Cloud Computing and Computer Clouds

Dan C. Marinescu
Computer Science Division
Department of Electrical Engineering & Computer Science
University of Central Florida, Orlando, FL 32816, USA
Email: dcm@cs.ucf.edu

February 3, 2012

Contents

1 Preface

The contents of this book represent a series of lectures given in the Fall 2011 to a graduate level class on cloud computing.

Cloud computing is a movement started sometime during the middle of the first decade of the new millennium; the movement is motivated by the idea that information processing can be done more efficiently on large farms of computing and storage systems accessible via the Internet. The appeal of cloud computing is that it 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 cost effective because of the multiplexing of resources. 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 security are ensured by service providers; the service providers can operate more efficiency due to specialization and centralization.

The cloud movement is not without skeptics and critics. The skeptics question what is actually a cloud, what is new, how does it differ from other types of large-scale distributed system, why cloud computing could be successful when grid computing had only limited success. For example, Larry Ellison, the CEO of Oracle, was quoted in the Wall Street Journal on September 26, 2008: "the interesting thing about Cloud Computing is that we've redefined Cloud Computing to include everything that we already do...;" Andy Iserwood, the Vice-President of HP's European Software Sales, was quoted in the ZDnet News on December 11, 2008 as saying: "a lot of people are jumping into the [cloud] bandwagon, but I have not heard two people say the same things about it. The are multiple definitions of the cloud."

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. Cloud security and the privacy of data are major concerns for the cloud computing users.

The applications discussed in Chapter 12 were developed by several students as follows: Tim Preston contributed to 12.2: Shameek Bhattacharjee to 12.3 and 12.9, Charles Schneider to 12.4, Michael Riera to 12.5, Kumiki Ogawa to 12.6, Wei Dai to 12.7, Gettha Priya Balasubramanian to 12.8.

2 Introduction

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. The network-centric computing idea led to the *Grid computing* movement in the early 1990s and, since 2005, to *utility computing* and *computer clouds*.

In *utility computing* the hardware and the software resources are concentrated in large data centers and the users can pay as they consume computing, storage, and communication resources. While utility computing often requires a cloud-like infrastructure, its focus is on the business model for providing the computing services is based; cloud computing is a path to utility computing embraced by major IT companies such as IBM, HP, Amazon, Oracle, Microsoft, Sun Microsystems, and others.

We can summarize the main features of cloud computing as follows:

- Cloud computing uses Internet technologies to offer scalable and elastic services; the term "elastic computing" refers to the ability to dynamically acquire computing resources and to support a variable workload.

- 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.

- The service providers can operate more efficiency due to specialization and centralization.

- Cloud computing is cost-effective because of the multiplexing of resources; lower costs for the service provider are past 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, as well as security and lowers communication costs.

Cloud computing is a technical and social reality and, at the same time, it is 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, is 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 a 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.

In the early 2011 Apple announced the `iCloud,` a network centric alternative for content such as music, videos, movies, and personal information; such content was previously confined to personal devices such as workstations, laptops, tablets, or smart phones. 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 [99], in other words, to store it in the cloud where the application runs. This insight lead us to believe that several new classes of cloud computing applications could emerge in the next few years [24].

As always, a good idea has generated a high level of excitement translated into a flurry of articles some of a scholarly depth, others with little merit or even bursting with misinformation. In this set of lecture notes 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 we 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; for this reason we start our discussion with an overview of parallel computing and distributed systems.

2.1 Network-centric computing and network-centric content

The concepts and technologies for network-centric computing and content evolved along the years and 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 Grid, initiated in the early 1990s by National Laboratories and Universities; used primarily for applications in the area of science and engineering.
- Computer clouds, promoted since 2005 as a form of service-oriented computing by large IT companies; used for enterprise computing, high-performance computing, Web hosting, storage for network-centric content.

The need to share data from high energy physics experiments motivated Sir Tim Berners-Lee who worked at CERN in the 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. 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. 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, as well as 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 paradigm 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 the early 1990s to facilitate access to grid services.

The goal to give to a grid user the illusion of a very large virtual supercomputer was hindered by the autonomy of the individual systems and the fact that they were connected by a wide-area network with limited bandwidth and latency considerably higher than those of the interconnection network of a physical supercomputer. Nevertheless, several Grand Challenge problems, such as protein folding, financial modelling, earthquake simulation, and climate/weather modelling, 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 generate 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 were expected to cooperate seamlessly. Heterogeneity of the hardware and of the system software poses insurmountable challenges for application mobility, the optimization of resource allocation, fault-tolerance, and other critical areas of system management. The fact that resources were in different administrative domains further complicated many already difficult problems related to security and resource management. While very popular in the science and engineering community, the grid movement did not address the major concerns of enterprise computing community and did not make a noticeable impact for 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. Cloud computing seems 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, Navys Next Generation Enterprize 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.

Network-centric content. 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. *Information* is the result of functions applied to content. The creation and consumption of audio and visual content is likely to transform the Internet to support increased quality in terms of resolution, frame rate, color depth, stereoscopic information and it seems reasonable to assume that the Future Internet will be content-centric. The content should be treated as having meaningful semantic connotations rather than a string of bytes; the focus will be 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.

Network-centric computing and network-centric contents share a number of characteristics:

- Most applications are data intensive; computer simulation becomes 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 industry. The widespread use of sensors contribute to the increase of the volume of data; multimedia applications are increasingly more popular.

- Virtually all applications are network-intensive; indeed, transferring large volumes of data requires high bandwidth networks; parallel computing, computation stirring, data streaming require low latency networks.

- The systems are accessed using *thin clients* running on systems with limited resources. In June 2011 Google releases 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.

The advantages of network-centric computing and network-centric content paradigms are, at the same time, sources for concern; we discuss some of them:

- Computing and communication resources (CPU cycles, storage, network bandwidth) are shared and resources can be aggregated to support data-intensive applications;

when multiple applications share a system their peak demands for resources are not synchronized thus, multiplexing leads to a higher resource utilization. On the other hand, the management of large pools of resources poses new challenges as complex systems are subject to phase transitions; new resource management strategies such as self-organization, and decisions based on approximate knowledge of the state of the system must be considered. Ensuring Quality of Service (QoS) guarantees is extremely challenging in such environments as total performance isolation is elusive.

- 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. Data sharing poses not only security and privacy challenges but also requires mechanisms for access control for authorized users, as well as, 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 costs for a private computing infrastructure as well as, the significant maintenance and operation costs.

- User convenience and elasticity, the ability to accommodate workloads with very large peak to average ratios.

It is very hard to point out a single technological or architectural development that triggered the movement towards network-centric computing and 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 to support distributed and parallel computing.

Along the years we have witnessed the breathtaking evolution of solid state technologies which led to the development of multi- and many-core processors; quad-core processors such as the AMD Phenom II X4, the Intel i3, i5, and i7, and hexa-core processors such as AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X are now used in servers populating computer clouds. The proximity of multiple cores on the same die allows the cache coherency circuitry to operate at a much higher clock-rate than it would be possible if the 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 and over 4GB/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 in the range of $30. Optical storage technologies and flash memories are widely used nowadays.

The *three-tier model* is a software architecture and a software design pattern. The *presentation tier* is the topmost level of the application; typically, it runs on a desktop PC or workstation and uses a standard graphical user interface and displays information related to services such as browsing merchandize, purchasing, and shopping cart contents. The presentation tier communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. The *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*. The *data tier* controls the servers where the information is stored; it runs an RDBMS 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 of the tiers can be replaced independently; for example, a change of operating system in the presentation tier would only affect the user interface code.

2.2 Cloud computing - an old idea whose time has come

Once the technological elements were in place it was only a matter of time until the economical advantages of cloud computing became apparent. Table 1 from [24] shows that due to the economy of scale, large data centers based on commodity computers experience a 5 to 7 times decrease of resource consumption including energy, compared to medium size centers.

Table 1: The costs for the computing and communication infrastructure for medium and large data centers: (a) networking - in dollars per Mbit/sec/month; (b) storage - dollars per GByte/month; and (c) system administrators. The ratio of the costs for medium size (with around 1,000 systems) versus large (with more than 50,000 systems) data centers.

Resource	Medium size	Large	Ratio
Networking	95	13	7.1
Storage	2.20	0.40	5.7
System admin	One per 140 systems	One for more than 1,000 systems	7.1

Data centers could be placed at sites with low energy cost (e.g., 1 KWh costs in Idaho 3.6 cents, in California 10 cents, and in Hawaii 18 cents). Storing and managing large amounts of data lead to energy inefficiency because of the power consumption and of energy used for cooling the data centers. For example, there are $6,000$ data centers in the U.S and reportedly consumed 1.5% of all electric energy in the U.S. at a cost of $4.5 billion and 61 billion KWh in 2006. The power demanded by data centers was predicted to double from 2006 to 2011. Peak instantaneous demand was predicted to increase from 7 GW in 2006 to 12 GW in 2011, requiring the construction of 10 new power plants.

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 the premises or off the premises of the organization.

- 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 could provide the computing resources needed for a large organization, e.g., a research institution, a university, or a corporation. [24] argues that a private cloud does not support utility computing when the user pays as it consumes resources. 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 limited to a specific domain.

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; indeed, 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 homogeneous set of hardware and software resources in a single administrative domain. Security, resource management, fault-tolerance, and quality of service are less challenging than in a heterogeneous environment with resources in multiple administrative domains.
- It is focused on enterprise computing; the adoption of cloud computing by industrial organizations, financial institutions, healthcare organizations and so on, has a potential 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 breakthrough that have made feasible cloud computing, 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 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[1]. A temporary fix with negative economical implications is *overprovisioning*, i.e., having enough resources to satisfy the largest projected demand.

- Data lock-in; once a customer is hooked to one provider it is hard to move to another. The standardization efforts at NIST attempt to answer this problem.

- Data confidentiality and auditability; this is indeed a serious problem. We analyze it in Section 10.

- Data transfer bottlenecks; many applications are data-intensive. A very important strategy is to store the data as close to the site where it is needed as possible. 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 isolations are discussed in Section 6.4.

- 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 a promising avenue.

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

2.3 Cloud computing paradigms and services

According to the NIST reference model in Figure 1 [176] 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

[1]SLAs are discussed in Section 4.5.

how well it conforms to a set of established criteria. For example, a security audit evaluates cloud security, a privacy-impact audit evaluates the privacy-impact, while a performance audit evaluates the cloud performance.

We start with the observation that it is difficult to distinguish the services associated with cloud computing from those that any computer operations center would include [227]. 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. Three paradigms that can be identified with computing services payed on demand, SaaS, PaaS, and IaaS, will be discussed next.

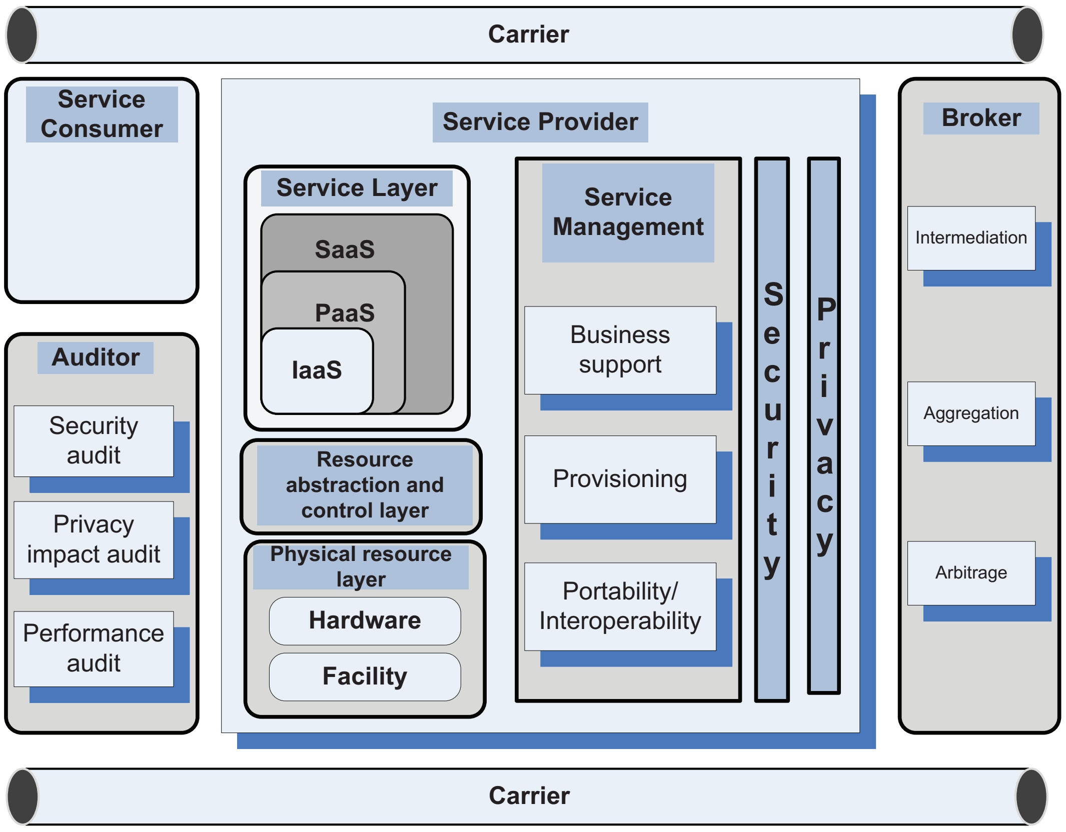


Figure 1: The entities involved in a service-oriented computing and, in particular, in cloud computing according to the NIST chart. The carrier provides connectivity between service providers, service consumers, brokers, and auditors.

a. *Software as a Service (SaaS)* - the capability to use applications supplied by the service provider in a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The user does 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 (CRM), 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.

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

- Many competitors use the same product, such 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.

b. *Platform as a Service (PaaS)* gives the capability to deploy consumer-created or acquired applications 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 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 particulary 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 the area of software development when multiple developers and users have to work together and the there is a need to automate the deployment and testing services.

c. *Infrastructure as a Service (IaaS)* - 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 select networking components (e.g., host firewalls). Services offered by this paradigm include: server hosting, web servers, storage, computing hardware, operating systems, virtual instances, load balancing, Internet access, and bandwidth provisioning.

Other classes of services offered by cloud computing are necessary to support the following paradigms:

1. Service management and provisioning including: 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 including: 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 including data management and development.

The *IaaS* cloud computing paradigm 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 particulary 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.

2.4 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: (i) the control is relinquished to third party services; (ii) the data is stored on multiple sites administered by several organizations; and (iii) multiple services interoperate across the network.

Unauthorized access, data corruption, infrastructure failure, or unavailability are some of the risks related to relinquishing the control to third party services; moreover, it is difficult to identify the source of the problem and the entity causing it. Systems can span the boundaries of multiple organizations and cross the security borders, a process is 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" [239].

The complex structure of cloud services can make it difficult to determine who is responsible in case something undesirable happens. In a complex chain of events or systems, many entities contribute to an action with undesirable consequences, some of them have the opportunity to prevent these consequences, and therefore 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 or ability of individuals to exercise personal control over the collection, use and disclosure of their personal data by others; this tests the confidence and trust in todays 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 and they 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 therefore will be determined by privacy issues addressed by these companies and the countries where the data centers are located. Privacy is affected by cultural differences, while Western cultures favor privacy, other cultures emphasis community; this leads to an ambivalent attitude towards the question of 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. But the need for rules and regulations for the governance of cloud computing are obvious. The term *governance* means the manner in which 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; indeed, private companies are less constraint by ethics oversight and governance arrangements are more conducive to profit generation.

Accountability is a necessary ingredient for cloud computing; adequate information about how data is handled within the cloud and about allocation of responsibility are key elements to 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.

2.5 Cloud storage diversity and vendor lock-in

When a large organization relies solely on a single cloud provider there are several risks involved. As the short history of cloud computing shows, cloud services may be unavailable for a short, or even for an extended period of time; such an interruption of service is likely to impact negatively the organization and possibly diminish, or cancel complectly, the benefits of utility computing for that organization. The potential for permanent data loss in case of a catastrophic system failure poses an equally 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 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 tera or possibly peta bytes of data over the network takes a fairly long time, it incurs substantial charges for the bandwidth used and requires substantial manpower.

Chapter 4 discusses the storage models supported by the cloud infrastructure provided by Amazon, Google, and Microsoft and Chapter 9 covers the architecture of storage systems. Reliability is a major concern and in this section we discuss a solution which avoidance of vendor lock-in and addresses storage reliability.

A solution to guard 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. A more effective solution is 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 striping with distributed parity, over a disk array, see Figure 2(a); the disk controller distributes the 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 is lost then we still have all the blocks of the third file, $c1, c2$, and $c3$ and we can recover the missing blocks for the others as follows:

$$\begin{array}{c} a2 = (a1) \text{ XOR } (aP) \text{ XOR } (a3) \\ b2 = (b1) \text{ XOR } (bP) \text{ XOR } (b3) \\ d1 = (dP) \text{ XOR } (d2) \text{ XOR } (d3) \end{array} . \tag{1}$$

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(b) strips the 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, it 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 raises immediately 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 question is reported [5]; the RACS system uses the same data model and mimics the interface to the *S3* (Simple Storage System) provided by AWS (Amazon Web Services). The *S3* system, discussed in Section 4.1, 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 [5] led the authors to conclude that the costs increases and the performance penalties of the RACS systems are relatively minor. The paper also suggests an implementation to avoid the single point of failure. Several proxies under allow clients to access the data stored on multiple clouds; the system is able to recover from the failure of a single proxy.

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 next future. The pervasive nature of scalability dominates all aspects of cloud management and cloud applications; solutions which perform well on small systems are are no longer feasible when the number of systems, or the volume of the input data of an application, increase by one or more orders of magnitude. Experimentations 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 particular data-intensive application.

