listNextBatchOfObjects(listing)};
```



	Name	Messages Available	Messages in Flight	Created
☑	vera	0	0	2012-09-13 15:49:32 GMT-04:00

1 SQS Queue selected.

Name:	vera	Default Visibility Timeout:	30 seconds
URL:	https://queue.amazonaws.com/532677841526/vera	Message Retention Period:	4 days
ARN:	arn:aws:sqs:us-east-1:532677841526:vera	Maximum Message Size:	64 KB
Created:	2012-09-13 15:49:32 GMT-04:00	Messages Available (Visible):	0
Last Updated:	2012-09-13 15:49:32 GMT-04:00	Messages in Flight (Not Visible):	0
Delivery Delay:	0 seconds	Messages Delayed:	0

FIGURE A.7

Queue actions in SQS.

To account for a large number of objects in the bucket, the previous example becomes

```
AmazonS3Client s3 = new AmazonS3Client(
new BasicAWSCredentials("access_key", "secret_key"));
List<String> allKeys = new ArrayList<String>();
ObjectListing listing = s3.listObjects("bucket_name");
while (true)
  {
   for (S3ObjectSummary summary :
       listing.getObjectSummaries())
       {
        allKeys.add(summary.getKey());
       }
    if (!listing.isTruncated())
       {
        break;
       }
   listing = s3.listNextBatchOfObjects(listing);
  }
```

A.6 HOW TO MANAGE AWS SQS SERVICES IN C#

Recall from Section 2.3 that SQS is a system for supporting automated workflows. Multiple components can communicate with messages sent and received via SQS. An example showing the use of message queues is presented in Section 7.6. Figure A.7 shows the actions available for a given queue in *SQS*.

The following steps can be used to create a queue, send a message, receive a message, delete a message, delete the queue in C#:

1. Authenticate an SQS connection

```
NameValueCollection appConfig =
            ConfigurationManager.AppSettings;
        AmazonSQS sqs = AWSClientFactory.CreateAmazonSQSClient
            (appConfig["AWSAccessKey"], appConfig["AWSSecretKey"]);
```

2. Create a queue

```
CreateQueueRequest sqsRequest = new CreateQueueRequest();
sqsRequest.QueueName = "MyQueue";
CreateQueueResponse createQueueResponse =
    sqs.CreateQueue(sqsRequest);
String myQueueUrl;
myQueueUrl = createQueueResponse.CreateQueueResult.QueueUrl;
```

3. Send a message

```
SendMessageRequest sendMessageRequest =
    new SendMessageRequest();
sendMessageRequest.QueueUrl =
    myQueueUrl; //URL from initial queue
sendMessageRequest.MessageBody = "This is my message text.";
sqs.SendMessage(sendMessageRequest);
```

4. Receive a message

```
ReceiveMessageRequest receiveMessageRequest =
    new ReceiveMessageRequest();
receiveMessageRequest.QueueUrl = myQueueUrl;
ReceiveMessageResponse receiveMessageResponse =
    sqs.ReceiveMessage(receiveMessageRequest);
```

5. Delete a message

```
DeleteMessageRequest deleteRequest =
    new DeleteMessageRequest();
deleteRequest.QueueUrl = myQueueUrl;
deleteRequest.ReceiptHandle = messageRecieptHandle;
DeleteMessageResponse DelMsgResponse =
    sqs.DeleteMessage(deleteRequest);
```

6. Delete a queue

```
DeleteQueueRequest sqsDelRequest = new DeleteQueueRequest();
sqsDelRequest.QueueUrl =
    createQueueResponse.CreateQueueResult.QueueUrl;
DeleteQueueResponse delQueueResponse =
sqs.DeleteQueue(sqsDelRequest);
```

A.7 HOW TO INSTALL SNS ON UBUNTU 10.04

SNS, the Simple Notification Service, is a web service for: monitoring applications, workflow systems, time-sensitive information updates, mobile applications, and other event-driven applications which require a simple and efficient mechanism for message delivery. SNS "pushes" messages to clients, rather than requiring a user to periodically poll a mailbox or another site for messages.

SNS is based on the publish-subscribe paradigm; it allows a user to define the topics, the transport protocol used (HTTP/HTTPS, Email, SMS, SQS), and the end-point (URL, Email address, phone number, SQS queue) for notifications to be delivered.

Ubuntu is an open source operating system for personal computers based on Debian Linux distribution. The desktop version of Ubuntu[1] supports Intel x86 32-bit and 64-bit architectures.

SNS supports the following actions:

- Add/Remove Permission
- Confirm Subscription
- Create/Delete Topic
- Get/Set Topic Attributes
- List Subscriptions/Topics/Subscriptions By Topic
- Publish/Subscribe/Unsubscribe

The site http://awsdocs.s3.amazonaws.com/SNS/latest/sns-qrc.pdf provides detailed information about each one of these actions.

The following steps must be taken to install an SNS client:

1. Install Java in the *root* directory and then execute the commands

```
deb http://archive.canonical.com/lucid partner
update
install sun-java6-jdk
```

   Then change the default Java settings

```
update-alternatives -config java
```

2. Download the *SNS* client, unzip the file and change permissions

```
wget http://sns-public-resources.s3.amazonaws.com/
       SimpleNotificationServiceCli-2010-03-31.zip
chmod 775 /root/ SimpleNotificationServiceCli-1.0.2.3/bin
```

3. Start the AWS management console and go to *Security Credentials*. Check the *Access Key ID* and the *Secret Access Key* and create a text file */root/credential.txt* with the following content:

```
AWSAccessKeyId= your_Access_Key_ID
AWSSecretKey= your_Secret_Access_Key
```

4. Edit the *.bashrc* file and add

```
export AWS_SNS_HOME=~/SimpleNotificationServiceCli-1.0.2.3/
export AWS_CREDENTIAL_FILE=$HOME/credential.txt
export PATH=$AWS_SNS_HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-6-sun/
```

[1]Ubuntu is an African humanist philosophy; "ubuntu" is a word in the Bantu language of South Africa meaning "humanity towards others."

5. Reboot the system
6. Enter on the command line

   ```
   sns.cmd
   ```

   If the installation was successful the list of *SNS* commands will be displayed.

A.8 HOW TO CREATE AN EC2 PLACEMENT GROUP AND USE MPI

An *EC2 Placement Group*, is a logical grouping of instances which allows the creation of a virtual cluster. When several instances are launched as an *EC2 Placement Group* the virtual cluster has a high bandwidth interconnect system suitable for network-bound applications. The cluster computing instances require an HVM (Hardware Virtual Machine) ECB-based machine image, while other instances use a PVM (Paravirtual Machine) image. Such clusters are particularly useful for high performance computing when most applications are communication intensive.

Once a placement group is created, MPI can be used for communication among the instances in the placement group. MPI is a de-facto standard for parallel applications using message passing, designed to ensure high performance, scalability, and portability; it is a language-independent "message-passing application programmer interface, together with a protocol and the semantic specifications for how its features must behave in any implementation" [206]. MPI supports point-to-point, as well as collective communication; it is widely used by parallel programs based on the SPMD (Same Program Multiple Data) paradigm.

The following *C* code [206] illustrates the startup of MPI communication for a process group, *MPI_COM_PROCESS_GROUP* consisting of *nprocesses*; each process is identified by its *rank*. The runtime environment *mpirun* or *mpiexec* spawns multiple copies of the program, with the total number of copies determining the number of process ranks in *MPI_COM_PROCESS_GROUP*.

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#define TAG 0
#define BUFSIZE 128

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int nprocesses;
  int my_processId;
  int i;
  MPI_Status stat;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COM_PROCESS_GROUP,&nprocesses);
  MPI_Comm_rank(MPI_COM_PROCESS_GROUP,&my_processId);
```

MPI_SEND and *MPI_RECEIVE* are blocking send and blocking receive, respectively; their syntax is:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag,MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
           int source, int tag, MPI_Comm comm, MPI_Status *status)
```

with

buf	–	initial address of send buffer (choice)
count	–	number of elements in send buffer (nonnegative integer)
datatype	–	data type of each send buffer element (handle)
dest	–	rank of destination (integer)
tag	–	message tag (integer)
comm	–	communicator (handle).

Once started, every process other than the coordinator, the process with $rank = 0$, sends a message to the entire group and then receives a message from each of the other members of the process group.

```
  if(my_processId == 0)
  {
    printf("%d: We have %d processes\n", my_processId, nprocesses);
    for(i=1;i<nprocesses;i++)
    {
      sprintf(buff, "Hello %d! ", i);
      MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_PROCESS_GROUP);
    }
    for(i=1;i<nprocesses;i++)
    {
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_PROCESS_GROUP, &stat);
      printf("%d: %s\n", my_processId, buff);
    }
  }
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_PROCESS_GROUP, &stat);
    sprintf(idstr, "Processor %d ", my_processId);
    strncat(buff, idstr, BUFSIZE-1);
    strncat(buff, "reporting for duty\n", BUFSIZE-1);
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COM_PROCESS_GROUP);
  }

  MPI_Finalize();
  return 0;
}
```

An example of cloud computing using the MPI is described in [167]. An example of MPI use on EC2 is at http://rc.fas.harvard.edu/faq/amazonec2.

A.9 STARCLUSTER – A CLUSTER COMPUTING TOOLKIT FOR EC2

StarCluster, http://star.mit.edu/cluster/, is an open source cluster-computing toolkit for EC2. The system assigns user-friendly names to the nodes of the virtual cluster and the cluster is so configured to allow *ssh* from any node of the cluster to any other nodes. It allows to attach EBS volumes to the cluster for persistent storage and provides and API for executing OS commands such as copying files. StarCluster supports dynamically cluster reconfiguration, as well as lunching spot instances to reduce the service costs.

StarCluster AMIs consist of several scientific libraries:

1. OpenMPI – for writing parallel applications.
2. Automatically Tuned Linear Algebra Software (ATLAS) – optimized for Amazon EC2 larger instances, see http://math-atlas.sourceforge.net/.
3. NumPy/SciPy compiled against the optimized ATLAS install. SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. NumPy is a set of tools for integrating C/C++ and Fortran code useful linear algebra, Fourier transform, and random number capabilities, see https://www.scipy.org/scipylib/.
4. IPython – interactive parallel computing in Python, see https://ipython.org/.

Several other important features of the StarCluster are: (1) Support for starting/stopping EBS-backed clusters on EC2. (2) Elastic Load Balancing – using Sun Grid Engine queue statistics. (3) Support for specifying instance types on a per-node basis. (4) A number of commands for EC2 and S3 operations including the ones in Table A.2.

A.10 AN ALTERNATIVE SETTING OF AN MPI VIRTUAL CLUSTER

An alternative setting up for an MPI cluster is described in [312]. The instances used are *cc2.8xlarge* with 2x Intel Xeon E5-2670 processors and $\approx$ 60 GB of RAM per node and spot instances rather than reserved ones were used thus, saving about 90% of the cost. The VM virtualization layer used by the *cc2.8xlarge* instances is thinner than the one of other instances.

Setting user's instances involves several steps:

1. Select a VM image and an instance type.
2. Define a placement group.
3. Configure the storage.
4. Define the VM tags to manage the instances.
5. Create of a key pair.
6. Setup the security group.
7. Launch the instances.

The nest phase is the configuration of the virtual cluster (VC). All instances of the virtual cluster are booted with the same operating system and share a 10 Gbps Ethernet subnet. The public IP address of the booted instances is available from the EC2 Dashboard by selecting their entry under the "Instances" page. The preliminary steps for the VC configurations are:

- Create nodes aliases – add their internal IP addresses to */etc/hosts* as *node1, node2, node3, node4*.
- Create aliases for files to be transferred from the master (*node1*) to workers *node2, node3, node4*.
- Enable password-less ssh between instances.

Table A.2 StartCluster commands for EC2 and S3 operations.

Command	Function
listinstances	List all running EC2 instances
listspots	List all EC2 spot instance requests
listimages	List all registered EC2 images (AMIs)
listpublic	List all public StarCluster images on EC2
listkeypairs	List all EC2 keypairs
createkey	Create a new Amazon EC2 keypair
removekey	Remove a keypair from Amazon EC2
s3image	Create a new instance-store (S3) AMI from a running EC2 instance
ebsimage	Create a new EBS image (AMI) from a running EC2 instance
removeimage	Deregister an EC2 image (AMI)
createvolume	Create a new EBS volume for use with StarCluster
listvolumes	List all EBS volumes
resizevolume	Resize an existing EBS volume
removevolume	Delete one or more EBS volumes
spothistory	Show spot instance pricing history stats
showconsole	Show console output for an EC2 instance
listregions	List all EC2 regions
listzones	List all EC2 availability zones in the current region
listbuckets	List all S3 buckets
showbucket	Show all files in an S3 bucket

There are two options to install and lunch MPI in every node, the first for *OpenMPI* and the second for *mpich*, both use the *yum* package.

```
sudo yum install openmpi-devel
sudo yum install mpich-devel
```

Once the GCC compiler, the MPI runtime, and the mpicc wrapper are available in every node the following commands must be added to the *.bashrc* file on all compute nodes

```
export PATH=/usr/lib64/openmpi/bin:$PATH
export LD_LIBRARY_PATH=/usr/lib64/openmpi/lib
```

The following command runs a program "application.xx" available on every node

```
mpirun -np 32 -hostfile ~/nodefile ~/application.xx
```

with the hostfile called "nodefile" compatible with OpenMPI created as follows:

```
$ cat ~/nodefile
node1 slots=16
node2 slots=16
node3 slots=16
node4 slots=16
```

A 64-way MPI job is created. Each node has 16 cores and 16 hyperthreads.

A.11 HOW TO INSTALL HADOOP ON ECLIPSE ON A WINDOWS SYSTEM

Eclipse (http://www.eclipse.org/) is a software development environment. Eclipse consists of an integrated development environment (IDE) and an extensible plug-in system. It is written mostly in Java and can be used to develop applications in Java and, by means of various plug-ins, in C, C++, Perl, PHP, Python, R, Ruby, and several other languages. The IDE is often called Eclipse CDT for C/C++, Eclipse JDT for Java, and Eclipse PDT for PHP.

The software packages used are:

- Apache Hadoop is a software framework that supports data-intensive distributed applications under a free license. *Hadoop* was inspired by Google's MapReduce; see Section 7.5 for a discussion of MapReduce and Section 7.6 for an application using *Hadoop*.
- *Cygwin* is a Unix-like environment for Microsoft Windows. It is open source software, released under the GNU General Public License version 2. *Cygwin* consists of: (1) a dynamic-link library (DLL) as an API compatibility layer providing a substantial part of the POSIX API functionality; and (2) an extensive collection of software tools and applications that provide a Unix-like look and feel.

A. Pre-requisites

- *Java 1.6*; set JAVA_Home = path where *JDK* is installed.
- *Eclipse Europa 3.3.2*
  Note: the *hadoop* plugin was specially designed for Europa and newer releases of *Eclipse* might have some issues with *hadoop* plugin.

B. SSH Installation

1. Install *cygwin* using the installer downloaded from http://www.cygwin.com. From the *Select Packages* window select the *openssh* and *openssl* under Net.
   Note: Create a desktop icon when asked during installation.
2. Display the "Environment Variables" panel

```
Computer -> System Properties -> Advanced System Settings
                                      -> Environment Variables
```

   Click on the variable named *Path* and press *Edit*; append the following value to the path variable

```
    ;c:\cygwin\bin;c:\cygwin\usr\bin
```

3. Configure the *ssh daemon* using *cygwin*. Left click on the *cygwin* icon on desktop and click "Run as Administrator". Type in the command window of *cygwin*

```
   ssh-host-config.
```

4. Answer "Yes" when prompted *sshd should be installed as a service*; answer "No" to all other questions.

5. Start the *cygwin* service by navigating to

   ```
   Control Panel -> Administrative Tools -> Services
   ```

   Look for *cygwin sshd* and start the service.
6. Open *cygwin* command prompt and execute the following command to generate keys

   ```
   ssh-keygen
   ```

7. When prompted for filenames and pass phrases press ENTER to accept default values. After the command has finished generating keys, enter the following command to change into your *.ssh* directory:

   ```
   cd ~/.ssh
   ```

8. Check if the keys were indeed generated

   ```
   ls -l
   ```

9. The two files *id_rsa.pub* and *id_rsa* with recent creation dates contain authorization keys.
10. To register the new authorization keys, enter the following command (Note: the sharply-angled double brackets are very important)

    ```
    cat id_rsa.pub >> authorized_keys
    ```

11. Check if the keys were set up correctly

    ```
    ssh localhost
    ```

12. Since it is a new *ssh* installation, you will be warned that authenticity of the host could not be established and will be asked whether you really want to connect. Answer YES and press ENTER. You should see the *cygwin* prompt again, which means that you have successfully connected.
13. Now execute again the command:

    ```
    ssh localhost
    ```

    this time no prompt should appear.

C. Download *hadoop*

1. Download *hadoop 0.20.1* and place in a directory such as

   ```
   C:\Java
   ```

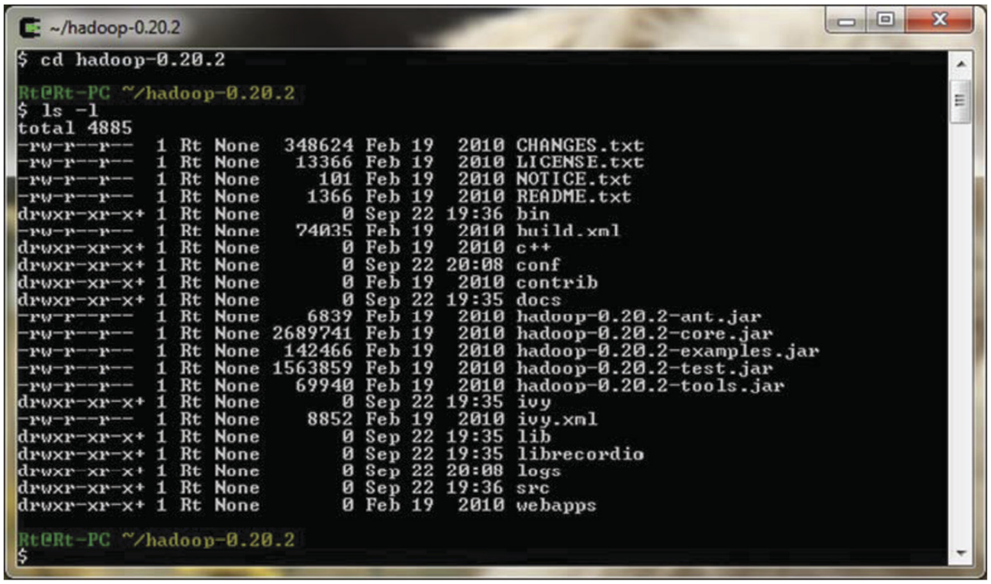
2. Open the *cygwin* command prompt and execute

   ```
   cd
   ```

3. Enable the home directory folder to be shown in the Windows Explorer window

   ```
   explorer
   ```

4. Open another Windows Explorer window and navigate to the folder that contains the downloaded *hadoop* archive.
5. Copy the *hadoop* archive into the home directory folder.



```
$ cd hadoop-0.20.2

Rt@Rt-PC ~/hadoop-0.20.2
$ ls -l
total 4885
-rw-r--r--  1 Rt None  348624 Feb 19  2010 CHANGES.txt
-rw-r--r--  1 Rt None   13366 Feb 19  2010 LICENSE.txt
-rw-r--r--  1 Rt None     101 Feb 19  2010 NOTICE.txt
-rw-r--r--  1 Rt None    1366 Feb 19  2010 README.txt
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:36 bin
-rw-r--r--  1 Rt None   74035 Feb 19  2010 build.xml
drwxr-xr-x+ 1 Rt None       0 Feb 19  2010 c++
drwxr-xr-x+ 1 Rt None       0 Sep 22 20:08 conf
drwxr-xr-x+ 1 Rt None       0 Feb 19  2010 contrib
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:35 docs
-rw-r--r--  1 Rt None    6839 Feb 19  2010 hadoop-0.20.2-ant.jar
-rw-r--r--  1 Rt None 2689741 Feb 19  2010 hadoop-0.20.2-core.jar
-rw-r--r--  1 Rt None  142466 Feb 19  2010 hadoop-0.20.2-examples.jar
-rw-r--r--  1 Rt None 1563859 Feb 19  2010 hadoop-0.20.2-test.jar
-rw-r--r--  1 Rt None   69940 Feb 19  2010 hadoop-0.20.2-tools.jar
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:35 ivy
-rw-r--r--  1 Rt None    8852 Feb 19  2010 ivy.xml
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:35 lib
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:35 librecordio
drwxr-xr-x+ 1 Rt None       0 Sep 22 20:08 logs
drwxr-xr-x+ 1 Rt None       0 Sep 22 19:36 src
drwxr-xr-x+ 1 Rt None       0 Feb 19  2010 webapps

Rt@Rt-PC ~/hadoop-0.20.2
$
```

FIGURE A.8

The result of unpacking *hadoop*.

D. Unpack *hadoop*

1. Open a new *cygwin* window and execute

```
tar -xzf hadoop-0.20.1.tar.gz
```

2. List the contents of the home directory

```
ls -l
```

A newly created directory called *hadoop-0.20.1* should be seen. Execute

```
cd hadoop-0.20.1
ls -l
```

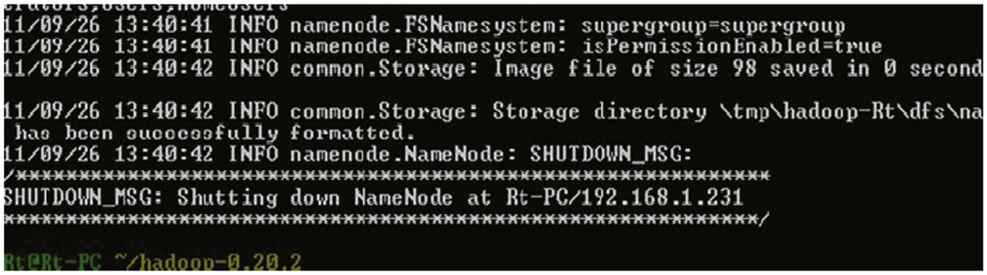
The files listed in Figure A.8 should be seen.

E. Set properties in configuration file

1. Open a new *cygwin* window and execute the following commands

```
cd hadoop-0.20.1
cd conf
explorer
```

2. The last command will cause the Explorer window for the *conf* directory to pop up. Minimize it for now or move it to the side.



```
11/09/26 13:40:41 INFO namenode.FSNamesystem: supergroup=supergroup
11/09/26 13:40:41 INFO namenode.FSNamesystem: isPermissionEnabled=true
11/09/26 13:40:42 INFO common.Storage: Image file of size 98 saved in 0 second

11/09/26 13:40:42 INFO common.Storage: Storage directory \tmp\hadoop-Rt\dfs\na
 has been successfully formatted.
11/09/26 13:40:42 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at Rt-PC/192.168.1.231
************************************************************/

Rt@Rt-PC ~/hadoop-0.20.2
```

FIGURE A.9

The creation of HDFS.

3. Launch *Eclipse* or a text editor such as *Notepad* ++ and navigate to the *conf* directory and open the file *hadoop*-site to insert the following lines between $< configuration >$ and $< /configuration >$ tags.

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9100</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>localhost:9101</value>
</property>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
```

F. Format the Namenode

Format the *namenode* to create a Hadoop Distributed File System (HDFS). Open a new *cygwin* window and execute the following commands:

```
cd hadoop-0.20.1
mkdir logs
bin/hadoop namenode -format
```

When the formatting of the *namenode* is finished, the message in Figure A.9 appears.

A.12 EXERCISES AND PROBLEMS

Problem 1. Establish an account to AWS. Use the AWS management console to launch an EC2 instance and connect to it.

Problem 2. Launch three EC2 instances; the computations carried out by the three instances should consist of two phases and the second phase should be started only after all instances have finished the first stage. Design a protocol and use *Simple Queue Service (SQS)* to implement the barrier synchronization after the first phase.

Problem 3. Use the *Zookeeper* to implement the coordination model in Problem 2.

Problem 4. Use the *Simple Workflow Service (SWF)* to implement the coordination model in Problem 2. Compare the three methods.

Problem 5. Upload several (10–20) large image files to an S3 bucket. Start an instance which retrieves the images from the S3 bucket and compute the retrieval time. Use the *ElastiCache* service and compare the retrieval time for the two cases.

Problem 6. Numerical simulations are ideal applications for cloud computing. Output data analysis of a simulation experiment requires the computation of confidence intervals for the mean for the quantity of interest [296]. This implies that one must run multiple batches of simulation, compute the average value of the quantity of interest for each batch, and then calculate say 95% confidence intervals for the mean. Use the *CloudFormation* service to carry out a simulation using multiple cloud instances which store partial results in S3 and then another instance computes the confidence interval for the mean.

Problem 7. Run an application which takes advantage of the *Autoscaling* service.

Problem 8. Use the *Elastic Beanstalk* service to run an application and compare it with the case when the *Autoscaling* service was used.

Problem 9. Design a cloud service and a testing environment; use the *Elastic Beanstalk* service to support automatic scaling up and down and use the *Elastic Load Balancer* to distribute the incoming service request to different instances of the application.

CLOUD PROJECTS B

Several projects for students enrolled in a cloud computing class and a research project involving an extensive simulation are discussed in this section. These projects reflect the great appeal of cloud computing for several types of applications. Large-scale simulation of complex systems such as the one discussed in Sections B.1 and B.2 can only be done using the large pool of resources provided by clouds. Cloud services such as the one discussed in Sections B.3 and B.4 are another important class of applications.

Design projects where multiple alternatives must be evaluated and compared, such as the one discussed in Section B.5, benefit from a cloud environment. Several, possibly many design alternative can be simulated concurrently; then the selection of the best alternatives based on different performance objective can be done by multiple instances running concurrently. Big Data applications such as the one discussed in Section B.6 require resources that can only be provided by super computers or computer clouds. Clouds are by far the better alternative to supercomputers due to access and lower cost.

B.1 CLOUD-BASED SIMULATION OF A DISTRIBUTED TRUST ALGORITHM

Mobile wireless applications are likely to benefit from cloud computing, as discussed in Chapter 12. This expectation is motivated by several reasons:

- The convenience of data access from any site connected to the Internet.
- The data transfer rates of wireless networks are increasing; the time to transfer data to and from a cloud is no longer a limiting factor.
- The mobile devices have limited resources; while new generations of smart phones and tablet computers are likely to use multi-core processors and have a fair amount of memory, power consumption is and will continue to be a major concern in the near future. Thus, it seems reasonable to delegate compute-intensive and data-intensive tasks to an external entity, e.g., a cloud.

The first application we discuss is a cloud-based simulation for trust evaluation in a Cognitive Radio Networks (CRN) [67]. The available communication spectrum is a precious commodity and the objective of a CRN is to use the communication bandwidth effectively, while attempting to avoid interference with licensed users. Two main functions necessary for the operation of a CRN are spectrum sensing and spectrum management; the former detects unused spectrum and the later decides the optimal use of the available spectrum. Spectrum sensing in CRNs is based on information provided by the nodes of the network. The nodes compete for the free channels and some may supply deliberately distorted information to gain advantage over the other nodes; thus, trust determination is critical for the management of CRNs.

Cognitive radio networks. Research in the last decade reveals a significant temporal and spatial underutilization of the allocated spectrum. Thus, the motivation to opportunistically harness the vacancies of spectrum at a given time and place.

The original goal of cognitive radio, first proposed at Bell Labs [346,347], was to develop a software-based radio platform which allows a reconfigurable wireless transceiver to automatically adapt its communication parameters to network availability and to user demands. Today the focus of cognitive radio is on spectrum sensing [78].

We recognize two types of devices connected to a CRN, the primary and the secondary ones; *primary* devices have exclusive rights to specific regions of the spectrum, while *secondary* devices enjoy dynamic spectrum access and are able to use a channel, provided that the primary, licensed to use that channel, is not communicating. Once a primary starts its transmission, the secondary using the channel is required to relinquish it and identify another free channel to continue its operation; this mode of operation is called an *overlay mode.*

Cognitive radio networks are often based on *cooperative spectrum sensing* strategy. In this mode of operation each device determines the occupancy of the spectrum based on its own measurements combined with information from its neighbors and then shares its own spectrum occupancy assessment with its neighbors [181,472,473].

Information sharing is necessary because a device alone cannot determine the true spectrum occupancy. Indeed, a secondary device has a limited transmission and reception range; device mobility combined with typical wireless channel impairments such as multi path fading, shadowing, and noise add to the difficulties of gathering accurate information by a single device.

Individual devices of a centralized, or infrastructure-based CRN, send the results of their measurements regarding spectrum occupancy to a central entity, be it a base station, an access point, or a cluster head. This entity uses a set of *fusion rules* to generate the spectrum occupancy report and then distributes it to the devices in its jurisdiction. The area covered by such networks is usually small as global spectrum decision are affected by the local geography.

There is another mode of operation based on the idea that a secondary device operates at a much lower power level than a primary one. In this case the secondary can share the channel with the primary as long as its transmission power is below a threshold, μ, that has to be determined periodically. In this scenario the receivers wishing to listen to the primary are able to filter out the "noise" caused by the transmission initiated by secondaries if the signal-to-noise ratio, (S/N), is large enough.

We are only concerned with the overlay mode whereby a secondary device maintains an *occupancy report*, which gives a snapshot of the current status of the channels in the region of the spectrum it is able to access. The occupancy report is a list of all the channels and their state, e.g., 0 if the channel is free for use and 1 if the primary is active. Secondary devices continually sense the channels they can access to gather accurate information about available channels.

The secondary devices of an ad hoc CRN compete for free channels and the information one device may provide to its neighbors could be deliberately distorted; malicious devices will send false information to the fusion center in a centralized CRN. Malicious devices could attempt to deny the service, or to cause other secondary devices to violate spectrum allocation rules. To *deny the service* a device will report that free channels are used by the primary. To entice the neighbors to commit FCC violations, the occupancy report will show that channels used by the primary are free. This attack strategy is called *secondary spectrum data falsification (SSDF)* or Byzantine attack.[1] Thus, trust determination is a critical issue for CR networks.

Trust. The actual meaning of *trust* is domain and context specific. Consider for example networking; at the MAC-layer (Medium Access Control) the multiple-access protocols assume that all senders follow the channel access policy, e.g., in CSMA-CD a sender senses the channel and then attempts to transmit if no one else does. In a store-and-forward network, trust assumes that all routers follow a best-effort policy to forward packets towards their destination. We shall use the term node instead of device throughout the remaining of this section.

In the context of cognitive radio trust is based on the quality of information regarding the channel activity provided by a node. The status of individual channels can be assessed by each node based on the results of its own measurements combined with the information provided by its neighbors, as is the case of several algorithms discussed in the literature [105,472].

The alternative discussed in Section B.3 is to have a cloud-based service which collects information from individual nodes, evaluates the state of each channel based on the information received, and supplies this information on demand. Evaluation of the trust and identification of untrustworthy nodes are critical for both strategies [393].

A distributed algorithm for trust management in cognitive radio. The algorithm computes the trust of node $1 \leq i \leq n$ in each node in its vicinity, $j \in V_i$, and requires several preliminary steps. The basic steps executed by a node i at time t are:

1. Determine node i's version of the occupancy report for each one of the K channels:

$$S_i(t) = \{s_{i,1}(t), s_{i,2}(t), \ldots, s_{i,K}(t)\} \tag{B.1}$$

   In this step node i measures the power received on each of the K channels.
2. Determine the set $V_i(t)$ of the nodes in the vicinity of node i. Node i broadcasts a message and individual nodes in its vicinity respond with their *NodeId*.
3. Determine the distance to each device $j \in V_i(t)$ using the algorithm described in this section.
4. Infer the power as measured by each device $j \in V_i(t)$ on each channel $k \in K$.
5. Use the location and power information determined in the previous two steps to infer the status of each channel

$$s_{i,k,j}^{infer}(t) \quad \text{with} \;\; 1 \leq k \leq K, \;\; j \in V_i(t) \tag{B.2}$$

   a secondary node j should have determined: 0 if the channel is free for use, 1 if the primary device is active, and X if it cannot be determined.

$$s_{i,k,j}^{infer}(t) = \begin{cases} 0 & \text{if secondary node j decides that channel k is free} \\ 1 & \text{if secondary node j decides that channel k is used by the primary} \\ X & \text{if no inference can be made} \end{cases} \tag{B.3}$$

6. Receive the information provided by neighbor $j \in V_i(t)$, $S_{i,k,j}^{recv}(t)$.

[1] See section for 3.12 for a brief discussion of Byzantine attacks.