We can also conclude that cloud computing poses difficult 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 7. At the same

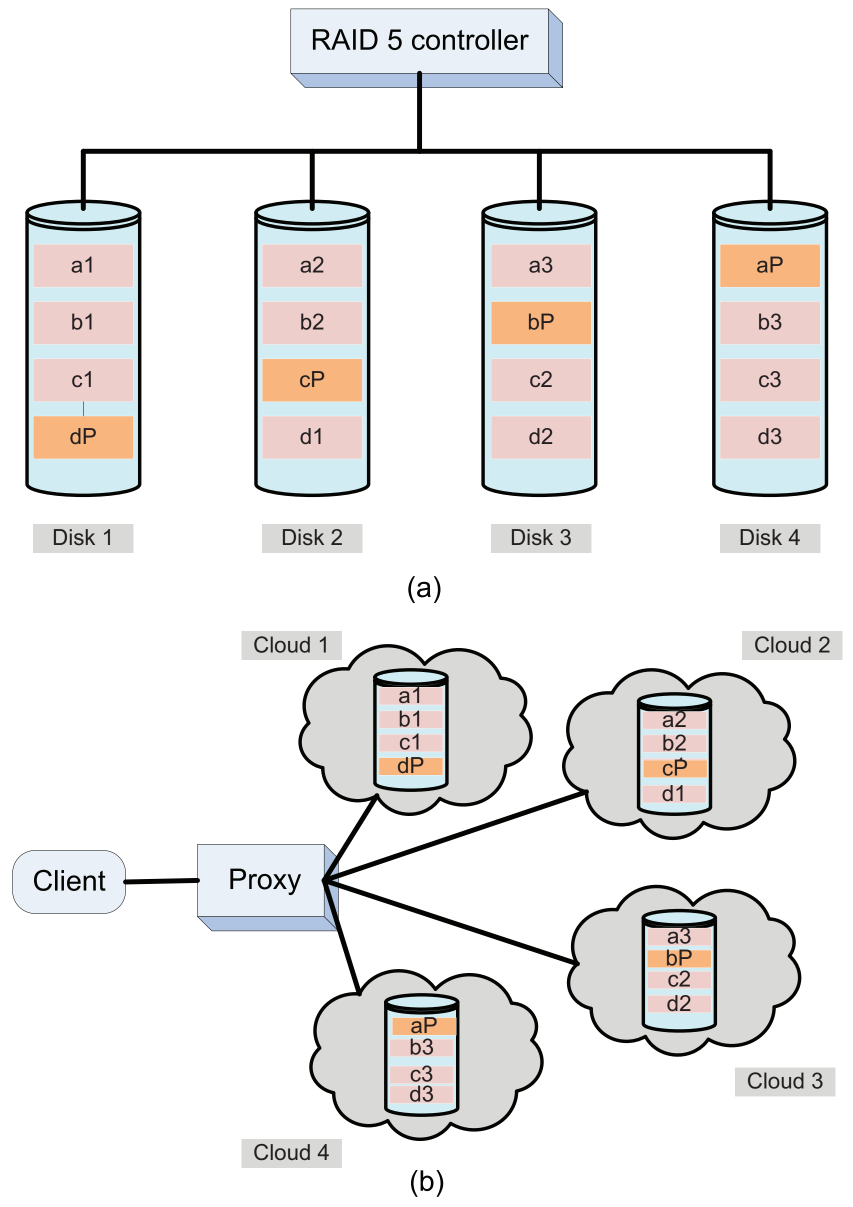


Figure 2: (a) A $(3, 4)$ RAID-5, configuration where individual blocks are stripped over three disks and a parity block is added; the parity block is constructed by `XOR`-ing the data blocks, e.g., $aP = a1$ XOR $a2$ XOR $a3$. The parity blocks are distributed among the 4 disks, aP is on disk 4, bP on 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.

time, the cloud application developers have to be aware of the limitations of the cloud computing model.

2.6 Further readings

A very good starting point for understanding the major issues in cloud computing is the 2009 paper "Above the clouds: a Berkeley view of cloud computing" [24]. The standardization effort at NIST is described by a wealth of documents [175, 176, 177, 178, 179, 180, 181, 182, 183] on the Web site http://collaborate.nist.gov.

A recent book by Hwang, Fox, and Dongarra [115] is a good introduction to a broad range of topics in the area of distributed systems, including grids, peer-to-peer systems, and clouds.

2.7 History notes

John McCarthy was a visionary in computer science; in the early 1960s he formulated the idea that computation may be organized as a public utility, like water and electricity.

Google was funded by Larry Page and Sergey Brin who, at the time, were graduate students in Computer Science at Stanford University; in 1998 the company was incorporated in California after receiving a contribution of $100,000 from the co-funder and chief hardware designer of SUN Microsystems, Andy Bechtolsheim.

Amazon *EC2 (Elastic Computing)* was initially released as a limited public beta cloud computing service on August 25, 2006. The system was developed by a team from Cape Town, South Africa.

In October 2008 Microsoft announced the *Windows Azure* platform; in June 2010 the platform becomes commercially available.

iCloud is a cloud storage and cloud computing service from Apple Inc. announced on June 6, 2011.

3 Basic concepts

Cloud computing is based on a large number of ideas and experience accumulated since the first electronic computer was used to solve computationally challenging problems. In this chapter we overview concepts in parallel and distributed systems important for understanding the basic challenges in the design and use of computer clouds.

Cloud computing is intimately tied to parallel and distributed computing. All cloud applications are based on the client-server paradigm with a relatively simple software, a thin-client, running on the user's machine while the computations are carried out on the cloud. Many cloud applications are data-intensive and use a number of instances which run concurrently. Transaction processing systems, e.g., 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 an in-order delivery of messages.

The concepts introduced in this section are very important in practice. Communication protocols which support coordination of distributed processes travel through noisy and unreliable communication channels which may lose messages or deliver duplicate, distorted, or out of order messages. To ensure reliable and in order delivery of messages such protocols stamp each message with a sequence number; in turn, a receiver sends an acknowledgment with its own sequence number to confirm the receipt of a message. The clocks of a sender and a receiver may not be synchronized thus these sequence numbers act as logical clocks. Timeouts are used to request the retransmission of lost or delayed messages.

The concept of consistent cuts and distributed snapshots are at the heart of *checkpoint-restart* procedures for long-lasting computations. Indeed, many cloud computations are data-intensive and run for extended periods of time on multiple computers in the cloud. 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.

Many functions of a computer cloud require information provided by *monitors*, system components which collect state information from the individual systems. For example, controllers for cloud resource management discussed in Chapter 7 require accurate state information; security and reliability can only be implemented using information provided by specialized monitors. Coordination of multiple instances is a critical function of an application controller.

3.1 Parallel computing

As demonstrated by nature, the ability to work in parallel as a group represents a very efficient way to reach a common target; the human beings have learned to aggregate themselves, and to assemble man-made devices in organizations where each entity may have modest ability, but a network of entities can organize themselves to accomplish goals that an individual entity cannot. Thus, we should not be surprised that the thought that individual computing systems should work in concert for complex applications was formulated early on.

Parallel computing allows us to solve large problems by splitting them into smaller ones and solving them concurrently. Parallel computing was considered for many years the holly grail for solving data-intensive problems encountered in many areas of science, engineering, and enterprise computing; it requires major advances in several areas including, algorithms, programming languages and environments, and computer architecture.

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 speed-up measures the effectiveness of parallelization; in the general case the *speed-up* of the parallel computation is defined as

$$S(N) = \frac{T_1}{T_N} \tag{2}$$

with T_1 the execution time of the sequential computation and T_N the execution time when N parallel computations are carried out. Amdhal's law[2] gives the potential speed-up of a parallel computation; it states that the portion of the computation which cannot be parallelized determines the overall speed-up, if α is the fraction of running time a sequential program spends on non-parallelizable segments of the computation then

$$S = \frac{1}{\alpha} \tag{3}$$

Amdhal's law applies to a fixed problem size; when the problem size is allowed to change Gustafson's law gives the speed-up with N processing elements as

$$S(N) = N - \alpha(N-1). \tag{4}$$

Coordination of concurrent computations could be quite challenging and involve overhead which ultimately reduces the speed-up of parallel computations. Often the parallel computation involves multiple stages and all concurrent activities must finish one stage before starting the execution of the next one; this *barrier synchronization* further reduce the speed-up.

The subtasks of a parallel program are called *processes* while *threads* are light-weight subtasks. Concurrent execution could be very challenging, e.g., it could lead to *race conditions*, un undesirable effect when the results of concurrent execution depend on the sequence of events. Often, shared resources must be protected by *locks* to ensure serial access. Another potential problem for concurrent execution of multiple processes is the presence of deadlocks; a *deadlock* occurs when processes competing with one another for resources are forced to wait for additional resources held by other processes and none of the processes can finish.

We distinguish the *fine-grain* from the *course-grain* parallelism; in the former case relatively small blocks of the code can be executed in parallel without the need to communicate or synchronize with other threads or processes, while in the later case large blocks of code can be executed in parallel. The speed-up of applications displaying fine-grain parallelism is considerably lower that those of coarse-grained applications; indeed, the processor speed is orders of magnitude higher than the communication speed even on systems with a fast interconnect.

[2]Gene Amdhal is a theoretical physicist turned computer architect who contributed significantly to the development of several IBM systems including System/360 and then started his own company, Amdhal Corporation; his company produced high performance systems in the 1970s. Amdhal is best known for Amdhal's law formulated in 1960.

In many cases discovering the parallelism is quite challenging and the development of parallel algorithms requires a considerable effort. For example, many numerical analysis problems such as solving large systems of linear equations, or solving systems of PDEs (Partial Differential Equations) require algorithms based on domain decomposition methods.

Data parallelism is based on partitioning the data into several blocks and running multiple copies of the same program concurrently, each running on a different data block, thus the name of the paradigm, Same Program Multiple Data (SPMD).

Decomposition of a large problem into a set of smaller problems that can be solved concurrently is sometimes trivial. For example, assume that we wish to manipulate the display of a three-dimensional object represented as a *3D* lattice of $(n \times n \times n)$ points; to rotate the image we would apply the same transformation to each one of the n^3 points. Such a transformation can be done by a geometric engine, a hardware component which can carry out the transformation of a subset of n^3 points concurrently.

Suppose that we want 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 processing elements is reduced by a factor of N.

A very appealing class of applications of cloud computing are numerical simulation of complex systems which require an optimal design; in such instances multiple design alternatives must be compared and optimal ones selected based on several optimization criteria. Consider for example the design of a circuit using FPGAs. An FPGA (Field-programmable Gate Array) 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 the cross-talk among the wires or to minimize the overall noise. Each alternative configuration requires hours or maybe days of computing on a cloud thus, running them concurrently reduces the design time considerably.

3.2 Parallel computer architecture

From the very beginning it was clear that parallel computing required specialized hardware and system software and a communication fabric was necessary to link the systems. The realization of the difficulties in developing new programming models, the software supporting the development of parallel applications and last, but not least, the parallel algorithms came gradually only later. The list of companies which aimed to support parallel computing and ended up as a casualty of this effort is long and includes names such as: Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCube, Sequent, Tandem, and Thinking Machines.

Our discussion of parallel computer architectures starts with the recognition that parallelism at different levels can be exploited; these levels are:

1. *Bit level parallelism.* The number of bits processed per clock cycle, often called a word size, has increased gradually from 4-bit processors to 8-bit, 16-bit, 32-bit, and since 2004 to 64-bit. The larger number of bits allows instructions to reference a larger

address space, and reduces the number of instructions required to process large size operands.

2. *Instruction-level parallelism.* Today's computers use multi-stage processing pipelines to speed up execution. Once an n-stage pipeline is full an instruction is completed at every cycle. For example, the pipeline for a RISC (Reduced Instruction Set Computing) architecture consists of five stages: instruction fetch, instruction decode, instruction execution, memory access, and write back. A CISC (Complex Instruction Set Computing) architecture could have a much large number of pipelines stages, e.g., an Intel Pentium 4 processor has a 35-stage pipeline.

3. *Data parallelism.* The program loops can be processed in parallel.

4. *Task parallelism.* The problem can be decomposed into tasks that can be carried out concurrently. A widely use type of task parallelism is the SPMD (Same Program Multiple Data). As the name suggests individual processors run the same program but on different segments of the input-data.

In 1966 Michael Flynn proposed a classification of computer architectures based on the number of *concurrent control/instruction* and *data streams*: SISD (Single Instruction Single Data), SIMD (Single Instruction, Multiple Data), and MIMD (Multiple Instructions, Multiple Data)[3].

The first use of SIMD instructions was in vector supercomputers such as the CDC Star-100 and the Texas Instruments ASC in the 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, in its UltraSPARC I microprocessor; the first widely-deployed SIMD for gaming was Intel's MMX extensions to the x86 architecture. IBM and Motorola then added AltiVec to the POWER architecture and there have been several extensions to the SIMD instruction sets for both architectures. These developments geared toward supporting real-time graphics with vectors of two, three, or four dimensions led to the development of GPUs (Graphic Processing Units). GPUs are very efficient at manipulating computer graphics, and their highly parallel structures based on SIMD execution support parallel processing of large blocks of data. GPUs produced by Intel, NVIDIA, and AMD/ATI are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.

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. The processors can share a common memory of a MIMD and we distinguish several types of systems, UMA (Uniform Memory Access), COMA (Cache Only Memory Access) and NUMA (Non-Uniform Memory Access). A MIMD system could have a distributed memory; in this case the processors and the memory communicate with one another using an interconnection network, such as a hypercube, a `2D` torus, a `3D` torus, an omega network, or other network topology. Today, most supercomputers are MIMD

[3]Another category, MISI (Multiple Instruction Single Data) is a fourth possible architecture, but it is very rarely used, and then only for fault tolerance.

machines and some use GPUs. Multi-core processors with multiple processing units are now ubiquitous.

The next natural step was triggered by advances in communication networks when low-latency and high-bandwidths WANs (Wide Area Networks) 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.

3.3 Distributed systems

The systems we analyze are distributed and in this section we introduce basic concepts necessary to understand the problems posed by the design of such systems. A *distributed system* is a collection of autonomous computers, connected through a network and 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.

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, and the resources 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.

Distributed systems have been around for several decades. For example, distributed file systems and network file systems have been used for user convenience and to improve reliability and functionality of file systems for many years. Modern operating systems allow a user to *mount* a remote file system and access it the same way a local file system is accessed, yet 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 a procedure running in a different address space, possibly on a remote system. RPCs have been introduced in the early 1970s by Bruce Nelson and used for the first time at Xerox; 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 middleware should support a set of desirable properties of a distributed system:

- Access transparency - local and remote information objects are accessed using identical operations;
- Location transparency -information objects are accessed without knowledge of their location;
- Concurrency transparency - several processes run concurrently using shared information objects without interference among them;
- Replication transparency - multiple instances of information objects are used to increase reliability without the knowledge of users or applications;

- Failure transparency - the concealment of faults;
- Migration transparency - the information objects in the system are moved without affecting the operation performed on them;
- Performance transparency - the system can be reconfigured based on the load and quality of service requirements;
- Scaling transparency - the system and the applications can scale without a change in the system structure and without affecting the applications.

3.4 Communication in a distributed system

To understand the important properties of distributed systems we use a model, an abstraction based on two critical components, processes 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 is characterized by its *state*; the state is the ensemble of information about the process we need to restart it after it has been suspended.

A *communication channel* provides the means for processes or 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. We also assume that the time a process needs to traverse a set of states is of no concern and that processes may fail.

The activity of any process is modelled as a sequence of events; an *event* is a change of the state of a process. There are two types of events, local and communication events. The cause of a *local event* is internal to the process, it is not affected by other processes or by the environment. A *communication event* is either the sending of a message to another process or receiving a message from another process.

The *local history of a process* is a sequence of events, possibly an infinite one, and can be presented graphically as a *space-time diagram* where events are ordered by their time of occurrence. For example, in Figure 3(a) the history of process p_1 consists of 11 events, $e^1, e^2, ..., e^{11}$. The process is in state σ_1 immediately after the occurrence of event e^1 and remains in that state until the occurrence of event e^2.

Distributed systems consist of multiple processes active at any one time and communicating with each other. *Communication events* involve more than one process and occur when the algorithm implemented by a process requires sending or receiving a message to

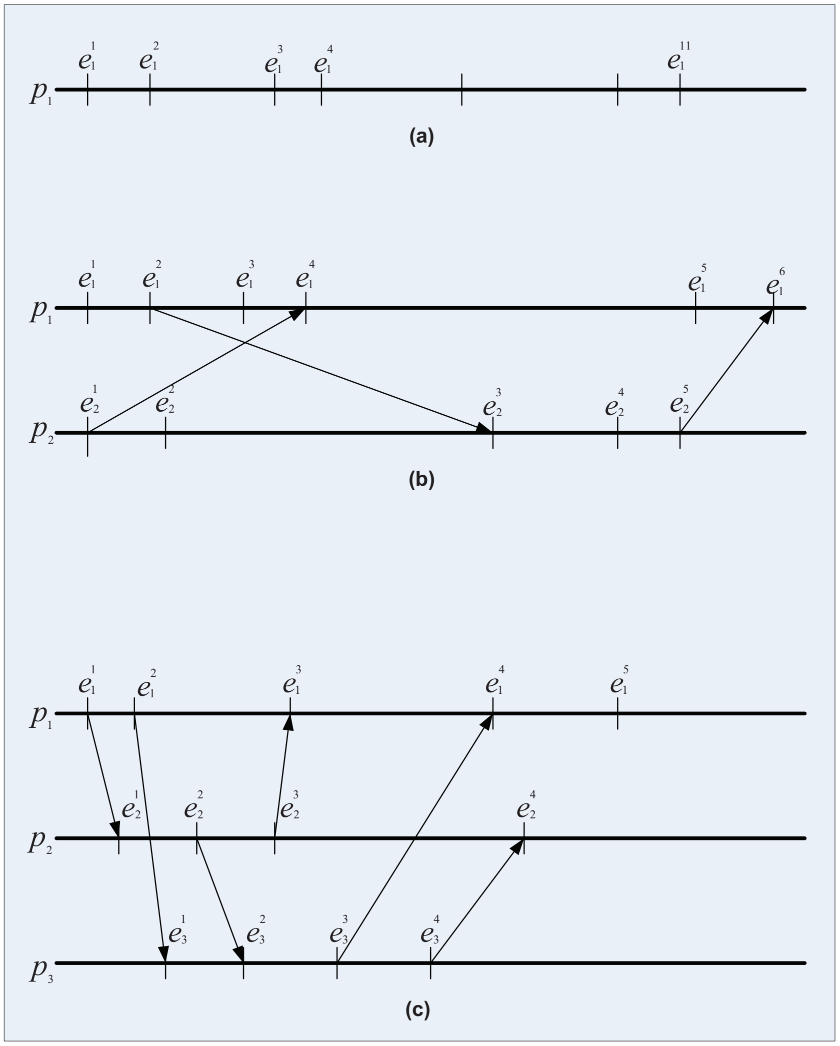


Figure 3: Space-time diagrams show the events during a process lifetime. (a) All events in case of a single process p_1 are local; the process is in state σ_1 immediately after the occurrence of event e^1 and remains in that state until the occurrence of event e^2. (b) Two processes p_1 and p_2; event e_1^2 is a communication event, p_1 sends a message to p_2; event e_2^3 is a communication event, process p_2 receives the message sent by p_1. (c) Three processes interact by means of communication events.

another process. The space-time diagram in Figure 3(b) shows two processes, p_1 and p_2 with local histories, respectively,

$$h_1 = (e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_1^6) \quad \text{and} \quad h_2 = (e_2^1, e_2^2, e_2^3, e_2^4, e_2^5). \tag{5}$$

A *protocol* is a finite set of messages exchanged among processes to help them coordinate their actions. Figure 3(c) 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 protocol requires each process to send messages to all other processes in response to the a message from the coordinator, process p_1.

The informal definition of the state of a single process can be extended to collections of communicating processes. 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 [32].

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, $p_1, p_2, \ldots, p_n$; then, its global state is an n-tuple of local states

$$\Sigma = (\sigma_1, \sigma_2, ..., \sigma_n). \tag{6}$$

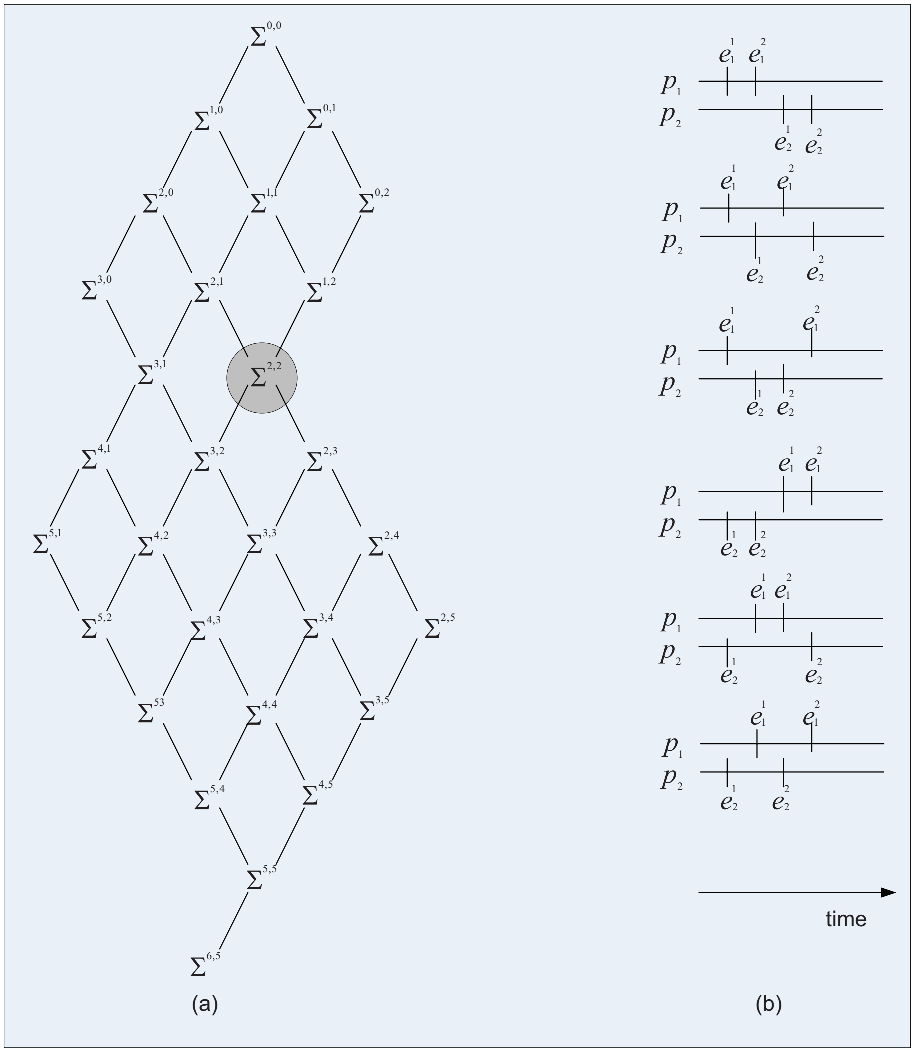


Figure 4: (a) The lattice of the global states of two processes with the space-time diagrams in Figure 3(b). (b) The sequences of events leading to the state $\Sigma^{2,2}$.

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 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 $\Sigma^{j1,j2,....jn}(\sigma_1^{j1}, \sigma_2^{j2}, ..., \sigma_n^{jn})$.

Figure 4(a) shows the lattice of global states of the distributed computation in Figure 3(b). This is a two-dimensional lattice because we have two processes, p_1 and p_2. The lattice of global states for the distributed computation in Figure 3(c) 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 4(b) 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 4(b) shows the six possible sequences of events to reach the global state $\Sigma^{2,2}$:

$$(e_1^1, e_1^2, e_2^1, e_2^2), (e_1^1, e_2^1, e_1^2, e_2^2), (e_1^1, e_2^1, e_2^2, e_1^2), (e_2^1, e_2^2, e_1^1, e_1^2), (e_2^1, e_1^1, e_1^2, e_2^2), (e_2^1, e_1^1, e_2^2, e_1^2). \quad (7)$$

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 a function of the global state of the system. In many instances we need to perform an action when the state of the system satisfies a particular condition.

3.5 Process coordination

A major concern in any distributed system is *process coordination* in the presence of channel failures. There are multiple modes for a channel to fail and some lead to messages being lost. In the most general case, it is impossible to guarantee that two processes will reach an agreement in case of channel failures, see Figure 5.

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 is a finite sequence 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.

The coordination problem can be solved sometimes by constructing fairly complex communication protocols. In other cases even though no theoretical solution exists, in practice one may use channels with very low error rates and may tolerate extremely low probabilities of failure.

We need to measure *time intervals*, the time elapsed between two events and 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 collaborating with each other are networked computers the precision of the clock synchronization is critical [140]. 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

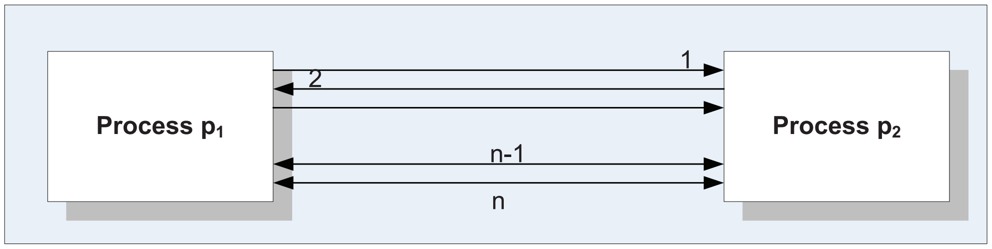


Figure 5: Process coordination in the presence of errors; each message may be lost with probability ϵ. If a protocol consisting of n messages exists, then the protocol should be able to function properly with $n-1$ messages reaching their destination, one of them being lost.

rate. That explains why we need to measure time very accurately; indeed, we have atomic clocks with an accuracy of about 10^{-6} seconds 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, e.g., 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., that the sensor triggering an alarm has indeed changed 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.

Timestamps are often used for event ordering using a global time-base constructed on local virtual clocks [162]. Δ-protocols [66] 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 [245]. 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 modelled as a sequence of *events*; hence, the binary relation cause-effect should be expressed in terms of events and should express our intuition that *the cause must precede the effects.* Again, we need to distinguish between local events and communication events. The latter 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

$$\text{if } \ e_i^k, e_i^l \in h_i \ \text{ and } \ k < l \ \text{ then } \ e_i^k \rightarrow e_i^l. \tag{8}$$

2. Causality of communication events:

$$\text{if } \ e_i^k = \mathit{send}(m) \ \text{ and } \ e_j^l = \mathit{receive}(m) \ \text{ then } \ e_i^k \rightarrow e_j^l. \tag{9}$$

3. Transitivity of the causal relationship

$$\text{if } \ e_i^k \rightarrow e_j^l \ \text{ and } \ e_j^l \rightarrow e_m^n \ \text{ then } e_i^k \rightarrow e_m^n. \tag{10}$$

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.*

3.6 Logical clocks

A *logical clock* is an abstraction necessary to ensure the clock condition 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, $TS(m) = LC(\mathit{send}(m))$. The rules to update the logical clock are specified by Equation 11:

$$LC(e) := \begin{cases} LC + 1 & \text{if } \ e \text{ is a local event or a } \mathit{send}(m) \text{ event} \\ \max(LC, TS(m) + 1) & \text{if } e = \mathit{receive}(m). \end{cases} \tag{11}$$

The concept of logical clocks is illustrated in Figure 6 using a modified *space-time diagram* where the events are labelled 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 11. 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 6. Nevertheless, communication events allow different processes to coordinate their logical

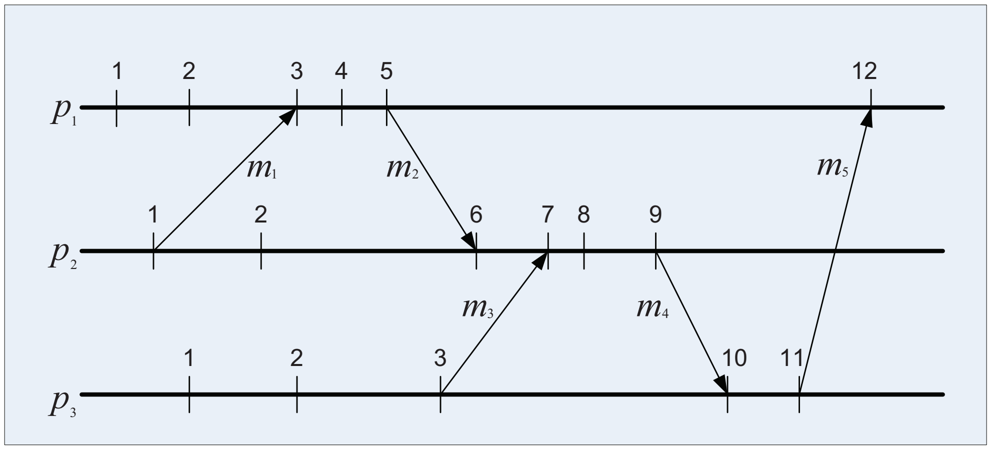


Figure 6: Three processes and their logical clocks; The usual labelling of events as $e_1^1, e_1^2, e_1^3, \ldots$ is omitted to avoid overloading the figure; only the logical clock values for the local and for the communication events are marked. The correspondence between the events and the logical 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. Process p_2 labels event e_2^3 as 6 because of message m_2, which carries information about the logical clock value at the time it was sent as 5. 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.

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

$$LC(e) < LC(e'') < LC(e'). \tag{12}$$

For example, there is an event, e_1^4, between the events e_1^3 and e_1^5 in Figure 3.6; 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 $LC(e_3^4) = 10$.

3.7 Message delivery rules; causal delivery

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 7

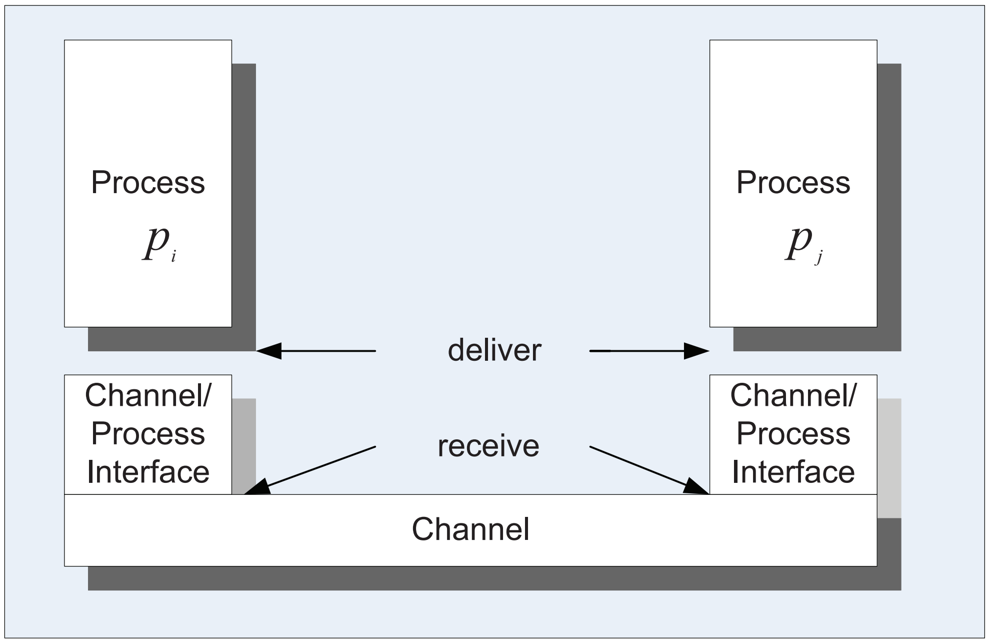


Figure 7: Message receiving and message delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.

$$receive(m) \rightarrow deliver(m). \tag{13}$$

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

$$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m'). \tag{14}$$

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.

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

$$send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m'). \tag{15}$$

When more than two processes are involved in a message exchange, the message delivery may be FIFO, but not causal as shown in Figure 8 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.

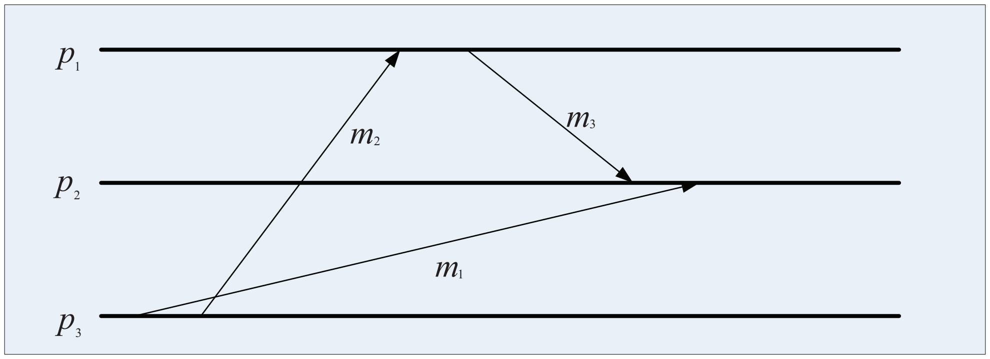


Figure 8: Violation of causal delivery when more than two processes are involved; message m_1 is delivered to process p_2 after message m_3, though message m_1 was sent before m_3. Indeed, message m_3 was sent by process p_1 after receiving m_2, which in turn was sent by process p_1 after sending m_1.

- $send(m_2) \rightarrow deliver(m_2)$.
- $send(m_3) \rightarrow deliver(m_3)$.

The transitivity property the causality relations above imply that $send(m_1) \rightarrow deliver(m_3)$.

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. Each process includes in every message the time stamp $RC(e)$, 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

$$e \rightarrow e' \Rightarrow RC(e) < RC(e'), \ \forall e, e'. \tag{16}$$

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, 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 time stamps

$$\forall e, e', \quad e \rightarrow e' \equiv TC(e) < TC(e'). \tag{17}$$

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.

3.8 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, they 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

$$R = (e_1^{j1}, e_2^{j2}, ..., e_n^{jn}) \tag{18}$$

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

For example, consider the three processes in Figure 9. We can construct a three-dimensional lattice of global states following a procedure similar to the one in Figure 4 starting from the initial state Σ_{000} to any reachable state Σ_{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 one; this run is valid, the system has traversed the global states

$$\Sigma^{000}, \Sigma^{100}, \Sigma^{110}, \Sigma^{111}, \Sigma^{211} \tag{19}$$

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

$$C = \{h_i^j\} \quad \text{with} \quad i \in \{1, n\} \text{ and } j \in \{1, n_i\}. \tag{20}$$

The frontier of the cut is an n-tuple consisting of the last event of every process included in the cut. Figure 9 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, and C_2 has the frontier $(5, 6, 3)$.