7. Compare the information provided by neighbor $j \in V_i(t)$

$$S^{recv}_{i,k,j}(t) = \{s^{recv}_{i,1,j}(t), s^{recv}_{i,2,j}(t), \ldots, s^{recv}_{i,K,j}(t)\} \tag{B.4}$$

with the information inferred by node i about node j

$$S^{infer}_{i,k,j}(t) = \{s^{infer}_{i,1,j}(t), s^{infer}_{i,2,j}(t), \ldots, s^{infer}_{i,K,j}(t)\} \tag{B.5}$$

8. Compute the number of matches, mismatches, and cases when no inference is possible, respectively,

$$\alpha_{i,j}(t) = \mathcal{M}\left[S^{infer}_{i,k,j}(t), S^{recv}_{i,k,j}(t)\right] \tag{B.6}$$

with $\mathcal{M}$ the number of matches between the two vectors,

$$\beta_{i,j}(t) = \mathcal{N}\left[S^{infer}_{i,k,j}(t), S^{recv}_{i,k,j}(t)\right] \tag{B.7}$$

with $\mathcal{N}$ the number of mismatches between the two vectors, and $X_{i,j}(t)$ the number of cases where no inference could be made.

9. Use the quantities $\alpha_{i,j}(t)$, $\beta_{i,j}(t)$, and $X_{i,j}(t)$ to assess the trust in node j. For example, compute the trust of node i in node j at time t as

$$\zeta_{i,j}(t) = \left[1 + X_{i,j}(t)\right] \frac{\alpha_{i,j}(t)}{\alpha_{i,j}(t) + \beta_{i,j}(t)} \tag{B.8}$$

Simulation of the distributed trust algorithm. The cloud application is a simulation of a CRN to assess the effectiveness of a particular trust assessment algorithm. Multiple instances of the algorithm run concurrently on an AWS cloud. The area where the secondary nodes are located is partitioned in several overlapping sub-areas as in Figure B.1. The secondary nodes are identified by an instance ID, *iId*, as well as a global ID, *gId*. The simulation assumes that the primary nodes cover the entire area thus their position is immaterial.

The simulation involves a controller and several cloud instances; in its initial implementation, the controller runs on a local system under Linux Ubuntu 10.04 LTS. The controller supplies the data, the trust program, and the scripts to the cloud instances; the cloud instances run under the Basic 32-bit Linux image on AWS, the so called *t1.micro*. The instances run the actual trust program and compute the instantaneous trust inferred by a neighbor; the results are then processed by an *awk*[2] script to compute the average trust associated with a node as seen by all its neighbors. On the next version of the application the data is stored on the cloud using the *S3* service and the controller also runs on the cloud.

In the simulation discussed here the nodes with

$$gId = \{1, 3, 6, 8, 12, 16, 17, 28, 29, 32, 35, 38, 39, 43, 44, 45\} \tag{B.9}$$

[2]The AWK utility is based on a scripting language and used for text processing; in this application it is used to produce formatted reports.

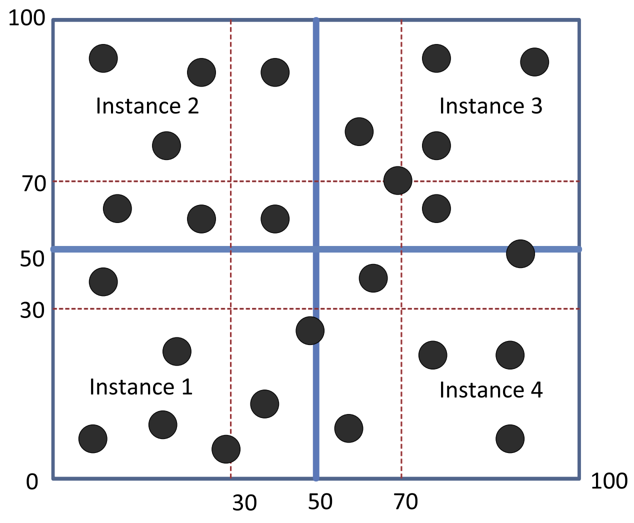


FIGURE B.1

Data partitioning for the simulation of a trust algorithm; the area covered is of size 100×100 units. The nodes in the four sub-areas of size 50×50 units are processed by an instance of the cloud application. The sub-areas allocated to an instance overlap to allow an instance to have all the information about a node in its coverage area.

were programmed to be dishonest. The results show that the nodes programmed to act maliciously have a trust value lower than that of the honest nodes; their trust value is always lower than 0.6 and, in many instances lower than 0.5, see Figure B.2. We also observe that the node density affects the accuracy of the algorithm; the algorithm predicts more accurately the trust in densely populated areas. As expected, nodes with no neighbors are unable to compute the trust.

In practice the node density is likely to be non-uniform, high density in a crowded area such as a shopping mall, and considerably lower in surrounding areas. This indicates that when the trust is computed using the information provided by all secondary nodes we can expect a higher accuracy of the trust determination.

B.2 SIMULATION OF TRAFFIC MANAGEMENT IN A SMART CITY

The objective of the project is to study the traffic crossing the center of a city for different traffic intensities and traffic light scheduling strategies.

The layout of the city center. Rectangular grid with n rows and m columns. There are NS (North–South) avenues and EW (East–West) streets. All avenues and streets are one way and have multiple lanes.

- The NS and SN avenues are $\mathcal{A}^{NS}(i)$ and $\mathcal{A}_i^{SN}, i \le m$, respectively. The EW and WE streets are $\mathcal{S}_j^{EW}$ and $S_j^{WE}, j \le n$, respectively.

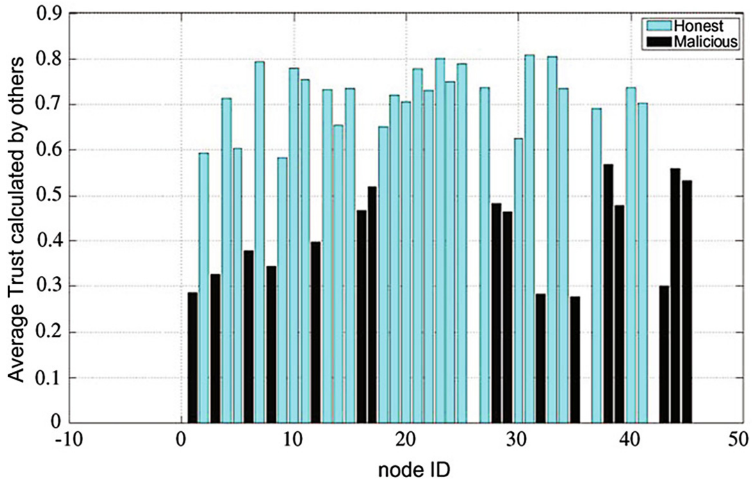


FIGURE B.2

The trust values computed using the distributed trust algorithm. The secondary nodes programmed to act maliciously have a trust value less than 0.6 and many less than 0.5, lower than that of the honest nodes.

- All distances are measured in c units; one unit equals the average car length plus an average required distance from the car ahead.
- The distance between rows i and $i+1$ and between columns j and $j+1$ are denoted by d_i, $1 \leq i \leq n$ and d_j, $1 \leq j \leq m$, respectively and $\min(d_i, d_j) > kc$ with $k > 100$.
- The direction of one-way traffic alternates on both avenues and streets; $\mathcal{A}_j^{NS}$, $j \in \{1, 3, \ldots\}$ and $\mathcal{A}^{SN}$, $j \in \{2, 4, \ldots\}$; similarly, $\mathcal{S}_i^{EW}$, $i \in \{1, 3, \ldots\}$ and $\mathcal{S}^{WE}$, $i \in \{2, 4, \ldots\}$. Avenues and streets have either two or three lines. Call L_j^{NS} the number of lanes of $\mathcal{A}_j^{NS}$.

The traffic lights are installed at all intersections $\mathcal{I}_{i,j}$, $1 \leq i \leq n,\ 1 \leq j \leq m$. The traffic lights $\mathcal{I}_{i,j}$, $1 < i < n,\ 1 < j < m$ allow left turns. Call $\tau_{i,j}^{gNS}(t)$, $\tau_{i,j}^{gNW}(t)$, $\tau_{i,j}^{gEW}(t)$ and $\tau_{i,j}^{gES}(t)$ the duration of the green light for the cycle starting at time t for directions NS, NW, EW, and ES, respectively, of traffic light $\mathcal{I}_{i,j}$.

Cars enter and exit the grid from all directions, NS, SN, EW, and WE.

- Each car has an associated path. For example, the path P^k of car C_i^k, $1 \leq i \leq MaxConv$ entering the grid is described by the pair entry point $\mathcal{I}_{i,j}^{k,entry}$ and exit point $\mathcal{I}_{i,j}^{k,exit}$ with $i = 1$ or $i = n$ and $j = 1$ or $j = m$ and at most two intermediate turning points $\mathcal{I}_{i,j}^{k,turn1}$ and $\mathcal{I}_{i,j}^{k,turn2}$ with $1 < i < n$ and $1 < j < m$. Call t_i^{in} the time when a car enters the grid.
- Cars entering the grid are grouped in “convoys” of different sizes. Convoys can be split when cars leave it to turn left or right or when the light turns red. Convoys are merged when cars enter the convoy or when cars from a different convoy join one convoy stopped at a traffic light.

- Cars whose paths requires a right turn should be in the right lane of an avenue or street and cars whose path requires a left turn should be on the left lane. Cars that do not turn should be on the center lane.
- Call $\nu_{i,j}^{in}(t,s)$ and $\nu_{p,q}^{out}(t,s)$ the number of cars in the convoys entering and respectively exiting the grid at intersections $\mathcal{I}_{i,j}$ and $\mathcal{I}_{p,q}$, in the interval $s-t$.
- Call $\Phi_{i,j}^{in}(t,s)$ the traffic intensity as the number of cars entering the grid in the interval $s-t$.

Hints for project implementation. Some of the suggestions for the project:

1. Create a description file with separate sections describing:
   (a) The layout including n, m, c; for example, $n=100$, $m=80$, and $c=12$ m.
   (b) The initial setting of the traffic lights.
   (c) The car arrival process and the convoys at each entry point.

   The simulation will require several data structures including:

- Car records. Each record includes:
  1. static data such as: the CarId, the entry point, the exit point, the entry time, the path;
  2. dynamic data such as: a list of all traffic lights it had to wait for the green light and the waiting time, the speed on each block, the time of exit.
- Traffic light records. Each record should include:
  1. static data such as: TrafficLightId; the location as the intersection of avenue $\mathcal{A}_i^{XX}$ with street $\mathcal{S}_j^{YY}$;
  2. dynamic data such as: the number of cycles (red + yellow + green); for each cycle it should include the times for green in each direction and the identity of convoys waiting.

Test several algorithms for traffic management.

1. Dumb scheduling – traffic lights follow a static, deterministic schedule.
2. Individual self-managed scheduling – a traffic light switches to green in direction DD when the length of the queue of cars waiting to cross the intersection exceeds a threshold L_{sw}.
3. Coordinated scheduling – modify the previous algorithm to include communication among neighboring nodes; use a publish-subscribe algorithm in which each traffic light subscribes to messages sent by its four neighbors.
4. Convoy-aware algorithm – attempts to create a green wave for the largest convoys and anticipate the setting of green lights to next intersections at the time the convoy reaches the intersection.

To evaluate the performance of an algorithm compute the *average transition time* of all cars in the interval (s,t).

Project implementation.[3] The simulation includes a graphical user interface displaying the smart city center. The four traffic scheduling algorithms are called: Dumb, Self-Managed, Coordinated, and Convoy.

The implementation uses Java Swing API and defines several classes:

1. Road – manages the layout of the grid.

[3]The implementation of the project is due to the group of five students: Ahmed Alhazmi, Austin Jerome, Ahmad Qutbuddin, Sai Lalitha Renduchintala, and Wendelyn Sanabria.

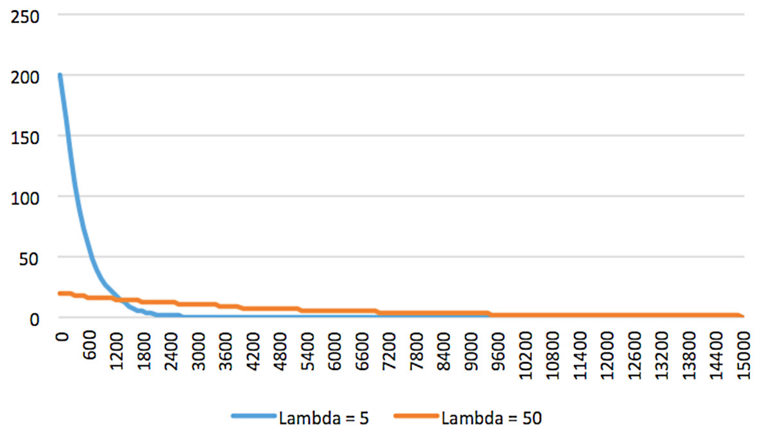


FIGURE B.3

The number of cars entering the system function of time represented by simulation cycles. Higher values of the parameter λ in Equation (B.10) scatters cars for a larger number of simulation cycles.

2. Traffic Point – handles intersections, as well as entry and exit points.
3. PaintGrid – is responsible for continuously painting and repainting the grid for a given time interval.
4. Frame – sets the size of the Frame Window used by the Canvas drawn the grid.
5. Car – creates cars and feeds them to a Java hashmap dynamic list.
6. Schedule – manages changing of the traffic lights and implements the logic of the four scheduling algorithms and controls the car arrival process.
7. Convoy – generates new convoys and adds cars to a convoy.
8. Statistics – gather traffic statistics.
9. StatWindow – display traffic statistics.

The car arrival stochastic process has an exponential distribution of the interarrival-times

$$p(t) = \lambda e^{-\lambda t}. \tag{B.10}$$

Lower values of parameter λ in Equation (B.10) results in higher concentration of cars for the first few simulation cycles, while higher values of $lambda$ scatters cars on larger number of simulation cycles see Figure B.3. A record describing the path of a car is created at the time when a car enters the grid. The path of a car could have zero turns if the car enters and exit on the same street or avenue. The path will have one turn if the car enters on one street and exits from an avenue, or enters an avenue and exits a street. The path will have two turns if the car enters one street and exits from another street or enters an avenue and exits from a different avenue.