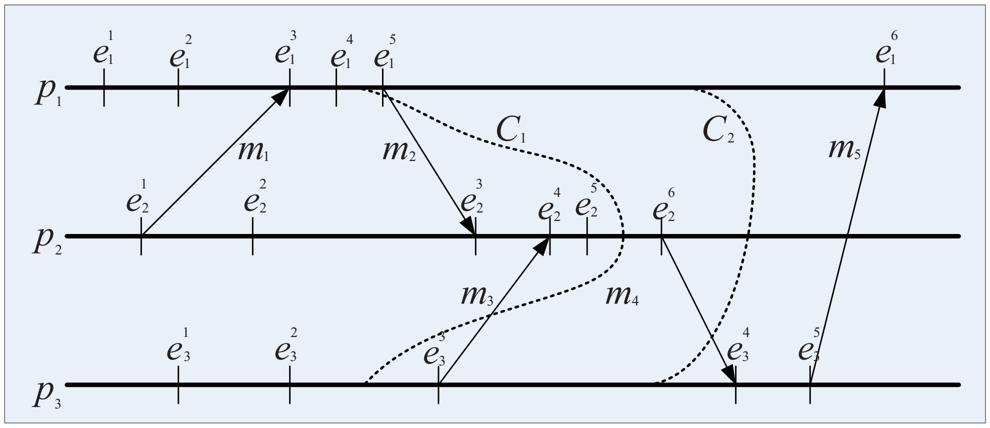


Figure 9: Inconsistent and consistent cuts: the cut $C_1 = (e_1^4, e_2^5, e_3^2)$ is inconsistent because it includes e_2^4, the event triggered by the arrival of the message m_3 at process p_2, but does not include e_3^3, the event triggered by process p_3 sending m_3 thus, the cut C_1 violates causality. 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.

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 9 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 a real 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 iff for all events

$$\forall e, e', \quad (e \in C) \ \wedge \ (e' \ \rightarrow \ e) \ \Rightarrow \ e' \in C. \tag{21}$$

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

$$\gamma(e) = \{e' \in G \mid e' \rightarrow e\} \ \cup \ \{e\}. \tag{22}$$

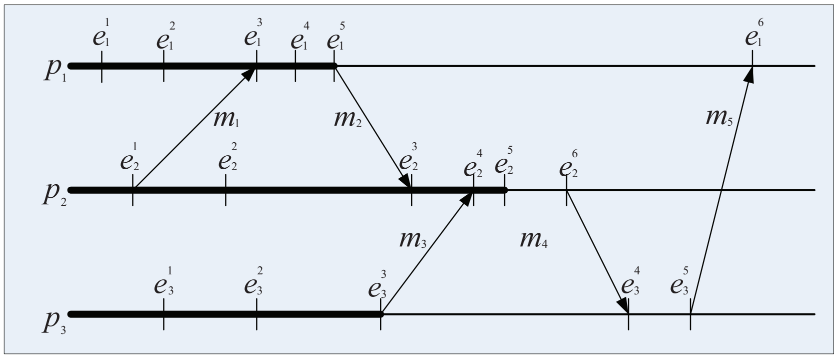


Figure 10: The causal history of event e_2^5, $\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}$, is the smallest consistent cut including e_2^5.

The causal history of event e_2^5 in Figure 10 is:

$$\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}. \tag{23}$$

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 sending of 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.

$$e \rightarrow e' \;\equiv\; \gamma(e) \subset \gamma(e'). \tag{24}$$

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

$$\gamma(e_i) \;=\; \gamma(e_j) \cup \gamma(e_k) \tag{25}$$

  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 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. The snapshot protocol of Chandy and Lamport consists of three steps [52]

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.

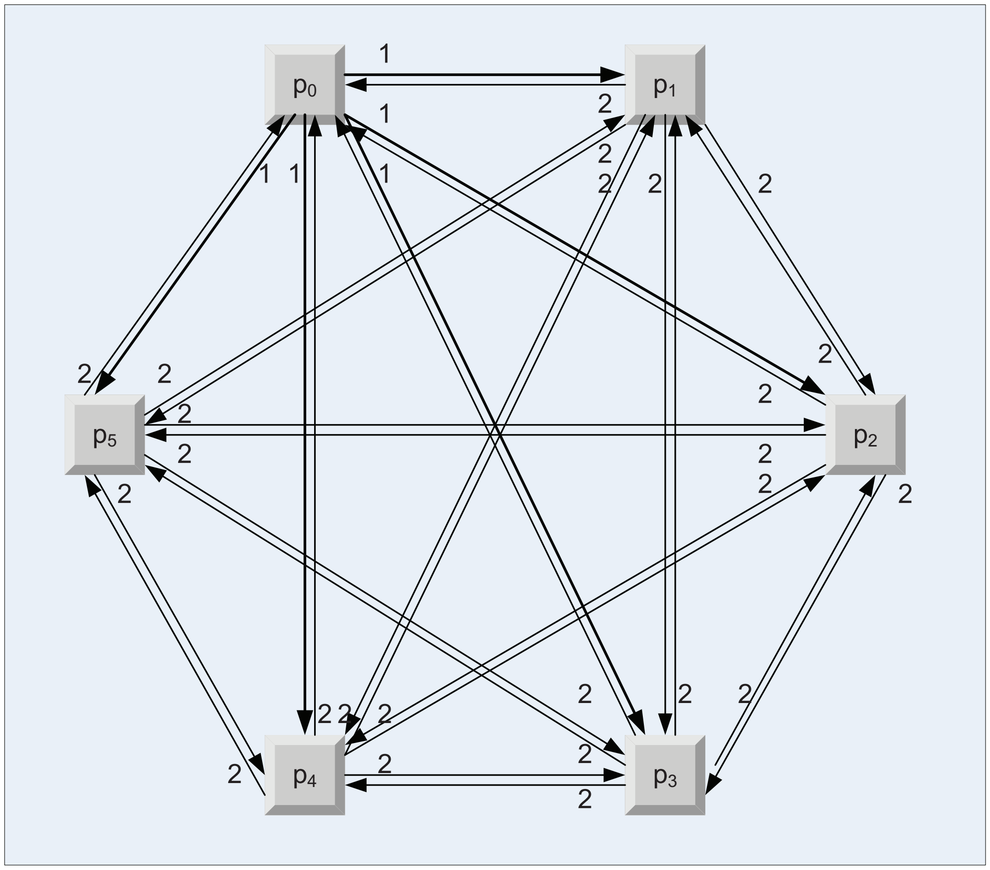


Figure 11: Six processes executing the snapshot protocol.

For example, consider a set of six processes, each pair of processes being connected by two unidirectional channels as shown in Figure 11. 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 labelled (1) in Figure 11.
- In step 2, each of the five processes, p_1, p_2, p_3, p_4, and p_5 sends a “take snapshot” message labelled (2).

A “take snapshot” message crosses each channel from process p_i to p_j exactly once and $6 \times 5 = 30$ messages are exchanged.

3.9 Enforced modularity; the client-server paradigm

Modularity is a basic concept in the design of man-made systems; a complex 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. It is thus, no surprise that the hardware, as well as, the software systems are composed of modules which interact with one another through well-defined interfaces.

In this section we are only concerned with software modularity. We distinguish *soft modularity* from *hard modularity.* The former means to divide a program into modules which call each other and communicate using shared memory or follow the procedure call convention. The steps involved in the transfer of the flow of control between the caller and the callee are: (i) the caller saves its state including the registers, the arguments, and the return address on the stack; (ii) the callee loads the arguments from the stack, carries out the calculations and then transfers control back to the caller; (iii) the caller adjusts the stack, restores its registers, and continues its processing.

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 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.

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; the server may fail and then come up without the clients being affected or even notice 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 allows systems with different processor architecture, e.g., 32-bit or 64-bit, different operating systems, e.g., multiple versions of operating systems such as Linux, Mac OS, or Windows, libraries and other system software, to cooperate. The client-server paradigm increases the 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.

System heterogeneity is a blessing in disguise. It 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 the others or may have a lower workload. A major difference between the basic models of grid and cloud computing is that the former did not impose any restrictions regarding heterogeneity of the computing platforms while the a computer cloud is a collection of homogeneous systems.

The clients and the servers communicate through a network that itself 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.

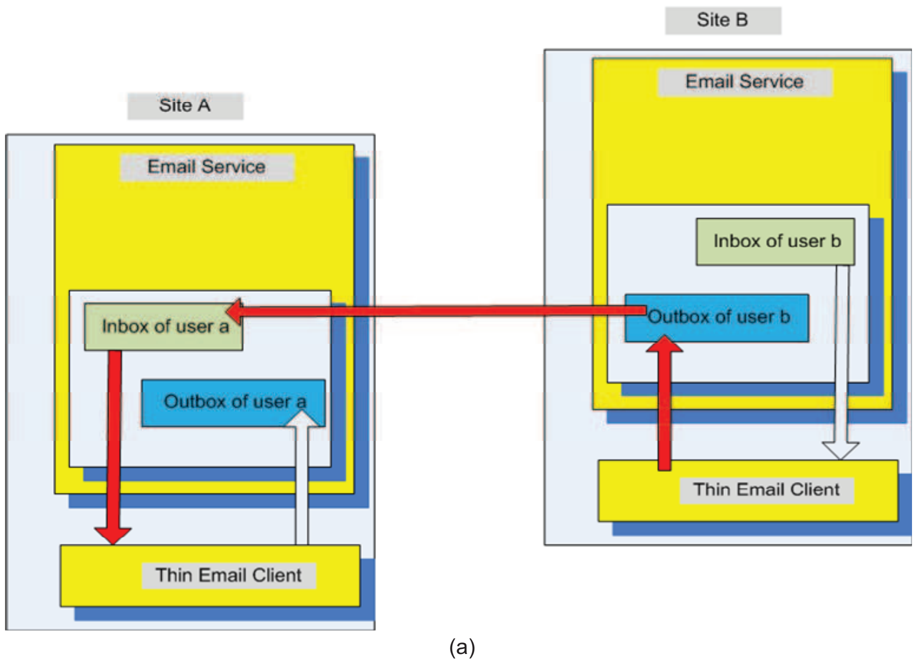
RPC (Remote Procedure Call) is often used for the implementation of client-server systems interactions. The RPC standard is described in RFC 1831. To use an RPC a process may use special services `PORTMAP` or `RPCBIND` available at port 111 to register and for service lookup. RPC messages must be well-structured; 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 so called fate sharing between caller and the callee but take longer than local calls due to communication delays. Several RPC semantics are implemented:

- *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 request an acknowledgment from the server.



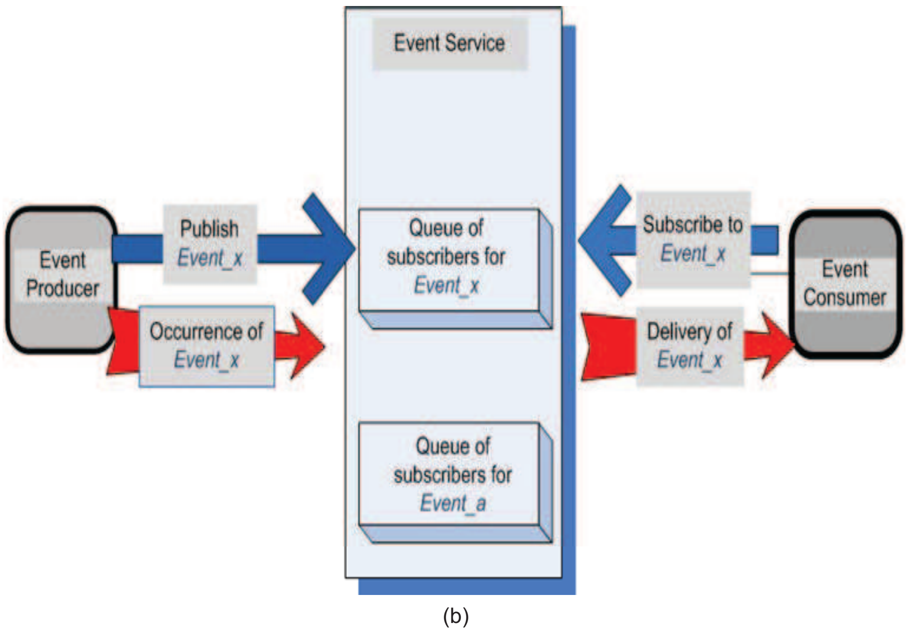


Figure 12: (a) Email service; the sender and the receiver communicate asynchronously using inboxes and outboxes. Mail demons run at each site. (b) An event service supports coordination in a distributed system environment. The service is based on the publish-subscribe paradigm; an event producer publishes events and an event consumer subscribes to events. The server maintains queues for each event and delivers notifications to clients when an event occurs.

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 electronic mail, see Figure 12(a), the Domain Name System (DNS), the X-windows, event services, see Figure 12(b), and so on.

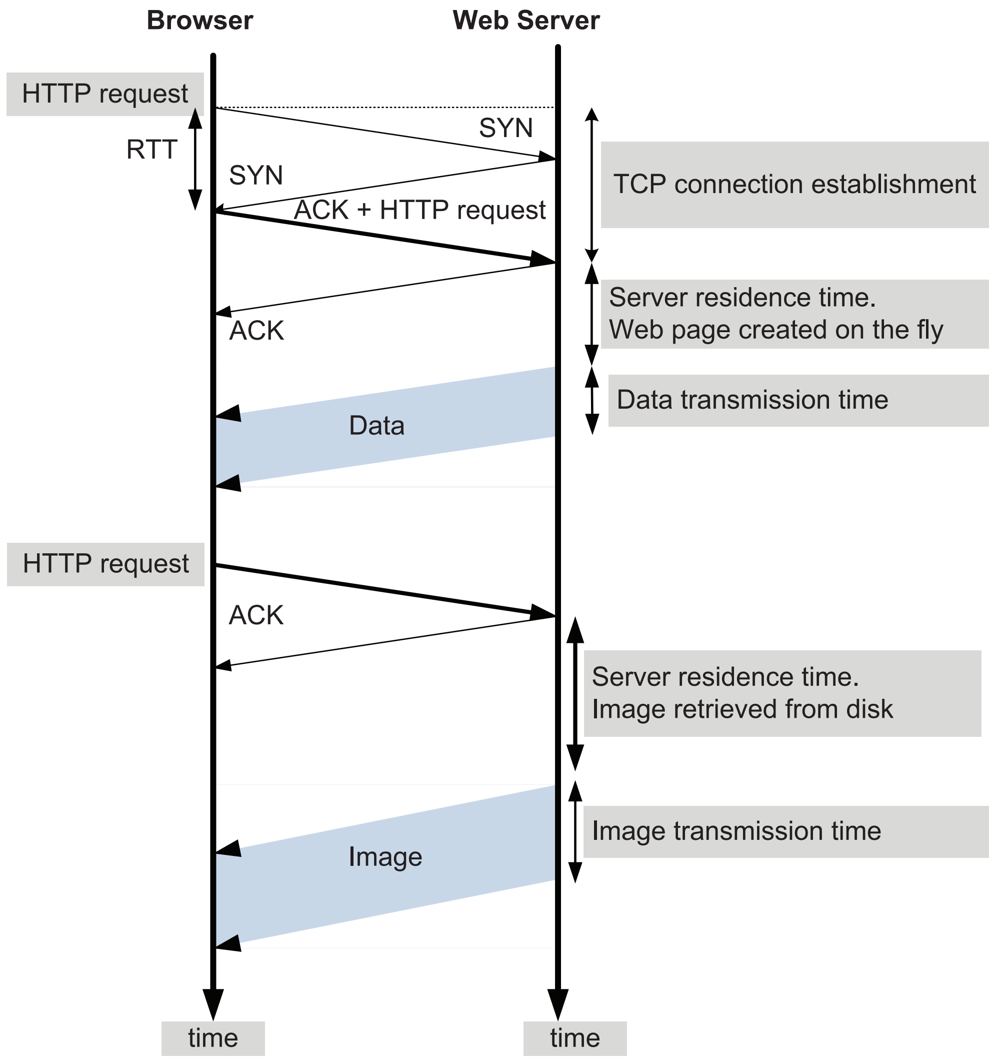


Figure 13: Client-server communication, the World Wide Web. Once the TCP connection is established the HTTP server takes its time to construct the page to respond the first request; to satisfy the second request the HTTP server must retrieve an image from the disk. The *response time* includes the RTT, the server residence time, and the data transmission time.

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 websites. 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 sever 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 13 shows the sequence of events when a client sends an HTTP request to a server to retrieve some information and the server constructs the page on the fly and then it requests 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 to 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 14. Proxy servers could provide multiple functions; for example, they may filter client requests and decide wether or not to forward the request based on some filtering rules. The proxy server may redirect the request to a server in close proximity to the client or to a less loaded server; a proxy can also act as caches and provide a local copy of a resource, rather than forward the request to the server.

Another type of client-server communication is HTTP-tunnelling used most often as a means for communication from network locations with restricted connectivity. Tunnelling 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 14.

3.10 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 [88], 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*[4]. In Section 5.4 we present a consensus service, the *ZooKeeper*, based on the Paxos protocol.

[4]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

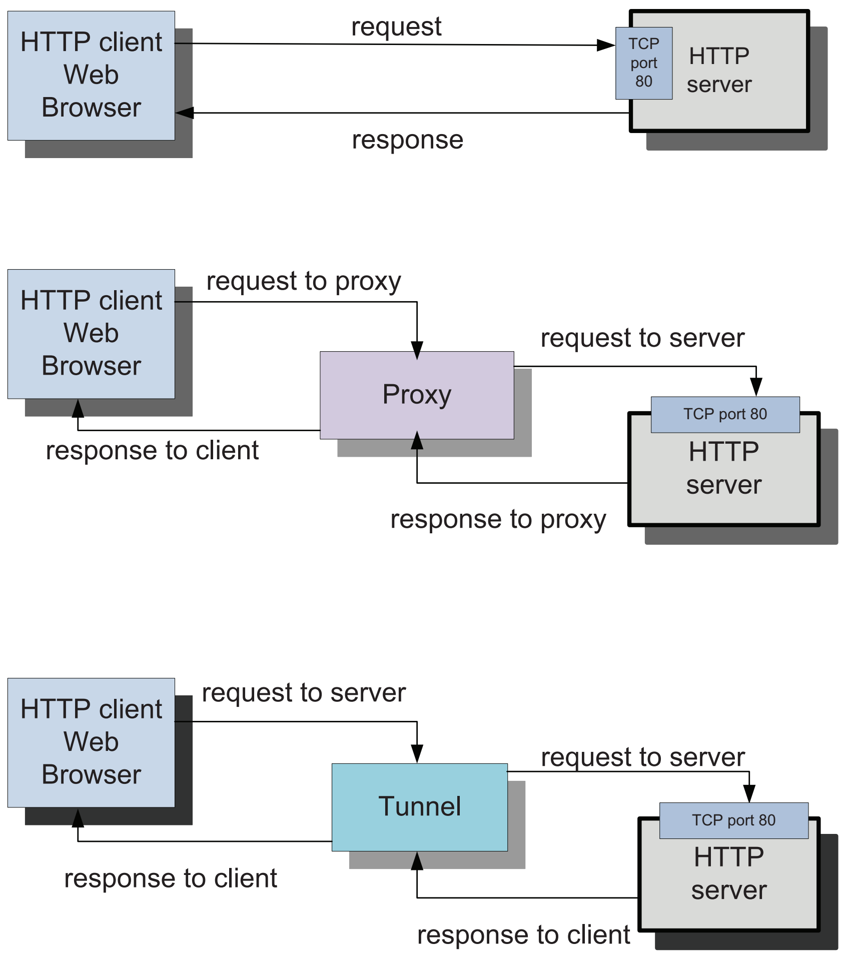


Figure 14: A client can communicate directly with the server, it can communicate through a proxy, or may use tunnelling to cross the network.

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 Paxos* and *Leaderless Byzantine Paxos*; he has also published a paper on the fictional part-time parliament in Paxos [141] and a layman's dissection of the protocol [142].

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[5].

remarkable correspondence to the problem faced by todays fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing" according to Leslie Lamport [141] (for additional papers see http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html).

[5]A Byzantine failure in a distributed system could be an *omission failure*, e.g., a crash failure, failure

- 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.

Each process advocates a value proposed by a client and could play one, two, or all three of the following roles: *acceptors* are the persistent, or fault-tolerant, memory of the system who decide which value to choose; *proposers* propose a value sent by a client to be chosen by the acceptors; *learners* learn which value was chosen and act as the replication factor of the protocol; the *leader* is an elected proposer. A *quorum* is a subset of all acceptors; any two quorums share at least one member.

When each process plays all three roles, proposer, acceptor, and learner, the flow of messages can be described as follows [142]: "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."

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 iff 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 tread as an acceptor that chooses not to respond.

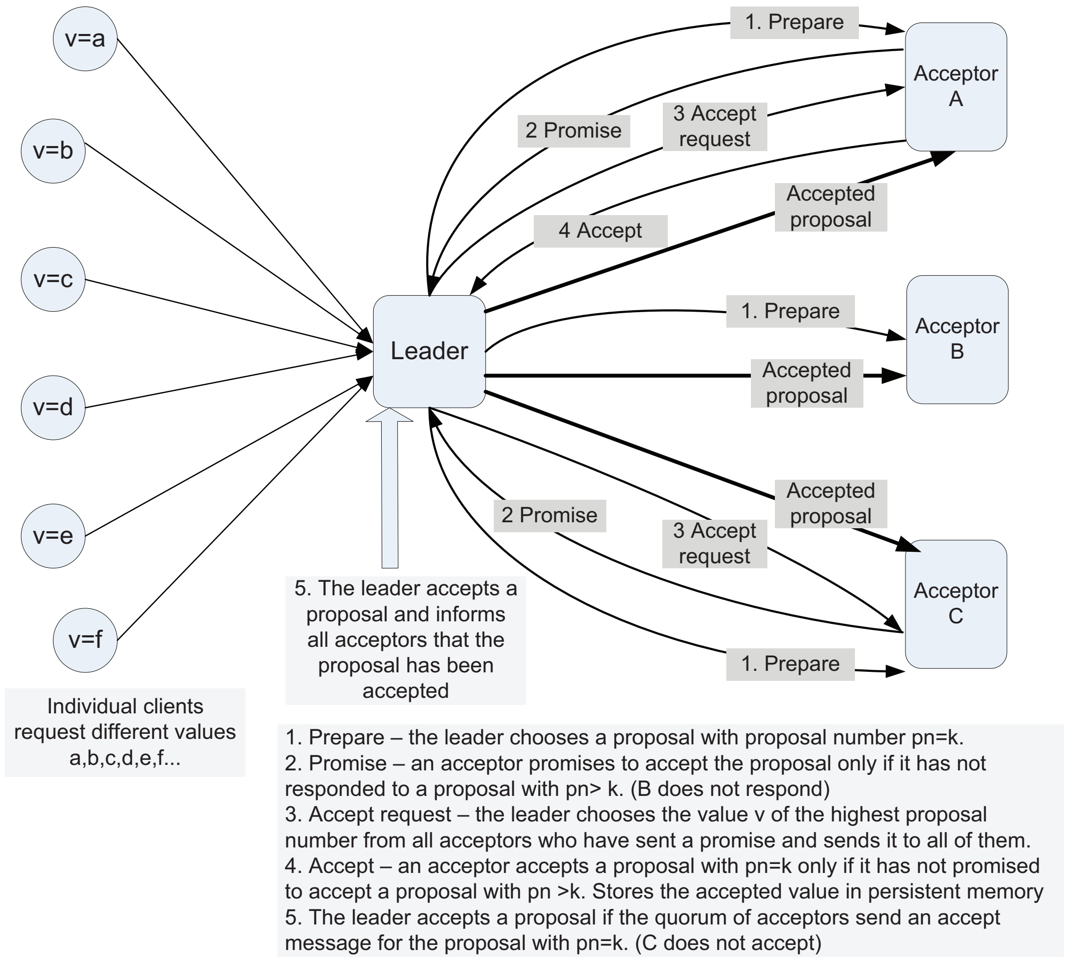
Phase II.

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.

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 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 propose and to every leaner; if it does not accept the proposal it should ignore the request.



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 15: 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.

Figure 15 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 [142]. 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.

3.11 Modelling concurrency with Petri Nets

In 1962 Carl Adam Petri introduced a family of graphs, the so-called Petri Nets (PNs) [199]. PNs are bipartite graphs populated with tokens that flow through the graph and used to model the dynamic rather than static behavior of systems, e.g. detect synchronization anomalies.

A *bipartite graph* is one with two classes of nodes; arcs always connect a node in one class with one or more nodes in the other class. In the case of Petri Nets the two classes of nodes 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.

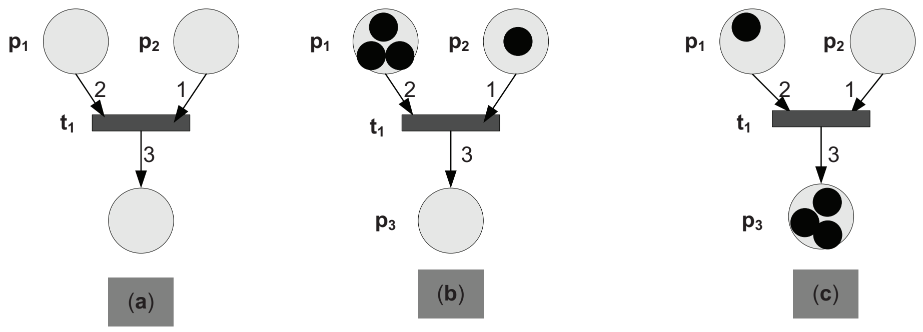


Figure 16: Petri Nets firing rules. (a) An unmarked net with one transition t_1 with two input places, p_1 and p_2, and one output place, p_3. (b) The marked net, the net with places populated by tokens; the net before firing the enabled transition t_1. (c) The marked net after firing transition t_1, two tokens from place p_1 and one from place p_2 are removed and transported to place p_3.

To 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 16.

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 modelled. PNs are very powerful abstractions and can express both concurrency and choice as we can see in Figure 17.

Petri nets can model concurrent activities. For example, the net in Figure 17(a) 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 on concurrent transitions are t_1 and t_3 in Figures 17(b) and (c).

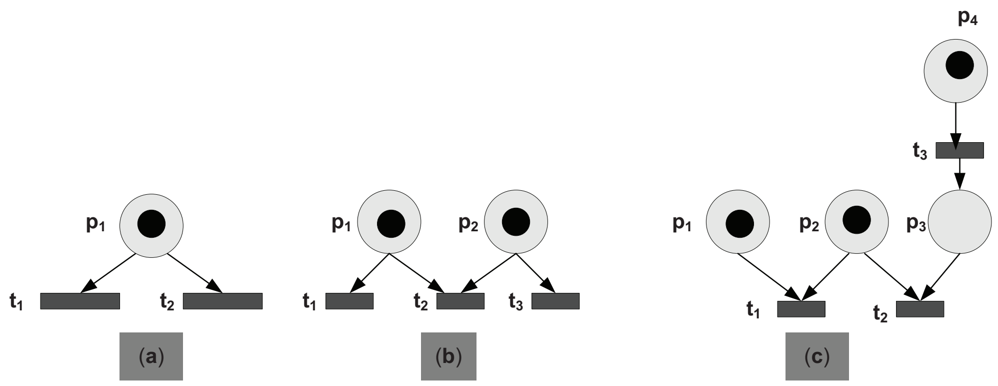


Figure 17: Petri Nets modelling. (a) Choice; only one of transitions t_1, or t_2 may fire. (b) Symmetric confusion; transitions t_1 and t_3 are concurrent and, at the same time, they are in conflict with t_2. If t_2 fires, then t_1 and t_3 are disabled. (c) Asymmetric confusion; transition t_1 is concurrent 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 17(b), 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 17(c).

The concurrent transitions t_2 and t_3 in Figure 18(a) model concurrent execution of two processes. Transition t_4 and its input places p_3 and p_4 in Figure 18(b) model synchronization; t_4 can only fire if the conditions associated with p_3 and p_4 are satisfied.

Petri Nets can be used to model *priorities*. The net in Figure 18(c) models a system with two processes modelled by transitions t_1 and t_2; the process modelled by t_2 has a higher priority than the one modelled by $_1$. If 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, modelling the activation of the second process. Only after t_2 is activated transition t_1, modelling of activation of the first process, will fire.

Petri Nets are able to model *exclusion*; for example, the net in Figure 18(d), models a group of n concurrent processes executing in a shared-memory environment. All processes can read at the same time, but only one may write. 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.

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

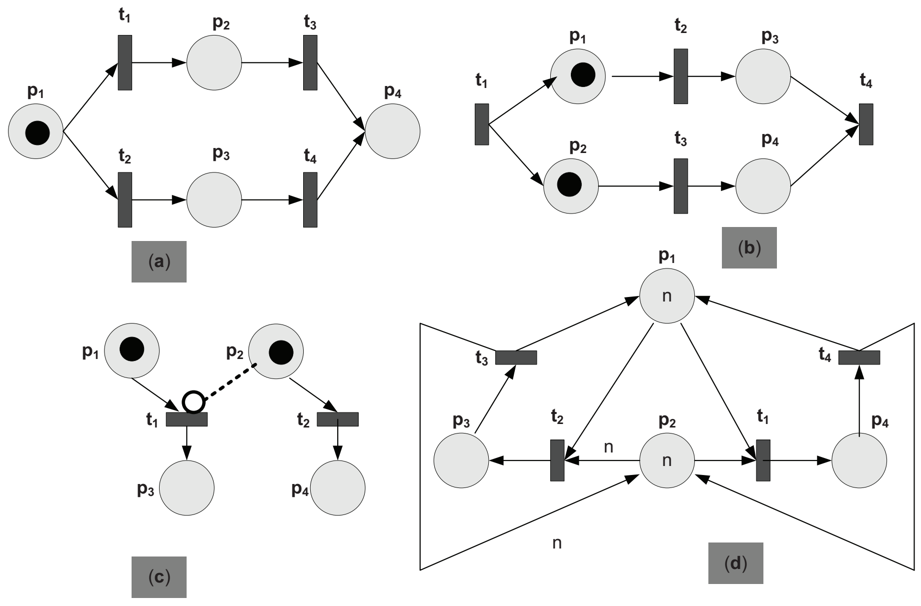


Figure 18: (a) A state machine; there is the choice of firing t_1, or t_2; only one transition fires at any given time, concurrency is not possible. (b) A marked graph; can model concurrency but not choice; transitions t_2 and t_3 are concurrent, there is no causal relationship between them. (c) An extended net used to model priorities; the arc from p_2 to t_1 is an inhibitor arc. The process modelled by transition t_1 is activated only after the process modelled by transition t_2 is activated. (d) Modelling exclusion: transitions t_1 and t_2 model writing and, respectively, reading with n processes in a shared memory. At any given time only one process may write but all n may read.

Labelled 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 labelling 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. Labelled 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 an labelled PN and s is a bag[6] over p denoting the markings of the net.

[6]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 $A(x) > 0$.

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 16(a) 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 16(a) 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 18(a) 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 18(a), (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* iff 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) with $N = (p, t, f, l)$ is called a *firing sequence* iff there exist markings $s_1, s_2, ...s_n \in \mathcal{B}(p)$ and transitions $t_1, t_2, ...t_n \in t$ such that $\sigma = t_1 t_2 ....t_n$ 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 making 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, $s_n \in \sigma(s_0)$. Reachability is a fundamental concern for dynamic systems; the reachability problem is decidable, but 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 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 $f_{i,j} = w(i, j) - w(j, i)$. Here $w(i, j)$ is the weight of the flow relation (arc) from transition t_i to its output place p_j, and $w(j, i)$ is the weight of the arc from the input place p_j to transition t_i. In this expression $w(i, j)$ represents the number of tokens added to the output place p_j and $w(j, i)$ the ones removed from the input place p_j when transition t_i fires. F^T is the transpose of the incidence matrix.

A marking s_k c an 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_i to be enabled at a marking s is that $w(j, i) \leq s(j) \quad \forall s_j \in \bullet t_i$, 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; *inhibitor arcs* that transitions to be enabled. For example, the arc from p_2 to t_1 in the net in Figure 18(a) is an inhibitor arc; the process modelled by transition t_1 is activated only after the process modelled 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 18 (c) is not enabled while place p_2 holds a token.

Based on their structural properties Petri Nets can be partitioned in several classes:

- State Machines - are used to model finite state machines and cannot model concurrency and synchronization;
- Marked Graphs - cannot model choice and conflict;
- Free-choice Nets - cannot model confusion;
- Extended Free-choice Nets - cannot model confusion but allow inhibitor arcs;
- Asymmetric Choice Nets - can model asymmetric confusion but not 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 19.

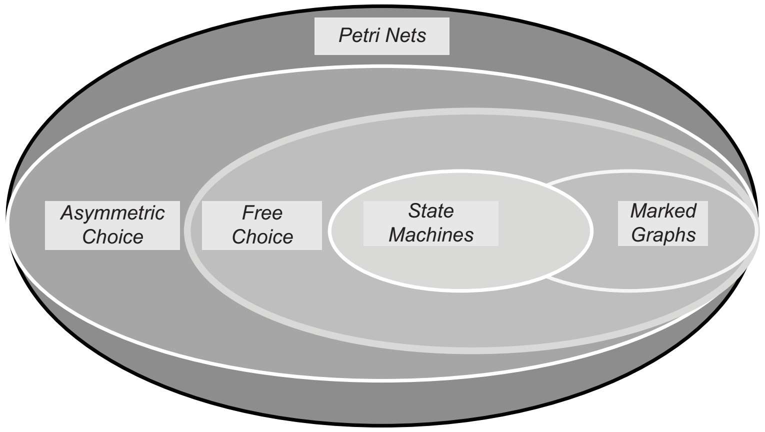


Figure 19: Classes of Petri Nets.

State Machine: a Petri Net is a *state machine* iff $\forall t_i \in t \ (\mid \bullet t_i \mid \ = 1 \wedge \mid t_i \bullet \mid \ = 1)$. 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 18(a) have only one input and 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* iff $\forall p_i \in p \mid \bullet p_i \mid = 1 \wedge \mid p_i \bullet \mid = 1$; in a marked graph each place has only one incoming and one outgoing flow relation; thus, marked graphs do no not allow modelling 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* iff

$$(\bullet t_i) \cap (\bullet t_j) \ = \ \emptyset \ \Rightarrow \mid \bullet t_i \mid \ = \ \mid \bullet t_j \mid \quad \forall t_{i,j} \in t \tag{26}$$

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

In an extended free-choice net if two transition 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 (CPSs) allow tokens of different colors thus, increase the expressivity of the PNs but do not simplify their analysis. Several extension of Petri Nets to support performance analysis by associated 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 modelled 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 [150]; 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 modelled.

3.12 Analysis of communicating processes

This brief discussion of distributed systems leads to the observation that the analysis of communicating processes requires a more formal framework. Hoare realized that a language based on execution traces is insufficient to abstract the behavior of communicating processes and developed *communicating sequential processes* (CSP) [112]. More recently, Milner initiated an axiomatic theory called the *Calculus of Communicating System* (CCS), [168].

Process algebra is the study of concurrent communicating processes within an algebraic framework. The process behavior is modelled 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. Here we only outline the theory called *Basic Process Algebra* (BPA).

An *algebra* $\mathcal{A}$ consists of a set A of elements and a set of operators, f. A is called the domain of the algebra $\mathcal{A}$ and consists of a set of constants and variables. The operators map A^n to A, the domain of an algebra is closed with respect to the operators in f. For example, in Boolean algebra $\mathcal{B} = (B, xor, and, not)$ with $B = \{0, 1\}$.