Shown next is a segment of the configuration file specifying the layout and some of parameters related to cars.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Configuration {
    // Simulation class:
    //  Exponential Car Insertion Rate; Number of Cars
    protected int Lambda = 15;
    protected int NumberOfCars = 350;
    // Grid class:
    //  Number of Streets and Avenues
    //  Maximum and Minimum Block Side Length in c units
    protected int NumberOfStreets = 3;
    protected int NumberOfAvenues = 3;
    protected int MaximumBlockSide = 35;
    protected int MinimumBlockSide = 35;
    // Road class:
    //  Number of Forward and Turning Lanes
    protected int NumberOfForwardLanes = 2;
    protected int NumberOfTurningLanes = 1;
    // TrafficLight class
    //  Maximum Red and Green time in seconds
    //  Maximum Red Time could be Maximum Green Time + Yellow Time
    //  Yellow Time in seconds; Intersection light initial status (TBD)
    //  Scheduling Scheme D, S, C, V
    protected int MaxRedTime = 4000;
    protected int MaxGreenTime = 3000;
    protected int YellowTime = 1000;
    protected char SchedulingScheme = 'C';
    // Car class:
    //  Maximum Car Speed in c/second unit
    //  Car Acceleration in c/second2 unit; Car length and width in pixels
    protected int CarSpeed = 5;
    protected int CarAcceleration = 1;
    protected int CarLength = 6;
    protected int CarWidth = 3;
    protected int Clearance = 2;
```

The next step involves the logic of car movement through an intersection. The traffic light checks if the car is moving straight or turning using car path information. The state of the traffic light status determines whether the car should decelerate to stop or move ahead and accelerate towards the next intersection. If the car turns, then a car speed switch function is called.

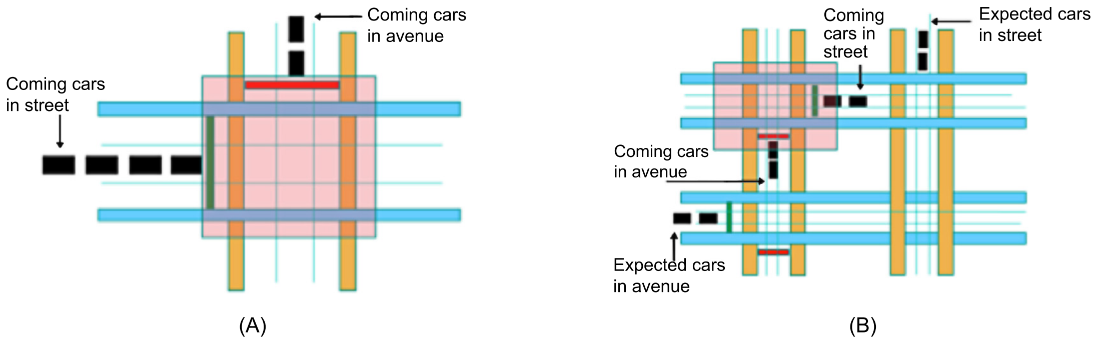


FIGURE B.4

(A) Self-managed scheduling uses the information about the number of cars to change the traffic light. (B) In the coordinated scheduling self-managed intersections coordinate their traffic lights.

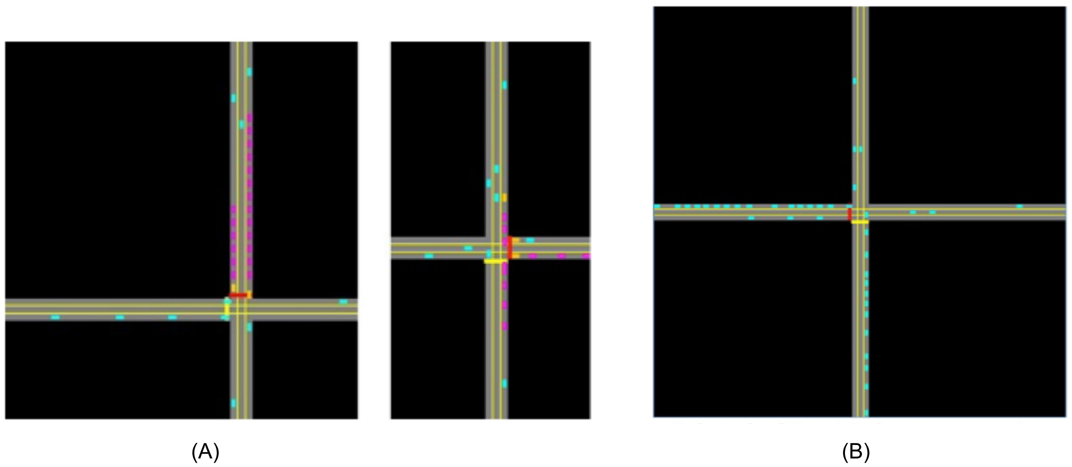


FIGURE B.5

(A) Convoy of cars waiting at an intersection (left); traffic lights extend the yellow lights to allow a convoy to cross the intersection (right). (B) The simulation of maximum capacity scenario based on an ideal orchestration.

The self-managed scheduling uses two queues for the traffic lights at the intersection of streets and avenues. The length of the car queue at the traffic lights triggers changes of the traffic lights, as seen in Figure B.4A

The coordinated scheduling algorithm was implemented as an extension to the self-managed scheduling algorithm. Now an intersection communicates with its neighboring intersections to determine the number of cars that are passing through. This allows the next intersection along the path to turn Green when a larger number of cars will reach the intersection, see Figure B.4B.

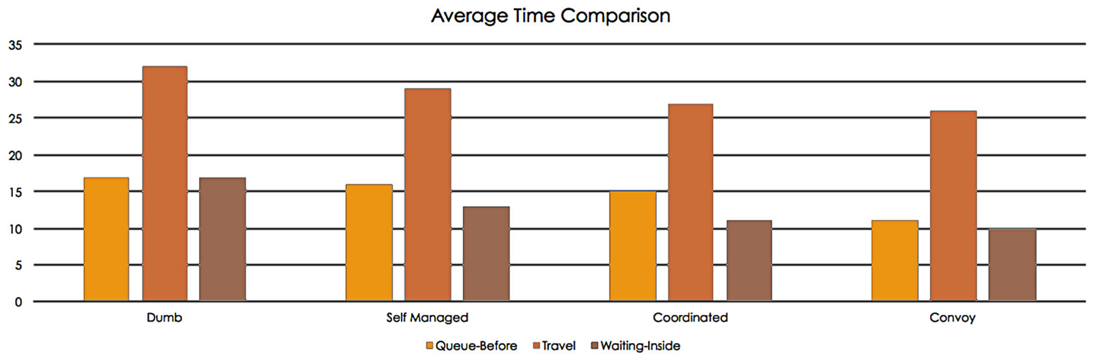


FIGURE B.6

The three bars for each one of the four traffic lights scheduling algorithms represent: (a) the waiting time before entering the grid; (b) the transit time through the grid; and (c) the waiting time at traffic lights while transiting the grid. The four traffic lights scheduling algorithms are: (1) dumb; (2) self-managed; (3) correlated; and (4) convoys.

The convoy scheduling algorithm treats a number of n cars separated by a car length, c, to be assimilated with larger car of length $n \times c$. Once a convoy is in motion, a traffic light extends its yellow state to allow the entire convoy to pass. Once a car turns, the whole convoy is broken up, see Figure B.5A.

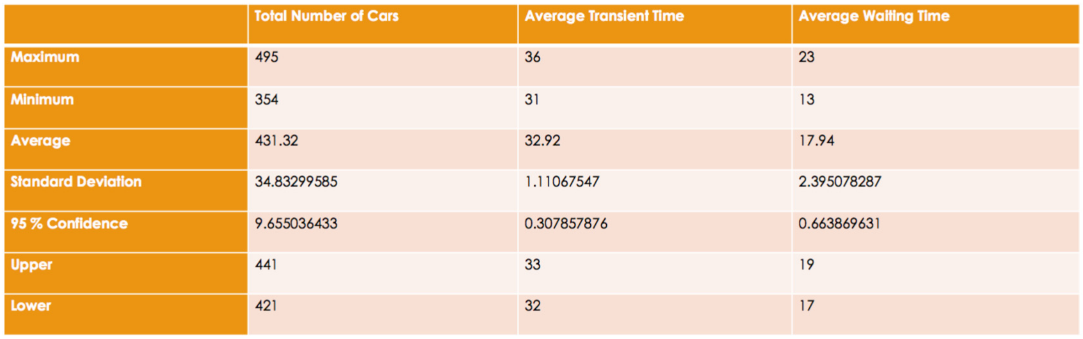
We have also investigated an ideal orchestrations scenario when the distance between cars allows the two continuous streams of cars entering every intersection from the two directions to pass alternatively through without stopping. This ideal traffic light-less scenario seen in Figure B.5B allows us to determine the maximum capacity of the grid.

Some of the simulation results are discussed next. The average transit time decreases steadily from about 36.8 units of simulated time for the Dumb traffic lights scheduling algorithm to 32.6 for the self-managed, 32 for coordinated, and 31.4 for the convoy. The average waiting time is 19.8, 17.4, and 15.2 for the self-managed, coordinated, and convoys traffic lights scheduling algorithms, respectively.

Figure B.6 shows also a comparison of four algorithms. For each algorithm it shows the waiting time for entering the grid, the transit time through the grid, and the waiting time at traffic lights while transiting through grid. The convoy scheduling algorithm performs best. Figure B.7 shows the confidence intervals from 50 simulation runs for the coordinated scheduling algorithm.

B.3 A CLOUD TRUST MANAGEMENT SERVICE

The cloud service discussed in this section [67] is an alternative to the distributed trust management scheme analyzed in Section B.1. Mobile devices are ubiquitous nowadays and their use will continue to increase. Clouds are emerging as the computing and the storage engines of the future for a wide range of applications. There is a symbiotic relationship between the two; mobile devices can consume, as well as produce very large amounts of data, while computer clouds have the capacity to store and



	Total Number of Cars	Average Transient Time	Average Waiting Time
Maximum	495	36	23
Minimum	354	31	13
Average	431.32	32.92	17.94
Standard Deviation	34.83299585	1.11067547	2.395078287
95 % Confidence	9.655036433	0.307857876	0.663869631
Upper	441	33	19
Lower	421	32	17

FIGURE B.7

Confidence intervals for the car transit time.

deliver such data to the user of a mobile device. To exploit the potential of this symbiotic relationship we propose a new cloud service for the management of wireless networks.

Mobile devices have limited resources. While new generations of smart phones and tablet computers are likely to use multicore processors and have a fair amount of memory, power consumption is still, and will continue to be in the near future, a major concern. It seems thus, reasonable to delegate compute-intensive and data-intensive tasks to the cloud.

Transferring computations related to CRN management to a cloud supports the development of new, possibly more accurate, resource management algorithms. For example, algorithms to discover communication channels currently in use by a primary transmitter could be based on past history, but are not feasible when the trust is computed by the mobile device. Such algorithms require massive amounts of data and can also identify malicious nodes with high probability.

Mobile devices such as smart phones and tablets are able to communicate using two networks: (i) a cellular wireless network; and (ii) a WiFi network. The service we propose assumes that a mobile device uses the cellular wireless network to access the cloud, while the communication over the WiFi channel is based on cognitive radio (CR). The amount of data transferred using the cellular network is limited by the subscriber's data plan, but no such limitation exists for the WiFi network. The cloud service discussed next will allow mobile devices to use the WiFi communication channels in a cognitive radio network environment and will reduce the operating costs for the end-users.

While the focus of our discussion is on trust-management for CRN networks, the cloud service we propose can be used for tasks other than the bandwidth management. For example, routing in a mobile ad hoc network, detection and isolation of non-cooperative nodes, and other network management and monitoring functions could benefit from the identification of malicious nodes.

Model assumptions. The cognitive radio literature typically analyzes networks with a relatively small number of nodes active in a limited geographic area; thus, all nodes in the network sense the same information on channel occupancy. Channel impairments such as signal fading, noise, and so on, cause errors and lead trustworthy nodes to report false information. We consider networks with a much larger number of nodes distributed over a large geographic area; as the signal strengths decays with the distance we consider several rings around a primary tower. We assume a generic fading model given by the following expression

$$\gamma_k^i = T_k \times \frac{A^2}{s_{ik}^{\alpha}} \tag{B.11}$$

where γ_k^i is the received signal strength on channel k at location of node i, A is the frequency constant, $2 \leq \alpha \leq 6$ is path loss factor, s_{ik}^{α} is the distance between primary tower P_k and node i, and T_k is the transition power of primary tower P_k transmitting on channel k.

In our discussion we assume that there are K channels labeled $1, 2, \ldots, K$ and that the primary transmitter P^k transmits on channel k. The algorithm is based on several assumptions regarding the secondary nodes, the behavior of malicious nodes, and the geometry of the system. First, we assume that the secondary nodes:

- Are mobile devices; some are slow-moving, while others are fast-moving.
- Cannot report their position because they are not equipped with a GPS system.
- The clocks of the mobile devices are not synchronized.
- The transmission and reception range of a mobile device can be different.
- The transmission range depends on the residual power of each mobile device.

We assume that the malicious nodes in the network are a minority and their behavior is captured by the following assumptions:

- The misbehaving nodes are malicious, rather than selfish; their only objective is to hinder the activity of other nodes whenever possible, a behavior distinct from the one of selfish nodes motivated to gain some advantage.
- The malicious nodes are uniformly distributed in the area we investigate.
- The malicious nodes do not collaborate in their attack strategies.
- The malicious nodes change the intensity of their Byzantine attack in successive time slots; similar patterns of malicious behavior are easy to detect and an intelligent attacker is motivated to avoid detection.

The geometry of the system is captured by Figure B.8. We distinguish primary and secondary nodes and the cell towers used by the secondary nodes to communicate with service running on the cloud.

We use majority voting rule for a particular ring around a primary transmitter; the global decision regarding the occupancy of a channel requires a majority of the votes. Since the malicious nodes are a minority and they are uniformly distributed, the malicious nodes in any ring are also a minority; thus, a ring based majority fusion is a representative of accurate occupancy for the channel associated with the ring.

All secondary nodes are required to register first and then to transmit periodically their current power level, as well as their occupancy report for each one of the K channels. As mentioned in the introductory discussion, the secondary nodes connect to the cloud using the cellular network. After a mobile device is registered, the cloud application requests the cellular network to detect its location; the towers of the cellular network detect the location of a mobile device by triangulation with an accuracy which is a function of the environment and is of the order of 10 meters. The location of the mobile device is reported to the cloud application every time they provide an occupancy report.

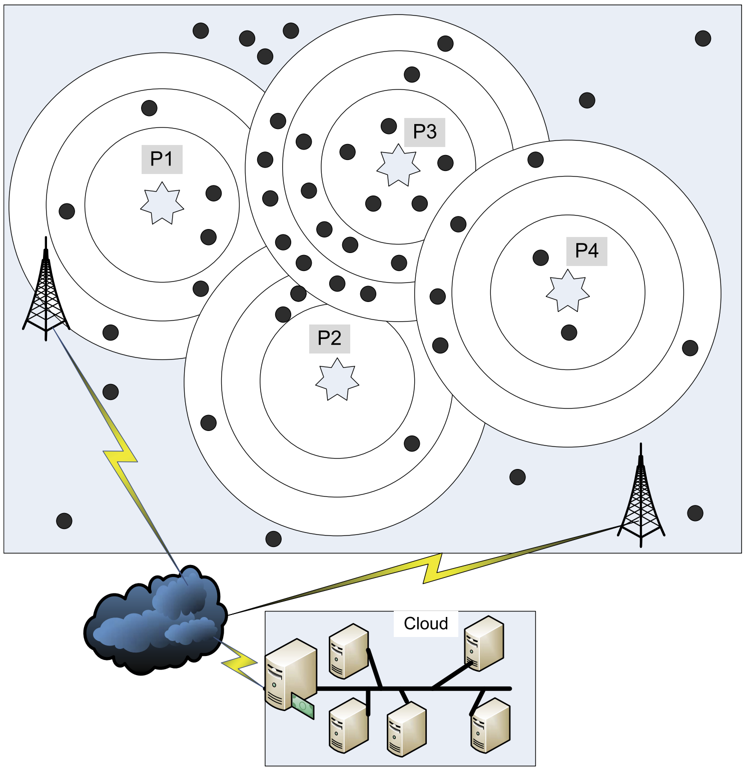


FIGURE B.8

Schematic representation of a CR layout; four primary nodes, P_1–P_4, a number of mobile devices, two towers for a cellular network and a cloud are shown. Not shown are the hotspots for the WiFi network.

The nodes which do not participate in the trust computation will not register in this cloud-based version of the resource management algorithm thus, they do not get the occupancy report and cannot use it to identify free channels. Obviously, if a secondary node does not register it cannot influence other nodes and prevent them from using free channels, or tempt them to use busy channels.

In the registration phase a secondary node transmits its MAC address and the cloud responds with the tuple (Δ, δ_s). Here, Δ is the time interval between two consecutive reports, chosen to minimize the communication, as well as the overhead for sensing the status of each channel. To reduce the communication overhead secondary nodes should transmit only the changes from the previous status report. $\delta_s < \Delta$ is the time interval to the first report expected from the secondary node. This scheme provides a pseudo-synchronization, so that the data collected by the cloud and used to determine the trust is based on observations made by the secondary nodes at about the same time.

An algorithm for trust evaluation based on historical information. The cloud computes the probable distance d_i^k of each secondary node i from the known location of a primary transmitter, P^k. Based on signal attenuation properties we conceptualize N circular rings centered at the primary where each ring is denoted by $\mathcal{R}_r^k$, with $1 \leq r \leq N$ the ring number.

The radius of a ring is based on the distance d_r^k to the primary transmitter P^k. A node at a distance $d_i^k \leq d_1^k$ is included in the ring $\mathcal{R}_1^k$, nodes at distance $d_1^k < d_i^k \leq d_2^k$ are included in the ring $\mathcal{R}_2^k$, and so on. The closer to the primary, the more accurate the channel occupancy report of the nodes in the ring should be. Call n_r^k the number of nodes in ring $\mathcal{R}_r^k$.

At each report cycle at time t_q, the cloud computes the occupancy report for channel $1 \leq k \leq K$ used by primary transmitter P^k. The status of channel k reported by node $i \in \mathcal{R}_r^k$ is denoted as $s_i^k(t_q)$. Call $\sigma_{one}^k(t_q)$ the count of the nodes in the ring $\mathcal{R}_r^k$ reporting that the channel k is not free (reporting $s_i^k(t_q) = 1$) and $\sigma_{zero}^k(t_q)$ the count of those reporting that the channel is free (reporting $s_i^k(t_q) = 0$):

$$\sigma_{one}^k(t_q) = \Sigma_{i=1}^{n_r^k} s_i^k(t_q) \quad \text{and} \quad \sigma_{zero}^k(t_q) = n_r^k - \sigma_{one}^k(t_q). \tag{B.12}$$

Then the status of channel k reported by the nodes in the ring R_r^k is determined by majority voting as

$$\sigma_{R_r}^k(t_q) \begin{cases} = 1 & \text{when } \sigma_{one}^k(t_q) \geq \sigma_{zero}^k(t_q) \\ = 0 & \text{otherwise} \end{cases} \tag{B.13}$$

To determine the trust in node i we compare $s_i^k(t_q)$ with $\sigma_{R_r}^k(t_q)$; call $\alpha_{i,r}^k(t_q)$ and $\beta_{i,r}^k(t_q)$ the number of matches and, respectively, mismatches in this comparison for each node in the ring R_r^k. We repeat this procedure for all rings around P^k and construct

$$\alpha_i^k(t_q) = \Sigma_{r=1}^{n_r^k} \alpha_{i,r}^k(t_q) \quad \text{and} \quad \beta_i^k(t_q) = \Sigma_{r=1}^{n_r^k} \beta_{i,r}^k(t_q) \tag{B.14}$$

Node i will report the status of the channels in the set $C_i(t_q)$, the channels with index $k \in C_i(t_q)$; then quantities $\alpha_i(t_q)$ and $\beta_i(t_q)$ with $\alpha_i(t_q) + \beta_i(t_q) = |C_i(t_q)|$ are

$$\alpha_i(t_q) = \Sigma_{k \in C_i} \alpha_i^k(t_q) \quad \text{and} \quad \beta_i(t_q) = \Sigma_{k \in C_i} \beta_i^k(t_q). \tag{B.15}$$

Finally, the global trust in node i is a random variable

$$\zeta_i(t_q) = \frac{\alpha_i(t_q)}{\alpha_i(t_q) + \beta_i(t_q)}. \tag{B.16}$$

The trust in each node at each iteration is determined using a similar strategy as the one discussed earlier; its status report, $S_j(t)$, contains only information about the channels it can report on and only if the information has changed from the previous reporting cycle.

Then a statistical analysis of the random variables for a window of time W, $\zeta_j(t_q), t_q \in W$ allows us to compute the moments as well as a 95% confidence interval. Based on these results we assess if node j is trustworthy and eliminate the untrustworthy ones when we evaluate the occupancy map at the next cycle. We continue to assess the trustworthiness of all nodes and may accept the information from node j when its behavior changes.

Let us now discuss the use of historical information to evaluate trust. We assume a sliding window $W(t_q)$ consists of n_w time slots of duration τ. Given two decay constants k_1 and k_2, with $k_1 + k_2 = 1$, we use an exponential averaging giving decreasing weight to old observations. We choose $k_1 << k_2$ to give more weight to the past actions of a malicious node. Such nodes attack only intermittently and try to disguise their presence with occasional good reports; the misbehavior should affect the trust more than the good actions. The history-based trust requires the determination of two quantities

$$\alpha_i^H(t_q) = \Sigma_{i=0}^{n_w-1} \alpha_i(t_q - i\tau)k_1^i \quad \text{and} \quad \beta_i^H(t_q) = \Sigma_{i=0}^{n_w-1} \beta_i(t_q - i\tau)k_2^i \tag{B.17}$$

Then the history-based trust for node i valid only at times $t_q \geq n_w\tau$ is:

$$\zeta_i^H(t_q) = \frac{\alpha_i^H(t_q)}{\alpha_i^H(t_q) + \beta_i^H(t_q)}. \tag{B.18}$$

For times $t_q < n_w\tau$ the trust will be based only on a subset of observations rather than a full window on n_w observations.

This algorithm can also be used in regions where the cellular infrastructure is missing. An ad hoc network could allow the nodes that cannot connect directly to the cellular network to forward their information to nodes closer to the towers and then to the cloud-based service.

Simulation of the history-based algorithm for trust management. The aim of the history-based trust evaluation is to distinguish between trustworthy and malicious nodes. We expect the ratio of malicious to trustworthy nodes, as well as the node density, to play an important role in this decision. The node density, ρ, is the number of nodes per unit of area. In our simulation experiments the size of the area is constant, but the number of nodes increases from 500 to 2 000 thus, the node density increases by a factor of four. The ratio of the number of malicious to the total number of nodes varies between $\alpha = 0.2$ to a worst case of $\alpha = 0.6$.

The performance metrics we consider are: the average trust for all nodes, the average trust of individual nodes, and the error of honest/trustworthy nodes. We wish to see how the algorithm behaves when the density of the nodes increases; we consider four cases with 500, 1000, 1 500 and 2 000 nodes on the same area thus, we allow the density to increase by a factor of four. We also investigate the average trust when α, the ratio of malicious nodes to the total number of nodes increases from $\alpha = 0.2$ to $\alpha = 0.4$ and, finally, to $\alpha = 0.6$.

This straightforward data partitioning strategy for the distributed trust management algorithm is not a reasonable one for the centralized algorithm because it would lead to excessive communication among the cloud instances. Individual nodes may contribute data regarding primary transmitters in a different sub-area; to evaluate the trust of each node the cloud instances would have to exchange a fair amount of information. This data partitioning would also complicate our algorithm which groups together secondary nodes based on the distance from the primary one.

Instead, we allocate to each instance a number of channels and all instances share the information about the geographic position of each node; the distance of a secondary node to any primary one can then be easily computed. This data partitioning strategy scales well in the number of primaries. Thus, it is suitable for simulation in large metropolitan areas, but may not be able to accommodate cases when the number of secondaries is of the order of 10^8–10^9.

The objectives of our studies are to understand the limitations of the algorithm; the aim of the algorithm is to distinguish between trustworthy and malicious nodes. We expect that the ratio of malicious to trustworthy nodes, as well as the node density should play an important role in this decision. The measures we examine are the average trust for all nodes, as well as the average trust of individual nodes.

The effect of the malicious versus trustworthy node ratio on the average trust. We report the effect of the malicious versus trustworthy node ratio on the average trust when the number of nodes increases. The average trust is computed separately for the two classes of nodes and allows us to determine if the algorithm is able to clearly separate them.

Recall that the area is constant thus, when the number of nodes increases so does the node density. First, we consider two extreme cases; the malicious nodes represent only 20% of the total number of nodes and an unrealistically high presence, of 60%. Then we report on the average trust when the number of nodes is fixed and the malicious nodes represent an increasing fraction of the total number of nodes.

Results reported in [67] show that when the malicious nodes represent only 20% of all nodes, there is a clear distinction between the two groups. The malicious nodes have an average trust of 0.28 and the trustworthy ones have an average trust index of 0.91, regardless of the number of nodes.

When the malicious nodes represent 60% of all the nodes then the number of nodes plays a significant role; when the number of nodes is small the two groups cannot be distinguished their average trust index is almost equal, 0.55 although the honest nodes have a slightly more average trust value. When the number of nodes increases to 2 000 and the node density increases four folds then the average trust of the first (malicious) group decreases to 0.45 and for the second (honest) group it increases to about 0.68.

This result is not unexpected; it only shows that the history-based algorithm is able to classify the nodes properly even when the malicious nodes are a majority, a situation we do not expect to encounter in practice. This effect is somewhat surprising; we did not expect that under these extreme condition the average of the trust of all nodes will be so different for the two groups. A possible explanation is that our strategy to reward constant good behavior, rather than occasional good behavior, designed to mask the true intentions of a malicious node, works well.

Figure B.9 shows the average trust function of α, the ratio of malicious versus total number of nodes. The results confirm the behavior discussed earlier; we see a clear separation of the two classes only when the malicious nodes are in minority. When the density of malicious nodes approaches a high value so that they are in majority, the algorithm still performs as evident from the figure that the average trust for honest nodes even at high value of α is more than for malicious nodes. Thus the trusts reflect the aim of isolating the malicious from the honest set of nodes. We also observe that the separation is more clear when the number of nodes in the network increases.

The benefits of a cloud-based service for trust management. A cloud service for trust management in cognitive networks can have multiple technical as well as economical benefits [97]. The service is likely to have a broader impact than the one discussed here, it could be used to support a range of important policies in wireless network where many decisions require the cooperation of all nodes. A history-based algorithm to evaluate the trust and detect malicious nodes with high probability is at the center of the solution we have proposed [67].

A centralized, history-based algorithm for bandwidth management in CRNs has several advantages over the distributed algorithms discussed in the literature:

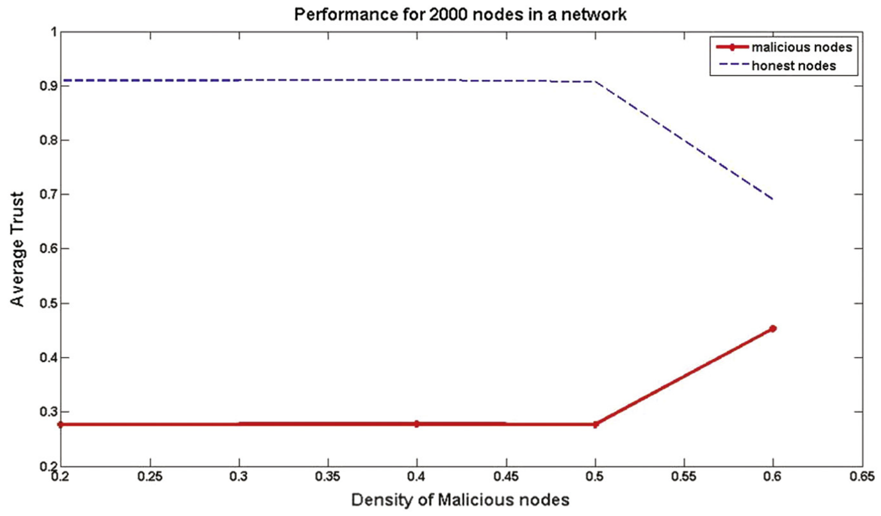


FIGURE B.9

The average trust function of α for a population size of 2000 nodes. As long as malicious nodes represent 50% or less of the total number of nodes the average trust of malicious nodes is below 0.3, while the one of trustworthy nodes is above 0.9 in a scale of 0 to 1.0. As the number of nodes increases, the distance between the average trust of the two classes becomes larger and even larger when $\alpha > 0.5$. i.e., the malicious nodes are in majority.

- Drastically reduces the computations a mobile device is required to carry out to identify free channels and avoid penalties associated with interference with primary transmitters.
- Allows a secondary node to get information about channel occupancy as soon as it joins the system and later on demand; this information is available even when a secondary node is unable to receive reports from its neighbors, or when it is isolated.
- Does not require the large number of assumptions critical to the distributed algorithms.
- The dishonest nodes can be detected with high probability and their reports can be ignored; thus in time the accuracy of the results increases. Moreover, historic data could help detect a range of Byzantine attacks orchestrated by a group of malicious nodes.
- It is very likely to produce more accurate results than the distributed algorithm as the reports are based on information from all secondary nodes reporting on a communication channel used by a primary, not only those in its vicinity; a higher node density increases the accuracy of the predictions. The accuracy of the algorithm is a function of the frequency of the occupancy reports provided by the secondary nodes.

The centralized trust management scheme has several other advantages. First, it can be used not only to identify malicious nodes and provide channel occupancy reports but also to manage the allocation of free channels. In the distributed case, two nodes may attempt to use a free channel and collide; this situation is avoided in the centralized case. At the same time, malicious nodes can be identified with high probability and be denied access to the occupancy report.

The server could also collect historic data regarding the pattern of behavior of the primary nodes and use this information for the management of free channels. For example, when a secondary node requests access for a specific length of time the service may attempt to identify a free channel likely to be available for that time.

The trust management may also be extended to other network operations such as routing in a mobile ad hoc network; the strategy in this case would be to avoid routing through malicious nodes.

B.4 A CLOUD SERVICE FOR ADAPTIVE DATA STREAMING

In this section we discuss a cloud application related to data streaming [397]. Data streaming is the name given to the transfer of data at a high-rate with real-time constraints. Multi-media applications such as music and video streaming, high-definition television (HDTV), scientific applications which process a continuous stream of data collected by sensors, the continuous backup copying to a storage medium of the data flow within a computer, and many other applications require the transfer of real-time data at a high-rate. For example, to support real-time human perception of the data, multi-media applications have to make sure that enough data is being continuously received without any noticeable time lag.