$\mathcal{BPA} = (\Sigma_{BPA}, E_{BPA})$ is an algebra. Here Σ_{BPA} consists of two binary operators, $(+)$ and $(\times)$, as well as a number of constants, $a, b, c, \ldots$, and variables, $x, y, \ldots$. The operator $\times$ is called the *product* or the *sequential composition* and it is generally omitted, $x \times y$ is equivalent to xy and means a process that first executes x and then y. The operator $(+)$ is called the *sum* or the *alternative composition*, $x + y$ is a process that either executes x or y, but not both.

E_{BPA} consists of five axioms

$$\begin{array}{ll} x + y = y + x & \text{(A1) Commutativity of sum} \\ x + y) + z = x + (y + z) & \text{(A2)-Associativity of sum} \\ x + x = x & \text{(A3)- Idempotency of sum} \\ (x + y)z = xz + yz & \text{(A4)- Right distributivity of product} \\ (xy)z = x(yz) & \text{(A5)- Associativity of product} \end{array} \quad (27)$$

The alternative composition, $x + y$, implies a nondeterministic choice between x and y and can be represented as two branches in a state transition diagram. The fourth axiom $(x + y)z = xz + yz$ says that a choice between x and y followed by z is the same as a choice between xz, and yz and then either x followed by z or y followed by z. Note that the following axiom is missing from the definition of BPA

$$x(y + z) = xy + xz. \quad (28)$$

The reason for this omission is that in $x(y + z)$ the component x is executed first and then a choice between y and z is made while in $xy + xz$ a choice is made first and only then either x followed by y or x followed by z are executed. Processes are thus characterized by their branching structure and indeed the two processes $x(y+z)$ and $xy+xz$ have a different *branching structures*. The first process has two sub-processes and a branching in the second subprocess, whereas the second one has two branches at the beginning.

3.13 Further readings

Seminal papers in distributed systems are authored by Many Chandy and Leslie Lamport [52], by Leslie Lamport [140], [141], [142], Hoare [112], and Milner [168]. The collection of contributions with the title "Distributed systems", edited by Sape Mullender includes some of these papers.

Petri Nets were introduced in [199]. An in-depth discussion on concurrency theory and system modelling with PNs can be found in [200].

The text "Computer networks: a top-down approach featuring the Internet" by J.A. Kurose and K. W. Ross is a good introduction to networking. A recent text of Saltzer and Kaashoek [213] covers basic concepts in computer system design.

3.14 History notes

Two theoretical developments in 1930s were critical for the development of modern computers; the first was the publication of Alan Turing's 1936 paper [242]. 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[7] 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 was `ENIAC` built at the Moore School of Electrical Engineering at Penn State and became operational in July 1946; unlike `Z3`, `ENIAC` used a decimal number system and was program-controlled by patch cables and switches.

Third-generation computers were built during the $1964 - 1971$ period; they made extensive use of integrated circuits (ICs) and run under the control of an operating systems. The microprocessor was invented in 1971. In 1969 the UNIX operating system was developed for a DEC PDP minicomputer by Kenneth Thompson and Dennis Ritchie at the Bell Labs.

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 (Advanced Research Project Agency), encouraged Lawrence Roberts who had worked previously at MITs Lincoln laboratories to become the Chief Scientist at ISTO and to initiate a networking project based on packet switching rather than circuit switching. In the early 1960 Leonard Kleinrock at UCLA had developed the theoretical foundations for packet networks and in early 1970 for hierarchical routing in packet switching networks. The first two nodes of the ARPANET interconnected were the Network Measurement Center at the UCLA's School of Engineering and Applied Science and the SRI International in Menlo Park, California; by the end of 1971 there were 15 sites interconnected by ARPANET.

[7]A Turing complete computer is equivalent to a universal Turing machine modulo memory limitations

4 Cloud Infrastructure

In this chapter we overview the cloud computing infrastructure offered at this time by Amazon, Google, and Microsoft; they support one or more of the three cloud computing paradigms discussed in Section 2.3: IaaS (Infrastructure as a Service), SaaS (Software as a Service), and PaaS (Platform as a Service). Amazon is a pioneer in IaaS, while Google's efforts are focused on SaaS and PaaS paradigms. Sun and IBM offer their own cloud computing platforms, *Sun Cloud* and *Blue Cloud*, respectively. In 2011, HP announced plans to enter the cloud computing club.

Private clouds are an alternative to commercial cloud platforms. Open-source cloud computing platforms such as *Eucaliptus* [185], *OpenNebula*, and *Nimbus* can be used as a control infrastructure for a private cloud. We continue our discussion of the cloud infrastructure with an overview of SLAs (Service Level Agreements), followed by a brief discussion of software licensing and energy consumption and ecological impact of cloud computing. We conclude with a section covering user experiences with current systems.

4.1 Cloud computing at Amazon

Amazon was one of the first providers of cloud computing (*http://aws.amazon.com*); it announced a limited public beta release of its Elastic Computing platform called *EC2* in August 2006. *EC2* is based on the `Xen` virtualization strategy discussed in detail in Section 6.6. In *EC2* each virtual machine functions as a virtual private server and it 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.

Amazon uses two categories, *Region* and *Availability Zone* to describe the physical and virtual placement of the systems providing each type of service. For example, the *S3* storage service is available in the US Standard and US West, Europe, and Asia Pacific Regions; the corresponding storage facilities are located in Northern Virginia and Northern California, Ireland, Singapore and Tokyo, respectively. An application developer has the option to use these categories to reduce the communication latency, minimize costs, satisfy address regulatory requirements, and increase reliability and security.

The *AWS - Amazon Web Services* infrastructure offers a palette of services available through the *AWS Management Console* discussed next; these services includ:

- *Elastic Compute Cloud (EC2)*;
- *Simple Storage System (S3)*;
- *Elastic Block Store (EBS)*;
- *Simple DB*;
- *Simple Queue Service (SQS)*;
- *CloudWatch*; and
- *Virtual Private Cloud (VPC)*.

Elastic Compute Cloud[8] 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. *EC2* allows a user to load instances of an application with a custom application environment, manage networks access permissions, and run the images using as many or as few systems as desired. *EC2* instances boot from an AMI (Amazon Machine Image) digitally signed and stored in *S3*; one could use the few images provided by Amazon or customize an image and store it in *S3*.

A user can interact with *EC2* using a set of SOAP (Simple Object Access Protocol) messages and can list available AMI images, boot an instance from an image, terminate an image, display user's running instances, 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.

EC2 allows the import of virtual machine 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. Another facility called *auto scaling* empowers the user to scale seamlessly up and down the number of instances used by an application. The *EC2* system offers several instance types:

- Standard instances: micro (StdM), small (StdS), large (StdL), extra large (StdXL); small is the default.

- High memory instances: high-memory extra large (HmXL), high-memory double extra large (Hm2XL), and high-memory quadruple extra large (Hm4XL).

- High CPU instances: high-CPU extra large (HcpuXL).

- Cluster computing: cluster computing quadruple extra large (Cl4XL).

Table 2 summarizes the features and the amount of resources supported by each instance. The resources supported by each configuration are: main memory, virtual computers (VCs) with a 32 or 64-bit architecture, instance memory (I-memory) on persistent storage, and I/O performance at two levels, moderate (M) of high (H). The computing power of a virtual core is measured in EC2 compute units (CUs).

A main attraction of the Amazon cloud computing is the low cost; the dollar amounts charged for one hour of running under Linux or Unix and Windows at the time of this writing are summarized in Table 3.

Simple Storage System is a storage service with a minimal set of functions: write, read, and delete. It allows an application to handle an unlimited number of objects ranging in size from 1 byte to 5 TB. 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

[8]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.

Table 2: The nine instances supported by *EC2*. The cluster computing `Cl4XL` (quadruple extra large) instance uses 2 Intel Xeon X5570, quad-core "Nehalem" architecture processors. The instance memory (I-memory) refers to persistent storage; the I/O performance can be moderate (M) or high(H).

Instance name	API name	Platform (32/64-bit)	Memory (GB)	Max EC2 compute units	I-memory (GB)	I/O (M/H)
StdM		32 and 64	0. 633	1 VC; 2 CUs		
StdS	`m1.small`	32	1.7	1 VC; 1 CU	160	M
StdL	`m1.large`	64	7.5	2 VCs; 2×2 CUs	85	H
StdXL	`m1.xlarge`	64	15	4 VCs; 4×2 CUs	1,690	H
HmXL	`m2.xlarge`	64	17.1	2 VCs; 2×3.25 CUs	420	M
Hm2XL	`m2.2xlarge`	64	34.2	4 VCs; 4×3.25 CUs	850	H
Hm4XL	`m2.4xlarge`	64	68.4	8 VCs; 8×3.25 CUs	1,690	H
HcpuXL	`c1.xlarge`	64	7	8 VCs; 8×2.5 CUs	1,690	H
Cl4XL	`cc1.4xlarge`	64	18	33.5 CUs	1,690	H

Table 3: The costs in dollars for one hour of running under Linux or Unix and for running under Windows for several EC2 instances.

Instance	Linux/Unix	Windows
StdM	0.007	0.013
StdS	0.03	0.048
StdL	0.124	0.208
StdXL	0.249	0.381
HmXL	0.175	0.231
Hm2XL	0.4	0.575
Hm4XL	0.799	1.1
HcpuXL	0.246	0.516
Cl4XL	0.544	N/A

object: the name, modification time, an access control list, and up to 4 KB 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. The *Amazon S3 SLA* guarantees reliability. *S3* uses standards-based `REST`[9] and `SOAP`[10] interfaces; the

[9]REST is a style of software architecture for distributed hypermedia systems with clients initiating requests and servers processing the requests and returning responses. Requests and responses are built around a representations of resources; a representation of a resource is typically a document that captures the current or intended state of a resource.

[10]SOAP is a protocol specification for Web Services. The message format is XML (Extensible Markup Language); message transmission is based on RPC (Remote Procedure Call) and HTTP (Hypertext Transfer Protocol). SOAP consists of three parts: an envelope, which defines what is in the message and how to process it; a set of encoding rules for expressing instances of application-defined datatypes; and a convention for representing procedure calls and responses.

default download protocol is `HTTP` and a `BitTorrent`[11] protocol interface is provided to lower costs for high-scale distribution. *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. *S3* computes the *MD5* [12] 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. *S3* is designed to store large objects.

Elastic Block Store provides persistent block level storage volumes for use with Amazon *EC2* instances. A volume appears to an application as a raw, unformatted and reliable physical disk. The size of the storage volumes range from 1 GB to 1 TB; 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. The *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.

SimpleDB 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. *SimpleDB* 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 is a hosted message queue. *SQS* is a system for supporting automated workflows; it allows multiple Amazon *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 *QS* 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 5.6.

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 free of charge either seven or eight pre-selected metrics collected at one or at five minute intervals and then view graphs and statistics for these metrics.

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

[12]MD5 (Message-Digest Algorithm) is a widely used cryptographic hash function; it produces a 128-bit hash value.

Virtual Private Cloud 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.

In 2007 Garfinkel reported the results of an early evaluation of the Amazon Web Services [92]. The paper 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* 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 that the *S3* system could deliver to a single client thread. From the measurements the author concluded that the data throughput for large objects was considerably larger than for small objects 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, roughly three times the aggregate bandwidth of one thread. In 107556 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.

The AWSLA (Amazon Web Services Licensing Agreement) allows the company to terminate service to any customer at any time for any reason and contains a covenant not to sue Amazon or its affiliates as the result of any damages that might arise out of the use of AWS. As noted in [92], 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, libellous, 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."

4.2 Cloud computing, the Google perspective

Google's efforts are concentrated in the area of Software as a Service (SaaS); *Gmail, Google docs, Google calendar, Picassa* and *Google Groups* are Google services 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 such as iPhones, iPads, BlackBerries, and laptops and tablets; the data for these services is stored at data centers on the cloud.

The *Gmail* service hosts the 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 and to view the history of document changes and it has a spell checker. The service allows users to import and export files in several formats including 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. 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 is also a leader in the Platform-as-a-Service (PaaS) space. AppEngine is a developer platform hosted on the cloud; initially it supported only Python and support for Java was added later; detailed documentation for Java is available. The database for code development can be accessed with GQL (Google Query Language) with an SQL-like syntax.

The concept of *structured data* is important for Google's service strategy. The change of search philosophy reflects the transition from unstructured Web content to structured data, data which contain additional information, e.g., the place where a photograph was taken, information about the singer of a digital recording of a song, the local services at a geographic location, and so on [157].

Search engine crawlers rely on hyperlinks to discover new content. The *deep web* is content stored in databases and served as pages created dynamically by querying HTML forms; such content is unavailable to crawlers which are unable to fill out such forms. Examples of deep Web sources are: sites with geographic-specific information such as local stores, services, and business; sites which report statistics and analysis produced by governmental and non governmental organizations; art collections; photo galleries; bus, train, and airlines schedules, and so on. Structured content is created by labelling; *Flickr* and *Google Co-op* are examples of structures where labels and annotations are added to objects, images and pages, stored on the Web. *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.

Specialized structure-aware search engines for several areas, including travel, weather and local services, have already been implemented. But 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 tablet running *Chrome-OS*. Cloud-based apps, extreme portability, built-in 3G connectivity, almost instant-on, and all-day battery life are the main attractions of this tablet with a keyboard.

Google adheres to a bottom-up, engineer-driven, and 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.

4.3 Windows Azure and Online Services

Azure and *Online Services* are PaaS (Platforms as a Service) and, respectively, SaaS (Software as a Service) cloud platforms from Microsoft. *Windows 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.

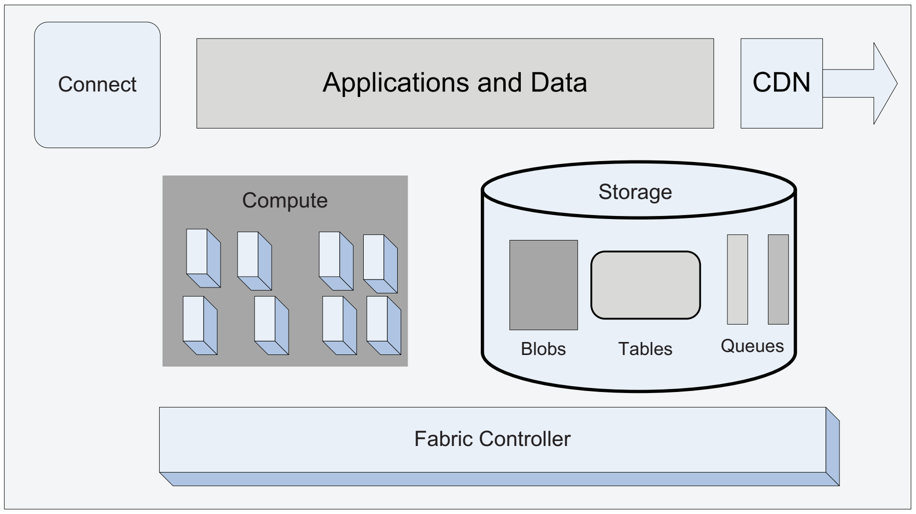


Figure 20: The components of Windows Azure: Compute - runs cloud applications; Storage - uses blobs, tables, and queues to store data; Fabric Controller - deploys, manages, and monitors applications; CDN - maintains cache copies of data; Connect - allows IP connections between the user systems and applications running on Windows Azure.

Windows Azure has three core components (see Figure 20): *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 speed up 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 CRM.* 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 which run a user-provided Windows Server 2008 R2 image.

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 of the resources in its environment: computers, switches, load balancers, and it is aware of every Windows Azure application. The fabric controller decides where new applications should run; it chooses the physical servers to optimize utilization using configuration information uploaded with each Windows Azure application. The configuration information is an XML-based description of how many Web role instances, how many Worker role instances, and of other needs of the application; 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[13] 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 [102].

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 Google with Gmail, Google docs, Google calendar, Google Groups, and Picassa 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 Amazon 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

[13]NTFS (New Technology File System) is the standard file system of Windows

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 the administrative costs are virtually zero for PaaS.

4.4 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, Virtual Machines Monitors (VMM) running on individual systems, an archive containing disk images of Virtual Machines (VMs), a front end for communication with the user, and a cloud control infrastructure. Open-source cloud computing platforms such as *Eucaliptus* [185], *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 VMM (Virtual Machine Monitor) running on that system to setup a VM;
- invokes the DHCP[14] and the IP bridging software to set up a MAC and IP address for the VM.

We now discuss briefly the three open-source software systems, *Eucalyptus, OpenNebula*, and *Nimbus.*

Eucalyptus (http://www.eucalyptus.com/) can be viewed as an open-source counterpart of Amazon's *EC2*. 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, the analog of Amazons *S3* system called Walrus. The procedure to construct a virtual machine is based on the generic one described above [219]:

[14]The Dynamic Host Configuration Protocol (DHCP) is an automatic configuration protocol; it 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, and each client computer on the LAN is configured to request an IP address from the DHCP server during network initialization. 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 table with MAC address/IP address pairs, which are manually filled in; only clients with a MAC address listed in this table will be allocated an IP address.

- the *euca2ools* front-end is used to request a VM;
- the VM disk image is transferred to a compute node;
- this disk image is modified for use by the VMM on the compute node;
- the compute node sets up network bridging to provide a virtual NIC[15] with a virtual MAC address[16];
- in the head node the DHCP is set up with the MAC/IP pair;
- VMM 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 5 configurations for 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 virtual machine consists of several steps: (i) a user signs in to the head node using `ssh`[17]; (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 VMM are transferred to the compute node via the NFS; (vii)) the VMM on the compute note starts the VM; (viii) the user is able to `ssh` directly to the VM on the compute node.

According to the analysis in [219], 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 the running Nimbus process can `ssh` into all compute nodes. Customization in this system can only be done by the system administrators.

Table 4 from [219] summarizes the features of the three systems. The conclusions of the comparative analysis are as follows: *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

[15]A Network Interface Controller is the hardware component connecting a computer to a LAN (Local Area Network); it is also known as a network interface card, network adapter, or LAN adapter.

[16]A Media Access Control address (MAC address) is a unique identifier assigned to network interfaces for communications on the physical network the device is connected to.

[17]Secure Shell (`ssh`) is a 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.

Table 4: A side-by-side comparison of *Eucaliptus*, *OpenNebula*, and *Nimbus*.

	Eucaliptus	*OpenNebula*	*Nimbus*
Philosophy	Mimics EC2	Highly customizable private cloud	Many Globus features
Best suited for	Private clouds with many users	Small configurations with trusted users	Scientific computing
Customizability	Administrators limited users	Administrators and users	All but image storage and credentials
Internal security	Root access required for many actions	Lose; can be made tighter	Tight
User security	User credentials via Web interface	Users log to the head	x509 credentials with the cloud
DHCP	On cluster controller	-	On each compute node

adequate for a scientific community less interested in the technical internals of the system, but with broad customization requirements.

4.5 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 customers 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.

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 "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 of 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: (i) abandonment rate - percentage of calls abandoned while waiting to be answered; (ii) average speed to answer - average time before the service desk answers a call; (iii) time service factor - percentage of calls answered within a definite time frame; (iv) first-call resolution - percentage of incoming calls that can be resolved without a callback; and (v) 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 fulfilment in real-time. In turn, automated negotiation has three main components: (i) the *object of negotiation* which define the attributes and constraints under negotiation; (ii) the *negotiation protocols* which describe the interaction between negotiating parties, and (iii) the *decision models* responsible for processing proposals and generating counter proposals.

The concept of compliance in cloud computing is discussed in [45] 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) [20] 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 [258].

4.6 Software licensing

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

Only very 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 [47]. The *Software as a Service (SaaS)* 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). 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 licence and billing Web-based services [47]. The architecture of the *elasticLM* license service has several layers: coallocation, 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 licence 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.

4.7 Energy use and ecological impact of large-scale data centers

In this section we discuss the energy use of data centers and its economical and ecological impact. We start our discussion with a brief analysis of the concept of energy-proportional systems. This is a very important concept because a strategy for resource management in a computing cloud is that, whenever possible, to concentrate the load on a subset of servers and switch the rest of them to a standby mode. This strategy aims to reduce the power consumption and, implicitly, the cost for providing computing and storage services; we analyze it in depth in Chapter 7

The operating efficiency of a system is captured by an expression of performance per Watt of power. It is widely reported that during the last two decades the performance of computing systems has increased much faster than their operating efficiency; for example, during the period 1998 till 2007, the performance of supercomputers has increased 7000% while their operating efficiency has increased only 2000%.

In an ideal world, the energy consumed by an idle system should be near zero and should grow linearly with the system load. In real life, even machines whose power requirements scale linearly use when idle more than half the power they use at full load, see Figure 21 from [38].

Energy-proportional systems could lead to large savings in energy costs for computing clouds. 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 $1000 - 2000$ W during a strenuous, short time effort [38].

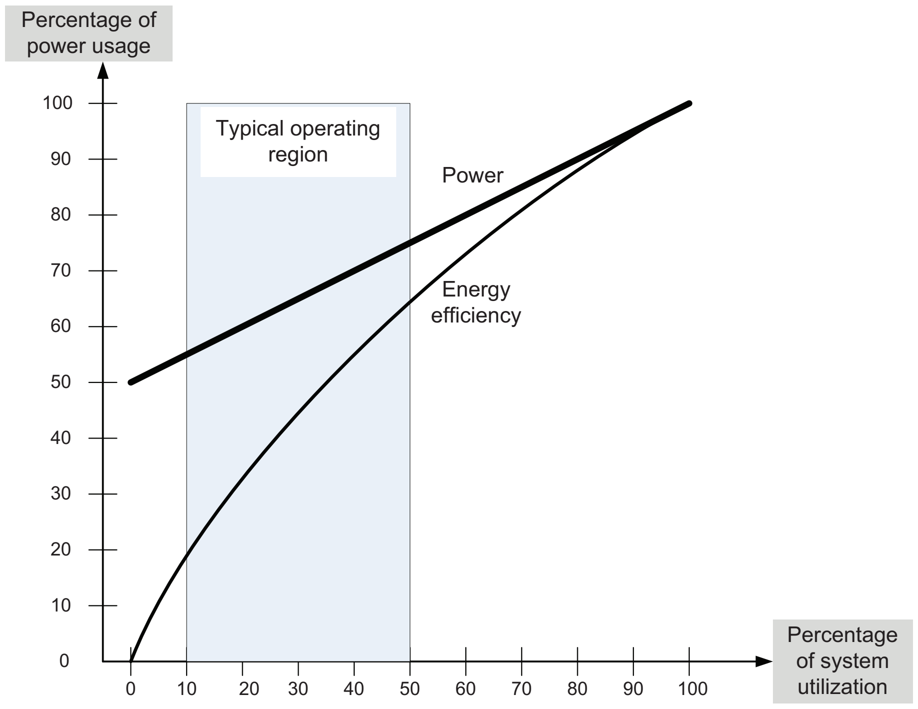


Figure 21: Even when power requirements scale linearly with the load the energy efficiency of a computing system is not a linear function of the load; even when idle, a system may use 50% of the power corresponding to the full load. Data collected over a long period of time show that the typical operating region for the servers at a data center is from about 10% to 50% of the load.

Different subsystems of a computing system behave differently in terms of energy efficiency; while many processors have reasonably 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. According to [38] the dynamic power range of other components of a system is much narrower: less than 50% for DRAM, 25% for disk drives, and 15% for networking switches.

A number of proposals have emerged for *energy proportional* networks; the energy consumed by such networks is proportional with the communication load. For example, in [6] the authors argue that a data center network based on a flattened butterfly topology is more energy and cost efficient. High-speed channels typically consist of multiple serial lanes with the same data rate; a physical unit is striped across all the active lanes. Channels commonly operate plesiochronously[18] and are always on, regardless of the load, because they must still send idle packets to maintain byte and lane alignment across the multiple lanes. An example of an energy proportional network is *InfiniBand*[19], a network with a dynamic range; its

[18]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 than that of the I/O channels.

[19]*InfiniBand* is a communications link with a switched fabric topology designed to be scalable; it has high throughput, low latency, and supports quality of service guarantees and failover - the capability to switch to

architectural specification defines multiple operational data rates. 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.

Energy saving in large-scale storage systems is also of concern. 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 [246]; measurement results indicate that it performs better than LRU, MRU, and LFU[20] policies for a range of file sizes, file availability, and number of client nodes and the power requirement are reduced by as much as 31%. Another technique is based on data migration. The system in [106] uses data storage in virtual nodes managed with a distributed hash table; the 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.

The energy consumption of large-scale data centers and their costs for energy and for cooling are significant now and are expected to increase substantially in the future. In 2006, the 6000 data centers in the U.S. reportedly consumed 61×10^9 KWh of energy, 1.5% of all electricity consumption in the country, at a cost of \$4.5 billion [246].

The predictions were dire: the energy consumed by the data centers was expected to double from 2006 to 2011; peak instantaneous demand was expected to increase from 7 GW in 2006 to 12 GW in 2011, requiring the construction of ten new power plants. The energy consumption of data centers and the network infrastructure is predicted to reach 10300 TWh/year[21] in 2030, based on 2010 levels of efficiency [202]. 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.

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 [6]. 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.

Many proposals argue that dynamic resource provisioning is necessary to minimize power consumption. Two main issues are critical for energy saving: the amount of resources allocated to each application and the placement of individual workloads. For example, a resource management framework combining a utility-based dynamic Virtual Machine provisioning manager with a dynamic VM placement manager to minimize power consumption

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; it supports several signalling rates. Links can be bonded together for additional throughput.

[20]LRU (Least Recently Used), MRU (Most Recently Used), and LFU(Least Frequently Used) are replacement policies used by memory hierarchies for caching and paging.

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

and reduce Service Level Agreement violations is presented in [243].

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 [202].

The same study reports that if the energy demand for bandwidth is 4 Watts-hour per Megabyte[22] and if the demand for network bandwidth is 3.2 Gbytes/day/person or 2570 Exabytes/year for the entire world population, then the energy required for this activity will be 1175 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.

The power consumption required by different types of human activities is partially responsible for the greenhouse gas emissions. According to a recent study [202], the greenhouse gas emission due to the data centers is estimated to increase from 116×10^6 tones of CO_2 in 2007 to 257 tones in 2020 due primarily to increased consumer demand. 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 [255].

4.8 User experience

There are 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 [193]. 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; (iv) data leaks; and (v) 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

[22]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.

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

An IBM study of a large customer pool identified the attractions for cloud computing and the percentage of responders who considered the element as critical [118]:

- 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%;
- Able to take advantage of the latest functionality: 40%;
- Relieve pressure on internal resources: 39%;
- Solve problems related to updating/upgrading: 39%;
- Rapid deployment: 39%;
- Able to scale up resources to meet the needs: 39%;
- Able to focus on core competencies: 38%;
- Take advantage of the improved economics of scale: 37%;
- Reduce infrastructure management needs: 37%;

- Lower energy costs: 29%;
- Reduce space requirements: 26%;
- Create new revenue streams: 23%;

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.

4.9 History notes

Amzon is one of the first providers of cloud computing; one year after the beta release of `EC2` in 2006, two new instance types (Large and Extra-Large) were added followed by two more types High-CPU Medium and High-CPU Extra Large in 2008 and new features including: static IP addresses, Availability Zones, and User Selectable Kernels as well as the Block Store (EBS). Amazon EC2 has been in full production mode since October 2008 and supports: a service level agreement, as well as the Windows operating system and the Microsoft SQL Server.

4.10 Further readings

Information about cloud computing at Amazon, Google, Microsoft, and HP and about the open source platforms is available online:

- 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
- *Eucalyptus:* http://www.eucalyptus.com/
- *Open-Nebula:* http://www.opennebula.org/
- *Nimbus:* http://www.nimbusproject.org/

Energy consumption is discussed in [6], [202], [243], and [246].

Additional information about SLAs is available online: a web service level agreement (WSLA) at *http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf.*, a white paper on SLA specification at *http://www.itsm.info/SLA%20description.pdf*, and a toolkit at *http://www.service-level-agreement.net.*

Energy use and ecological impact are discussed in [6], [106], [202], [243], and [246].

5 Cloud Computing

The efforts to support large-scale distributed computing encountered major difficulties along the years. The users of these systems discovered how difficulties to locate the systems needed to run an application and to distribute the computations and coordinate them. The users discovered that it is equally difficult to scale up and down to accommodate a dynamic load, to recover after a system failure, and to support effectively checkpoint/restarting procedures. At the same time, the providers of computing cycles and storage realized the difficulties to manage a large number of systems and to provide guarantees for the quality of service. The actual cost of assembling and managing and their relatively low utilization offset any economic advantage offered by large-scale distributed systems.

The main economic attractions for the users of cloud computing are: a very low infrastructure investment, as there is no need to assemble and maintain a large-scale system; low *utility-based computing costs,* as the customers are only billed for the infrastructure used. At the same time, the users benefit from the potential to reduce the execution time of compute-intensive and data-intensive applications through parallelization; if the application can partition the work in n chunks and spawn n instances of itself, then the execution time could be reduced by a factor close to n. Moreover, the application developers enjoy the advantages of a *just-in-time infrastructure*, they are free to design an application without being concerned with the system where the application will run; oftentimes, an application becomes the victim of its own success, it attracts a user population larger than the system could support. Cloud computing is also beneficial for the providers of computing cycles as it typically leads to a more efficient resource utilization.

The 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 on one hand into the ability to provide satisfactory solutions to a host of critical problems related to security, scalability, reliability, quality of service, and to the requirements enforced by Service Level Agreements (SLAs) and, on the other hand, into the number of applications suites available for customers.

The main appeal of cloud computing is its focus on enterprise computing. This clearly differentiates it from the grid computing effort which was largely focused on scientific and engineering applications. Of course, the other major advantage of the cloud computing approach over the grid computing is the concentration of resources in large data centers in a single administrative domain.

It is expected that utility computing providers such as IBM, HP, Google, Oracle, Microsoft and others will develop in the next future application suites to attract customers. Microsoft seems well positioned in the effort to attract customers to its Azure platform as it already has many applications for enterprise customers, while Red Hat and Amazon may choose to stay only with their "infrastructure only" approach [25].

The 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. In Section 7.5 we discuss *arbitrarily divisible* workloads which can be partitioned into an arbitrarily large number of segments; the *arbitrarily divisible load sharing model* is common to many applications and these are precisely the applications suitable for cloud computing.

Not all applications are suitable for cloud computing; Web services, database services, and transaction-based services are ideal applications for cloud computing, while applications where the workload cannot be arbitrarily partitioned, or require intensive communication among concurrent instances are unlikely to perform well on a cloud. The resource requirements of transaction-oriented services are dependent on the current load, which itself is very dynamic; the cost/performance profile of such applications benefits from an elastic environment where resources are available when needed and where one pays only for the resources it consumes. On the other hand, an application with a complex workflow and multiple dependencies, as is often the case in high-performance computing, could require longer execution times and higher costs on a cloud. The benchmarks for high-performance computing discussed in Section 5.8 show that communication and memory intensive applications may not exhibit the performance levels shown on supercomputers with low latency and high bandwidth interconnects.

5.1 Challenges for cloud computing

The development of efficient cloud applications inherits the challenges posed by the natural imbalance between computing, I/O, and communication bandwidths of physical systems; these challenges are greatly amplified due the scale of the system, its distributed nature, and by the fact that virtually all applications are data-intensive. Though cloud computing infrastructures attempt to automatically distribute and balance the load, the application developer is still left with the responsibility to place the data close to the processing site and to identify optimal storage for the data.

One of the main advantages of cloud computing, the shared infrastructure, could also have a negative impact as perfect performance isolation[23] is nearly impossible to reach in a real system, especially when the system is heavily loaded. The performance of virtual machines fluctuates based on the load, the infrastructure services, the environment including the other users. Reliability is also a major concern; node failures are to be expected whenever a large numbers of nodes cooperate for the computations. Choosing an optimal instance, in terms of performance isolation, reliability, and security, from those offered by the cloud infrastructure is another critical factor to be considered. Of course, cost considerations play also 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 the systems of the cloud and communicating among them. Thus, efficiency, consistency, and communication scalability of communication are major concerns for an application developer. Indeed, due to shared networks and unknown topology, cloud infrastructures exhibit inter-node latency and bandwidth fluctuations which affect the application performance.

The data storage plays a critical role in the performance of any data-intensive application; the organization of the storage, the storage location, as well as, the storage bandwidth must be carefully analyzed to lead to an optimal application performance. Clouds support many storage options to set up a file system similar to the *Hadoop* file system; among them

[23]Performance and security isolation of virtual machines is discussed in Section 6.4.

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 a JPEG audio file may include the name of the song, 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. Performance considerations limit the amount of data logging, while the ability to identify the source of unexpected results and errors is helped by frequent 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.

5.2 Existing cloud applications and new application opportunities

Existing cloud applications can be divided in several broad categories: (i) Processing pipelines; (ii) Batch processing systems; and (iii) Web applications [244].

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 crawl 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 *Google* (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 cover also 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 economical sense to store the data in the cloud close to where the application runs; 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* from Google. 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[24], 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.

5.3 Workflows - coordination of multiple activities

Many cloud applications require the completion of multiple interdependent tasks; the description of a complex activity involving such an ensemble of tasks is known as an 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 conclude with a parallel between traditional transaction systems and cloud workflows.

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 modelling; a task is a unit of work to be performed on the cloud and it is characterized by several attributes, such as

[24] A meshup is an application that uses and combines data, presentation 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 meshups.

- 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 pairs `<event, action>`. The exceptions included in the task exception list are called *anticipated exceptions*, as opposed to unanticipated exceptions. In our model, events not included in the exception list trigger replanning. *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; and
3. iterative - the tasks between the fork and the join are executed repeatedly.

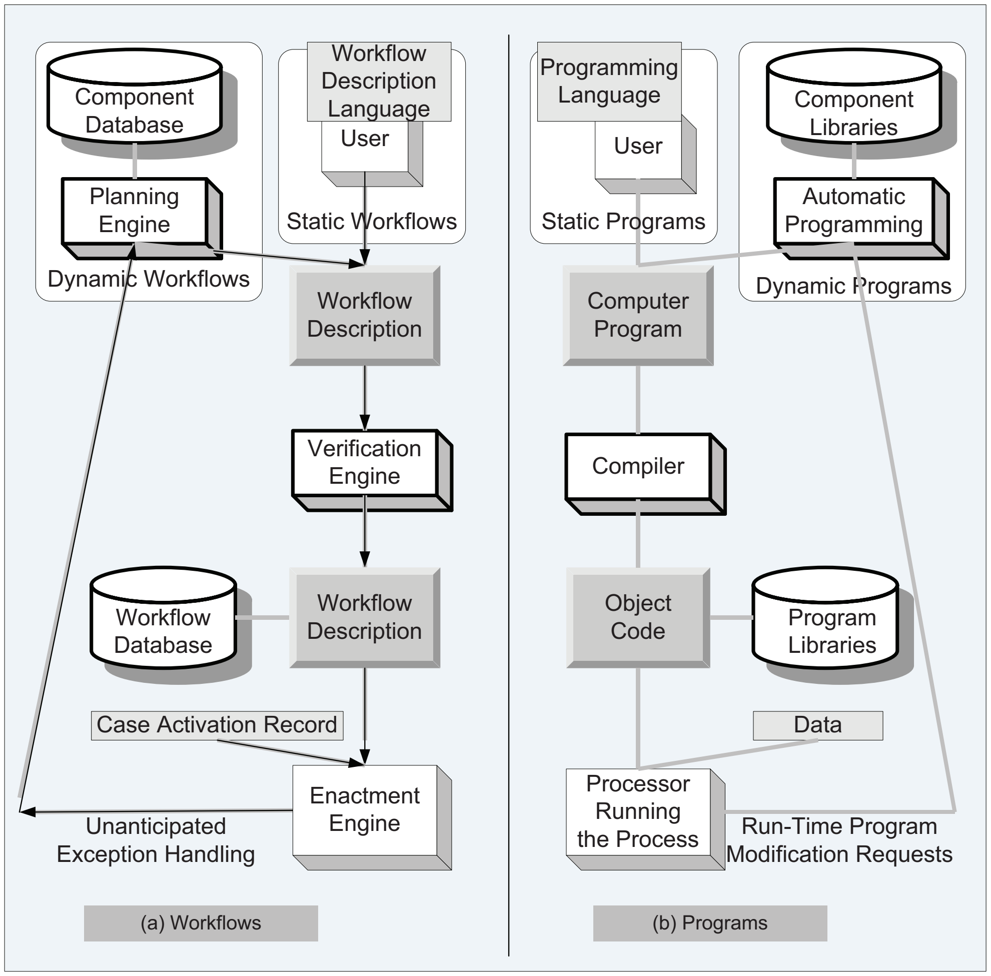


Figure 22: (a) The life cycle of a workflow. (b) The life cycle of a computer program. The workflow definition is analogous to writing a program; planning is analogous to automatic program generation; verification corresponds to syntactic verification of a program; and workflow enactment mirrors the execution of the compiled program.