We are concerned with the case when the data streaming involves a multi-media application connected to a service running on a computer cloud. The stream could originate from the cloud, as is the case of the iCloud service provided by Apple, or could be directed toward the cloud, as in the case of a real-time data collection and analysis system.

Data streaming involves three entities, the sender, a communication network, and a receiver. The resources necessary to guarantee the timing constraints include CPU cycles and buffer space at the sender and the receiver and network bandwidth. Adaptive data streaming determines the data rate based on the available resources. Lower data rates imply lower quality, but reduce the demands for system resources.

Adaptive data streaming is possible only if the application permits trade-offs between quantity and quality. Such trade-offs are feasible for audio and video streaming which allow lossy compression, but are not acceptable for many applications which processes a continuous stream of data collected by sensors.

Data streaming requires accurate information about all resources involved and this implies that the network bandwidth has to be constantly monitored; at the same time, the scheduling algorithms should be coordinated with memory management to guarantee the timing constraints. Adaptive data streaming poses additional constraints because the data flow is dynamic. Indeed, once we detect that the network cannot accommodate the data rate required by an audio or video stream we have to reduce the data rate thus, to convert to a lower quality audio or video. Data conversion can be done on the fly and, in this case, the data flow on the cloud has to be changed.

Accommodating dynamic data flows with timing constraints is non-trivial; only about 18% of the top 100 global video web sites use ABR (Adaptive Bit Rate) technologies for streaming [469].

This application stores the music files in S3 buckets and the audio service runs on the EC2 platform. In EC2 each virtual machine functions as a virtual private server and is called an *instance;* an instance specifies the maximum amount of resources available to an application, the interface for that instance, as well as the cost per hour.

EC2 allows the import of VM images from the user environment to an instance through a facility called *VM import*. It also distributes automatically the incoming application traffic among multiple instances using the *elastic load balancing* facility. EC2 associates an *elastic IP address* with an account. This mechanism allows a user to mask the failure of an instance and re-map a public IP address to any instance of the account, without the need to interact with the software support team.

The adaptive audio streaming involves a multi-objective optimization problem. We wish to convert the highest quality audio file stored on the cloud to a resolution corresponding to the rate that can be sustained by the available bandwidth; at the same time, we wish to minimize the cost on the cloud site, and also minimize the buffer requirements for the mobile device to accommodate the transmission jitter. Finally, we wish to reduce to a minimum the start-up time for the content delivery.

A first design decision is if data streaming should only begin after the conversion from the WAV to MP3 format has been completed, or it should proceed concurrently with conversion, in other words start as soon as several MP3 frames have been generated; another question is if the converted music file should be saved for later use or discarded.

To answer these question we experimented with conversion from the highest quality audio files which require a 320 Kbps data rate to lower quality files corresponding to 192, 128, 64, 32 and finally 16 Kbps. If the conversion time is small and constant there is no justification for pipelining data conversion and streaming, a strategy which complicates the processing flow on the cloud. It makes sense to cache the converted copy for a limited period of time with the hope that it will be reused in the next future.

Another design decision is how the two services should interact to optimize the performance; two alternatives come to mind:

1. The audio service running on the EC2 platform requests the data file from the S3, converts it, and, eventually, sends it back. The solution involves multiple delays and it is far from optimal.
2. Mount the S3 bucket as an EC2 drive. This solution reduces considerably the start-up time for audio streaming.

The conversion from a high-quality audio file to a lower quality, thus a lower bit rate is performed using the LAME library.

The conversion time depends on the desired bit-rate and the size of the original file. Tables B.1, B.2, B.3, and B.4 show the conversion time in seconds when the source MP3 file are of, 320 Kbps and 192 Kbps, respectively; the size of the input files is also shown.

The platforms used for conversion are: (a) the *EC2 t1.micro* server for the measurements reported in Tables B.1 and B.2 and (b) the *EC2 c1.medium* for the measurements reported in Tables B.3 and B.4. The instances run the Ubuntu Linux operating system.

The results of our measurements when the instance is the *t1.micro* server exhibit a wide range of the conversion times, 13–80 seconds, for the large audio file of about 6.7 MB when we convert from 320 to 192 Kbps. A wide range, 13–64 seconds, is also observed for an audio file of about 4.5 MB when we convert from 320 to 128 Kbps. For poor quality audio the file size is considerably smaller, about 0.56 MB and the conversion time is constant and small, 4 seconds.

Figure B.10 shows the average conversion time for the experiments summarized in Tables B.1, B.2. It is somewhat surprising that the average conversion time is larger when the source file is smaller as is the case when the target bit rates are 64, 32 and 16 Kbps. Figure B.11 shows the average conversion time for the experiments summarized in Tables B.3 and B.4.

Table B.1 Conversion time in seconds on a *EC2 t1.micro* server platform; the source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C 7	C8	C9	C10	$\bar{T}_c$
192	6.701974	73	43	19	13	80	42	33	62	66	36	46.7
128	4.467982	42	46	64	48	19	52	52	48	48	13	43.2
64	2.234304	9	9	9	9	10	26	43	9	10	10	14.4
32	1.117152	7	6	14	6	6	7	7	6	6	6	7.1
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table B.2 Conversion time in seconds on a *EC2 t1.micro* server platform; the source file is of high audio quality, 192 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C 7	C8	C9	C10	$\bar{T}_c$
128	4.467982	14	15	13	13	73	75	56	59	72	14	40.4
64	2.234304	9	9	9	32	44	9	23	9	45	10	19.9
32	1.117152	6	6	6	6	6	6	20	6	6	6	7.4
16	0.558720	6	6	6	6	6	6	20	6	6	6	5.1

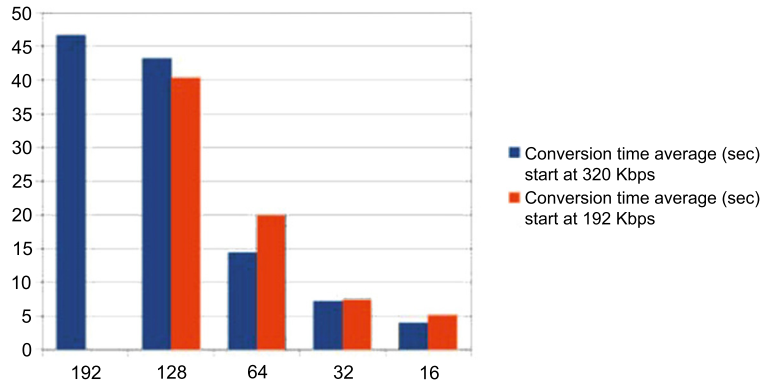


FIGURE B.10

The average conversion time on a *EC2 t1.micro* platform. The left bars and the right bars correspond to the original file at: the highest resolution (320 Kbps data rate) and next highest resolution (192 Kbps data rate), respectively.

The results of our measurements when the instance runs on the EC2 c1.medium platform show consistent and considerably lower conversion times; Figure B.11 presents the average conversion time.

To understand the reasons for our results we took a closer look at the two types of EC2 instances, "micro" and "medium", and their suitability for the adaptive data streaming service. The t1.micro supports bursty applications, with a high average-to-peak ratio for CPU cycles, e.g., transaction processing systems. EBS provides block level storage volumes; the "micro" instances are only EBS-backed.

Table B.3 Conversion time T_c in seconds on a *EC2 c1.medium* platform; the source file is of the highest audio quality, 320 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C 7	C8	C9	C10	$\bar{T}_c$
192	6.701974	15	15	15	15	15	15	15	15	15	15	15
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	11	11	11	11	11	11	11	11	11	11	11
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

Table B.4 Conversion time in seconds on a *EC2 c1.medium* platform; the source file is of high audio quality, 192 Kbps. The individual conversions are labeled $C1$ to $C10$; $\bar{T}_c$ is the mean conversion time.

Bit-rate (Kbps)	Audio file size (MB)	C1	C2	C3	C4	C5	C6	C 7	C8	C9	C10	$\bar{T}_c$
128	4.467982	15	15	15	15	15	15	15	15	15	15	15
64	2.234304	10	10	10	10	10	10	10	10	10	10	10
32	1.117152	7	7	7	7	7	7	7	7	7	7	7
16	0.558720	4	4	4	4	4	4	4	4	4	4	4

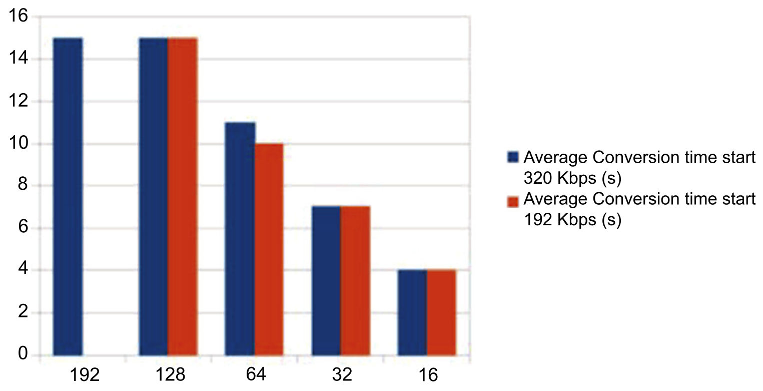


FIGURE B.11

The average conversion time on a *EC2 c1.medium* platform. The left bars and the right bars correspond to the original file at: the highest resolution (320 Kbps data rate) and next highest resolution (192 Kbps data rate), respectively.

The "medium" instances support compute-intensive application with a steady and relatively high demand for CPU cycles. Our application is compute-intensive thus, there should be no surprise that our measurements for the EC2 c1.medium platform show consistent and considerably lower conversion times.

B.5 CLOUD-BASED OPTIMAL FPGA SYNTHESIS

In this section we discuss another class of applications that could benefit from cloud computing. The benchmarks presented in Section 7.10 compared the performance of several codes running on a cloud with runs on supercomputers. As expected, the results showed that a cloud is not an optimal environment for applications exhibiting fine- or medium-grain parallelism.

Indeed, the communication latency is considerably larger on a cloud than on supercomputer with a more expensive, custom interconnect. This simply means that we have to identify applications which do not involve extensive communication, or applications exhibiting coarse-grain parallelism.

A cloud is an ideal running environment for scientific applications involving model development when multiple cloud instances could concurrently run slightly different models of the system. When the model is described by a set of parameters, the application can be based on the SPMD paradigm combined with an analysis phase when the results from the multiple instances are ranked based on a well-defined metric.

In this case there is no communication during the first phase of the application, when partial results are produced and then written to storage server. The individual instances signal the completion and a new instance to carry out the analysis and display the results is started. A similar strategy can be used by engineering applications of mechanical, civil, electrical, electronic, or any other system design area. In this case the multiple instances run concurrent design for different sets of parameters of the system.

A cloud application for optimal design of field-programmable gate arrays (FPGAs) is discussed next. As the name suggests, an FPGA is an integrated circuit designed to be configured/adapted/programmed in the field to perform a well-defined function [432]. Such a circuit consists of *logic blocks* and *interconnects* that can be "programmed" to carry out logical and/or combinatorial functions, see Figure B.12.

The first commercially viable FPGA, XC2064, was produced in 1985 by Xilinx. Today FPGAs are used in many areas including digital signal processing, CRNs, aerospace, medical imaging, computer vision, speech recognition, cryptography, and computer hardware emulation. FPGAs are less energy efficient and slower than application-specific integrated circuits (ASICs). The widespread use of FPGAs is due to their flexibility and the ability to reprogram them.

Hardware description languages (HDLs) such as VHDL and Verilog are used to program FPGAs; HDLs are used to specify a register-transfer level (RTL) description of the circuit. Multiple stages are used to synthesize FPGA.

A cloud-based system was designed to optimize the routing and placement of components. The basic structure of the tool is shown in Figure B.13. The system uses the PlanAhead tool from Xilinx, see http://www.xilinx.com/, to place system components and route chips on the FPGA logical fabric. The computations involved are fairly complex and take a considerable amount of time; for example, a fairly simple system consisting of a software core processor (Microblaze), a block random access memory (BRAM), and a couple of peripherals can take up to forty minutes to synthesize on a powerful workstation. Running N design options in parallel on a cloud speeds-up the optimization process by a factor close to N.

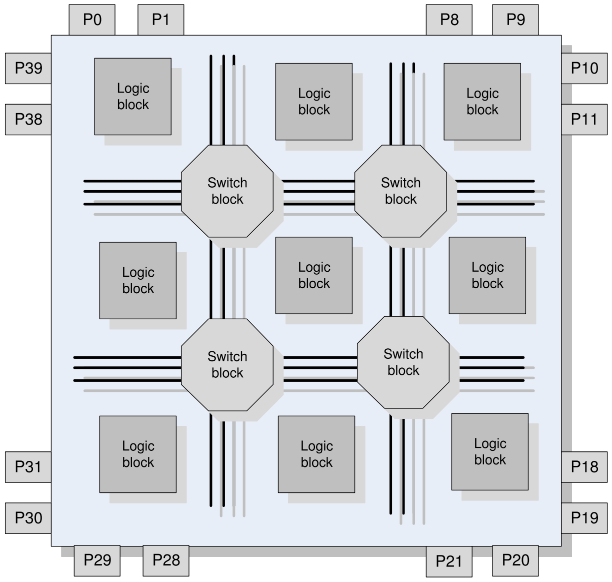


FIGURE B.12

The structure of a Field Programmable Gate Array (FPGA) with 30 pins, $P1–P29$, 9 logic blocks and 4 switchblocks.

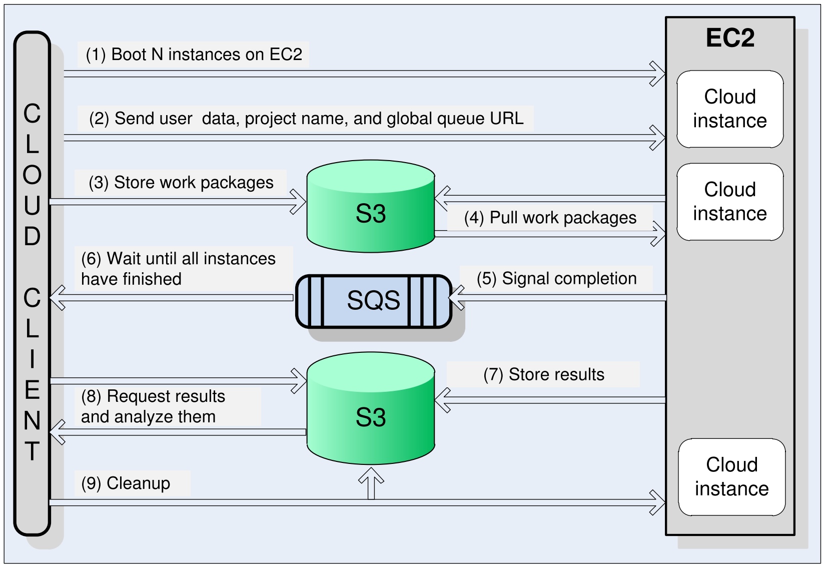

CLOUD CLIENT
EC2
(1) Boot N instances on EC2
(2) Send user data, project name, and global queue URL
(3) Store work packages
S3
(4) Pull work packages
Cloud instance
Cloud instance
(6) Wait until all instances have finished
SQS
(5) Signal completion
(7) Store results
(8) Request results and analyze them
S3
(9) Cleanup
Cloud instance

FIGURE B.13

The architecture of a cloud-based system to optimize the routing and placement of components on an FPGA.

B.6 TENSOR NETWORK CONTRACTION ON AWS

A numerical simulation project related to research in condensed matter physics is discussed in [328] and overviewed in this section. To illustrate the problems posed by Big Data we analyze different options offered by 2016 vintage Amazon Web Services for running the application. M4 and C4 seem to be the best choices for applications such as Tensor Network Contraction (TNC).

Tensor contraction. In linear algebra the *rank* $\mathcal{R}$ of an object is given by the number of indices necessary to describe its elements. A scalar has rank 0, a vector $a = (a_1, a_2, \dots, a_n)$ has rank 1 and n elements, a matrix $\mathcal{A} = [a_{ij}],\ 1 \le i \le n,\ \ 1 \le j \le m$ has rank 2 and $n \times m$ elements

$$\mathcal{A} = \begin{bmatrix} a_{11} & a_{12} & \dots a_{1m} \\ a_{21} & a_{22} & \dots a_{2m} \\ \vdots & & \\ a_{n1} & a_{n2} & \dots a_{nm} \end{bmatrix}. \tag{B.19}$$

Tensors have rank $\mathcal{R} \ge 3$; the description of tensor elements is harder. For example, consider a rank 3 tensor $\mathcal{B} = [b_{jkl}]$ with elements $b_{jkl} \in \mathbb{R}^{2\times2\times2}$. The eight elements of this tensor are: $\{b_{111}, b_{112}, b_{121}, b_{122}\}$ and $\{b_{211}, b_{212}, b_{221}, b_{222}\}$. We can visualize the tensor elements as the vertices of a cube where the first group of elements are in the plane $j = 1$ and $j = 2$, respectively. Similarly, the tensor elements $\{b_{111}, b_{211}, b_{112}, b_{212}\}$ and $\{b_{121}, b_{221}, b_{122}, b_{222}\}$ are in the planes $k = 1$ and $k = 2$, respectively, while $\{b_{111}, b_{121}, b_{211}, b_{221}\}$ and $\{b_{112}, b_{122}, b_{212}, b_{222}\}$ are in the planes $l = 1$ and $l = 2$, respectively.

Tensor contraction is the summation over repeated indices of the two tensors or of a vector and a tensor. Let $\mathcal{C}$ be the contraction of two arbitrary tensors $\mathcal{A}$ and $\mathcal{B}$. The rank of the tensor resulting after contraction is

$$\mathcal{R}(\mathcal{C}) = \mathcal{R}(\mathcal{A}) + \mathcal{R}(\mathcal{B}) - 2. \tag{B.20}$$

For example, when $\mathcal{A} = [a_{ij}]$, $\mathcal{B} = [b_{jkl}]$ and we contract over j we obtain $\mathcal{C} = [c_{ikl}]$ with

$$c_{ikl} = \sum_j a_{ij} b_{jkl}. \tag{B.21}$$

The rank of $\mathcal{C}$ is $\mathcal{R}(\mathcal{C}) = 2 + 3 - 2 = 3$. Tensor $\mathcal{C}$ has 8 elements

$$\begin{array}{l|l} c_{111} = \sum_{j=1}^{2} a_{1j} b_{j11} = a_{11}b_{111} + a_{12}b_{211} & c_{121} = \sum_{j=1}^{2} a_{1j} b_{j21} = a_{11}b_{121} + a_{12}b_{221} \\ c_{212} = \sum_{j=1}^{2} a_{2j} b_{j12} = a_{21}b_{112} + a_{22}b_{212} & c_{222} = \sum_{j=1}^{2} a_{2j} b_{j22} = a_{21}b_{122} + a_{22}b_{222} \\ c_{112} = \sum_{j=1}^{2} a_{1j} b_{j11} = a_{11}b_{112} + a_{12}b_{212} & c_{122} = \sum_{j=1}^{2} a_{1j} b_{j21} = a_{11}b_{122} + a_{12}b_{222} \\ c_{211} = \sum_{j=1}^{2} a_{2j} b_{j11} = a_{21}b_{111} + a_{22}b_{211} & c_{221} = \sum_{j=1}^{2} a_{2j} b_{j21} = a_{21}b_{121} + a_{22}b_{221} \end{array} \tag{B.22}$$

Tensor networks and tensor network contraction (TNC). A *tensor network* is defined as follows: let $[A_1], \dots, [A_n]$ be n tensors with index sets $x^{(1)}, \dots, x^{(n)}$ where each $\{x^{(i)}\}$ is a subset of $\{x_1, \dots, x_N\}$

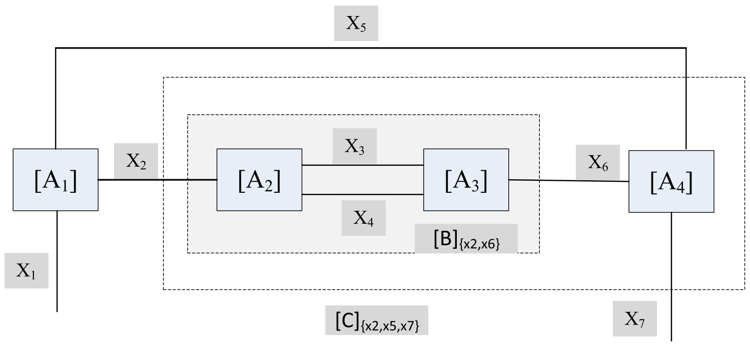


FIGURE B.14

The ordering of tensor contraction when the index set is $\{x_1, x_2, \ldots, x_7\}$ and the four "small" tensors are $[A1]_{\{x_1,x_2,x_5\}}$, $[A2]_{\{x_2,x_3,x_4\}}$, $[A3]_{\{x_3,x_4,x_6\}}$ and $[A4]_{\{x_5,x_6,x_7\}}$. Tensors $[B]$ and $[C]$ are the results of contraction of $[A_2]$, $[A_3]$, and $[A_4]$, $[B]$, respectively.

with N very large. We assume that the "big" tensor $[A]_{\{x_1,\ldots,x_N\}}$ can be expressed as the product of the "smaller" tensors $[A_1], \ldots, [A_n]$

$$[A]_{\{x_1,\ldots,x_K\}} = [A_1]_{\{x^{(1)}\}} \ldots [A_n]_{\{x^{(n)}\}}. \tag{B.23}$$

We wish to compute the scalar

$$Z_A = \sum_{\{x_1,\ldots,x_N\}} [A_i]_{\{x_1,\ldots,x_N\}} \tag{B.24}$$

For example, $N = 7$ and $n = 4$ and the index set is $\{x_1, x_2, \ldots, x_7\}$ for the TNC in Figure B.14. The four "small" tensors and their respective subsets of the index set are

$$[A1]_{\{x_1,x_2,x_5\}},\ [A2]_{\{x_2,x_3,x_4\}},\ [A3]_{\{x_3,x_4,x_6\}} \text{ and } [A4]_{\{x_5,x_6,x_7\}}. \tag{B.25}$$

The "big" tensor $[A]$ is the product of the four "small" tensors

$$[A]_{\{x_1,x_2,x_3,x_4,x_5,x_6,x_7\}} = [A1]_{\{x_1,x_2,x_5\}} \otimes [A2]_{\{x_2,x_3,x_4\}} \otimes [A3]_{\{x_3,x_4,x_6\}} \otimes [A4]_{\{x_5,x_6,x_7\}}. \tag{B.26}$$

To calculate Z_A we first contract $[A_2]$ and $[A_3]$ and the result is tensor $[B]$

$$[B]_{\{x_2,x_6\}} = \sum_{x_3} \sum_{x_4} [A2]_{\{x_2,x_3,x_4\}} \otimes [A3]_{\{x_3,x_4,x_6\}}. \tag{B.27}$$

Next we contract $[B]$ and $[A_4]$ to produce $[C]$

$$[C]_{\{x_2,x_5,x_7\}} = \sum_{x_6} [B]_{\{x_2,x_6\}} \otimes [A4]_{\{x_5,x_6,x_7\}}. \tag{B.28}$$

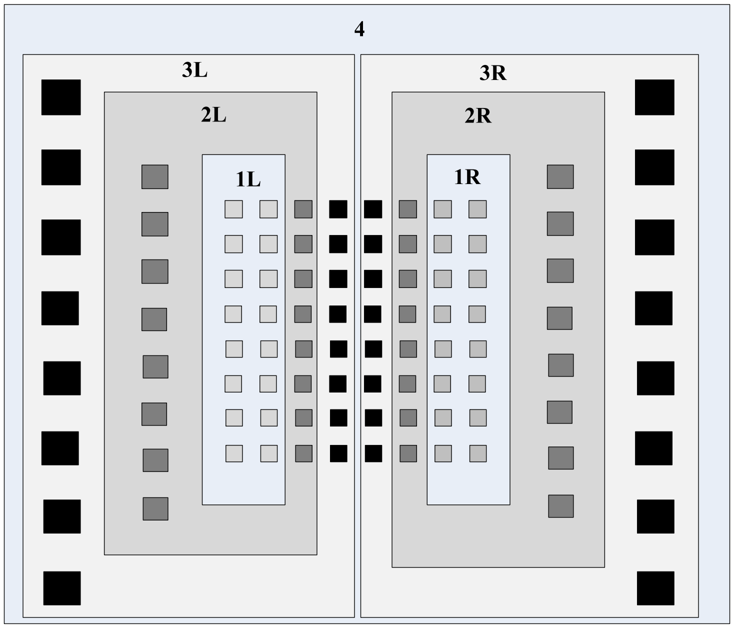


FIGURE B.15

Contraction when $L = 8$ and we have an 8×8 tensor network. The first iteration contracts columns 1 and 2 and columns 7 and 8, see 1L and 1R boxes. During the second iteration the two resulting tensors are contracted with columns 3 and 6, respectively, as shown by 2L and 2R boxes. During the 3-rd iteration the new tensors are contracted with columns 4 and 5, respectively, as shown by the 3L and 3R boxes. Finally, during the 4-th iteration the "big" vector is obtained by contracting the two tensors produced by the 3-rd iteration.

Finally, we compute

$$Z_{\{x_1,x_7\}} = \sum_{x_2} \sum_{x_5} [A_1]_{\{x_1,x_2,x_5\}} \otimes [C]_{\{x_2,x_5,x_7\}}. \tag{B.29}$$

Tensor network contraction is CPU- and memory-intensive. If the tensor network has an arbitrary topology TNC is considerably more intensive than in the case of a regular topology, e.g., a 2-D lattice.

A TNC example. We now discuss the case of an application where the tensors form a 2-D, $L \times L$ rectangular lattice. Each tensor in the interior of the lattice has four indices each one running from 1 to D^2, while outer tensors have only three indices, and the ones at the corners have only two. The resulting tensors form a product of vectors (top and bottom tensors) and matrices (interior tensors) with vertical orbitals running from 1 to D^{2L}. The space required for tensor network contraction can be very large, we expect parameter values as large as $D = 20$ and $L = 100$. This is a Big Data application, 20^{200} is a very large number indeed!!

Figure B.15 illustrates the *generic TLC algorithm* for $L = 8$. The first iteration of the computation contracts the left-most (1L) and the right-most columns (1R) of the tensor network. The process continues until we end up with the "big" vector after $L/2 = 4$ iterations. The left and right contractions, (1L, 2L, and 3L) and (1R, 2R, and 3R) are mirror images of one another and are carried out concurrently.

A TNC algorithm for condensed matter physics. In quantum mechanics vectors in an n-dimensional Hilbert space describe quantum states and tensors describe transformation of quantum states. Tensor network contraction has applications in condensed-matter physics and our discussion is focused only on the algorithmic aspects of the problem.

The algorithm for tensor network contraction should be flexible, efficient, and cost effective. Flexibility means the ability to run problems of different sizes, with a range of values for D and L parameters. An efficient algorithm should support effective multithreading and optimal use of available system resources.

The notations used to describe the contraction algorithms for tensor network T are:

- $N^{(i)}$ – number of vCPUs for iteration i; $N = 2$ for all iterations of Stage 1, while the number of vCPUs for Stage 2 may increase with the number of iterations;
- m – the amount of memory available on the vCPU of the current instance;
- $\mathcal{T}^{(i)} = [\mathcal{T}^{(i)}_{j,k}]$ – version of the $[T]$ after the i-th iteration;
- $L^{(i)}$ – the number of columns of $[T]$ at iteration i;
- $\mathcal{T}^{(i)}_{j,k}$ – tensor in row j and column k of $\mathcal{T}^{(i)}$; $T^{1}_{j,k} = T_{j,k}$;
- $\mathcal{T}^{(i)}_{k}$ – column k of $\mathcal{T}^{(i)}$;
- $\mathbb{C}(\mathcal{T}^{(i)}_{k}, T_j),\ i > 1$ – contraction operator applied to columns $\mathcal{T}^{(i)}_{k}$ and T_j in Stage 1;
- $\mathbb{V}(\mathcal{T}^{(L_{col})})$ – vertical contraction operator applied to the "big tensor" obtained after L_{col} column contractions;
- μ – amount of memory for $T_{j,k}$, a tensor of the original T;
- $\mu^{(i)}$ – storage for a tensor $\mathcal{T}^{(i)}_{j,k}$ created at iteration i;
- I_{max} – maximum number of iterations for Stage 1 of the TNC algorithm.

The *generic contraction algorithm* for a 2-D tensor network with L_{row} rows and L_{col} columns, $T = [T_{j,k}],\ 1 \le j \le 2L_{row}\ 1 \le k \le 2L_{col}$ is an extension of the one in Figure B.15. TNC is an iterative process, at each iteration two pairs of columns are contracted concurrently. During the first iteration the two pairs of columns of $\mathcal{T}$, $(1,2)$, and $(2L, 2L-1)$ are contracted. At iterations $2 \le i \le L$ the new tensor network has $L^{(i)} = L - 2i$ columns and the contraction is applied to column pairs: the column resulting from the contractions at iteration $(i-1)$, now columns 1 and $L^{(i)}$ with columns 2 and $L^{(i)} - 1$, respectively.

The TNC algorithm is organized in multiple stages with different AWS instances for different stages. Small to medium size problems need only the first stage to produce the results and use low end instances with 2 vCPUs, while large problems must feed the results of the first stage to a second one running on more powerful AWS instances. A third stage may be required for extreme cases when the size of one tensor exceeds the amount of vCPU memory, some 4 GB at this time. The three stages of the algorithm are discussed next.

- *Stage 1.* An entire column of $\mathcal{T}^{(i)}$ can be stored in the vCPU memory and successive contraction iterations can proceed seamlessly when

$$L_{row}\left(2\mu + \mu^{(i-1)} + \mu^{(i)}\right) < m. \tag{B.30}$$

  This is feasible for the first iterations of the algorithm and for relatively small values of L_{row}. Call I_{max} the largest value of i which satisfies Equation (B.30).
  Use a low-end M4 or C4 instance with 2 vCPUs, $N = 2$. The computation runs very fast with optimal use of the secondary storage and network bandwidth. Each vCPU is multithreaded, multiple threads carry out the operations required by the contraction operator $\mathbb{C}$ while one thread reads the next column of the original tensor network, $\mathcal{T}$ in preparation for the next iteration.
- *Stage 2.* After a number of iterations the condition in Equation (B.30) is no longer satisfied and the second phase should start. Now an iteration consists of partial contractions when subsets of column tensors are contracted independently. In this case the number of vCPUs is $N > 2$.
- *Stage 3.* As the amount of space needed for a single tensor increases and the vCPU memory cannot store a single tensor

$$\mu_i > m. \tag{B.31}$$

  In this extreme case we use several instances with the largest number of vCPUs, e.g., either M4.10xlarge or C4.10xlarge.

Stage 1 TNC algorithm. The algorithm is a straightforward implementation of the generic TNC algorithm:

1. Start an instance with $N = 2$, e.g., C4.large;
2. Read input parameters e.g., L_{row}, L_{col};
3. Compute I_{max};
4. First iteration
   (a) vCP1 – read T_1 and T_2, apply $\mathbb{C}(T_1, T_2)$; start reading T_3;
   (b) vCP2 – read $T_{L_{col}}$ and $T_{L_{col}-1}$, apply $\mathbb{C}(T_{L_{col}}, T_{L_{col}-1})$, start reading $T_{L_{col}-2}$;
5. Iterations $2 \leq i \leq \min[I_{max}, L_{col}]$. The column numbers correspond to the contracted tensor network with $L_{col}^{(i)} = L_{col} - 2(i-1)$ columns
   (a) vCP1 – apply $\mathbb{C}(\mathcal{T}_1^{(i)}, T_2)$; start reading T_3;
   (b) vCP2 – apply $\mathbb{C}(\mathcal{T}_{L_{col}^{(i)}}^{(i)}, T_{L_{col}^{(i)}-1})$; start reading $T_{L_{col}^{(i)}-2}$;
6. If $L_{col} \leq I_{max}$ carry out vertical compression of the "big tensor" and finish;
   (a) Apply $\mathbb{V}(T^{(L_{col}})$;
   (b) Write result;
   (c) Kill the instance;
7. Else prepare the data for the Stage 2 algorithm;
   (a) vCPU1 – save $\mathcal{T}_i^{(i)}$;
   (b) vCPU2 – save $\mathcal{T}_{L_{col}-i}^{(i)}$;
   (c) Kill the instance.

Stage 2 TNC algorithm. This stage starts with a tensor network $\mathcal{T}^{(I_{max})}$ with $2(L_{col} - I_{max})$ columns and L_{row} rows. Multiple partial contractions will be done for each column of $\mathcal{T}^{(I_{max})}$ during this stage.

The number of vCPUs for the instance used for successive iterations may increase. Results of a partial iteration have to be saved at the end of the partial iteration. The parameters for this phase are:

- $\mu_{pc}^{(i)}$ – the space per tensor required for partial contraction at iteration i
$$\mu_{pc}^{(i)} = \mu + \mu^{(i-1)} + \mu^{(i)}, \tag{B.32}$$
partial contraction increases the space required by each tensor;
- $\mathbb{C}_{pc}\left(\mathcal{T}_k^{(i)}, T_j, s\right)$ – partial contraction operator applied to segment s of columns $\mathcal{T}_k^{(i)}$ and T_j in Stage 2;
- $n_r^{(i)}$ – number of rows of a column segment for each partial contraction at iteration i given by
$$n_r^{(i)} = \left\lceil \frac{m}{\mu_{pc}^{(i)}} \right\rceil. \tag{B.33}$$
- $p^{(i)}$ – number of partial contractions per column at iteration i;
$$p^{(i)} = \left\lceil \frac{L_{row}}{n_r^{(i)}} \right\rceil. \tag{B.34}$$
The total number of partial contractions at iteration i is $2p^{(i)}$;
- The number of vCPUs for iteration i is
$$N^{(i)} = 2p^{(i)}. \tag{B.35}$$
- $L_{col}^{(i)}$ – the number of columns at iteration i of Stage 2;
- $I_{Max} = L_{col} - I_{max}$ – the number of iterations of Stage 2 assuming that Stage 3 is not necessary;
- $\mathbb{A}_{pc}\left(\mathcal{T}^i_{k,p^{(i)}}\right)$ – assembly operator for the $p^{(i)}$ segments resulting from partial contraction of column k at iteration i.

Stage 2 TNC consists of the following steps:

```
1. For $i = 1, I_{Max}$
   (a) Compute $\mu_{pc}, n_r^{(i)}, p^{(i)}, N^{(i)}$;
   (b) If $N \leq 40$ start an instance with $N = N^{(i)}$; else start multiple C4.10xlarge instances to run
       concurrently all partial contractions.
   (c) For $j = 1, p^{(i)}$
       i. $vCPU_j$
          • Read $\mathcal{T}_{1,j}^{(i)}$ and $T_{2,j}$ and apply $\mathbb{C}_{pc}\left(\mathcal{T}_{1,j}^{(i)}, T_{2,j}\right)$;
          • Store the result $\mathcal{T}_{1,j}^{(i+1)}$
```

ii. $vCPU_{j+p^{(i)}}$

- Read $\mathcal{T}^{(i)}_{L^{(i)}_{col},j}$ and $T_{L^{(i)}_{col}-1,j}$ and apply $\mathbb{C}_{pc}\left(\mathcal{T}^{(i)}_{L^{(i)}_{col},j}, T_{L^{(i)}_{col}-1,j}\right)$;
- Store the result, $\mathcal{T}^{(i+1)}_{L^{(i)}_{col},j}$;

(d) Assemble partial contractions

i. $vCPU_1$

- Apply $\mathbb{A}_{pc}\left(\mathcal{T}^{i}_{1}, p^{(i)}\right)$,
- Store $\mathcal{T}^{(i+1)}_{1}$.

ii. $vCPU_2$

- Apply $\mathbb{A}_{pc}\left(\mathcal{T}^{i}_{L^{(i)}_{col}}, p^{(i)}\right)$,
- Store $\mathcal{T}^{(i+1)}_{L^{(i+1)}_{col}}$.

2. If $i < I_{Max}$ proceed to next iteration, $i = i + 1$; else
   (a) Apply $\mathbb{V}\mathcal{T}^{(I_{Max})}$;
   (b) Write TNC result;
   (c) Kill the instance.

Stage 3 TNC algorithm. The algorithm is similar with the one for Stage 2 but now a single tensor is distributed to multiple vCPUs.

An analysis of memory requirements for TNC. Let us assume that we have L tensors per column and each tensor has dimension D. Consider the leftmost, or equivalently the rightmost column, and note that the number of bonds differs for different tensors, the top and the bottom tensors have 2 bonds and the other $L-1$ have 3 bonds, so the total number of elements in this column is

$$\mathcal{N}^{(0)}_{1} = 2D^2 + (L-2)D^3. \tag{B.36}$$

The top and bottom tensors of the next column have three bonds and the remaining $L-2$ have four bonds thus the total number of elements in the second column is

$$\mathcal{N}^{(0)}_{2} = 2D^3 + (L-2)D^4. \tag{B.37}$$

After contraction the number of elements becomes

$$\mathcal{N}^{(1)}_{1} = 2D^3 + (L-2)D^5. \tag{B.38}$$

Each tensor element requires two double precision floating point numbers thus, the amount of memory needed for the first iteration is

$$\begin{aligned}\mathcal{M}^{(1)} &= 2 \times 8 \times [D^2 + (L-2)D^3 + 2D^3 + (L-2)D^4 + 2D^3 + (L-2)D^5] \\ &= 16 \times [2D^3 + (L-2)D^4 + 2D^2(1+D) + (L-2)D^3(1+D^2)]\end{aligned} \tag{B.39}$$

The amount of memory needed for iterations 2 and 3 are

$$\begin{aligned}\mathcal{M}^{(2)} &= 16 \times [2D^3 + (L-2)D^5 + 2D^3 + (L-2)D^4 + 2D^4 + (L-2)D^7] \\ &= 16 \times [2D^3 + (L-2)D^4 + 2D^3(1+D) + (L-2)D^5(1+D^2)]\end{aligned} \tag{B.40}$$

and

$$\begin{aligned}\mathcal{M}^{(3)} &= 16 \times [2D^4 + (L-2)D^7 + 2D^3 + (L-2)D^4 + 2D^5 + (L-2)D^9] \\ &= 16 \times [2D^3 + (L-2)D^4] + 2D^4(1+D) + (L-2)D^7(1+D^2))\end{aligned} \tag{B.41}$$

It follows that the amount of memory for iteration i is

$$\mathcal{M}^{(i)} = 16 \times [2D^3 + (L-2)D^4 + 2D^{i+1}(1+D) + (L-2)D^{2i+1}(1+D^2] \tag{B.42}$$

When $D = 20$ and $L = 100$ the amount of memory for the first iteration is

$$16 \times [2 \times 20^3 + 98 \times 20^4 + 2 \times 20^2 \times (1+20) + 98 \times (20^3 + 20^5)] = 5,281,548,800\ bytes. \tag{B.43}$$

This example shows why only the most powerful systems with ample resources can be used for TNC. It also shows that an application has to adapt, the best it can, to the packages of resources provided by the CSP, while in an better world an application-centric view should prevail, and the system should assemble and offer precisely the resources needed by an application neither more nor less.

Literature

[1] W. M. P. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros. "Workflow patterns." *Technical Report*, Eindhoven University of Technology, 2000.

[2] D. Abadi et al. "The Beckman report on database research." *Communications of the ACM*, **59**(2):92–99, 2016.

[3] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. "Performance guarantees for web server end-system: a control theoretic approach." *IEEE Transactions Parallel and Distributed Systems*, **13**(1):80–96, 2002.

[4] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads." *Proc. VLDB Endowment*, **2**(1):922–933, 2009.

[5] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D. Beyer, and F. Safai. "Self-adaptive SLA-driven capacity management for Internet services." *Proc. IEEE/IFIP Network Operations and Management Symp.*, pp. 557–568, 2006.

[6] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. "RACS: A case for cloud storage diversity." *Proc. ACM Symp. Cloud Computing, (CD Proc.)*. ISBN: 978-1-4503-0036-0, 2010.

[7] D. Abts. "The Cray XT4 and Seastar 3-D torus interconnect." *Encyclopedia of Parallel Computing, Part 3*, Ed. D. Padua, Springer-Verlag, pp. 470–477, 2011.

[8] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. "Energy proportional datacenter networks." *Proc. ACM IEEE Int. Symp. Comp. Arch*, pp. 338–347, 2010.

[9] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang. "Autonomic management of cloud service centers with availability guarantees." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 220–227, 2010.

[10] S. Agarwal, H. Milner, A. Kleiner, A. Talwarkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. "Knowing when you're wrong: building fast and reliable approximate query processing systems." *Proc. ACM SIGMOD Int. Conf. on Management of Data*. ACM Press, pp. 481–492, 2014.

[11] M. Ahmadian, F. Plohan, Z. Roessler, and D. C. Marinescu. "SecureNoSQL: An approach for secure search of encrypted NoSQL databases in the public cloud." *International Journal of Information Management*, **37**:63–74, 2017.

[12] T. Akidau, A. Balikov, K. Bekiröglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. "MillWheel: fault-tolerant stream processing at Internet scale." *Proc. VLDB Endowment*, **6**(11):1033–1044, 2013.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and Whittle. "The data-flow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." *Proc. VLDB Endowment*, **8**(12):1792–1803, 2015.

[14] R. Albert, H. Jeong, and A.-L. Barabási. "The diameter of the world wide web." *Nature*, **401**:130–131, 1999.

[15] R. Albert, H. Jeong, and A.-L. Barabási. "Error and attack tolerance of complex networks." *Nature*, **406**:378–382, 2000.

[16] R. Albert and A-L. Barabási. "Statistical mechanics of complex networks." *Reviews of Modern Physics*, **72**(1):48–97, 2002.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. "A scalable, commodity data center network architecture." *Proc. ACM SIGCOM Conf. on Data Communication*, pp. 63–74, 2008.

[18] "Amazon web services: Overview of security processes." *http://s3.amazonaws.com*. Accessed August 2016.

[19] "Amazon elastic compute cloud." *http://aws.amazon.com/ec2/*. Accessed August 2016.

[20] "Amazon virtual private cloud." *http://aws.amazon.com/vpc/*. Accessed August 2016.

[21] "Amazon CloudWatch." *http://aws.amazon.com/cloudwatch*. Accessed August 2016.

[22] "Amazon elastic block store (EBS)." *http://aws.amazon.com/ebs/*. Accessed August 2016.

[23] "AWS management console." *http://aws.amazon.com/console/*. Accessed August 2016.

[24] "Amazon elastic compute cloud." *http://aws.amazon.com/ec2/*. Accessed April 2016.

[25] "Amazon Docker." *http://aws.amazon.com/docker*. Accessed August 2016.

[26] G. M. Amdahl. "Validity of the single-processor approach to achieving large-scale computing capabilities." *Proc. Conf. American Federation of Inf. Proc. Soc. Conf*. AFIPS Press, pp. 483–485, 1967.

[27] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren. "An opportunity cost approach for job assignment in a scalable computing cluster." *IEEE Transactions Parallel and Distributed Systems*, **11**(7):760–768, 2000.

[28] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. "Disk-locality in datacenter computing considered irrelevant." *Proc. 13th USENIX Conf. on Hot Topics in Operating Systems*, pp. 12–17, 2011.

[29] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. "Photon: fault-tolerant and scalable joining of continuous data streams." *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 577–588, 2013.

[30] T. Andrei. "Cloud computing challenges and related security issues." *http://www1.cse.wustl.edu/jain/cse571-09/ftp/cloud/index.html*. Accessed August 2015.

[31] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwing, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. "Web Service Agreement Specification." *http://www.gridforum.org/Meetings/GGF11/Documents/draftggfgraap-agreement.pdf*, 2004. Accessed August 2016.

[32] D. P. Anderson. "BOINC: A system for public-resource computing and storage." *Proc. 5th IEEE/ACM Int. Workshop Grid Computing*, pp. 4–10, 2004.

[33] R. Aoun, E. A. Doumith, and M. Gagnaire. "Resource provisioning for enriched services in cloud environment." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 296–303, 2010.

[34] Apache. "Apache capacity scheduler." *https://svn.apache.org/repos/asf/hadoop/common/tags/release-0.19.1/docs/capacity_scheduler.pdf*. Accessed August 2016.

[35] D. Ardagna, M. Trubian, and L. Zhang. "SLA based resource allocation policies in autonomic environments." *J. Parallel Distrib. Comp.*, **67**(3):259–270, 2007.

[36] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. "Energy-aware autonomic resource allocation in multi-tier virtualized environments." *IEEE Transactions Services Computing*, **5**(1):2–19, 2012.

[37] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Paterson, A. Rabkin, I. Stoica, and M. Zaharia. "Above the clouds: a Berkeley view of cloud computing." *Technical Report UCB/EECS-2009-28*, 2009, Also *http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf*. Accessed August 2015.

[38] D. Artz and Y. Gil. "A survey of trust in computer science and the Semantic Web." *J. Web Semantics: Science, Services, and Agents on the World Wide Web*, **1**(2):58–71, 2007.

[39] M. J. Atallah, C. Lock Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. "Models and algorithms for co-scheduling compute-intensive tasks on a network of workstations." *Journal of Parallel and Distributed Computing*, **16**:319–327, 1992.

[40] M. Auty, S. Creese, M. Goldsmith, and P. Hopkins. "Inadequacies of current risk controls for the cloud." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 659–666, 2010.

[41] L. Ausubel and P. Cramton. "Auctioning many divisible goods." *J. Euro. Economic Assoc.*, **2**(2–3):480–493, 2004.

[42] L. Ausubel, P. Cramton, and P. Milgrom. "The clock-proxy auction: a practical combinatorial auction design." Chapter 5, *Combinatorial Auctions*, Eds. P. Cramton, Y. Shoham, and R. Steinberg, MIT Press, Cambridge, Mass, 2006.

[43] A. Avisienis, J. C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." *IEEE Transactions Dependable and Secure Computing*, **1**(1):11–33, 2004.

[44] Y. Azar, A. Broder, A. Karlin, and E. Upfal. "Balanced allocations." *Proc. 26th ACM Symp. on the Theory of Computing*, pp. 593–602, 1994.

[45] Ö. Babaoğlu and K. Marzullo. "Consistent global states." *Distributed Systems*, (2nd edition) Ed. Sape Mullender, Addison-Wesley, Reading, MA, pp. 55–96, 1993.

[46] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh. "A Remote-file cache for RFS." *Proc. Summer 1987 USENIX Conf.*, pp. 275–280, 1987.

[47] X. Bai, H. Yu, G. Q. Wang, Y. Ji, G. M. Marinescu, D. C. Marinescu, and L. Böloni. "Coordination in intelligent grid environments." *Proceedings of the IEEE*, **93**(3):613–630, 2005.

[48] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services." *Proc. 5th Biennial Conf. Innovative Data Systems Research*, pp. 223–234, 2011.

[49] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. "A security analysis of Amazon's elastic compute cloud service." *Proc. 27th Annual ACM Symp. Applied Computing*, pp. 1427–1434, 2012.

[50] J. Baliga, R. W. A. Ayre, K. Hinton, and R. S. Tucker. "Green cloud computing: balancing energy in processing, storage, and transport." *Proceedings of the IEEE*, **99**(1):149–167, 2011.

[51] A-L. Barabási and R. Albert. "Emergence of scaling in random networks." *Science*, **286**(5439):509–512, 1999.

[52] A-L. Barabási, R. Albert, and H. Jeong. "Scale-free theory of random networks; the topology of World Wide Web." *Physica A*, **281**:69–77, 2000.

[53] P. Barham. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the art of virtualization." *Proc. 19th ACM Symp. Operating Systems Principles*, pp. 164–177, 2003.

[54] L. A. Barroso, J. Dean, and U. Hözle. “Web search for a planet: the Google cluster architecture.” *IEEE Micro*, **23**(2):22–28, 2003.

[55] L. A. Barroso and U. Hözle. “The case for energy-proportional computing.” *IEEE Computer*, **40**(12):33–37, 2007.

[56] L. A. Barossso, J. Clidaras, and U. Hözle. *The Datacenter as a Computer; an Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool, 2013.

[57] L. Bottou. “Large-scale machine learning with stochastic gradient descent.” *Proc. Int. Conf. on Computational Statistics*. Physica/Springer Verlag, Heidelberg, pp. 177–186, 2010.

[58] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunningberg. “SILK: Scout paths in the Linux kernel.” *Technical Report 2002-009*, Uppsala University, Department of Information Technology, Feb. 2002.

[59] G. Bell. “Massively parallel computers: why not parallel computers for the masses?.” *Proc. 4-th Symp. Frontiers of Massively Parallel Computing*. IEEE, pp. 292–297, 1992.

[60] R. Bell, M. Koren, and C. Volinsky. “The BellKor 2008 Solution to the Netflix Prize.” *Technical report*, AT&T Labs, 2008.

[61] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har’El, A. Gordon, A. Liguori, O. Wasserman, and B-A. Yassour. “The Turtles project: design and implementation of nested virtualization.” *Proc. 9th USENIX Conf. on OS Design and Implementation*, pp. 423–436, 2010.

[62] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow. “Blueprint for the Intercloud – protocols and formats for cloud computing interoperability.” *Proc. 4th Int. Conf. Internet and Web Applications and Services, ICIW ’09*, pp. 328–336, 2009.

[63] D. Bernstein and D. Vij. “Intercloud security considerations.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 537–544, 2010.

[64] D. Bernstein, D. Vij, and S. Diamond. “An Intercloud cloud computing economy – technology, governance, and market blueprints.” *Proc. SRII Global Conference*, pp. 293–299, 2011.

[65] D. Bertsekas and R. Gallagher. *Data Networks*, Second Edition. Prentice Hall, 1992.

[66] S. Bertram, M. Boniface, M. Surridge, N. Briscombe, and M. Hall-May. “On-demand dynamic security for risk-based secure collaboration in clouds.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 518–525, 2010.

[67] S. Bhattacharjee and D. C. Marinescu. “A cloud service for trust management in cognitive radio networks.” *Int. J. Cloud Computing*, **3**(14):326–353, 2014.

[68] M. Blackburn and A. Hawkins. “Unused server survey results analysis.” *www.thegreengrid.org/media/White/Papers/Unused%20Server%20Study_WP_101910_v1.ashx?lang=en*. Accessed August 2016.

[69] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K. Su. “Myrinet – A gigabit-per-second local-area network.” *IEEE Micro*, **5**(1):29–36, 1995.

[70] A. Boldyreva, N. Chenette, Y. Lee, and A. Oneill. “Order-preserving symmetric encryption.” Advances in Cryptology, Springer-Verlag, pp. 224–241, 2009.

[71] B. Bollobás. *Random Graphs*. Academic Press, London, 1985.

[72] K. Boloor, R. Chirkova, Y. Viniotis, and T. Salo. “Dynamic request allocation and scheduling for context aware applications subject to a percentille response time SLA in a distributed cloud.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 464–472, 2010.

[73] I. Brandic, S. Dustdar, T. Ansett, D. Schumm, F. Leymann, and R. Konrad. “Compliant cloud computing (C3): Architecture and language support for user-driven compliance management in clouds.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 244–251, 2010.

[74] S. Brandt, S. Banachowski, C. Lin, and T. Bisson. “Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes.” *Proc. IEEE Real-Time Systems Symp*, pp. 396–409, 2003.

[75] S. Brin and L. Page. “The anatomy of a large-scale hypertextual web search engine.” *Journal Computer Networks and ISDN Systems*, **30**(1–7):107–117, 1998.

[76] N. F. Britton. *Essential Mathematical Biology*. Springer, 2004.

[77] C. Brooks. *Enterprise NoSQL for Dummies*. Wiley, New Jersey, NJ, 2014.

[78] M. Buddhikot and K. Ryan. “Spectrum management in coordinated dynamic spectrum access based cellular networks.” *Proc. IEEE Int. Symp. New Frontiers in Dynamic Spectrum Access Networks*, pp. 299–307, 2005.

[79] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. “HaLoop: efficient iterative data processing on large clusters.” *Proc. VLDB Endowment*, **3**:285–296, 2010.

[80] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. “An evaluation of distributed data stores using the AppScale cloud platform.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 305–312, 2010.

[81] A. W. Burks, H. H. Goldstine, and J. von Neumann. "Preliminary Discussion of the Logical Design of an Electronic Computer Instrument." *Report to the US Army Ordnance Department*, 1946. Also in: *Papers of John von Neumann*, Eds. W. Asprey and A. W. Burks, MIT Press, Cambridge, MA, pp. 97–146, 1987.

[82] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. "Borg, Omega, and Kubernetes; lessons learned from three container-management systems over a decade." *ACM Queue*, **14**(1):70–93, 2016.

[83] M. Burrows. "The Chubby lock service for loosely-coupled distributed systems." *Proc. USENIX Symp. OS Design and Implementation*, pp. 335–350, 2006.

[84] R. Buyya, R. Ranjan, and R. Calheiros. "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services." *Proc. 10th Int. Conf. Algorithms and Architectures for Parallel Processing, Part I*, pp. 13–31, 2010.

[85] C. Cacciari, F. D'Andria, M. Gonzalo, B. Hagemeier, D. Mallmann, J. Martrat, D. G. Perez, A. Rumpl, W. Ziegler, and C. Zsigri. "elasticLM: A novel approach for software licensing in distributed computing infrastructures." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 67–74, 2010.

[86] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. C. Rosu, and M. Steiner. "Highly-scalable searchable symmetric encryption with support for boolean queries." *Advances in Cryptology*. Springer-Verlag, pp. 353–373, 2013.

[87] A. J. Canty, A. C. Davison, D. V. Hinkley, and V. Ventura. "Bootstrap diagnostics and remedies." *The Canadian Journal of Statistics*, **34**(1):5–27, 2006.

[88] A. G. Carlyle, S. L. Harrell, and P. M. Smith. "Cost-effective HPC: The community or the cloud?." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 169–176, 2010.

[89] E. Caron, F. Desprez, and A. Muresan. "Forecasting for grid and cloud computing on-demand resources based on pattern matching." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 456–463, 2010.

[90] R. Cattell. "Scalable SQL and NoSQL data stores." *http://cattell.net/datastores/Datastores.pdf*, 2011. Accessed August 2015.

[91] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. "How to enhance cloud architectures to enable cross-federation." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 337–345, 2010.

[92] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. "FlumeJava: easy, efficient data-parallel pipelines." *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 363–375, 2010.

[93] A. Chandra, P. Goyal, and P. Shenoy. "Quantifying the benefits of resource multiplexing in on-demand data centers." Computer Science Department Faculty Publication Series, vol. 20, 2003. *http://scholarworks.umass.edu/cs_faculty_pubs/20*. Accessed January 2017.

[94] T. D. Chandra, R. Griesemer, and J. Redstone. "Paxos made live: an engineering perspecitce." *Proc. 26th ACM Symp. Principles of Distributed Computing*, pp. 398–407, 2007.

[95] K. M. Chandy and L. Lamport. "Distributed snapshots: determining global states of distributed systems." *ACM Transactions on Computer Systems*, **3**(1):63–75, 1985.

[96] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: a distributed storage system for structured data." *Proc. USENIX Symposium on OS Design and Implementation*, pp. 205–218, 2006.

[97] V. Chang, G. Wills, and D. De Roure. "A review of cloud business models and sustainability." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 43–50, 2010.

[98] F. Chang, J. Ren, and R. Viswanathan. "Optimal resource allocation in clouds." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 418–425, 2010.

[99] L. Chang, Z. Wang, T. Ma, L. Jian, A. Golgshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar. "HAWQ: a massively parallel processing SQL engine in Hadoop." *Proc. ACM SIGMOD Int. Conf. Management of Data*, pp. 1223–1234, 2014.

[100] K. Chard, S. Caton, O. Rana, and K. Bubendorfer. "Social cloud: Cloud computing in social networks." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 99–106, 2010.

[101] A. Chazalet. "Service level checking in the cloud computing context." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 297–304, 2010.

[102] P. M. Chen and B. D. Noble. "When virtual is better than real." *Proc. 8-th Workshop Hot Topics in Operating Systems*, pp. 133–141, 2001.

[103] H. Chen, P. Liu, R. Chen, and B. Zang. "When OO meets system software: Rethinking the design of VMMs." *Fudan University, Parallel Processing Institute, Technical Report PPITR-2007-08003*, 2007, pp. 1–9. Also, *http://ppi.fudan.edu.cn/system/publications/paper/OVM-ppi-tr.pdf*, 2007. Accessed August 2015.

[104] T. M. Chen and S. Abu-Nimeh. "Lessons from Stuxnet." *Computer*, **44**(4):91–93, 2011.

[105] R. Chen, J-M. Park, and K. Bian. "Robust distributed spectrum sensing in cognitive radio networks." *Proc. IEEE Conf. Computer Communications*, pp. 1876–1884, 2008.

[106] Y. Chen, S. Alspaugh, and R. Katz. "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads." *Proc. VLDB Endowment*, **5**(12):1802–1813, 2012.

[107] J. Cheney, L. Chiticariu, and W-C. Tan. "Provenance in databases: why, how, and where." *Foundations and Trends in Databases*, **1**(4):379–474, 2007.

[108] D. Chiu and G. Agarwal. "Evaluating cache and storage options on the Amazon Web services cloud." *Proc. IEEE/ACM Int. Symp. Cluster, Cloud and Grid Computing*, pp. 362–371, 2011.

[109] R. Chow, P. Golle, M. Jackobsson, E. Shi, J. Staddon, R. Masouka, and J. Mollina. "Controlling data on the cloud: outsourcing computations without outsourcing control." *Proc. ACM Cloud Computing Security Workshop*, pp. 85–90, 2009.

[110] B. Clark, T. Deshane, E. Dow, S. Evabchik, M. Finlayson, J. Herne, and J. N. Matthews. "Xen and the art of repeated research." *Proc. USENIX Annual Technical Conference*, pp. 135–144, 2004. Also, *http://web2.clarkson.edu/class/cs644/xen/files/repeatedxen-usenix04.pdf*. Accessed August 2015.

[111] R. A. Clarke and R. K. Knake. *Cyber War: The Next Threat to National Security and What to Do About It*. Harper Collins, 2012.

[112] C. Clos. "A Study of non-blocking switching networks." *Bell System Technical Journal*, **32**(2):406–425, 1953.

[113] E. F. Codd. "A Relational model of data for large shared data banks." *Communications of the ACM*, **13**(6):377–387, 1970.

[114] E. Coffman, M. J. Elphick, and S. Shoshani. "System deadlocks." *Computing Surveys*, **3**(2):67–78, 1971.

[115] L. Colitti, S. H. Gunderson, E. Kline, and T. Refice. "Evaluating IPv6 adoption in the Internet." *Proc. Passive and Active Measurement Conf.*, Lecture Notes on Computer Science, vol. 6032, Springer-Verlag, Berlin, pp. 141–150, 2010.

[116] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. "Breaking up is hard to do: security and functionality in a commodity hypervisor." *Proc. 23rd ACM Symp. Operating Systems Principles*, pp. 189–202, 2011.

[117] F. J. Corbatò and V. A. Vyssotsky. "Introduction and overview of the Multix system." *Proc. AFIPS, Fall Joint Computer Conf.*, pp. 185–196, 1965.

[118] F. J. Corbatò. "On building systems that will fail." *Turing Award Lecture 1991*, *http://larch-www.lcs.mit.edu:8001/~corbato/turing91/*, 2011.

[119] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's globally-distributed database." *Proc. USENIX Symp. on Operating Systems Design and Implementation*, **31**(3), 2012, paper #8.

[120] *Combinatorial Auctions*, Eds. P. Cramton, Y. Shoham, and R. Steinberg, MIT Press, 2006.

[121] F. Cristian, H. Aghili, R. Strong, and D. Dolev. "Atomic broadcast from simple message diffusion to Byzantine agreement." *Proc. 15th Int. Symp. Fault Tolerant Computing*. IEEE Press, pp. 200–206, 1985. Also *Information and Computation*, **118**(1):158–179, 1995.

[122] Cloud Security Alliance. "Security guidance for critical areas of focus in cloud computing V2.1." *https://cloudsecurityalliance.org/csaguide.pdf*, 2009.

[123] Cloud Security Alliance. "Top threats to cloud computing V1.0." *https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf*, 2010. Accessed August 2015.

[124] Cloud Security Alliance. "Security guidance for critical areas of focus in cloud computing V3.0." *https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf*, 2011. Accessed August 2015.

[125] CUDA. "CUDA 7.0 performance report." *http://on-demand.gputechconf.com/gtc/2015/webinar/gtc-express-cuda7-performance-overview.pdf*, 2015. Accessed April 2016.

[126] E. Cuervo, A. Balasubramanian, D-K Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. "MAUI: making smartphones last longer with code offload." *Proc. 8th Int. Conf. on Mobile Systems, Applications, and Services*, pp. 49–62, 2010.

[127] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. "Reservation-based scheduling: if you're late don't blame us!." *Proc. ACM Symp. Cloud Computing*, pp. 1–14, 2014.

[128] B. Das, Y. Z. Zhang, and J. Kiszka. "Nested virtualization; state of the art and future directions." *KVM Forum*, 2014, Also *http://www.linux-kvm.org/images/3/33/02x03-NestedVirtualization.pdf*. Accessed January 2017.

[129] H. A. David. *Order Statistics*. Wiley, New York, NY, 1981.

[130] J. Dean and S. Ghernawat. "MapReduce: Simplified Data Processing on Large Clusters." *Proc. USENIX 6-th Symp. OS Design and Implementation*, pp. 137–149, 2004.
[131] J. Dean and L. A. Barroso. "The tail at scale." *Communications of the ACM*, **56**(2):74–80, 2013.
[132] J. Dean. "The rise of cloud computing systems." *Proc. ACM Symp. OS Principles*, 2015, article no. 12 (also http://sigops.org/sosp/sosp15/history/10-dean-slides.pdf). Accessed August 2016.
[133] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. "Anti-caching: a new approach to database management system architecture." *Proc. VLDB Endowment*, **6**:1942–1953, 2013.
[134] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's highly available key-value store." *Proc. 21st ACM/SIGOPS Symp. OS Principles*, pp. 205–220, 2007.
[135] C. Delimitrou and C. Kozyrakis. "QoSAware scheduling for heterogeneous data centers with Paragon." *ACM Transactions on Computer Systems*, **31**(4):12–24, 2013.
[136] C. Delimitrou and C. Kozyrakis. "The Netflix challenge: datacenter edition." *IEEE Computer Architecture Letters*, **12**(1):29–32, 2013.
[137] C. Delimitrou and C. Kozyrakis. "Quasar: Resource-efficient and QoS-aware cluster management." *Proc. ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 127–144, 2014.
[138] Y. Demchenko, C. de Laat, and D. R. Lopes. "Security services lifecycle management in on-demand infrastructure services provisioning." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 644–650, 2010.
[139] A. Demers, S. Keshav, and S. Shenker. "Analysis and simulation of a fair queuing algorithm." *Proc. ACM SIGCOMM'89 Symp. Communications Architectures and Protocols*, pp. 1–12, 1989.
[140] J. B. Dennis. "General parallel computation can be performed with a cycle-free heap." *Proc. 1998 Int. Conf. on Parallel Architectures and Compiling Techniques*. IEEE, pp. 96–103, 1998.
[141] J. B. Dennis. "Fresh Breeze: a multiprocessor chip architecture guided by modular programming principles." *ACM SIGARCH Computer Architecture News*, **31**(1):7–15, 2003.
[142] J. B. Dennis. "The fresh breeze model of thread execution." *Proc. Workshop on Programming Models for Ubiquitous Parallelism*, 2006. Also *http://csg.csail.mit.edu/Users/dennis/pmup-final.pdf*. Accessed January 2017.
[143] D. Denisson. "Continuous pipelines at Google." *https://www.usenix.org/sites/default/files/continuouspipelinesatgoogle-final.pdf*, 2015. Accessed May 2016.
[144] M. Devarakonda, B. Kish, and A. Mohindra. "Recovery in the Calypso file system." *ACM Transactions Computer Systems*, **14**(3):287–310, 1996.
[145] D. J. DeWitt, R. V. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. "Split query processing in polybase." *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 1255–1266, 2013.
[146] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali. "Cloud computing: distributed internet computing for IT and scientific research." *IEEE Internet Computing*, **13**(5):10–13, 2009.
[147] E. W. Dijkstra. "Cooperating sequential processes." *Programming Languages*, Ed. F. Genuys, Academic Press, New York, pp. 43–112, 1968, Originally appeared in 1965, E.W. Dijkstra Archive: Cooperating sequential processes (EWD 123) *https://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html*.
[148] E. W. Dijkstra. "Self-stabilizing systems in spite of distributed control." *Communications of the ACM*, **17**(11):643–644, 1974.
[149] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. "A survey of mobile cloud computing: architecture, applications, and approaches." *Wireless Communication and Mobile Computing*, 2011. http://dx.doi.org/10.1002/wcm.1203.
[150] DONA. "Data Oriented Network Architecture." *https://www2.eecs.berkeley.edu/Research/Projects/Data/102146.html*. Accessed December 2016.
[151] P. Donnelly, P. Bui, and D. Thain. "Attaching cloud storage to a campus grid using Parrot, Chirp, and Hadoop." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 488–495, 2010.
[152] T. Dörnemann, E. Juhnke, T. Noll, D. Seiler, and B. Freieleben. "Data flow driven scheduling of BPEL workflows using cloud resources." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 196–203, 2010.
[153] S. Drossopoulou, J. Noble, M. S. Miller, and T. Murray. "Reasoning about risk and trust in an open word." *http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44272.pdf*, 2015. Accessed August 2016.
[154] L. Ducas and D. Micciancio. "FHEW: bootstrapping homomorphic encryption in less than a second." *Advances in Cryptology–EUROCRYPT 2015*. Springer-Verlag, pp. 617–640, 2015.
[155] K. J. Duda and R. R. Cheriton. "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler." *Proc. 17th ACM Symp. OS Principles*, pp. 261–276, 1999.

[156] N. Dukkipati, T. Refice, Y.-C. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. "An argument for increasing TCP's initial congestion window." *ACM SIGCOMM Computer Communication Review*, **40**(3):27–33, 2010.

[157] X. Dutreild, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. "From data center resource allocation to control theory and back." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 410–417, 2010.

[158] G. Dyson. *Turing's Cathedral: The Origins of the Digital Universe*. Pantheon, 2012.

[159] D. L. Eager, E. D. Lazokwska, and J. Zahorjan. "Adaptive load sharing in homogeneous distributed systems." *IEEE Transactions on Software Engineering*, **12**:662–675, 1986.

[160] D. Ebneter, S. G. Grivas, T. U. Kumar, and H. Wache. "Enterprise architecture frameworks for enabling cloud computing." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 542–543, 2010.

[161] V. M. Eguiluz and K. Klemm. "Epidemic threshold in structured scale-free networks." *arXiv:cond-mat/0205439v1*, 2002.

[162] J. Ejarque, R. Sirvent, and R. M. Badia. "A multi-agent approach for semantic resource allocation." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 335–342, 2010.

[163] M. Elhawary and Z. J. Haas. "Energy-efficient protocol for cooperative networks." *IEEE/ACM Transactions Networking*, **19**(2):561–574, 2011.

[164] Enterprise Management Associates. "How to make the most of cloud computing without sacrificing control." White paper, prepared for IBM, pp. 18, September 2010. *http://www.siliconrepublic.com/reports/partner/26-ibm/report/311-how-to-make-the-most-of-clo/*. Accessed August 2015.

[165] P. Erdös and A. Rényi. "On random graphs." *Publicationes Mathematicae*, **6**:290–297, 1959.

[166] R. M. Esteves and C. Rong. "Social impact of privacy in cloud computing." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 593–596, 2010.

[167] C. Evanghelinos and C. N. Hill. "Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2." *Workshop on Cloud Computing and Its Applications*, 2008. *http://cisc.gmu.edu/scc/readings/CCPShpcA.pdf*. Accessed January 2017.

[168] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. "Rich queries on encrypted data: beyond exact matches." *Proc. 20th Euro Symp. Research in Computer Security*, Lecture Notes on Computer Science, vol. 9327, Springer-Verlag, Berlin, pp. 123–145, 2015.

[169] A. D. H. Farwell, M. J. Sergot, M. Salle, C. Bartolini, D. Tresour, and A. Christodoulou. "Performance monitoring of service-level agreements for utility computing." *Proc. IEEE. Int. Workshop Electronic Contracting*, pp. 17–24, 2004.

[170] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki. "Clearing the clouds." *Proc. 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, pp. 37–48, 2012.

[171] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini. "QoS-aware clouds." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 321–328, 2010.

[172] Federal Trade Comission. "Privacy online: for information practice in the electronic marketplace." *https://www.ftc.gov/reports/privacy-online-fair-information-practices-electronic-marketplace-federal-trade-commission*, 2000. Accessed August 2015.

[173] A. Fikes. "Storage architecture and challenges." *https://www.systutorials.com/3306/storage-architecture-and-challenges/*. Accessed March 2017.

[174] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of distributed consensus with one faulty process." *J. of the ACM*, **32**(2):374–382, 1985.

[175] A. Floratou, U. F. Minhas, and F. Ožcan. "SQLonHadoop: Full circle back to shared-Nothing database architectures." *Proc. VLDB Endowment*, **7**(12):1295–1306, 2014.

[176] S. Floyd and V. Jacobson. "Link-sharing and resource management models for packet networks." *IEEE/ACM Transactions Networking*, **3**(4):365–386, 1995.

[177] B. Ford. "Icebergs in the clouds the other risks of cloud computing." *Proc. 4th USENIX Workshop Hot Topics in Cloud Computing*, 2012. *arXiv:1203.1979v2*.

[178] M. Franklin, A. Halevy, and D. Maier. "From databases to dataspaces: A new abstraction for information management." *SIGMOD Record*, **34**(4):27–33, 2005.

[179] J. Franklin, K. Bowler, C. Brown, S. Edwards, N. McNab, and M. Steele. "Mobile device security – cloud and hybrid builds." *NIST Special Publication 1800-4b*, 2015. Also, *https://nccoe.nist.gov/publication/draft/1800-4b*.

[180] E. Gafni and D. Bertsekas. "Dynamic control of session input rates in communication networks." *IEEE Transactions Automatic Control*, **29**(10):1009–1016, 1984.

[181] G. Ganesan and Y. G. Li. "Cooperative spectrum sensing in cognitive radio networks." *Proc. IEEE Symp. New Frontiers in Dynamic Spectrum Access Networks*, pp. 137–143, 2005.
[182] A. G. Ganek and T. A. Corbi. "The dawning of the autonomic computing era." *IBM Systems Journal*, **42**(1):5–18, 2003.
[183] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.
[184] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. "Terra: a virtual machine-based platform for trusted computing." *Proc. ACM Symp. Operating Systems Principles*, pp. 193–206, 2003.
[185] S. Garfinkel and M. Rosenblum. "When virtual is harder than real: security challenges in virtual machines based computing environments." *Proc. 10th Conf. Hot Topics in Operating Systems*, pp. 20–25, 2005.
[186] S. Garfinkel. "An evaluation of Amazon's grid computing services: EC2, S3, and SQS." *Technical Report, TR-08-07*, Harvard University, 2007.
[187] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. "Building a high-level dataflow system on top of MapReduce: the Pig experience." *Proc. VLDB Endowment*, **2**(2):1414–1425, 2009.
[188] M. Gell-Mann. "Simplicity and complexity in the description of nature." *Engineering and Sciences*, **51**(3):2–9, 1988, California Institute of Technology, *http://resolver.caltech.edu/CaltechES:51.3.Mann*.
[189] C. Gentry. "A fully homomorphic encryption scheme." *Ph.D. Thesis*, Stanford University, 2009.
[190] C. Gentry. "Fully homomorphic encryption using ideal lattices." *Proc. 41st ACM Symp. Theory of Computing*, pp. 169–178, 2009.
[191] M. Gerla. "Vehicular cloud computing." *Proc. 1st Int. Workshop Vehicular Communication and Applications*. IEEE, pp. 152–155, 2012, Published in *11th Med. Ad Hoc Networking Workshop*.
[192] C. Gkantsidis, M. Mihail, and A. Saberi. "Random walks in peer-to-peer networks." *Performance Evaluation*, **63**(3):241–263, 2006.
[193] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system." *Proc. 19th ACM Symp. OS Principles (SOSP'03)*, pp. 15–25, 2003.
[194] D. Gmach, J. Rolia, and L. Cerkasova. "Satisfying service-level objectives in a self-managed resource pool." *Proc. 3rd. Int. Conf. Self-Adaptive and Self-Organizing Systems*, pp. 243–253, 2009.
[195] R. P. Goldberg. "Architectural principles for virtual computer systems." *Thesis*, Harvard University, 1973.
[196] R. P. Goldberg. "Survey of virtual machine research." *IEEE Computer Magazine*, **7**(6):34–45, 1974.
[197] G. Gonnet. "Expected length of the longest probe sequence in hash code searching." *J. of the ACM*, **28**(2):289–304, 1981.
[198] Google. "Google cloud platform." *https://cloud.google.com/*. Accessed August 2016.
[199] Google Docker. *https://cloud.google.com/container-engine*. Accessed May 2015.
[200] P. Goyal, X. Guo, and H. M. Vin. "A hierarchial CPU scheduler for multimedia operating systems." *Proc. USENIX Symp. OS Design and Implementation*, pp. 107–121, 1996.
[201] J. Gray. "The transaction concept: virtues and limitations." *Proc. 7 Int. Conf. Very Large Databases*, pp. 144–154, 1981.
[202] J. Gray and D. Patterson. "A conversation with Jim Gray." *ACM Queue*, **1**(4):8–17, 2003.
[203] G. Graefe, H. Volos, H. Kimura, H. Kuno, and J. Tucek. "In memory performance for Big Data." *Proc. VLDB Endowment*, **8**(1):37–48, 2014.
[204] T. G. Griffn, F. B. Shepherd, and G. Wilfong. "The stable paths problem and interdomain routing." *IEEE/ACM Transactions Networking*, **10**(2):232–243, 2002.
[205] M. Gritter and D. R. Cheriton. "An architecture for content routing support in the Internet." *Proc. USENIX Symp. on Internet Technologies and Systems*, **3**:4, 2001.
[206] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, 1994.
[207] N. Gruschka and M. Jensen. "Attack surfaces: a taxonomy for attacks on cloud services." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 276–279, 2010.
[208] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. "MapReduce in the clouds for science." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 565–572, 2010.
[209] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. "Nectar: Automatic management of data and computation in data centers." *Proc. USENIX Symp. OS Design and Implementation*, pp. 75–88, 2010.
[210] I. Gupta, A. J. Ganesh, and A-M. Kermarrec. "Efficient and adaptive epidemic-style protocols for reliable and scalable multicast." *IEEE Transactions Parallel and Distributed Systems.*, **17**(7):593–605, 2006.
[211] V. Gupta and M. Harchol-Balter. "Self-adaptive admission control policies for resource-sharing systems." *Proc. 11th Int. Joint Conf. Measurement and Modeling Computer Systems*, pp. 311–322, 2009.

[212] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. "Mesa: Geo-replicated, near real-time, scalable data warehousing." *Proc. VLDB Endowment*, **7**(12):1259–1270, 2014.

[213] A. Gupta and J. Shute. "High-availability at massive scale: building Google's data infrastructure for ads." *Proc. 9th Workshop Business Intelligence for the Real Time Enterprise*. Springer-Verlag, pp. 81–89, 2015.

[214] J. O. Gutierrez-Garcia and K.-M. Sim. "Self-organizing agents for service composition in cloud computing." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 59–66, 2010.

[215] P. J. Haas. "Hoeffding inequalities for join-selectivity estimation and online aggregation." *IBM Research Report RJ 10040 (90568)*, IBM Almaden Research, 1996.

[216] T. Haig, M. Pristley, and C. Rope. "Los Alamos bets on ENIAC: nuclear Monte Carlo simulations, 1947–1948." *IEEE Annals of the History of Computing*, **36**(3):42–63, 2014.

[217] F. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. "Lest we remember: cold boot attacks on encryption keys." *Proc. 17th Usenix, Security Symposium*, pp. 45–60, 2008.

[218] S. Halevi and V. Shoup. "Algorithms in HElib." *Advances in Cryptology*, Lecture Notes on Computer Science, Vols. 8616 and 8617, Eds. J. A. Garay and R. Gennaro, Springer-Verlag, pp. 554–571, 2014.

[219] J. D. Halley and D. A. Winkler. "Classification of emergence and its relation to self-organization." *Complexity*, **13**(5):10–15, 2008.

[220] J. Hamilton. "The stunning scale of AWS and what it means for the future of the cloud." *http://highscalability.com/blog/2015/1/12/the-stunning-scale-of-aws-and-what-it-means-for-the-future-o.html*, 2014. Accessed May 2016.

[221] P. B. Hansen. "The evolution of operating systems." *Classic Operating Systems: From Batch Processing To Distributed Systems*, Ed. P. B. Hansen, Springer-Verlag, pp. 1–36, 2001.

[222] R. F. Hartl, S. P. Sethi, and R. G. Vickson. "Survey of the maximum principles for optimal control problems with state constraints." *SIAM Review*, **37**(2):181–218, 1995.

[223] T. Härder and A. Reuter. "Principles of transaction-oriented database recovery." *ACM Computing Surveys*, **15**(4):287–317, 1983.

[224] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebreaker. "OLTP through the looking glass, and what we found there." *Proc. ACM/SIGMOD Int. Conf. on Management of Data*, pp. 981–992, 2008.

[225] K. Hasebe, T. Niwa, A. Sugiki, and K. Kato. "Power-saving in large-scale storage systems with data migration." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 266–273, 2010.

[226] R. L. Haskin. "Tiger Shark – a scalable file system for multimedia." *IBM J. of Research and Development*, **42**(2):185–197, 1998.

[227] J. L. Hellerstein. "Why feedback implementations fail: the importance of systematic testing." *5th Int. Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*. *http://eurosys2010-dev.sigops-france.fr/workshops/FeBID2010/hellerstein.pdf*, 2015. Accessed August 2016.

[228] J. L. Hennessy and D. A. Patterson. *Computer Architecture – A quantitative Approach*, Fifth Edition. Morgan Kaufmann, Waltham, MA, 2012.

[229] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*, Revised Reprint. Morgan Kaufmann, Waltham, MA, 2012.

[230] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. "Starfish: A self-tuning system for big data analytics." *Proc. 5th Conf. on Innovative Data Systems Research*, pp. 261–272, 2011.

[231] T. Hey, S. Tansley, and K. Tolle. "Jim Gray on eScience: a transformed scientific method." *The fourth paradigm. Data-intensive scientific discovery*. Microsoft Research, 2009. Also, *http://research.microsoft.com/en-us/collaboration/fourthparadigm/4th_paradigm_book_complete_lr.pdf*. Accessed August 2015.

[232] H. Scalability. "The stunning scale of AWS and what it means for the future of the cloud." *http://highscalability.com/blog/2015/1/12/the-stunning-scale-of-aws-and-what-it-means-for-the-future-o.html*. Accessed May 2016.

[233] M. Hilbert and P. López. "The worlds's technological capacity to store, communicate, and compute information." *Science*, **332**(6025):60–65, 2011.

[234] M. Hill, S. Eggers, J. R. Larus, G. Taylor, G. Adams, B. K. Bose, G. Gibson, P. Hansen, J. Keller, S. Kong, C. Lee, D. Lee, J. Pendlton, S. Richie, D. Wood, B. Zorn, P. Hilfinger, D. Hodges, R. Katz, J. K. Osterhout, and J. Petterson. "Design decisions in SPUR." *Computer*, **9**(11):8–22, 1986.

[235] M. D. Hill and M. R. Marty. "Amdahl's Law in the multicore era." *IEEE Computer*, **41**(7):33–38, 2008.

[236] M. Hinchey, R. Sterritt, C. Rouff, J. Rash, and W. Truszkowski. "Swarm-based space exploration." *ERCIM News*, **64**, 2006.

[237] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. "Mesos: A platform for fine-grained resource sharing in the data center." *Proc. 8th USENIX Symp. Networked Systems Design and Implementation*, pp. 295–308, 2011.

[238] C. A. R. Hoare. "Communicating sequential processes." *Comm. of the ACM*, **21**(8):666–677, 1978.

[239] U. Hölzle. "Brawny cores still beat wimpy cores, most of the time." *IEEE Micro*, **30**(4):3, 2010. Also *http://static.googleusercontent.com/media/research.google.com/en/pubs/archive/36448.pdf*, 2010. Accessed August 2016.

[240] J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities." *Proc. National Academy of Science*, **79**:2554–2558, 1982.

[241] C. Hopps. "Analysis of an equal-cost multi-path algorithm." *RFC 2992*. Internet Engineering Task Force, 2000.

[242] J. H. Howard, M. L. Kazer, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. "Scale and performance in a distributed file system." *ACM Transactions Computer Systems*, **6**(1):51–81, 1988.

[243] D. H. Hu, Y. Wang, and C.-L. Wang. "BetterLife 2.0: Large-scale social intelligence reasoning on cloud." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 529–536, 2010.

[244] K. Hwang, G. Fox, and J. Dongarra. *Distributed and Cloud Computing*. Morgan-Kaufmann, 2011.

[245] G. Iachello and J. Hong. "End-user privacy in human-computer interaction." *Foundations and Trends in Human-Computer Interactions*, **1**(1):1–137, 2007.

[246] IBM Smart Business. "Dispelling the vapor around the cloud computing. Drivers, barriers and considerations for public and private cloud adoption." White Paper, 2010. *ftp.software.ibm.com/common/ssi/ecm/en/ciw03062usen/CIW03062USEN.PDF*. Accessed August 2015.

[247] IBM Corporation. "General parallel file systems (version 3, update 4). Documentation Updates." *http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.gpfs.doc*. Accessed August 2015.

[248] IBM Corporation. "The evolution of storage systems." *IBM Research. Almaden Research Center Publications*. *http://www.almaden.ibm.com/storagesystems/pubs*. Accessed August 2015.

[249] IBM Corporation. "Bringing big data to the enterprise." *https://www-01.ibm.com/software/in/data/bigdata/*, 2015. Accessed May 2016.

[250] Intel Corporation. "Intel 64 and IA-32 Architectures Software Developer Manuals." *https://software.intel.com/en-us/node/699529#combined*, 2016. Accessed January 2017.

[251] IBM Corporation. "Big Data." *https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html*. Accessed January 2017.

[252] M. Iorga and A. Karmel. "Managing risk in a cloud ecosystem." *IEEE Cloud Computing*, **2**(6):51–57, 2015.

[253] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." *Proc. 2nd ACM SIGOPS/EuroSys European Conf. Computer Systems*, pp. 57–62, 2007.

[254] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. J. Wright. "Performance analysis of high performance computing applications on the Amazon Web services cloud." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 159–168, 2010.

[255] M. Jelasity, A. Montresor, and O. Babaoglu. "Gossip-based aggregation in large dynamic networks." *ACM Transactions Computer Systems*, **23**(3):219–252, 2005.

[256] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. "Gossip-based peer sampling." *ACM Transactions Computer Systems*, **25**(3):8, 2007.

[257] M. Jensen, S. Schäge, and J. Schwenk. "Towards an anonymous access control and accountability scheme for cloud computing." *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 540–541, 2010.

[258] H. Jin, X.-H. Sun, Y. Chen, and T. Ke. "REMEM: REmote MEMory as checkpointing storage." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 319–326, 2010.

[259] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, Y. Zhang, S. Madden, M. Stonebraker, J. Hugg, and D. J. Abadi. "H-Store: a high-performance, distributed main memory transaction processing system." *Proc. VLDB Endowment*, **1**(2):1496–1499, 2008.

[260] E. Kalyvianaki, T. Charalambous, and S. Hand. "Applying Kalman filters to dynamic resource provisioning of virtualized server applications." *Proc. 3rd Int. Workshop Feedback Control Implementation and Design in Computing Systems and Networks*, p. 6, 2008.

[261] E. Kalyvianaki, T. Charalambous, and S. Hand. "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters." *Proc. 6th Int. Conf. Autonomic Computing*, pp. 117–126, 2009.

[262] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G-Y. Wei, and D. Brooks. “Profiling a warehouse-scale computer.” *Proc. 42nd Annual Int. Sym. Computer Architecture, ISCA*, pp. 158–169, 2015.

[263] R. M. Karp, M. Luby, and F. Meyer auf der Heide. “Efficient PRAM simulation on a distributed memory machine.” *Proc. 24th ACM Symp. on the Theory of Computing*, pp. 318–326, 1992.

[264] K. Kc and K. Anyanwu. “Scheduling Hadoop jobs to meet deadlines.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 388–392, 2010.

[265] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy. “Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs.” *Proc. 4th Int. Conf. Autonomic Computing*, pp. 100–109, 2007.

[266] J. Kephart. “The utility of utility: Policies for autonomic computing.” *Proc. LCCC Workshop Control of Computing Systems*, 2011. *http://www.lccc.lth.se/media/LCCC2011/WorkshopDecember/Kephart.pdf*.

[267] W. O. Kermack and A. G. McKendrick. “A contribution to the theory of epidemics.” *Proc. Royal Soc. London, A*, **115**:700–721, 1927.

[268] A. Khajeh-Hosseini, D. Greenwood, and I. Sommeerville. “Cloud migration: A case study of migrating an enterprise IT system to IaaS.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 450–457, 2010.

[269] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[270] J. Kim, W. J. Dally, and D. Abts. “Flattened butterfly: a cost-efficient topology for high-radix networks.” *Proc. 34th Int. Symp. Computer Architecture*, pp. 126–137, 2007.

[271] J. Kim, W. J. Dally, and D. Abts. “Efficient topologies for large-scale cluster networks.” *Proc. 2010 Optical Fiber Communication Conf. and National Fiber Optic Engineers Conf*, pp. 1–3, 2010.

[272] S. T. King, P. M. Chen, Y-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. “SubVirt: Implementing malware with virtual machines.” *Proc. IEEE Symp. Security and Privacy*, pp. 314–327, 2006.

[273] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I. Jordan. “A general bootstrap performance diagnostic.” *Proc. 19th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*. ACM, New York, NY, pp. 419–427, 2013.

[274] L. Kleinrock. *Queuing Systems, Vols. I and II*. Wiley, New York, NY, 1965.

[275] D. E. Knuth. *The Art of Computer Programming I–III*, 2nd edition. Addison–Wesley, Reading, MA, 1973.

[276] F. Koeppe and J. Schneider. “Do you get what you pay for? Using proof-of-work functions to verify performance assertions in the cloud.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 687–692, 2010.

[277] B. Koley, V. Vusirikala, C. Lam, and V. Gill. “100Gb Ethernet and beyond for warehouse scale computing.” *Proc. 15th OptoElectronics and Communications Conf.*, pp. 106–107, 2010.

[278] J. G. Koomey. “Estimating total power consumption by servers in the US and world.” *http://hightech.lbl.gov/documents/data_centerssvrpwrusecompletefinal.pdf*. Accessed May 2016.

[279] J. G. Koomey, S. Berard, M. Sanchez, and H. Wong. “Implications of historical trends in the energy efficiency of computing.” *IEEE Annals of the History of Computing*, **33**(3):46–54, 2011.

[280] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kuma, A. Leblang, N. Li, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. “Impala: a modern, open-source SQL engine for Hadoop.” *Proc 7th Biennial Conf. on Innovative Data Systems Research*, (online proceedings), 2015. See also *cidrdb.org/cidr2015/CIDR15_Paper28.pdf*.

[281] G. Koslovski, W.-L. Yeow, C. Westphal, T. T. Huu, J. Montagnat, and P. Vicat-Blanc. “Reliability support in virtual infrastructures.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 49–58, 2010.

[282] P. R. Krugman. *The Self-organizing Economy*. Blackwell Publishers, 1996.

[283] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*, 6th Edition. Addison–Wesley, Reading, MA, 2013.

[284] S. Kumar, L. Shi, N. Ahmed, S. Gil, D. Katabi, and D. Rus. “CarSpeak: a content-centric network for autonomous driving.” *ACM SIGCOMM Computer Communication Review*, **42**(4):259–270, 2012.

[285] D. Kusic, J. O. Kephart, N. Kandasamy, and G. Jiang. “Power and performance management of virtualized computing environments via lookahead control.” *Proc. 5th Int. Conf. Autonomic Computing*, pp. 3–12, 2008.

[286] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. “KVM: the Linux virtual machine monitor.” *Proc. Linux Symposium*, Ottawa, pp. 225–230, 2007.

[287] C. Labovitz et al. “ATLAS Internet Observatory 2009 Annual Report.” *http://www.nanog.org/meetings/nanog47/presentations*, 2009. Accessed August 2015.

[288] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Khan, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff. “A brief history of the Internet.” *ACM SIGCOMM Computer Communication Review*, **1**(5):22–39, 2009.

[289] C. F. Lam. “FTTH look ahead – Technologies and architectures.” *Proc. 36th European Conf. Optical Communications (ECOC’10)*, pp. 1–18, 2010.

[290] L. Lamport and P. M. Melliar-Smith. “Synchronizing clocks in the presence of faults.” *Journal of the ACM*, **32**(1):52–78, 1985.

[291] L. Lamport. “The part-time parliament.” *ACM Transactions Computer Systems*, **2**:133–169, 1998.

[292] L. Lamport. “Paxos made simple.” *ACM SIGACT News*, **32**(4):51–58, 2001.

[293] L. Lamport. “Turing Lecture – the computer science of concurrency: the early years.” *Communications of the ACM*, **58**(6):71–77, 2015.

[294] B. W. Lampson and H. E. Sturfis. “Reflections on operating system design.” *Communications of the ACM*, **19**(5):251–265, 1976.

[295] P. A. Lascocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. “The inevitability of failure: the flawed assumption of security in modern computing environments.” *Proc. 21 National. Information Systems Security Conf*, pp. 303–314, 1998.

[296] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, 1982.

[297] D. Lea. *Concurrent Programming in Java*, Second Edition. Addison–Wesley, Reading, MA, 1999.

[298] P. J. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf. “The architecture of an integrated local area network.” *IEEE Transactions Selected Areas in Communication*, **1**(5):842–857, 1983.

[299] E. Lee, E-K. Lee, and M. Gerla. “Vehicular cloud networking: architecture and design principles .” *IEEE Communications Magazine*, **52**(2):142–155, 2014.

[300] E. Le Sueur and G. Heiser. “Dynamic voltage and frequency scaling: the laws of diminishing returns.” *Proc. Workshop Power Aware Computing and Systems*, pp. 2–5, 2010.

[301] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. “Tachyon: reliable, memory speed storage for cluster computing frameworks.” *Proc. ACM Sym. Cloud Computing*, pp. 1–15, 2014.

[302] Z. Li, N.-H. Yu, and Z. Hao. “A novel parallel traffic control mechanism for cloud computing.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 376–382, 2010.

[303] C. Li, A. Raghunathan, and N. K. Jha. “Secure virtual machine execution under an untrusted management OS.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 172–179, 2010.

[304] X. Li, J. B. Dennis, G. R. Gao, W. Lim, H. Wei, C. Yang, and R. Pavel. “FreshBreeze: a data flow approach for meeting DDDAS challenges.” *Proc. Int. Conf, On Computational Science (ICCS 2015)*, pp. 2573–2584, 2015.

[305] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh. “Automated control in cloud computing: challenges and opportunities.” *Proc. First Workshop Automated Control for Data Centers and Clouds*. ACM Press, pp. 13–18, 2009.

[306] C. Lin and D. C. Marinescu. “Stochastic high-level Petri nets and applications.” *IEEE Transactions on Computers*, **37**(7):815–825, 1988.

[307] X. Lin, Y. Lu, J. Deogun, and S. Goddard. “Real-time divisible load scheduling for cluster computing.” *Proc. 13th IEEE Real-time and Embedded Technology and Applications Symp.*, pp. 303–314, 2007.

[308] C. Lin and D. C. Marinescu. “Stochastic high level Petri Nets and applications.” *IEEE Transactions Computers*, **C-37**(7):815–825, 1988.

[309] S. Liu, G. Quan, and S. Ren. “On-line scheduling of real-time services for cloud computing.” *Proc. 6th World Congress on Services*. IEEE, pp. 459–464, 2010.

[310] D. Lo, L. Cheng, R. Govindaraju, L-A. Barroso, and C. Kozyrakis. “Towards energy proportionality for large-scale latency-critical workloads.” *Proc. ACM SIGARCH Computer Architecture News*, **42**(3):301–312, 2014.

[311] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and Kozyrakis. “Heracles: improving resource efficiency at scale.” *Proc. 42nd Annual Int. Symp. Computer Architecture*, pp. 450–462, 2015.

[312] G. K. Lockwood. “Quick MPI cluster setup on Amazon EC2.” *https://glennklockwood.blogspot.com/2013/04/quick-mpi-cluster-setup-on-amazon-ec2.html*. Accessed January 2017.

[313] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis. “Towards a reference architecture for semantically interoperable clouds.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 143–150, 2010.

[314] C. Lu, J. Stankovic, G. Tao, and S. Son. “Feedback control real-time scheduling: framework, modeling and algorithms.” *J. of Real-time Systems*, **23**(1–2):85–126, 2002.

[315] W. Lu, J. Jackson, J. Ekanayake, R. S. Barga, and N. Araujo. “Performing large science experiments on Azure: Pitfalls and solutions.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 209–217, 2010.

[316] A. Luckow and S. Jha. “Abstractions for loosely-coupled and ensemble-based simulations on Azure.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 550–556, 2010.
[317] J. Luna, N. Suri, M. Iorga, and A. Karmel. “Leveraging the potential of cloud security service level agreements through standards.” *IEEE Cloud Computing*, **2**(3):32–40, 2015.
[318] W-Y. Ma, B. Shen, and J. Brassil. “Content service networks: the architecture and protocols.” *Web Caching and Content Delivery*, Eds. A. Bestavros and M. Rabinovich, Elsevier, pp. 83–101, 2001.
[319] J. Madhavan, A. Halevy, S. Cohen, X. Dong, S. R. Jeffery, D. Ko, and C. Yu. “Structured data meets the web: A few observations.” *IEEE Data Engineering Bulletin*, **29**(3):19–26, 2006. Also, *http://research.google.com/pubs/pub32593.html*. Accessed August 2015.
[320] D. J. Magenheimer and T. W. Christian. “vBlades: Optimized paravirtualization for the Itanium processor family.” *Proc. 3rd VM Research and Technology Workshop*, San Jose, CA, pp. 73–82, 2004.
[321] S. Majumder and S. Rixner. “Comparing Ethernet and Myrinet for MPI communication.” *Proc. 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, pp. 1–7, 2004.
[322] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. “Rethinking main memory OLTP recovery.” *Proc. IEEE 30th Int. Conf. on Data Engineering*, pp. 604–615, 2014.
[323] D. C. Marinescu. *Internet-based Workflow Management*. Wiley, New York, NY, 2002.
[324] D. C. Marinescu, H. J. Siegel, and J. P. Morrison. “Options and commodity markets for computing resources.” *Market Oriented Grid and Utility Computing*, Eds. R. Buyya and K. Bubendorf, Wiley, New York, NY. ISBN: 9780470287682, pp. 89–120, 2009.
[325] D. C. Marinescu, C. Yu, and G. M. Marinescu. “Scale-free, self-organizing very large sensor networks.” *J. of Parallel and Distributed Computing*, **50**(5):612–622, 2010.
[326] D. C. Marinescu, A. Paya, J. P. Morrison, and P. Healy. “Distributed hierarchical control versus an economic model for cloud resource management.” *http://arXiv:.org/pdf/1503.01061.pdf*, 2015.
[327] D. C. Marinescu. “Cloud energy consumption.” Chapter 25 in *Encyclopedia of Cloud Computing*, Eds. S. Muguresan and I. Bojanova, Wiley, New York, NY, 2016.
[328] D. C. Marinescu. *Complex Systems and Clouds: A Self-Organization and Self-Management Perspective*. Morgan Kaufmann, Waltham, MA, 2016.
[329] D. C. Marinescu, A. Paya, and J. P. Morrison. “A cloud reservation system for Big Data applications.” *IEEE Transactions Parallel and Distributed Computing*, **28**(3):606–618, 2017.
[330] D. C. Marinescu, A. Paya, J. P. Morrison, and S. Olariu. “An approach for scaling cloud resource management.” *Cluster Computing*, **20**(1):909–924, 2017, Springer Verlag.
[331] P. Marshall, K. Keahey, and T. Freeman. “Elastic site: using clouds to elastically extend site resources.” *Proc. IEEE Int. Symp. Cluster Computing and the Grid*, pp. 43–52, 2010.
[332] L. Mashayekhy, M. M. Nejad, and D. Grosu. “Cloud federations in the sky: formation game and mechanisms.” *IEEE Transactions Cloud Computing*, **3**(1):14–27, 2015.
[333] F. Mattern. “Virtual time and global states of distributed systems.” *Proc. Int. Workshop Parallel and Distributed Algorithms*. Elsevier, New York, pp. 215–226, 1989.
[334] J. M. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, Waltham, MA, 2000.
[335] M. W. Mayer. “Architecting principles for system of systems.” *Systems Engineering*, **1**(4):267–274, 1998.
[336] M. Mazzucco, D. Dyachuk, and R. Deters. “Maximizing cloud providers revenues via energy aware allocation policies.” *Proc. IEEE 3rd Int. Conf. Cloud Computing*, pp. 131–138, 2010.
[337] S. McCartney. *ENIAC; The Triumphs and Tragedies of the World’s First Computer*. Walker & Company, New York, NY, 1999.
[338] P. McKenney. “On the efficient implementation of fair queuing.” *Internetworking: Research and Experience*, **2**:113–131, 1991.
[339] P. Mell. “What is special about cloud security?.” *IT-Professional*, **14**(4):6–8, 2012. *http://doi.ieeecomputersociety.org/10.1109/MITP.2012.84*. Accessed August 2015.
[340] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. “Diagnosing performance overheads in Xen virtual machine environments.” *Proc. 1st ACM/USENIX Conf. Virtual Execution Environments*, 2005.
[341] A. Menon, A. L. Cox, and W. Zwaenepoel. “Optimizing network virtualization in Xen.” *Proc. USENIX Annual Technical Conf*, pp. 15–28, 2006.
[342] A. P. Miettinen and J. K. Miettinen. “Energy efficiency of mobile clients in cloud computing.” *Proc. 2nd USENIX Conf. on Hot Topics in Cloud Computing*, pp. 4–11, 2010.

[343] R. A. Milner. *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlin, Heidelberg, NY, 1980.

[344] R. A. Milner. "Lectures on a calculus for communicating systems." *Proc. Seminar on Concurrency*, Lecture Notes in Computer Science, vol. 197, Springer-Verlag, Heidelberg, pp. 197–220, 1984.

[345] M. Miranda. "When every atom counts." *IEEE Spectrum*, **49**(7):32–37, 2012.

[346] J. Mitola and G. Q. Maguire. "Cognitive radio: making software radios more personal." *IEEE Personal Communications*, **6**:13–18, 1999.

[347] J. Mitola. "Cognitive radio: an integrated agent architecture for software defined radio." *Ph.D. Thesis*, KTH, Stockholm, 2000.

[348] M. Mitzenmacher. "The power of two choices in randomized load balancing." *Ph.D. Dissertation*, Computer Science Department, University of California at Berkeley, 1996.

[349] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. "The power of two random choices: a survey of techniques and results." *Handbook of Randomized Computing*, Eds. S. Rajasekaran, P. M. Pardalos, J. H. Reif, and J. Rolim, Kluwer Academic Publishers, pp. 255–312, 2001.

[350] M. Mitzenmacher. "The power of two choices in randomized load balancing." *IEEE Transactions on Parallel and Distributed Systems*, **12**(10):1094–1104, 2001.

[351] T. Miyamoto, M. Hayashi, and K. Nishimura. "Sustainable network resource management system for virtual private clouds." *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 512–520, 2010.

[352] A. Mondal, S. K. Madria, and M. Kitsuregawa. "Abide: A bid-based economic incentive model for enticing non-cooperative peers in mobile P2P networks." *Proc. 12th Int. Conf. Database Systems for Advanced Applications*, pp. 703–714, 2007.

[353] J. H. Morris, M. Satyanarayanan, M. H. Conner, M. H. Howard, D. S. Rosenthal, and F. D. Smith. "Andrew: a distributed personal computing environment." *Communications of the ACM*, **29**(3):184–201, 1986.

[354] R. J. T. Morris and B. J. Truskowski. "The evolution of storage systems." *IBM Systems Journal*, **42**(2):205–217, 2003.

[355] J. Nagle. "On packet switches with infinite storage." *IEEE Transactions Communications*, **35**(4):435–438, 1987.

[356] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. "Intel virtualization technology: hardware support for efficient processor virtualization." *Intel Technology Journal*, **10**(3):167–177, 2006.

[357] M. Nelson. "Virtual memory for the Sprite operating system." *Technical Report UCB/CSD 86/301*, Computer Science Division (EECS), University of California, Berkeley, 1986.

[358] M. N. Nelson, B. B. Welch, and J. K. Osterhout. "Caching in Sprite network file system." *ACM Transactions Computer Systems*, **6**(1):134–154, 1988.

[359] A. J. Nicholson, S. Wolchok, and B. D. Noble. "Juggler: virtual networks for fun and profit." *IEEE Transactions Mobile Computing*, **9**(1):31–43, 2010.

[360] H. Nissenbaum. "Can trust be secured online? A theoretical perspective." *Etica e Politica*, **I**(2), 24, 1999, Edizione Universita di Trieste, *http://hdl.handle.net/10077/5544*.

[361] NIST. "Cloud architecture reference models: A survey." *http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/Meeting4AReferenceArchitecture013111NIST_CCRATWG_004v2_ExistentReferenceModels_01182011.pdf*. Accessed February 2017.

[362] NIST–Reference Architecture Analysis Team. "Cloud computing reference architecture – Strawman model V2." Document *NIST CCRATWG 019*, p. 28, 2011. *http://www.ogf.org/pipermail/occi-wg/attachments/20110303/e63dee43/attachment-0001.pdf*. Accessed February 2017.

[363] NIST–ITLCCP. "NIST cloud computing reference architecture. (version 1)." *NIST – Information Technology Laboratory Cloud Computing Program*, 2011. *http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/ReferenceArchitectureTaxonomy/NIST_CC_Reference_Architecture_v1_March_30_2011.pdf*. Accessed February 2017.

[364] NIST. *Cloud specific terms and definitions*, *NIST Cloud Computing Collaboration Site*. *http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/ReferenceArchitectureTaxonomy/Taxonomy_Terms_and_definitions_version_1.pdf*. Accessed February 2017.

[365] NIST. "Basic security functional areas." *NIST Cloud Computing Collaboration Site. NIST Reference Architecture – Strawman Model*, *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing/RefArchDocuments*. Accessed February 2017.

[366] NIST. "Cloud security services architecture." *NIST Cloud Computing Collaboration Site*. *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing*. Accessed February 2017.

[367] NIST. “Threat sources by cloud architecture component.” *NIST Cloud Computing Collaboration Site*. *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing*. Accessed February 2017.

[368] NIST. “General cloud environments – SWG.” *NIST Cloud Computing Collaboration Site*. *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing*. Accessed February 2017.

[369] NIST. “Threat analysis of cloud services (initial thoughts for discussion).” *NIST Cloud Computing Collaboration Site*. *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing*. Accessed February 2017.

[370] NIST. “Top 10 cloud security concerns (Working list).” *NIST Cloud Computing Collaboration Site*, *http://collaborate.nist.gov/twiki-cloud-computing/bin/view/CloudComputing*. Accessed February 2017.

[371] NIST. “Mobile device security.” *https://nccoe.nist.gov/sites/default/files/library/sp1800/mds-nist-sp1800-4b-draft.pdf*, 2015. Accessed August 2016.

[372] Y. Nuevo. “Cellular phones as embedded systems.” *Digest of Technical Papers, IEEE Solid-State Circuits Conference*, pp. 32–37, 2004.

[373] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. “The Eucalyptus open-source cloud computing system.” *Proc. 9th IEEE/ACM Int. Symp. Cluster Computing and the Grid*, pp. 124–131, 2009.

[374] S. Oikawa and R. Rajkumar. “Portable RK: A portable resource kernel for guaranteed and enforced timing behavior.” *Proc. IEEE Real Time Technology and Applications Symp.*, pp. 111–120, June 1999.

[375] T. Okuda, E. Kawai, and S. Yamaguchi. “A mechanism of flexible memory exchange in cloud computing environments.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 75–80, 2010.

[376] D. Olmedilla. “Security and privacy on the semantic web.” *Security, Privacy and Trust in Modern Data Management*, Eds. M. Petkovic and W. Jonker, Springer-Verlag, 2006.

[377] M. O’Neill. “SaaS, PaaS, and IaaS: a security checklist for cloud models.” *http://www.csoonline.com/article/660065/saas-paas-and-iaas-a-security-checklist-for-cloud-models*. Accessed August 2015.

[378] OpenVZ. *http://wiki.openvz.org*. Accessed August 2015.

[379] A. M. Oprescu and T. Kielmann. “Bag-of-tasks scheduling under budget constraints.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 351–359, 2010.

[380] Oracle Corporation. “Lustre file system.” *http://en.wikipedia.org/wiki/Lustre_(file_system)*, 2010. Accessed February 2017.

[381] Oracle Corporation. “Oracle NoSQL Database.” *http://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf*, 2011. Accessed February 2017.

[382] OSA. “SP-011: Cloud computing pattern.” *http://www.opensecurityarchitecture.org/cms/library/patternlandscape/251-pattern-cloud-computing*. Accessed February 2017.

[383] D. L. Osisek, K. M. Jackson, and P. H. Gum. “ESA/390 interpretive-execution architecture, foundation for VM/ESA.” *IBM Systems Journal*, **30**(1):34–51, 1991.

[384] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. “Sparrow: distributed, low latency scheduling.” *Proc. 24th ACM Symp. on Operating Systems Principles*, pp. 69–84, 2013.

[385] N. Oza, K. Karppinen, and R. Savola. “User experience and security in the cloud – An empirical study in the Finnish Cloud Consortium.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 621–628, 2010.

[386] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef. “Performance management for cluster-based web services.” *IEEE J. Selected Areas in Communications*, **23**(12):2333–2343, 2005.

[387] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. “Performance evaluation of virtualization technologies for server consolidation.” *HP Technical Report HPL-2007-59*, 2007lso, *http://www.hpl.hp.com/techreports/2007/HPL-2007-59R1.pdf*. Accessed August 2015.

[388] P. Paillier. “Public-key cryptosystems based on composite degree residuosity classes.” *Advances in Cryptology*. Springer-Verlag, pp. 223–238, 1999.

[389] Y. Pan, S. Maini, and E. Blevis. “Framing the issues of cloud computing and sustainability: A design perspective.” *Proc. IEEE 2nd Int. Conf. Cloud Computing Technology and Science*, pp. 603–608, 2010.

[390] V. Pankratius, F. Lind, A. Coster, P. Erickson, and J. Semeter. “Space weather monitoring using multicore mobile devices.” *American Geographic Union (AGU) Fall Meeting Abstracts*, *http://adsabs.harvard.edu/abs/2013AGUFMSA31B..06P*. Also *https://mahali.mit.edu/sites/default/files/documents/Mahali-Overview.pdf*. Accessed November 2016.

[391] M. Paolino, A. Rigo, A. Spyridakis, J. Fanguède, P. Lalov, and D. Raho. “T-KVM: A trusted architecture for KVM ARM v7 and v8 virtual machines securing virtual machines by means of KVM, TrustZone, TEE, and SELinux.” *Proc, 6th Int. Conf. on Cloud Computing, GRIDs, and virtualization*. IARIA, Nice, France, pp. 39–45, 2015.

[392] D. L. Parnas. “On the criteria to be used in decomposing systems into modules.” *Communications of the ACM*, **15**(12):1053–1058, 1972.
[393] S. Parvin, S. Han, L. Gao, F. Hussain, and E. Chang. “Towards trust establishment for spectrum selection in cognitive radio networks.” *Proc. IEEE 24th Int. Conf. Advanced Information Networking and Applications*, pp. 579–583, 2010.
[394] A. M. K. Pathan and R. Buya. “A taxonomy of content delivery networks.” *http://cloudbus.org/reports/CDN-Taxonomy.pdf*, 2009. Accessed August 2015.
[395] B. Pawlowski, C. Juszezak, P. Staubach, C. Smith, D. Label, and D. Hitz. “NFS Version 3 design and implementation.” *Proc. Summer 1994 Usenix Technical Conference*, pp. 137–151, 1994.
[396] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Turlow. “The NFS Version 4 protocol.” *Proc. 2nd Int. SANE Conf. System Administration and Network Engineering*, 2000.
[397] A. Paya and D. C. Marinescu. “A cloud service for adaptive digital music streaming.” *Proc. 8th Int. Conf. Signal Image Technology and Internet Systems*, 2012.
[398] A. Paya and D. C. Marinescu. “Energy-aware load balancing and application scaling for the cloud ecosystem.” *IEEE Transactions on Cloud Computing*, **5**(1):15–27, 2017.
[399] J. Pearn. “How many servers does Google have?.” *https://plus.google.com/+JamesPearn/posts/VaQu9sNxJuY*, 2012. Accessed February 2017.
[400] S. Pearson and A. Benameur. “Privacy, security, and trust issues arising from cloud computing.” *Proc. Cloud Computing and Science*, pp. 693–702, 2010.
[401] M. Perrin. *Distributed Systems: Concurrency and Consistency*. Elsevier, Oxford, UK, 2017.
[402] C. A. Petri. “Kommunikation mit Automaten.” *Schriften des Rheinisch-Westfälisches Institutes für Instrumentelle Mathematik*, **2**, 1962, Bonn.
[403] C. A. Petri. *Concurrency theory*, Lecture Notes in Computer Science, vol. 254, Springer-Verlag, Heidelberg, pp. 4–24, 1987.
[404] A. Pnueli. “The temporal logic of programs.” *Proc. 18th Annual IEEE Symp. Foundations of Computer Science*, pp. 46–57, 1977.
[405] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. “CryptDB: protecting confidentiality with encrypted query processing.” *Proc. 23 ACM Symp. on Operating Systems Principles*, pp. 85–100, 2011. http://dx.doi.org/10.1145/2043556.2043566.
[406] G. J. Popek and R. P. Golberg. “Formal requirements for virtualizable third generation architecture.” *Communications of the ACM*, **17**(7):412–421, 1974.
[407] “A framework for hardware-software co-design of embedded systems.” *Embedded.eecs.berkeley.edu/Respep/Research/hsc*, 2012. Accessed August 2015.
[408] C. Preist and P. Shabajee. “Energy use in the media cloud.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 581–586, 2010.
[409] D. Price and A. Tucker. “Solaris Zones: operating systems support for consolidating commercial workloads.” *Proc. 18th Large Installation System Administration*. USENIX, pp. 241–254, 2004.
[410] M. Price. “The paradox of security in virtual environments.” *Computer*, **41**(11):22–28, 2008.
[411] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. “Understanding performance interference of I/O workload in virtualized cloud environments.” *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 51–58, 2010.
[412] P. Radzikowski. “SAN vs DAS: A cost analysis of storage in the enterprise.” *http://capitalhead.com/articles/san-vs-das-a-cost-analysis-of-storage-in-the-enterprise.aspx*, (updated 2010). Accessed February 2017.
[413] A. Ranabahu and A. Sheth. “Semantics centric solutions for application and data portability in cloud computing.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 234–241, 2010.
[414] F. Y. Rashid. “The dirty dozen: 12 cloud security threats.” *InfoWorld*, *www.infoworld.com/article/3041078/security/the-dirty-12-cloud-security-threats.html*, March 11, 2016.
[415] N. Regola and J.-C. Ducom. “Recommendations for virtualization technologies in high performance computing.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 409–416, 2010.
[416] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozuch. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis.” *Proc. ACM Symp. Cloud Computing*, 2012, Article #7.
[417] V. Reiss, R. Hanrahan, and K. Levendahl. “The Big Science of stockpile stewardship.” *Physics Today*, **68**(8):47–53, 2016.
[418] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. ““Google-wide profiling: A continuous profiling infrastructure for data centers.” *IEEE Micro*, **30**(4):65–79, 2010. *http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/36575.pdf*. Accessed August 2016.

[419] M. Riandato. “Jails Free BSD handbook.” *http://www.freebsd.org/doc/handbook/jails.html*. Accessed January 2017.

[420] RightScale. “Cloud computing trends: 2016 state of the cloud survey.” *http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey*, 2016. Accessed August 2016.

[421] D. M. Ritchie and K. Thompson. “The Unix time-sharing system.” *Communications of the ACM*, **17**(7):365–375, 1974.

[422] D. M. Ritchie. “The evolution of the Unix time-sharing system.” *Bell Labs Technical Journal*, **63**(2.2):1577–1593, 1984.

[423] L. M. Riungu, O. Taipale, and K. Smolander. “Research issues for software testing in the cloud.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 557–564, 2010.

[424] R. Rivest, A. Shamir, and L. Adleman. “A method for obtaining digital signatures and public-key cryptosystems.” *Communications of the ACM*, **21**(2):120–126, 1978.

[425] A. A. Rocha, T. Salonidis, T. He, and D. Towsley. “sLRFU: a data streaming based least recently frequently used caching policy.” *https://pdfs.semanticscholar.org/ea8d/df7b03109e18633c0b94a09a3dce03b18059.pdf*, 2015. Accessed March 2017.

[426] R. Rodrigues and P. Druschel. “Peer-to-peer systems.” *Communications of the ACM*, **53**(10):72–82, 2010.

[427] M. Rodriguez-Martinez, J. Seguel, and M. Greer. “Open source cloud computing tools: A case study with a weather application.” *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 443–449, 2010.

[428] J. Rolia, L. Cerkasova, M. Arlit, and A. Andrzejak. “A capacity management service for resource pools.” *Proc. 5th Int. Workshop on Software and Performance*. ACM, pp. 229–237, 2005.

[429] M. Rosenblum and T. Garfinkel. “Virtual machine monitors: Current technology and future trends.” *Computer*, **38**(5):39–47, 2005.

[430] D. M. Rousseau, S. B. Sitkin, R. S. Burt, and C. Camerer. “Not so different after all: a cross-disciplinary view of trust.” *Academy of Management Review*, **23**(3):393–404, 1998.

[431] T. L. Ruthkoski. “Exploratory project: State of the cloud, from University of Michigan and beyond.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 427–432, 2010.

[432] H. F-W Sadrozinski and J. Wu. *Applications of Field-Programmable Gate Arrays in Scientific Research*. Taylor and Francis Inc., Bristol, PA, USA, 2010.

[433] A. Salomaa. *Public-Key Cryptography*. Springer-Verlag, Heidelberg, 1990.

[434] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, Waltham, MA, 2009.

[435] N. Samaan. “A novel economic sharing model in a federation of selfish cloud providers.” *IEEE Transactions Parallel and Distributed Systems*, **25**(1):12–21, 2014.

[436] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xy, and G. R. Ganger. “Diagnosing performance changes by comparing request flows.” *Proc. 8th USENIX Conf. Networked Systems Design and Implementation*, p. 14, 2011.

[437] B. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. “Design and implementation of the Sun network file system.” *Proc. Summer Usenix, Technical Conference*, pp. 119–130, 1986.

[438] T. Sandholm and K. Lai. “Dynamic proportional share scheduling in Hadoop.” *Proc. 15th Workshop Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science, vol. 6253, Springer-Verlag, Berlin, Heidelberg, pp. 110–131, 2010.

[439] R. Sadhu. “Good-enough security; toward a pragmatic business-driven discipline.” *IEEE Internet Computing*, **7**(1):66–68, 2003.

[440] M. Satyanarayanan. “A survey of distributed file systems.” *CS Technical Report, CMU*, *http://www.cs.cmu.edu/~satya/docdir/satya89survey.pdf*, 1989. Accessed August 2015.

[441] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. “The case for VM-based cloudlets in mobile computing.” *IEEE Transactions Pervasive Computing*, **8**(4):14–23, 2009.

[442] M. Satyanarayanan. “Mobile computing: the next decade.” *Proc. 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, pp. 1–5, 2010.

[443] J. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, Reading, MA, 1998.

[444] F. Schmuck and R. Haskin. “GFPS: A shared-disk file system for large computing clusters.” *Proc. Conf. File and Storage Technologies*. USENIX, pp. 231–244, 2002.

[445] P. Schuster. “Nonlinear dynamics from Physics to Biology. Self-organization: An 53 old paradigm revisited.” *Complexity*, **12**(4):9–11, 2007.

[446] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. “Omega: flexible, scalable schedulers for large compute clusters.” *Proc. 8th ACM European Conf. on Computer Systems*, pp. 351–364, 2013.

[447] S. Scott, D. Abts, J. Kim, and W. J. Dally. "The Blackwidow highradix Clos network." *Proc. 33rd Annual Int. Symp. on Computer Architecture*. IEEE, pp. 16–28, 2006.

[448] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. "Adapting software fault isolation to contemporary CPU architectures." *Proc. 19th USENIX Security Symposium*, pp. 1–11, 2010.

[449] P. Sempolinski and D. Thain. "A comparison and critique of Eucalyptus, OpenNebula and Nimbus." *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 417–426, 2010.

[450] J. Shneidman, C. Ng, D. C. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. Chun. "Why markets could (but don't currently) solve resource allocation problems in systems." *Proc. 10th Workshop on Hot Topics in Operating Systems*, pp. 7–14, 2005.

[451] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. "F1: a distributed SQL database that scales." *Proc. VLDB Endowment*, **6**(11):1068–1079, 2013.

[452] H. A. Simon. *Administrative Behavior*. Macmillan, New York, NY, 1947.

[453] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu. "Storage management in virtualized cloud environment." *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 204–211, 2010.

[454] J. Skorupa, J. Hewitt, and E. Zeng. "Gartner report: use networking to differentiate your hyperconverged system." *https://www.gartner.com/doc/reprints?id=1-3BPNPNA&ct=160715&st=sg*, May 2016. Accessed September 2016.

[455] J. E. Smith and R. Nair. "The architecture of virtual machines." *Computer*, **38**(5):32–38, 2005.

[456] L. Snyder. "Type architectures, shared memory, and the corollary of modest potential." *Annual Review of Computer Science*, **1**:289–317, 1986.

[457] B. Snyder. "Server virtualization has stalled, despite the hype." *http://www.infoworld.com/article/2624771/server-virtialization-has-stalled-despite-the-hype.html*, 2010. Accessed February 2017.

[458] SNIA, OGF. "Cloud storage for cloud computing." Joint Paper of *Storage Networking Industry Association* and *Open Grid Forum*, pp. 1–12 *http://forge.gridforum.org/sf/docman/do/downloadDocument/projects.occi-wg*, 2009. Accessed August 2015.

[459] SoftwareInsider. "Compare cloud computing providers." *http://cloud-computing.softwareinsider.com*. Accessed February 2017.

[460] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatowska, J. McDermid, and R. Paige. "Large-scale IT complex systems." *Communications of the ACM*, **55**(7):71–77, 2012.

[461] T. Stanley, T. Close, and M. S. Miller. "Causeway: A message-oriented distributed debugger." *Technical Report HPL-2009-78*, 2009, p. 14. Also, *http://www.hpl.hp.com/techreports/2009/HPL-2009-78.pdf*. Accessed February 2017.

[462] M. Stecca and M. Maresca. "An architecture for a mashup container in vizualized environments." *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 386–393, 2010.

[463] P. Stingley. "Cloud architecture." *http://sites.google.com/site/cloudarchitecture/*. Accessed February 2017.

[464] R. Stoica and A. Ailamaki. "Enabling efficient OS paging for main-memory OLTP databases." *Proc. 9th Int. Workshop on Data Management on New Hardware*, 2013, Article #6.

[465] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. "Using a market economy to provision compute resources across planet-wide clusters." *Proc. IEEE Int. Symp. on Parallel and Distributed Processing*, pp. 1–8, 2009.

[466] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for Internet applications." *Proc. ACM/SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 149–160, 2001.

[467] M. Stonebreaker. "The "NoSQL" has nothing to do with SQL." *http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext*, 2009. Accessed February 2017.

[468] StratoKey. "Cloud data protection." *https://www.stratokey.com/solutions/cloud-access-security-brokers?gclid=CM7bp5eOh80CFQ8kgQodYu4L5g*, 2015. Accessed August 2016.

[469] StreamingMedia. "Only 18% using adaptive streaming, says Skyfire report." *http://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=79393*. Accessed August 2015.

[470] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. "Overcite: A distributed, cooperative citeseer." *Proc. 3rd Symp. on Networked Systems Design and Implementation*, pp. 69–79, 2006.

[471] J. Sugerman, G. Venkitachalam, and B. Lim. "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor." *Proc. USENIX Annual Technical Conference*, pp. 1–14, 2001.

[472] C. Sun, W. Zhang, and K. B. Letaief. "Cluster-based cooperative spectrum sensing for cognitive radio systems." *Proc. IEEE Conf. on Communications*, pp. 2511–2515, 2007.

[473] C. Sun, W. Zhang, and K. B. Letaief. "Cooperative spectrum sensing for cognitive radios under BW constraints." *Proc. IEEE Wireless Communications and Networking Conference*, pp. 1–5, 2007.

[474] Y. Sun, Z. Han, and K. J. R. Liu. "Defense of trust management vulnerabilities in distributed networks." *IEEE Communications Magazine, Special Issue, Security in Mobile Ad Hoc and Sensor Networks*, **46**(2):112–119, 2008.

[475] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. M. Vin. "Application performance in the QLinux multimedia operating system." *Proc. 8th ACM Int. Conf. on Multimedia*, pp. 127–136, 2000.

[476] M. Steinder, I. Walley, and D. Chess. "Server virtualization in autonomic management of heterogeneous workloads." *ACM SIGOPS Operating Systems Review*, **42**(1):94–95, 2008.

[477] T. Taleb and A. Ksentini. "Follow me cloud: interworking distributed clouds & distributed mobile networks." *IEEE Network*, **27**(5):12–19, 2013.

[478] D. Tancock, S. Pearson, and A. Charlesworth. "A privacy impact assessment tool for cloud computing." *Proc. 2nd IEEE Int. Conf. Cloud Computing*, pp. 667–674, 2010.

[479] L. Tang, J. Dong, Y. Zhao, and L.-J. Zhang. "Enterprise cloud service architecture." *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 27–34, 2010.

[480] J. Tang, Y. Cui, K. Ren, J. Liu, and R. Buyya. "Ensuring security and privacy preservation for cloud data services." *ACM Computing Surveys*, **49**(1), 2016, article13.

[481] J. Tate, F. Lucchese, and R. Moore. "Introduction to Storage Area Networks." IBM Redbooks, 2006. *http://www.redbooks.ibm.com/redbooks/pdfs/sg245470.pdf*. Accessed August 2015.

[482] D. Tennenhouse. "Layered multiplexing considered harmful." *Protocols for High-Speed Networks*, Eds. H. Rudin and R. C. Williamson, North Holland, pp. 143–148, 1989.

[483] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. "A hybrid reinforcement learning approach to autonomic resource allocation." *Proc. IEEE Int Conf. on Autonomic Computing*, pp. 65–73, 2006.

[484] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive a warehousing solution over a MapReduce Framework." *Proc. VLDB Endowment*, **2**(2):1626–1629, 2009.

[485] J. Timmermans, V. Ikonen, B. C. Stahl, and E. Bozdag. "The ethics of cloud computing. A conceptual review." *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 614–620, 2010.

[486] "Top 500 supercomputers." *http://top500.org/featured/top-systems/* (Accessed January 2016).

[487] TRIAD. "Translating relaying Internet architecture integrating active directories." *http://gregorio.stanford.edu/triad/*. Accessed December 2016.

[488] C. Tung, M. Steinder, M. Spreitzer, and G. Pacifici. "A scalable application placement controller for enterprise data centers." *Proc. 16th Int. Conf. on the World Wide Web*, 2007.

[489] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem." *Proc. London Math. Soc., Ser. 2*, **42**:230–265, 1937; and "On computable numbers, with an application to the Entscheidungsproblem: A correction," *Proc. London Math. Soc., Ser. 2*, **43**:544–546, 1937.

[490] A. M. Turing. "The chemical basis of morphogenesis." *Philosophical Transactions of the Royal Society of London, Series B*, **237**:37–72, 1952.

[491] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leun. "Dynamic service migration and workload scheduling in edge-clouds." *Performance Evaluation.*, **91**(C):205–228, 2015.

[492] L. G. Valiant. "A bridging model for parallel computation." *Communications of the ACM*, **33**(8):103–112, 1990.

[493] L. G. Valiant. "A bridging model for multicore computing." *Proc. 16th Annual European Symp. on Algorithms*, Lecture Notes on Computer Science, vol. 5193, pp. 13–28, 2008.

[494] J. Varia. "Cloud architectures." *http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf*.

[495] J. van Vliet, F. Paganelli, S. van Wel, and D. Dowd. *Elastic Beanstalk: Simple Cloud Scaling for Java Developers*. O'Reilly Publishers, Sebastopol, California, 2011.

[496] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. "Dynamically scaling applications in the cloud." *ACM SIGCOMM Computer Communication Review*, **41**(1):45–52, 2011.

[497] H. N. Van, F. D. Tran, and J.-M. Menaud. "Performance and power management for cloud infrastructures." *Proc. IEEE Conf. Cloud Computing*, pp. 329–336, 2010.

[498] K. Varadhan, R. Govindan, and D. Estrin. "Persistent route oscillations in interdomain routing." *Computer Networks*, **32**(1):1–16, 2000.

[499] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. "The power of choice in data-aware cluster scheduling." *Proc. 11th USENIX Symp. Operating Systems Design and Implementation*, pp. 301–314, 2014.

[500] P. Veríssimo and L. Rodrigues. “A posteriori agreement for fault-tolerant clock synchronization on broadcast networks.” *Proc. 22nd Annual Int. Symp. on Fault-Tolerant Computing*. IEEE Press, Los Alamitos, CA, pp. 527–536, 1992.

[501] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. “Server workload analysis for power minimization using consolidation.” *Proc. USENIX Annual Technical Conference*, p. 28, 2009.

[502] A. Verma, L. Pedrosaz, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. “Large-scale cluster management at Google with Borg.” *Proc. 10th European Conference on Computer Systems*, 2015, Article No.18.

[503] VMware. “VMware vSphere Storage Appliance.” *https://www.vmware.com/files/pdf/techpaper/VM-vSphere-Storage-Appliance-Deep-Dive-WP.pdf*. Accessed August 2015.

[504] J. von Neumann. “Probabilistic Logic and Synthesis of Reliable Organisms from Unreliable Components.” *Automata Studies*, Eds. C. E. Shannon and J. McCarthy, Princeton University Press, Princeton, NJ, 1956.

[505] J. von Neumann. “Fourth University of Illinois Lecture.” *Theory of Self-Reproducing Automata*, Ed. A. W. Burks, University of Illinois Press, Champaign, IL, 1966.

[506] S. V. Vrbsky, M. Lei, K. Smith, and J. Byrd. “Data replication and power consumption in data grids.” *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 288–295, 2010.

[507] B. Walker, G. Popek, E. English, C. Kline, and G. Thiel. “The LOCUS distributed operating system.” *Proc. 9th ACM Symp. Operating Systems Principles*, pp. 49–70, 1983.

[508] M. E. Wall, A. Rechtsteiner, and L. M. Rocha. “Singular value decomposition and principal component analysis.” *A Practical Approach to Microarray Data Analysis*, Eds. D. P. Berrar, W. Dubitzky, and M. Granzow, Kluwer, Norwell, MA, 2003.

[509] K. Walsh and E. G. Sirer. “Experience with an object reputation system for peer-to-peer file sharing.” *Proc. 3rd Symp. Networked Systems Design and Implementation*, pp. 1–14, 2006.

[510] C. Ward, N. Aravamudan, K. Bhattacharya, K. Cheng, R. Filepp, R. Kearney, B. Peterson, L. Shwartz, and C. C. Young. “Workload migration into clouds – challenges, experiences, opportunities.” *Proc. IEEE Conf. Cloud Computing*, pp. 164–171, 2010.

[511] L. Wang, L. Park, R. Pang, V. S. Pai, and L. Peterson. “Reliability and security in the CoDeeN content distribution network.” *Proc. USENIX Annual technical Conference*, p. 14, 2004.

[512] M. Wang, N. Kandasamy, A. Guez, and M. Kam. “Adaptive performance control of computing systems via distributed cooperative control: application to power management in computer clusters.” *Proc. 3rd IEEE Intl. Conf. on Autonomic Computing*, pp. 165–174, 2006.

[513] Y. Wang, I-R. Chen, and D-C. Wang. “A Survey of Mobile Cloud Computing Applications: Perspectives and Challenges.” *Wireless Personal Communications*, **80**(4):1607–1623, 2015.

[514] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leun. “Dynamic service migration in edge-clouds.” *Proc. IFIP Networking Conf.*. http://dx.doi.org/10.1109/IFIPNetworking.2015.7145316, 2015.

[515] D. J. Watts and S. H. Strogatz. “Collective-dynamics of small-world networks.” *Nature*, **393**:440–442, 1998.

[516] J. Webster. “Evaluating IBM’s SVC and TPC for server virtualization.” *ftp://ftp.boulder.ibm.com/software/at/tivoli/analyst_paper_ibm_svc_tpc.pdf*. Accessed August 2016.

[517] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, Waltham, MA, 2001.

[518] M. Whaiduzzaman, M. Sookhak, A. Gani, and R. Buyya. “A survey on vehicular cloud computing.” *J. of Network and ComputerApplications*, **40**:325–344, 2014.

[519] S. E. Whang and H. Garcia-Molina. “Managing information leakage.” *Proc. 5th Biennal Conf. on Innovative Data Systems Research*, 2011. Also *http://ilpubs.stanford.edu:8090/987/*. Accessed August 2015.

[520] J. Wilkes. “More Google cluster data.” *http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html*, 2011. Accessed August 2016.

[521] A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, Shelter Island, NY, 2012.

[522] A. Whitaker, M. Shaw, and S. D. Gribble. “Denali; lightweight virtual machines for distributed and networked applications.” *Technical Report 02-0201*, University of Washington, 2002.

[523] V. Winkler. *Securing the cloud: cloud computer security techniques and tactics*. Elsevier Science and Technologies Books, 2011.

[524] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. “Scalable thread scheduling and global power management for heterogeneous many-core architectures.” *Proc. 19th Int. Conf. Parallel Architectures and Compilation Techniques*, pp. 29–40, 2010.

[525] "Google 2 billion lines of code and in one place." *http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/*, 2015. Accessed August 2016.
[526] E. C. Withana and B. Plale. "Usage patterns to provision for scientific experimentation in clouds." *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 226–233, 2010.
[527] M. Witkowski, P. Brenner, R. Jansen, D. B. Go, and E. Ward. "Enabling sustainable clouds via environmentally opportunistic computing." *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 587–592, 2010.
[528] S. A. Wolf, A. Y. Chtchelkanova, and D. M. Treger. "Spintronics – a retrospective and perspective." *IBM J. of Research and Development.*, **50**(1):101–110, 2006.
[529] Xen Wiki. *http://wiki.xensource.com/xenwiki/CreditScheduler*, 2007.
[530] Z. Xiao and D. Cao. "A policy-based framework for automated SLA negotiation for internet-based virtual computing environment." *Proc. 16th IEEE Int. Conf. Parallel and Distributed Systems*, pp. 694–699, 2010.
[531] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. "Shark: SQL and rich analytics at scale." *ACM SIGMOD int. Conf. on Management of Data*, pp. 13–24, 2013.
[532] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos. "A survey of information-centric networking research." *IEEE Communications Surveys and Tutorials*, **16**(2):1024–1049, 2014.
[533] A. C. Yao. "Some complexity questions related to distributed computing." *Proc. 11th Symp. on the Theory of Computing*, pp. 209–213, 1979.
[534] A. C. Yao. "How to generate and exchange secrets." *Proc. 27th Annual Symp. on Foundations of Computer Science*, pp. 162–167, 1986.
[535] A. Yasin. "A Top–Down method for performance analysis and counters architecture." *Proc. IEEE Int. Symp. on Performance Analysis Systems and Software*, pp. 1–10, 2014.
[536] A. Yasin, Y. Ben-Asher, and A. Mendelson. "Deep-dive analysis of the data analytics workload in CloudSuite." *Proc. IEEE Int. Workshop/Symp. on Workload Characterization, Paper 67*, pp. 1–10, 2014.
[537] S. Yi, D. Kondo, and A. Andrzejak. "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud." *Proc. 3rd IEEE Int. Conf. Cloud Computing*, pp. 236–243, 2010.
[538] H. Yu, X. Bai, and D. C. Marinescu. "Workflow management and resource discovery for an intelligent grid." *Parallel Computing*, **31**(7):797–811, 2005.
[539] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Kumar, and G. J. Currey. "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language." *Proc. 8th USENIX Symp. Operating System Design and Implementation*, pp. 1–14, 2009.
[540] M. Zapf and A. Heinzl. "Evaluation of process design patterns: an experimental study." *Business Process Management*, Lecture Notes on Computer Science, vol. 1806, Eds. W. M. P. van der Aalst, J. Desel, and A. Oberweis, Springer-Verlag, Heidelberg, pp. 83–98, 2000.
[541] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling." *Proc. 5th European Conf. on Computer Systems*, pp. 265–278, 2010.
[542] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing." *Proc. 9th USENIX Conf. on Networked Systems Design and Implementation*, pp. 2–16, 2012.
[543] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters." *Proc. 4th USENIX conference on Hot Topics in Cloud Computing, USENIX Association*, pp. 10–16, 2012.
[544] P. Zech, M. Felderer, and R. Breu. "Towards a model-based security testing approach in cloud computing environments." *Proc. 6th IEEE Int. Conf. on Software Security and Reliability Companion*, pp. 47–56, 2012.
[545] Z. L. Zhang, D. Towsley, and J. Kurose. "Statistical analysis of the generalized processor sharing scheduling discipline." *IEEE J. Selected Areas in Communications*, **13**(6):1071–1080, 1995.
[546] X. Zhang, J. Liu, B. Li, and T-S. P. Yum. "CoolStreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming." *Proc. IEEE INFOCOM 2005*, pp. 2102–2111, 2005.
[547] C. Zhang and H. D. Sterck. "CloudBATCH: A batch job queuing system on clouds with Hadoop and HBase." *Proc. 2nd IEEE Int. Conf. Cloud Computing Technology and Science*, pp. 368–375, 2010.
[548] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. C. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. "Named data networks." *ACM SIGCOMM Computer Communication Review*, **44**(3):66–73, 2014.
[549] X. Zhao, K. Borders, and A. Prakash, "Virtual machine security systems." *Advances in Computer Science and Engineering*, pp. 339–365.

Glossary

A

Access Control List (ACL) list of pairs (subject, value) defining the set of access rights to an object; for example, read, write, execute permissions for a file.

Advanced Configuration and Power Interface (ACPI) open standard for device configuration and power management by the operating system. It defines four Global "Gx" states and six Sleep "Sx" states. For example, "S3" is referred to as Standby, Sleep, or Suspend to RAM.

Advanced Microcontroller Bus Architecture (AMBA) open standard, on-chip interconnect specification for the connection and management of a large number of controllers and peripherals.

Amazon Machine Image (AMI) a unit of deployment, an environment including all information necessary to set up and boot an instance including: (1) a template for the root volume for the instance; e.g., an operating system, an application server, and applications; (2) launch permissions controlling all AWS accounts that can use the AMI to launch instances; and (3) a block device mapping specifying the volumes to be attached to the instance when launched.

Amdahl's law formula used to predict the theoretical maximum speedup for a program using multiple processors/cores. Informally, it states that the portion of the computation which cannot be parallelized determines the overall speedup.

Anti-entropy a process, often using Merkle trees, of comparing the data of all replicas and updating each replica to the newest version.

AppEngine (AE) an ensemble of computer, storage, search, and networking services for building web and mobile applications and run them on Google servers.

Application Binary Interface (ABI) the projection of the computer system seen by a process or thread in execution. ABI allows the ensemble consisting of the application and the library modules to access the hardware. ABI does not include privileged system instructions, instead it invokes system calls.

Application Program Interface (API) defines the set of instructions the hardware was designed to execute and gives the application access to the Instruction Set Architecture (ISA) layer. It includes High Level Language (HLL) library calls which often invoke system calls. The API is the projection of the system from the perspective of the HLL program.

Application layer deployed software applications targeted towards end-user software clients or other programs, and made available via the cloud.

Auction a sale where items are sold to the highest bidder.

Auditor party conducting independent assessment of cloud services, information system operations, performance and security of the cloud implementation.

Audit systematic evaluation of a cloud system by measuring how well it conforms to a set of established criteria; e.g., security audit if the criteria is security, privacy-impact audit if the criteria is privacy assurance, performance audit if the criteria is performance.

Authentication credential something that an entity is, has, or knows that allows that entity to prove its identity to a system.

Auto Scaling AWS service providing automatic scaling of EC2 instances through grouping of instances, monitoring of the instances in a group, and defining *triggers*, pairs of CloudWatch alarms and policies, which allow the size of the group to be scaled up or down.

AWK utility utility for text processing based on a scripting language.

B

Bandwidth the number of operations per unit of time; for example, the bandwidth of a processor is expressed in Mips or Mflops while the memory and I/O bandwidth is expressed in Mbps.

Basic Core Equivalent (BCE) quantity describing how resources of a multicore processor are allocated to the individual cores. For example, a symmetric core processor can be configured as sixteen 1-BCE cores, eight 2-BCE cores, four 4-BCE cores, two 6-BCE cores, or one 16-BCE cores. An asymmetric core processor may have ten 1-BCE cores and one 6-BCE core.

Basic input/output system (BIOS) system component invoked after a computer system is powered on to load the operating system and later to manage the data flow between the OS and devices such as keyboard, mouse, disk, video adapter, and printer.

Bigquery fully-managed enterprise data warehouse for large-scale data analytics on Google cloud platform.
BigTable distributed storage system developed by Google to store massive amounts of data and to scale up to thousands of storage servers.
Bisection bandwidth the sum of the bandwidths of the minimal number of links that are cut when splitting a network into two parts.
Bit-level parallelism parallel computing based on increasing processor word size thus, lowering the number of instructions required to process larger size operands.
BitTorrent peer-to-peer communications protocol for file sharing.
Border Gateway Protocol (BGP) a path vector reachability protocol. It maintains a table of IP networks. It designates network reachability among autonomous systems and makes the core routing decisions in the Internet based on path, network policies and/or rule sets.
Borg management software for clusters consisting of tens of thousands of servers co-located and interconnected by a datacenter-scale network fabric.
Boundary value problem problem with conditions specified at the extremes of the independent variable(s).
Bounded input data defining property of batch processing. The computing engine has as input a dataset of known contents and size, as opposed to processing a continuous stream of incoming data.
Broker entity that manages the use, performance and delivery of cloud services, and negotiates relationships between cloud service providers and cloud users.
Buffer overflow anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.
BusyBox software providing several stripped-down Unix tools in a single executable file and running in environments such as Linux, Android, FreeBSD, or Debian.
Bus.Device.Function (BDF) data used to describe PCI devices.
Byte-range tokens used to specify the range of read and write operations to data files.
Byzantine failure a fault presenting different symptoms to different observers. In a distributed system a Byzantine failure could be: an *omission failure*, e.g., a crash failure, failure to receive a request or to send a response; it could also be a *commission failure*, e.g., process a request incorrectly, corrupt the local state, and/or send an incorrect or inconsistent response to a request.

C
Callback executable code passed as an argument to other code; the callee is expected to execute the argument either immediately or at a later time for synchronous and, respectively, asynchronous callbacks.
Callstack data structure storing information about the active subprograms invoked during the execution of a program. Also called execution stack, program stack, control stack, or run-time stack.
Carrier a networking organization that provides connectivity and transports data between communicating entities. Also, a carrier signal is a transmitted electromagnetic pulse or wave at a steady base frequency on which information can be imposed by modulation.
Causal delivery extension of the First-In-First-Out (FIFO) delivery to the case when a process receives messages from different sources.
Cell storage storage organization consisting of cells of the same size and objects fitting exactly in one cell.
Central Limit Theorem (CLT) statistical theory stating that the sum of a large number of independent random variables has a normal distribution.
Chaining in vector computers mechanisms allowing vector operations to start as soon as individual elements of vector source operands become available. Chaining operates on *convoys*, sets of vector instructions that can potentially be executed together.
Chameleon an NSF facility, an OpenStack KVM experimental environment for large-scale cloud research.
Command Line Interface (CLI) provides the means for a user to interact with a program.
Client-server paradigm software organization based on message-passing enforcing modularity. It allows systems with different processor architectures, different operating systems, libraries, and other system software, to cooperate.
Clock condition a strong clock condition in a distributed system requires an equivalence between the causal precedence and the ordering of the time stamps of messages.
Closed-box platforms systems with embedded cryptographic key that allow themselves to reveal their true identity to remote systems and authenticate the software running on them. Found on some cellular phones, game consoles, and ATMs.

Clos network multistage nonblocking network with an odd number of stages. In a Clos network all packets overshoot their destination and then hop back to it.
Cloud Bigtable high performance NoSQL database service for large analytical and operational workloads on Google cloud platform.
Cloud Datastore highly-scalable NoSQL database for web and mobile applications on Google cloud platform.
CloudFormation AWS service for creation of a stack describing the application infrastructure.
Cloud Functions (CF) a lightweight, event-based, asynchronous system to create single-purpose functions that respond to cloud events on Google cloud platform.
CloudLab an NSF facility, a testbed allowing researchers to experiment with cloud architectures and new applications.
CloudWatch AWS monitoring infrastructure used to collect and track metrics important for optimizing the performance of applications and for increasing the efficiency of resource utilization. Without installing any software a user can monitor pre-selected metrics and then view graphs and statistics for these metrics.
Coarse-grained parallelism execution mode when large blocks of code are executed before the concurrent threads/processes communicate with one another.
Cognitive radio wireless communication when an intelligent transceiver detects which communication channels are not in use and uses them while avoiding channels in use.
Cognitive radio trust trust regarding the information received by a intelligent transceiver from other nodes.
Combinatorial auction auction in which participants can bid on combinations of items or packages.
Community cloud a cloud infrastructure shared by several organizations and supporting a specific community with shared concerns (e.g., mission, security requirements, policy, and compliance considerations).
Communication channel physical system allowing two entities to communicate with one another.
Communication protocol a communication discipline involving a finite set of messages exchanged among entities. A protocol typically implements error control, flow control, and congestion control mechanisms.
Computation steering interactively guiding a computational experiment towards a region of interest.
Computer cloud a collection of systems in a single administrative domain offering a set of computing and storage services; a form of utility computing.
Computing grid a distributed system consisting of a large number of loosely coupled, heterogeneous, and geographically dispersed systems in different administrative domains. The name is a metaphor for accessing computer power with similar ease as accessing electric power supplied by the electric grid.
Concurrency simultaneous execution of related activities.
Concurrent write-sharing multiple clients can modify the data in a file at the same time.
Confidence interval statistical measure offering a guarantee of the quality of a result. A procedure is said to generate confidence intervals with a specified coverage $\alpha \in [0, 1]$ if, on a proportion exactly α of the set of experiments, the procedure generates an interval that includes the answer. For example, a 95% confidence interval $[a, b]$ means that in 95% of the experiments the result will be in $[a, b]$.
Conflict fraction average number of conflicts per successful transactions in a transaction processing system.
Congestion control mechanism ensuring that the offered load of a network does not exceed the network capacity.
Consistent hashing hashing technique for reducing the number of keys to be remapped when a hash table is resized. In average only K/n keys need to be remapped with K the number of keys and n the number of slots.
Container Engine cluster manager and orchestration system for Docker containers built on the Kubernetes system. It schedules and manages containers automatically according to user specifications on the Google cloud platform.
Content any type or volume of media, be it static or dynamic, monolithic or modular, live or stored, produced by aggregation, or mixed.
Container software system emulating a separate physical server; a container has its own files, users, process tree, IP address, shared memory, semaphores, and messages. Each container can have its own disk quotas.
Control flow architecture computer architecture when the program counter of a processor core determines the next instruction to be loaded in the instruction register and then executed.
Control sensitive instructions machine instructions changing either the memory allocation, or the execution to kernel mode.
Cooperative spectrum sensing mode of operation in which each node determines the occupancy of the spectrum based on its own measurements, combines it with information from its neighbors and then shares its own spectrum occupancy assessment with its neighbors.
Copy-on-write (COW) mechanism used by virtual memory operating systems to minimize the overhead of copying the virtual memory of a process when a process creates a copy of itself.

Cron a job scheduler for Unix-like systems used to periodically schedule jobs, often used to automate system maintenance and administration.

Cross-site scripting the most popular form of attack against web sites; a browser permits the attacker to insert client-scripts into the web pages and thus, bypass the access controls at the web site.

Compute Unified Device Architecture (CUDA) programming model invented by NVIDIA for using graphics processing units (GPUs) for general purpose processing.

Cut subset of the local history of all processes of a process group. *The frontier of the cut* is an n-tuple consisting of the last event of every process included in the cut.

Cut-through (wormhole) network routing routing mechanism when a packet is forwarded to its next hop as soon as the header is received and decoded. The packet can experience blocking if the outgoing channel expected to carry it to the next node is in use; in this case the packet has to wait until the channel becomes free.

D

Database as a Service (DBaaS) a cloud service where the database runs on the physical infrastructure of the cloud service provider.

Data Description Language (DDL) syntax similar to a computer programming language for defining data structures; it is widely used for database schemas.

Data Manipulation Language (DML) programming language used to retrieve, store, modify, delete, insert and update data in database; SELECT, UPDATE, INSERT statements or query statements are examples of DML statements.

Dataflow architecture computer architecture where operations are carried out at the time when their input becomes available.

Datagram basic transfer unit in a packet-switched network; it consists of a header containing control information necessary to transport its payload through the network.

Data hazards in pipelining potential danger situations when the instructions in a pipeline are dependent upon one another.

Data-level parallelism an extreme form of coarse-grained parallelism, based on partitioning the data into chunks/blocks/segments and running concurrently either multiple programs or copies of the same program, each on a different data block.

Data portability the ability to transfer data from one system to another without being required to recreate or reenter data descriptions or to significantly modify the application being transported.

Data object a logical container of data that can be accessed over a network, e.g., a blob; may be an archive, such as specified by the *tar* format.

Data-shipping allows fine-grained data sharing; an alternative to byte-range locking.

Deadlock synchronization anomaly occurring when concurrent processes or threads compete with one another for resources and reach a state when none of them can proceed.

Denial of service attack (DOS attack) Internet attack targeting a widely used network service and preventing legitimate access to the service. It forces the operating system of the targeted host(s) to fill the connection tables with illegitimate entries.

De-perimeterisation process allowing systems to span the boundaries of multiple organizations and cross the security borders.

Direct Memory Access (DMA) hardware feature allowing I/O devices and other hardware subsystems direct access to the system memory without the CPU involvement. Also used for memory-to-memory copying and for offloading expensive memory operations, such as scatter-gather operations, from the CPU to the dedicated DMA engine. Intel includes I/O Acceleration Technology (I/OAT) on high-end servers.

Distributed system collection of computers interconnected by a network. Users perceive the system as a single, integrated computing facility.

Dynamic binary translation conversion of blocks of guest instructions from a portable code format to the instructions understood by a host system. Such blocks can be cached and reused to improve performance.

Dynamic instruction scheduling architectural feature of modern processors supporting out of order instruction execution. It can reduce the number of pipeline stalls but adds to circuit complexity.

Dynamic power range interval between the lower and the upper limit of the device power consumption. A large dynamic range means that the device is able to operate at a lower fraction of its peak power when its load is low.

Dynamic voltage scaling a power conservation technique; often used together with frequency scaling under the name *dynamic voltage and frequency scaling* (DVFS).

Dynamic voltage and frequency scaling (DVFS) power management technique of increasing or decreasing the operating voltage or the clock frequency of a processor in order to increase the instruction execution rate and, respectively, to reduce the amount of heat generated and to conserve power.

E

EC2 Placement Group a logical grouping of instances which allows the creation of a virtual cluster.

Elastic Beanstalk AWS service handling automatically the deployment, the capacity provisioning, the load balancing, the auto-scaling, and the application monitoring functions. It interacts with other AWS services including EC2, S3, SNS, Elastic Load Balance, and AutoScaling.

Elastic Block Store (EBS) AWS service providing persistent block level storage volumes for use with EC2 instances. EBS supports the creation of snapshots of the volumes attached to an instance and then uses them to restart an instance. The storage strategy provided by EBS is suitable for database applications, file systems, and applications using raw data devices.

Elastic Compute Cloud (EC2) AWS service for launching instances of an application under several operating systems, such as several Linux distributions, Windows, OpenSolaris, FreeBSD, and NetBSD.

Elastic IP address AWS feature allowing an EC2 user to mask the failure of an instance and re-map a public IP address to any instance of the account, without the need to interact with the software support team.

Embarrassingly parallel application application when little or no effort is needed to extract parallelism and to run a number of concurrent threads with little communication among them.

Emergence generally understood as a property of a system that is not predictable from the properties of individual system components.

Energy proportional system system enjoying the propery that the energy consumed is proportional with the system's workload.

Enforced modularity software organization supported by the *client-server* paradigm when modules are forced to interact only by sending and receiving messages. The clients and the servers are independent modules and may fail separately. The servers are stateless, they do not have to maintain state information. Servers may fail and then come up without the clients being affected or even noticing the failure.

Error bar a line segment through a point on a graph, parallel to one of the axes, which represents the uncertainty or error of the corresponding coordinate of the point.

Event a change of state of a process or thread.

Event time the wall clock time when the event occurred.

Exception anomalous or exceptional conditions requiring special processing during the execution of a process. An exception breaks the normal flow of execution of a process/thread and executes a pre-registered exception handler from a known memory location provided by the first-level interrupt handler (FLIV).

Exception behavior preservation condition required for dynamic instruction scheduling. Any change in instruction order must not change the order in which exceptions are raised.

Explicitly Parallel Instruction Computing (EPIC) processor architecture allowing the processor to execute multiple instructions in each clock cycle. EPIC implements a form of Very Long Instruction Word (VLIW) architecture.

F

Fabric controller a distributed Windows Azure application replicated across a group of machines which owns all resources in its environment and it is aware of every application; it ensures scaling, load balancing, memory management, and reliability.

Facility layer heating, ventilation, air conditioning (HVAC), power, communications, and other aspects of the physical plant in a data center.

Failover-based software systems systems less affected by data center level failures; such systems only run at one site, but checkpoints are created periodically and sent to backup data centers.

FedRAMP common security model allowing joint authorizations and continuous security monitoring services for Government and Commercial cloud computing systems; intended for multi-agency use. The use of this common security risk model provides a consistent baseline for cloud-based technologies and ensures that the benefits of these cloud-based technologies are effectively integrated across a variety of cloud computing solutions. The risk model will enable the government to "approve once, and use often" by ensuring multiple agencies gain the benefit and insight of the FedRAMP's Authorization and access to service provider's authorization packages.

Field-programmable gate array (FPGA) an integrated circuit designed to be configured, adapted, and programmed in the field to perform a well-defined function.

Fine-grained parallelism concurrency when only relatively small blocks of the code can be executed in parallel without the need to communicate or synchronize with other threads or processes.

FISMA compliant environment environment that meets the requirements of the Federal Information Security Management Act of 2002. The law requires an inventory of information systems, the categorization of information and information systems according to risk level, security controls, a risk assessment, a system security plan, certification and accreditation of the system's controls, and continuous monitoring.

First-In-First-Out delivery delivery rule requiring that messages are delivered in the same order they were sent.

First-level interrupt handler (FLIH) software component of the kernel of an operating system activated in case of an interrupt or exception. It saves the registers of current process in the PCB (Process Control Block), determines the source of interrupt, and initiates the service of the interrupt.

Flash crowds an event which disrupts the life of a very significant segment of the population, such as an earthquake in a very populated area, and dramatically increases the load of computing and communication service; for example, an earthquake increases the phone and Internet traffic.

Flynn's taxonomy classification of computer architectures, proposed by Michael J. Flynn in 1966. Classifies the systems based on the number of control and data flows as: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD).

Flow control mechanism used to control the traffic in a network. Feedback from the receiver forces the sender to transmit only the amount of data the receiver is able to buffer and then process.

Front-end system component of a server system tasked to dispatch the client requests to multiple *back-end* systems for processing.

Full virtualization type of virtualization when each virtual machine runs on an exact copy of the actual hardware.

Future Internet a generic concept referring to all research and development activities involved in the development of new architectures and protocols for the Internet.

G

Geo replication operation when a system runs at multiple sites concurrently.

Gather operation operation supported by vector processing units to deal with sparse vectors. It takes an index vector and fetches the vector elements at the addresses given by adding a base address to the offsets given by the index vector; as a result a dense vector is loaded in a vector register. In parallel computing this operation is supported by MPI (Message Passing Interface) to take elements from many processes and gathers them for a single process.

Global agreement on time a necessary condition to trigger actions that should occur concurrently.

Go or Golang open source compiled, statically typed language like Algol and C; has garbage collection, limited structural typing, memory safety features, and CSP-style concurrent programming.

Guest operating system an operating system that runs under the control of a hypervisor, rather than directly on the hardware.

H

HaLoop extension of MapReduce with programming support for iterative applications and improved efficiency. Adds various caching mechanisms and makes the task scheduler loop-aware.

Hash function function used to map data of arbitrary size to data of fixed size. For example, a hash function can be applied to the name of a file and the n low-order bits of the hash value give the block number of the directory where the file information can be found. *Extensible hashing* is used to add a new directory block.

Hard deadline strict deadline with penalties, expressed precisely as milliseconds, or possibly seconds.

Hardware layer includes computers (CPU, memory), network (router, firewall, switch, network link and interface) and storage components (hard disk), and other physical computing infrastructure elements.

Head-of-line blocking situation when a long-running task cannot be preempted and other tasks waiting for the same resource are blocked.

Hedged requests short-term tail-tolerant techniques; the client issues multiple replicas of the request to increase the chance of a prompt reply.

Hot standby a method to achieve redundancy. The primary and the secondary (backup) system(s) run simultaneously. The data is mirrored to the secondary system(s) in real time so that both systems contain identical information.

Horizontal scaling application scaling by increasing the number of VMs as load increases and by reducing this number when load decreases; most common form of cloud application scaling.

Hybrid cloud an infrastructure consisting of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability.

HyperText Transfer Protocol (HTTP) application-level protocol built on top of the TCP transport protocol. HTTP is used by a web browser (the client) to communicate with the server.

HTTP-tunneling technique most often used as a means of communication from network locations with restricted connectivity. Tunneling means the encapsulation of a network protocol. In this case HTTP acts as a wrapper for the communication channel between the HTTP client and the HTTP server.

Hyper-convergence a software-centric architecture that tightly integrates compute, storage, networking, virtualization, and possibly other technologies in a commodity hardware box supported by a single vendor.

Hypervisor or virtual machine monitor (VMM) software that securely partitions the computer's resources of a physical processor into one or more virtual machines. Each virtual machine appears to be running on the bare hardware, giving the appearance of multiple instances of the same computer, but all are supported on a single physical system.

Hyper-threading term used to describe multiple execution threads possibly running concurrently but on a single core processor.

I

Idempotent action action that repeated several times has the same effect as when the action is executed only once.

IEEE 754 Standard for Floating-Point Arithmetic defines arithmetic formats, interchange formats, rounding rules, operations, and exception handling for floating point numbers.

Incommensurate scaling attribute of complex systems; when the size of the system, or when one of its important attributes, such as speed, increases, or when different system components are subject to different scaling rules.

InfiniBand switched fabric for supercomputer and data center interconnects. The serial link can operate at several data rates: single (SDR), double (DDR), quad (QDR), fourteen (FDR), and enhanced (EDR). The highest speed supported is 300 Gbps.

Infrastructure as a Service (IaaS) cloud delivery model that supplies resources for processing, storage, and communication, and allows the user to run arbitrary software, including operating systems and applications. The user does not manage or control the underlying cloud infrastructure, but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls).

Initial value problem computational problem when all conditions are specified at the same value of the independent variable in the equation.

Input/Output Memory Management Unit (IOMMU) connects the main memory with a DMA-capable I/O bus; it maps device-visible virtual addresses to physical memory addresses and provides memory protection from misbehaving devices.

Instruction flow preservation preservation of the flow of data between the instructions producing results and the ones consuming these results.

Instruction-level parallelism simultaneous execution of independent instructions of an execution thread.

Instruction Set Architecture (ISA) interface between the computer software and the hardware. It defines the valid instructions that a processor may execute. ISA allows the independent development of hardware and software.

Instruction pipelining technique implementing a form of parallelism called instruction-level parallelism within a single core or processor. A pipeline has multiple stages and at any given time several instructions are in different stages of processing. Each pipeline stage requires its own hardware.

Integrated Drive Electronics (IDE) interface for connecting disk drives; the drive controller is integrated into the drive, as opposed to a separate controller on, or connected to, the motherboard.

Intelligent Platform Management Interface (IPMI) standardized computer system interface developed by Intel and used by system administrators to manage a computer system and monitor its operation.

Interoperability capability to communicate, execute programs, or transfer data among various functional units under specified conditions.

Interrupt flag (IF) flag in the EFLAGS register used to control interrupt masking.

J

Jarvis short for *Just A Rather Very Intelligent Scheduler*; used to support Siri.

Java Database Connectivity (JDBC) API for Java defining how a client can access a database.

JobTracker and TaskTracker daemons handling processing of MapReduce jobs in Hadoop.

Journal storage storage for composite objects such as records consisting of multiple fields.

Java Message Service (JMS) middleware of the Java Platform for sending messages between two or more clients.

L

Large-scale-dynamic-data data captured by sensing instruments and controled in engineered, natural, and societal systems.

Last level cache (LLC) the cache called before accessing memory. Multicore processors have multiple level caches. Each core has its own L1 I-cache (instruction cache) and D-cache (data cache). Sometimes two cores share the same unified (instruction+data) L2 cache and all cores share an L3 cache. In this case the highest shared LLC is L3.

Latch a counter that triggers an event when it reaches zero.

Late binding dynamical correlation of tasks with data, depending on the state of the cluster.

Latency the time elapsed from the instance an operation is initiated until the instance its effect is sensed. Latency is context dependent.

LRU (Least Recently Used), MRU (Most Recently Used), and LFU (Least Frequently Used) replacement policies used by memory hierarchies for caching and paging.

Livelock condition appearing when two or more processes/threads continually change their state in response to changes in the other processes and none of the processes can complete execution.

Logical clock abstraction necessary to ensure the clock condition in the absence of a global clock.

Loopback file system (LOFS) virtual file system providing an alternate path to an existing file system. When other file systems are mounted onto an LOFS file system, the original file system does not change.

M

MAC address unique identifier permanently assigned to a network interface by the manufacturer. MAC stands for Media Access Control.

Maintainability a measure of the ease of maintenance of a functional unit.Synonymous with serviceability.

Malicious software (Malware) software designed to circumvent the authorization mechanisms and gain access to a computer system, gather private information, block access to a system, or disrupt the normal operation of a system; computer viruses, worms, spyware, and Trojan horses are examples of malware.

Man-in-the middle attack attacker impersonates the agents at both ends of a communication channel making them believe that they communicate through a secure channel.

Mapping a computation assign suitable physical servers to the application.

Mashup application that uses and combines data, presentations, or functionality from two or more sources to create a service.

Megastore a scalable storage for online services.

Memcaching a general purpose distributed memory system caches objects in main memory.

Message-Digest Algorithm (MD5) cryptographic hash function used for checksums. MD5 produces a 128-bit hash value. SHA-i (Secure Hash Algorithm, $0 \leq i \leq 3$) is a family of cryptographic hash functions; SHA-1 is a 160 bit hash function resembling MD5.

Merkle tree hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children.

Message delivery rule an additional assumption about the channel-process interface; establishes when a message received is actually delivered to the destination process.

Metering providing a measurement capability at some level of abstraction appropriate to the type of service.

Microkernel (μ-kernel) system software supporting only the basic functionality of an operating system kernel including low-level address space management, thread management, and inter-process communication. Traditional operating system components such as device drivers, protocol stacks, and file systems are removed from the microkernel and run in user space.

Middleware software enabling computers of a distributed system to coordinate their activities and to share their resources.

Mode sensitive instructions machine instructions whose behavior is different in the privileged mode.

Modularity basic concept in the design of man-made systems; a system is made out of components, or modules, with well-defined functions. A strong requirement for modularity is to define very clearly the interfaces between modules and to enable the modules to work together. Modularity can be *soft* or *enforced*.

Modularly divisible application application whose workload partitioning is decided a priori and cannot be changed.

Monitor process responsible for determining the state of a system.

Message Passing Interface (MPI) communication standard and communication library for a portable message-passing system.

Multi-homing a strategy to support high availability.

Multiple Instructions, Multiple Data architecture (MIMD) system with several processors/cores that function asynchronously and independently.

N

NAS Parallel Benchmarks benchmarks used to evaluate the performance of supercomputers. The original benchmark included five kernels: IS – Integer Sort, random memory access, EP – Embarrassingly Parallel, CG – Conjugate Gradient, MG – Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive, FT – discrete 3D Fast Fourier Transform, and all-to-all communication.

Network bisection bandwidth network attribute, measures the communication bandwidth between the two partitions when a network is partitioned into two networks of the same size.

Network bisection width minimum number of links cut when dividing a network into two halves.

Network diameter average distance between all pairs of two nodes; if a network is fully-connected its diameter is equal to one.

Network Interface Controller (NIC) the hardware component connecting a computer to a Local Area Network (LAN); also known as a network interface card, network adapter, or LAN adapter.

Network layer layer of a communication network responsible for routing packets through a packet switched network from the source to the destination.

NMap a security tool running on most operating systems to map the network, that is, to discover hosts and services in the network. The systems include *Linux, Microsoft Windows, Solaris, HP-UX, SGI-IRIX* and BSD variants such as *Mac OS X*.

Nonce a random or pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. Each time the authentication challenge response code is presented, the nonces are different, thus replay attacks are virtually impossible.

Non-privileged instruction machine instruction executed in user mode.

O

Object Request Broker (ORB) the middleware which facilitates communication of networked applications.

Ontology branch of metaphysics dealing with the nature of being. Provides the means for knowledge representation within a domain, it consists of a set of domain concepts and the relationships among these concepts.

Open-box platforms traditional hardware designed for commodity operating systems; does not have the same facilities as the *closed-box platforms*.

Open Database Connectivity (ODBC) open standard application API for database access.

Overclocking technique-based on DVFS; increases the clock frequency of processor cores above the nominal rate when the workload increases.

Overlay network a virtual network superimposed over a physical network.

Overprovisioning investment in a larger infrastructure than the *typical* workload warrants.

Oversubscription the ratio of the worst-case achievable aggregate bandwidth among the servers to the total bisection bandwidth of an interconnect.

P

Packet-switched network network transporting data units called *packets* through a maze of *switches* where packets are queued and routed towards their destination.

Pane a well defined area within a window for the display of, or interaction with, a part of that window's application or output.

Paragon Intel family of supercomputers launched in 1992 based on the Touchstone Delta supercomputer installed at CalTech for the Concurrent Supercomputing Consortium.

Parallel slackness method of hiding communication latency by providing each processor with a large pool of ready-to-run threads, while other threads wait for either a message, or for the completion of another operation.

Paravirtualization virtualization when each virtual machine runs on a slightly modified copy of the actual hardware; the reasons for paravirtualization: (i) some aspects of the hardware cannot be virtualized; (ii) to improve performance; (iii) to present a simpler interface.

Passphrase a sequence of words used to control access to a computer system; it is the analog of a password, but provides added security.

Paxos protocols a family of protocols to reach consensus based on a finite state machine approach.

Peer-to-Peer system (P2P) distributed computing system when resources (storage, CPU cycles) are provided by participant systems.

Peripheral Component Interconnect (PCI) computer bus for attaching hardware devices to a computer. The PCI bus supports the functions found on a processor bus, but in a standardized format independent of any particular processor.

Perf profiler tool for Linux 2.6+ systems; it abstracts CPU hardware differences in Linux performance measurements.

Petri nets bipartite graphs used to model the dynamic behavior of systems.

Phase transition thermodynamics concept describing the transformation, often discontinuous, of a system from one phase/state to another, as a result of a change in the environment.

Phishing attacks aiming to gain information from a site database by masquerading as a trustworthy entity.

Physical data container storage device suitable for transferring data between cloud-subscribers and clouds.

Physical resource layer includes all physical resources used to provide cloud services.

Pinhole mapping between the pair *(external address, external port)* and the *(internal address, internal port)* tuple carried by the network address translation function of the router firewall.

Pipelining splitting of an instruction into a sequence of steps that can be executed concurrently by different circuitry on the chip.
Pipeline scheduling separates dependent instruction from the source instruction by the pipeline latency of the source instruction. Its effect is to reduce the number of stalls.
Pipeline stages execution units of a pipeline. A basic pipeline has five stages for instruction execution: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back.
Pipeline stall the delay in the execution of an instruction in an instruction pipeline in order to resolve a hazard. Such stalls could drastically affect the performance.
Platform as a Service (PaaS) cloud delivery model supporting consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications.
Platform architecture layer software layer consisting of compilers, libraries, utilities, and other software tools and development environments needed to implement applications.
Plesiochronous operation operation when different parts of a system are almost, but not quite perfectly, synchronized; for example, when the core logic of a router operates at a frequency different from that of the I/O channels.
Pontryagin's principle method used in optimal control theory to find the best possible control which leads a dynamic system from one state to another, subject to a set of constrains.
Power consumption P of a CMOS-based circuit describes the power consumption function of the operating voltage frequency, $P = \alpha \cdot C_{eff} \cdot V^2 \cdot f$ with: α – the switching factor, C_{eff} – the effective capacitance, V – the operating voltage, and f – the operating frequency.
Privacy the assured, proper, and consistent collection, processing, communication, use and disposition of personal information and personally-identifiable information.
Private cloud infrastructure operated solely for the benefit of one organization; it may be managed by the organization or a third party and may exist on the premises or off the premises of the organization.
Privileged instructions machine instruction that can only be executed in kernel mode.
Process a program in execution.
Process group collection of cooperating processes.
Process/thread state ensemble of information needed to restart a process/thread after it was suspended.
Process or application virtual machine virtual machine running under the control of a normal OS and providing a platform-independent host for a single application, e.g., Java Virtual Machine (JVM).
Public-Key Infrastructure (PKI) model to create, distribute, revoke, use, and store digital certificates.
Pull paradigm distributed processing when resources are stored at the server site and the client pulls them from the server.

Q

Quick emulator (QEMU) a machine emulator; it runs unmodified OS images and emulates the guest architecture instructions on the host architecture it runs on.

R

Rapid provisioning automatically deploying cloud system based on the requested service, resources, and capabilities.
Recommender system system for predicting the preference of a user for an item by filtering information from multiple users regarding that item; used to recommend research articles, books, movies, music, news, and any imaginable item.
Red-black tree a self-balancing binary search tree where each node has a "color" bit (red or black) to ensure the tree remains approximately balanced during insertions and deletions.
Reference data infrequently used data, such as archived copies of medical or financial records, customer account statements, and so on.
Reliability measure of the ability of a functional unit to perform a required function under given conditions for a given time interval.
Remote Procedure Call (RPC) procedure for inter-process communication. RPC allows a procedure on a system to invoke a procedure running in a different address space, possibly on a remote system.
Resilience ability to reduce the magnitude and/or duration of the events disruptive to critical infrastructure.
Resilient Distributed Dataset (RDD) storage concept allowing a user to keep intermediate results and optimize their placement in the memory of a large cluster; used for fault-tolerant, parallel data structures.
Resource abstraction and control layer software elements used to realize the infrastructure upon which a cloud service can be established, such as hypervisor, virtual machines, virtual data storage.

Resource scale-out allocation of more servers to an application.

Resource scale up allocation of more resources to servers already allocated to an application.

Reservation station hardware used for dynamic instruction scheduling. A reservation station fetches and buffers an operand as soon as it becomes available. A pending instruction designates the reservation station it will send its output to.

Response time the time from the instance a request is sent until the response arrives.

Round-Trip Time (RTT) the time it takes a packet to cross the network from the sender to the receiver and back. Used to estimate the network load and detect network congestion.

Run total ordering of all the events in the global history of a distributed computation consistent with the local history of each participant process.

runC implementation of the Open Containers Runtime specification and the default executor bundled with Docker Engine.

S

Same Program Multiple Data (SPMD) parallel computing paradigm when multiple instances of one program run concurrently and each instance processes a distinct segment of the input data.

Scala general-purpose programming language supporting functional programming and a strong static type system. Scala code is compiled as Java byte code and runs on JVM (Java Virtual Machine).

Scalability ability of a system to grow without affecting its global function(s).

Scatter operation vector processing operation, the inverse of a gather operation, it scatters the elements of a vector register to addresses given by the index vector and the base address. In distributed computing MPI scatters data from one processor to a number of processors.

Searchable symmetric encryption (SSE) encryption method used when an encrypted databases is outsourced to a cloud or to a different organization. It supports conjunctive search and general Boolean queries on symmetrically encrypted data. SSE hides information about the database and the queries.

Secondary spectrum data falsification (SSDF) in software-defined radio the occupancy report from a malicious node showing that channels used by the primary node are free.

Security accreditation the organization authorizes (i.e., accredits) the cloud system for processing before operations and updates the authorization when there is a significant change to the system.

Security assessment risk assessment of the management, operational, and technical controls of the cloud system.

Security certification certification for the accreditation of a cloud system.

Self-organization process where some form of global order is the result of local interactions between parts of an initially disordered system. No single element acts as a coordinator and the global patterns of behavior are distributed.

Semantic Web term coined by Tim Berners-Lee to describe "a web of data that can be processed directly and indirectly by machines."

Sensitive instructions machine instructions behaving differently when executed in kernel and in user mode.

Sequential write-sharing condition when a file cannot be opened simultaneously for reading and writing by several clients.

Service aggregation operation when an aggregation brokerage service combines multiple services into one or more new services.

Service arbitrage/service aggregation grouping of cloud services. In service aggregation the services being aggregated are not fixed. Arbitrage provides flexibility and opportunistic choices for the service aggregator, e.g., provides multiple e-mail services through one service provider, or provides a credit-scoring service that checks multiple scoring agencies and selects the best score.

Service deployment activities and organization needed to make a cloud service available.

Service intermediation operation when an intermediation broker provides a service that directly enhances a given service delivered to one or more service consumers.

Service interoperability the capability to communicate, execute programs, or transfer data among various cloud services under specified conditions.

Service layer defines the basic services provided by cloud providers.

Service Level Agreement (SLA) a negotiated contract between the customer and the service provider explaining expected quality of service and legal guarantees. An agreement usually covers: services to be delivered, performance, tracking and reporting, problem management, legal compliance and resolution of disputes, customer duties and responsibilities, security, handling of confidential information, and termination.

Shared channel architecture network organization when all physical devices share the same bandwidth; the higher the number of devices connected to the channel the less bandwidth is available to each one of them.

Shard a horizontal partitioning of a database, a row in a table structured data.
Simple Object Access Protocol (SOAP) an application protocol developed in 1998 for web applications.
Singular Value Decomposition (SVD) given an $m \times n$ matrix $A = [a_{ij}]$, $1 \le i \le n, 1 \le j \le m$ with entries either real or complex numbers, $a_{ij} \in \mathbb{R}$ or $a_{ij} \in \mathbb{C}$, there exits a factorization

$$A = U \Sigma V^* \tag{259}$$

where: U is an $m \times n$ unitary matrix, Σ is a diagonal $m \times n$ matrix with non-negative real numbers on the diagonal, V is an $n \times n$ unitary matrix over the field, $\mathbb{R}$ or $\mathbb{C}$, and V^* is the complex conjugate transpose of V.
SLA management the ensemble of activities related to SLAs including SLA contract definition (basic schema with the quality of service parameters), SLA monitoring, and SLA enforcement.
Service management all service-related functions necessary for the management and operations of those services required by customers.
Service provider entity responsible for making a service available to service consumers.
Service orchestration the arrangement, coordination, and management of cloud infrastructure to provide different cloud services to meet IT and business requirements.
Servless computer service AWS service when applications are triggered by conditions and/or events specified by the end user. Lambda is an example of such service.
Shared cluster state a resilient master copy of the state of all cluster resources.
Sigmoid function $S(t)$ an "S-shaped" function defined as $S(t) = \frac{1}{1-e^{-t}}$. Its derivative can be expressed as function of itself, $S'(t) = S(t)(1 - S(t))$.
Simple DB AWS non-relational data store that allows developers to store and query data items via web services requests. It creates multiple geographically distributed copies of each data item and supports high performance web applications.
Simple Queue Service (SQS) AWS service for hosted message queues. It allows multiple EC2 instances to coordinate their activities by sending and receiving SQS messages.
Simple Storage System (S3) AWS storage service for large objects. It supports a minimal set of functions: write, read, and delete. S3 allows an application to handle an unlimited number of objects ranging in size from one byte to five terabytes.
Single Instruction, Single Data architecture (SISD) computer architecture supporting the execution of a single thread or process at any given time. Individual cores of a modern multicore processor are SISD.
Single Instruction, Multiple Data architecture (SIMD) computer architecture when one instruction processes multiple data elements. Used in vector processing.
S/KEY password system based on Leslie Lamport scheme. The real password of the user is combined with a short set of characters and a counter that is decremented at each use to form a single-use password. Used by several operating systems including, Linux, OpenBSD, and NetBSD.
Skype communication system using a proprietary voice-over-IP protocol. The system developed in 2003 was acquired by Microsoft in 2011. Nowadays it is a hybrid P2P and client-server system. It allows close to 700 million registered users from many countries around the globe to communicate.
Simultaneous multithreading (SMT) architectural feature allowing instructions from more than one thread to be executed in any given pipeline stage at a time.
Simple Mail Transfer Protocol (SMTP) application protocol defined in the early 1980s to support Email services.
Snapshot isolation guarantee that all reads made in a transaction will see a consistent snapshot of the database.
Soft deadline deadline that can be missed by fractions of the units. It is more of a guideline, no penalties are involved.
Soft modularity dividing a program into modules which call each other and communicate using shared memory or follow the procedure call convention. It hides the details of the implementation of a module. Once the interfaces of the modules are defined, the modules can be independently developed even in different programming languages, replaced, and tested.
Software as a Service (SaaS) cloud delivery model when cloud applications are accessible from various client devices through a thin client interface such as a web browser. The user does not manage or control the underlying cloud infrastructure.
Software development kit (SDK) a set of software tools for the creation of applications in a specific software environment.
Speed term used informally to describe the maximum data transmission rate, or the capacity of a communication channel; this capacity is determined by the physical bandwidth of the channel and this explains why the term channel "bandwidth" is also used to measure the channel capacity, or the maximum data rate.
Speedup measure of parallelization effectiveness.

SQL injection attack typically used against a web site; an SQL command entered in a web form causes the contents of a database used by the web site to be altered or to be dumped to the attacker site.

ssh (Secure Shell) network protocol that allows data to be exchanged using a secure channel between two networked devices; *ssh* uses public-key cryptography to authenticate the remote computer and allow the remote computer to authenticate the user. It also allows remote control of a device.

Streaming SIMD Extension (SSE) SIMD instruction set extension to the x86 architecture introduced by Intel in 1999. Its latest expansion is SSE4. It supports floating point operations and has a wider application than the MMX introduced in 1996.

Store-and-forward network packet switched network where a router buffers a packet, verifies its checksum, and then forwards it to the next router along the path from its source to the destination.

Structural hazards in pipelining hazards occurring when a part of the processor hardware is needed by two or more instructions at the same time.

Structured Query Language (SQL) special-purpose language for managing structured data in a relational database system (RDBMS). SQL has three components: a data definition language, a data manipulation language, and a data control language.

Structured overlay network network where each node has a unique key which determines its position in the structure. The keys are selected to guarantee a uniform distribution in a very large name space. Structured overlay networks use *key-based routing* (KBR); given a starting node v_0 and a key k the function $KBR(v_0, k)$ returns the path in the graph from v_0 to the vertex with key k.

Superscalar processor processor able to execute more than one instruction per clock cycle.

System history information about the past system evolution expressed as a sequence of events, each event corresponding to a change of the state of the system.

System portability the ability of a service to run on more than one type or size of cloud.

T

Task-level parallelism parallelism when the tasks of an application run concurrently on different processors. A job consists of multiple tasks scheduled either independently or co-scheduled when they need to communicate with one another.

Timestamps patterns used for event ordering using a global time-base constructed on local virtual clocks.

Thread of execution the smallest unit of processing that can be scheduled by an operating system.

Thread-level parallelism term describing the data-parallel execution using a GPU. A thread is a subset of vector elements processed by one of the lanes of a multithreaded processor.

Thread block scheduler GPU control software assigning thread blocks to multithreaded SIMD processors.

Thread scheduler GPU control software running on each multithreaded SIMD processor to assign threads to the SIMD lanes.

Three-way handshake process to establish a TCP connection between the client and the server. The client provides an arbitrary initial sequence number in a special segment with the *SYN* control bit on; then the server acknowledges the segment and adds its own arbitrarily chosen initial sequence number; finally, the client sends its own acknowledgment *ACK* as well as the HTTP request and the connection is established.

Threshold value of a parameter related to the system state that triggers a change in the system behavior.

Thrift framework for cross-language services.

Top-Down methodology hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application.

TPC BenchmarkH (TPC-H) decision support benchmark relevant for applications that examine large volumes of data and execute queries with a high degree of complexity; it consists of a suite of business oriented ad hoc queries and concurrent data modifications with broad industry-wide relevance.

TPC-DS de facto industry standard benchmark for assessing the performance of decision support systems.

Trusted application application with special privileges for performing security related functions.

TCP segmentation offload (TSO) procedure enabling a network adapter to compute the TCP checksum on transmit and receive, saves the host CPU the overhead for computing the checksum; large packets have larger savings.

Translation look-aside buffer (TLB) cache for dynamic address translation; it holds the physical address of recently used pages in virtual memory.

Transport layer network layer responsible for end-to-end communication, from the sending host to the destination host.

Trusted Computer Base (TCB) totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy.

Turing complete computer model of computation equivalent to a universal Turing machine except for memory limitations.

Tera Watt Hour (TWh) measure of energy consumption, one TWh is equal to 10^9 KWh.

U

Ubuntu open source operating system for personal computers. Ubuntu is an African humanist philosophy; "ubuntu" is a word in the Bantu language of South Africa meaning "humanity towards others."

Unbounded input data concept related to data streaming; the computing engine processes a dynamic data set where one never knows if the set is complete as new records are continually added and old ones are retracted.

Usability extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

Utility function relates the "benefits" of an activity or service with the "cost" to provide the service.

V

Vector computer computer operating on vector registers holding as many as 64 or 128 vector elements. Vector functional units carry out arithmetic and logic operations using data from vector registers as input and disperse the results back to memory.

Vector length register register of a SIMD processor for handling of vectors whose length is not a multiple of the length of the physical vector registers.

Vector mask register register of a SIMD processor used by conditional statements to disable/select vector elements.

Vertical scaling method to increase the resources of a cloud application. It keeps the number of VMs of an application constant, but increases the amount of resources allocated to each one of them.

Virtual Machine (VM) an isolated environment with access to a subset of the physical resources of a computer system. Each virtual machine appears to be running on the bare hardware, giving the appearance of multiple instances of the same computer, though all are supported by a single physical system.

Virtual Private Cloud (VPC) cloud organization providing a connection, via a Virtual Private Network, between an existing IT infrastructure of an organization and a set of isolated compute resources in the AWS cloud.

Virtual time warp abstraction allowing a thread to acquire an earlier effective virtual time, in other words, to borrow virtual time from its future CPU allocation.

Virtualization abstraction of hardware resources.

Virtualized infrastructure layer software elements, such as hypervisors, virtual machines, virtual data storage, and supporting middleware components used to realize the infrastructure upon which a computing platform can be established. While virtual machine technology is commonly used at this layer, other means of providing the necessary software abstractions are not precluded.

W

Work-conserving scheduling policy scheduling policy when the server cannot be idle while there is work to be done.

WebSphere Extended Deployment (WXD) middleware supporting setting performance targets for individual web applications and for the monitor response time.

Where-provenance information describing the relationship between the source and the output locations of data in a database.

Why-provenance information describing the relationship between the source tuples and the output tuples in the result of a database query.

Wide Area Network (WAN) packet switched network connecting systems located throughout a very large area.

Witness of database record the subset of database records ensuring that the record is the output of a query.

Work-conserving scheduler scheduler with the goal of keeping the resources busy, if there is work to be done; a *non-work conserving scheduler* may leave resources idle while there is work to be done.

Workflow description of a complex activity involving an ensemble of multiple interdependent tasks.

Write-ahead database technique when updates are written to persistent storage only after the log records have been written.

X

x86-32, i386, x86 and IA-32 CISC-based instruction set architecture of Intel processors. Now supplanted by x86-64 which supports vastly larger physical and virtual address spaces. The x86-64 specification is distinct from Itanium, initially known as IA-64 architecture.

x86 architecture architecture of Intel processors supporting memory segmentation with a segment size of 64K. The CR (code-segment register) points to the code segment. *MOV, POP*, and *PUSH* instructions serve to load and store segment registers, including CR.

Z

Zero-configuration network (zeroconf) computer network based on the TCP/IP and characterized by automatic assignment of numeric network addresses for networked devices, automatic distribution and resolution of computer hostnames, and automatic location of network services.

ZooKeeper a distributed coordination service implementing a version of the Paxos consensus algorithm.

Index