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 22. 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 the identification of 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 automatically, knowing a set of tasks and the preconditions and postconditions of 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.

Among the most desirable properties of a process description are the *safety* and *liveness* of the process. Informally, safety means that nothing "bad" ever happens and 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 23(a) violates the liveness requirement. As long as task C is chosen after completion of B, the process will terminate. But 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.

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 many 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 23(b) when at time t_1 task A acquires r

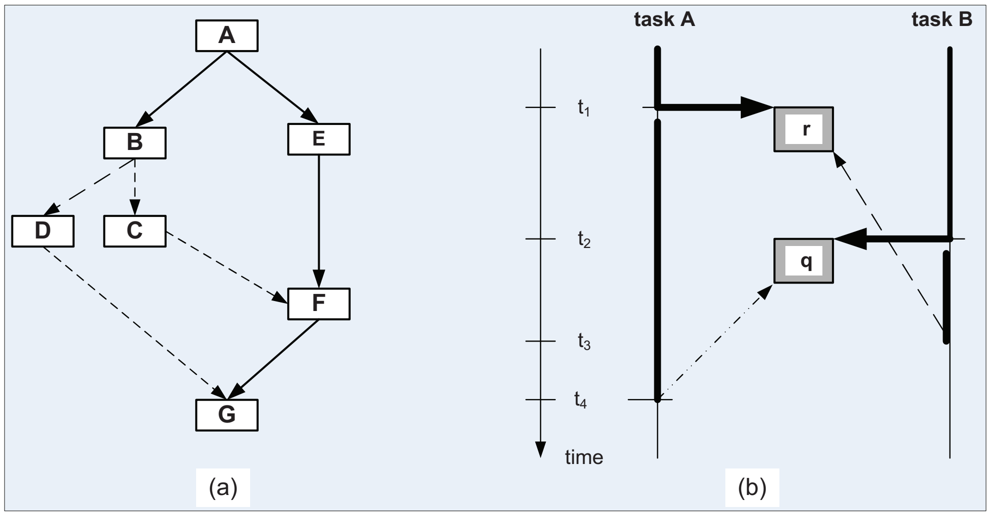


Figure 23: (a) A process description which violates the liveness requirement; if task C is chosen after completion of B, the process will terminate; if D is chosen, then F will never be instantiated because it requires the completion of both C and E. The process will never terminate because G requires 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.

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 23(b) can be avoided by requesting each task to acquire all resources at the same time; the price to pay is underutilization of resources; indeed, the idle time of each resource increases under this scheme.

Workflow pattern refers to the temporal relationship 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], and [260]. 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 illustrated in Figure 24 are:

- The *sequence* pattern occurs when several tasks have to be scheduled one after the completion of the other.

- The *AND split* pattern requires several tasks to be executed concurrently. Both tasks B and C are activated when task A terminates. 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

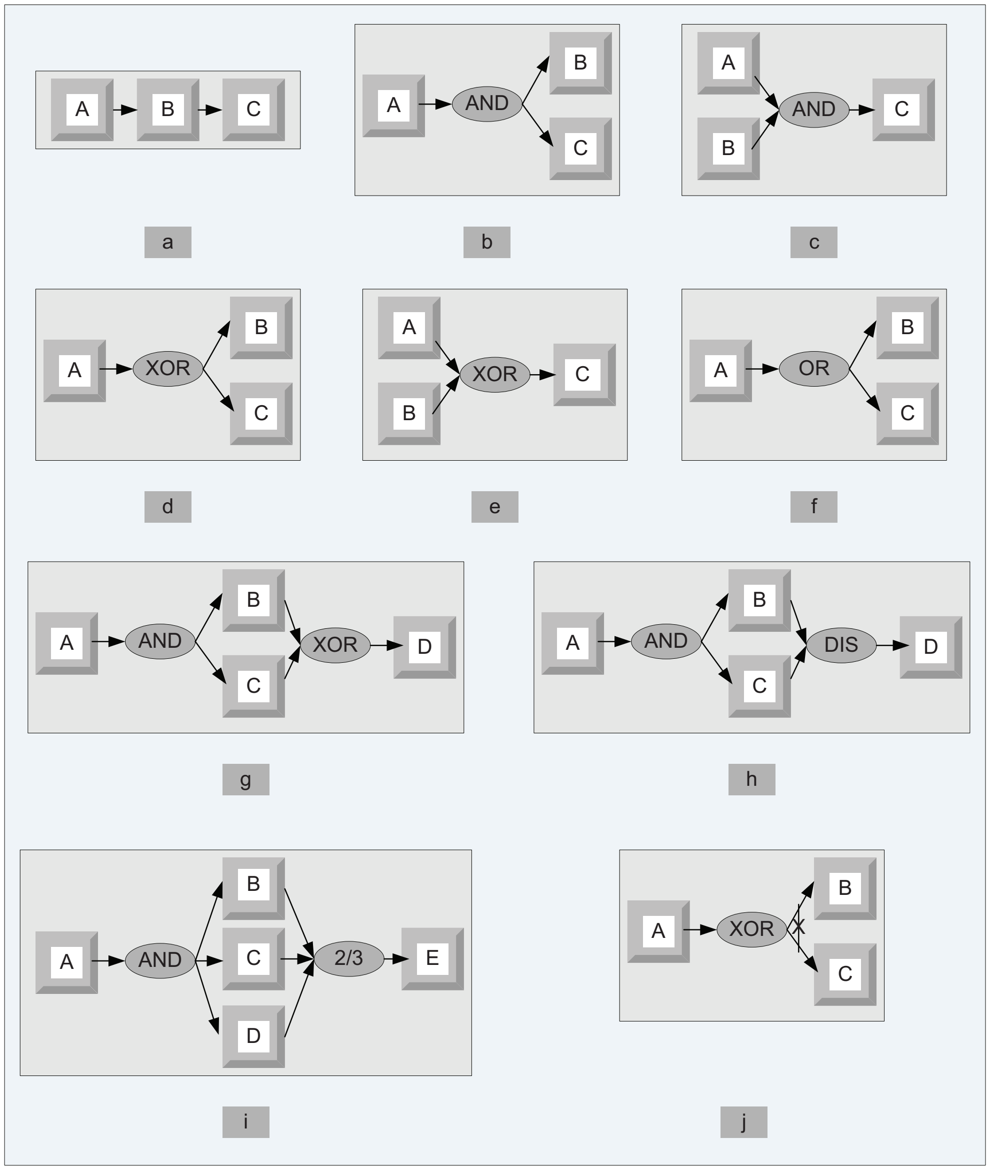


Figure 24: Basic workflow patterns. (a) Sequence; (b) AND split; (c) Synchronization; (d) XOR split; (e) XOR merge; (f) OR split; (g) Multiple Merge; (h) Discriminator; (i) N out of M join; (j) Deferred Choice.

implicit AND split, activities are connected directly and conditions can be associated with branches linking an activity with the next ones. Only when the conditions associated with a branch are true are the tasks activated.

- 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.

- The *XOR split* requires a decision; after the completion of task A, either B or C can be activated.

- The *XOR join*; several alternatives are merged into one, in our example task C is enabled when either A or B terminates.
- 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.
- The *multiple merge* construct allows multiple activation of a task and does not require synchronization after the execution of concurrent tasks. Once A terminates, tasks B and C execute concurrently. 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; then it waits for the remaining 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.
- 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.

Next we discuss the reachability of the goal state and we consider the following elements:

- 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, $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$.

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 exits such that the postconditions of one are preconditions of next process in the chain.

Generally, the preconditions of a process are 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 the 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 or processes. 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.

The most important feature of this coordination model is the ability to support 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 investigate the feasibility of the requested changes, and finally implement 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 to use 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 modelling level, whereas the differences affect in 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 a database transactional model is to maintain a consistent state of the database; a cloud is an open system, thus, its the 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.

5.4 Coordination based on a state machine model - the *ZooKeeper*

Cloud computing elasticity requires the ability to distribute computations and data across multiple systems; coordination among these systems is one of the critical function to be exercised in a distributed environment. The coordination model depends on the context, e.g., coordination of data storage, orchestration of multiple activities, blocking an activity until an event occurs, reaching consensus for the next action, or the 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. In the distributed data store model discussed in Section 2.5 the access of to data is mitigated by a proxy. The proxy is a single point of failure thus, an architecture with multiple proxies is desirable; these 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 which involves 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[25], advertising campaign planners, scenario testing, and so on. A solution to coordinate these activities is through configuration files which are 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 the re-distribution of the configuration file; moreover, in case of a system failure the configuration file does not allow recovery from the state of each server prior to the system crash, a more desirable alternative.

A solution for the proxy coordination problem is to consider a proxy as a deterministic finite state machine which performs the commands sent by clients in some sequence; the proxy has thus, a definite state and, when a command is received, it transitions to another

[25]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.

another state. When P proxies are involved, all of them must be synchronized and execute the same sequence of state machine commands; this can be ensured if all proxies implement a version of the Paxos consensus algorithm described in Section 3.10.

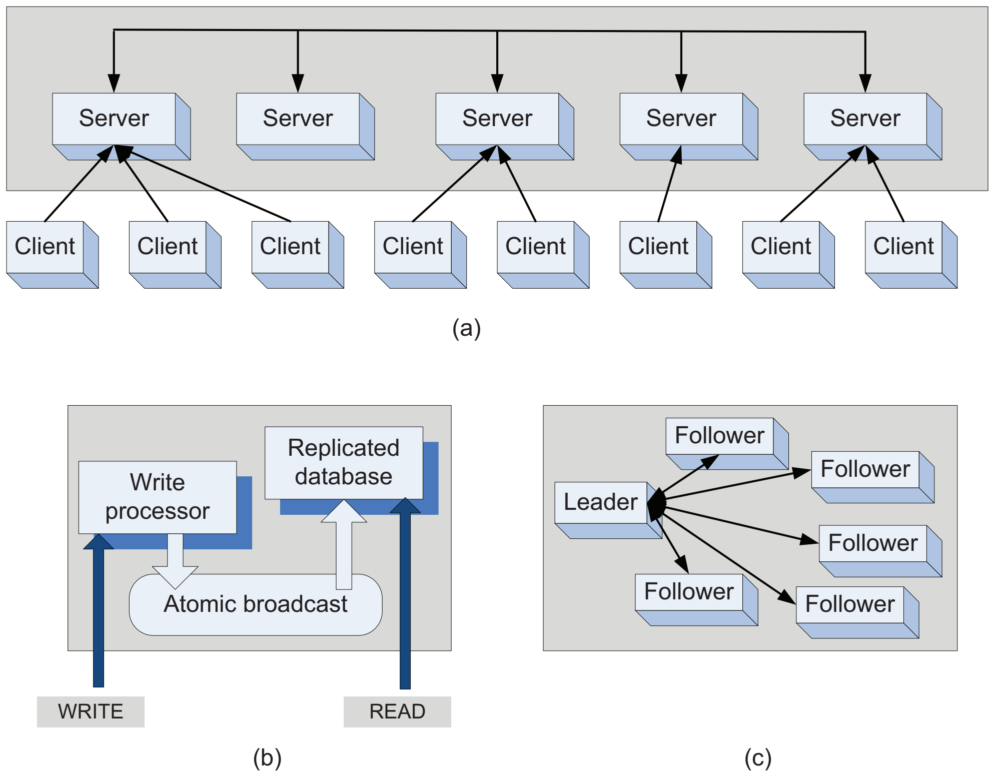


Figure 25: The *ZooKeeper* coordination service. (a) The service provides a single system image, clients can connect to any server. (b) Functional model of the *ZooKeeper* service; the replicated database is accessed directly by *READ* commands, while *WRITE* commands involve a more intricate processing based on atomic broadcast. (c) Processing a *WRITE* command - a server receiving the command from a client connected to it forwards the command to the *leader*; the *leader* uses atomic broadcast to reach consensus among all other servers which play the role of *followers*.

ZooKeeper is a distributed coordination service based on this model; a set of servers maintain consistency of the database replicated on each one of them. The open-source software is written in Java and has bindings for Java and C. The information about the project is available from the site http://zookeeper.apache.org/.

The organization of the service is shown in Figure 25; Figure 25(a) shows that the service provides a single system image, a client can connect to any server. Clients use TCP connections to servers and, if the server a client is connected to fails, the TCP connection times-out and the client detects the failure of the server and connects to another one. Figures 25(b) and (c) show that a *READ* operation directed to any server return the same result while the processing of a *WRITE* operation is more involved; the servers elect a *leader* and any *follower* receiving a request from one of the clients connected to it forwards it to the leader who 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. In the *ZooKeeper* the *znodes*, the equivalent of the *inodes* of a file system, can have data associated with it. Indeed, the system is designed to store state information; the data in each node includes version numbers for the data, changes of ACLs[26], and time stamps. This organization allows coordinated updates; the data retrieved by a client contains also a version number. The data stored in each node is read and written atomically, a *READ* returns all the data stored in the *znode*, while a *WRITE* replaces all the data.

The API to the *ZooKeeper* service 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 supports also the creation of *ephemeral* nodes, nodes created when a session starts and deleted when the session ends. The *ZooKeeper* service guarantees:

1. Atomicity - a transaction either completes of 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 will 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.

This brief description shows that the *ZooKeeper* service supports the finite state machine model of coordination; in this case a *znode* stores the state. The *ZooKeeper* service can be used to implement higher-level operations such as group membership, synchronization, and so on; the system is used by Yahoo's Message Broker and by several other applications.

[26]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.

5.5 The MapReduce programming model

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 [69]. 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* function outputs the pairs $< URL, 1 >$ and the *Reduce* function produces the pairs $< URL, totalcount >$. 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 from [69] 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 them 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.

- When all *Map* and *Reduce* tasks have been completed, the master wakes up the user program.

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 9.4.

An environment for experimenting with *MapReduce* is described in [69]: the computers are typically dual-processor x86 processors running Linux, with $2 - 4$ GB of memory per machine and commodity networking hardware typically $100 - 1000$ Mbps. A cluster consists of hundreds or thousands of machines. Data is stored on IDE[27] 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.

5.6 A case study: the GrepTheWeb application

An application called *GrepTheWeb* discussed in [244] is now in production at Amazon and we use it to illustrate 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 using as an argument a regular expression. The source of this search is a collection of document URLs produced by the *Alexa Web Search*, a software system that crawls the Web every night. The inputs to the applications are a regular expression and the large data set produced by the Web crawling software; the output is the set of records that satisfy the expression. The user is able to interact with the application and get the current status, see Figure 26 (a).

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. *MapReduce* was introduced by Google in 2004 for processing massive amounts of data. In the first step,

[27]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.

the "Map" step, the master node takes the input, partitions it into smaller sub-problems, and distributes them to worker nodes; a worker node may repeat the process, leading to a multi-level tree structure. A worker node processes the data allocated to it and passes the answer back to its master node. In the second phase, the "Reduce" phase, the master node merges the partial results and provides the answer to the problem it was originally trying to solve. MapReduce libraries are available for $C++$, $C\#$, Erlang, Java, OCaml, Perl, Python, PHP, Ruby, and other programming languages.

The details of the workflow of *GrepTheWeb* are captured in Figure 26 (b) from [244] and consist of the following steps:

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.

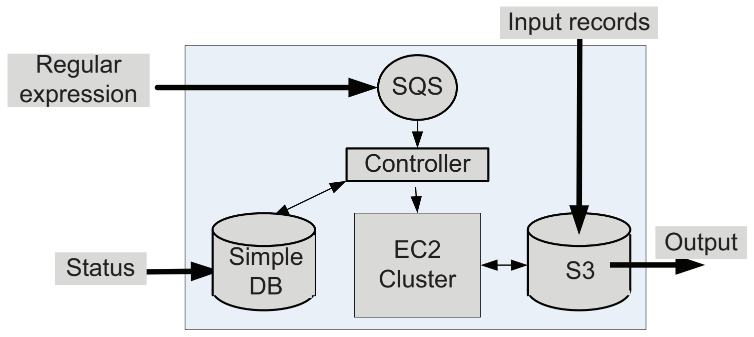
2. The processing phase: it is triggered by a `StartGrep` user request; then a launch message is enqueued in the launch queue. The launch controller thread picks up the message and executes the launch task; then, it updates the status and time stamps in the Amazon *SimpleDB* domain. Lastly, it enqueues a message in the monitor queue and deletes the message from the launch queue.

   (a) The launch task starts Amazon *EC2* instances using 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 Amazon *EC2* slave nodes in parallel; a map task takes files from Amazon *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) The final results are stored on Amazon *S3* in the output bucket.

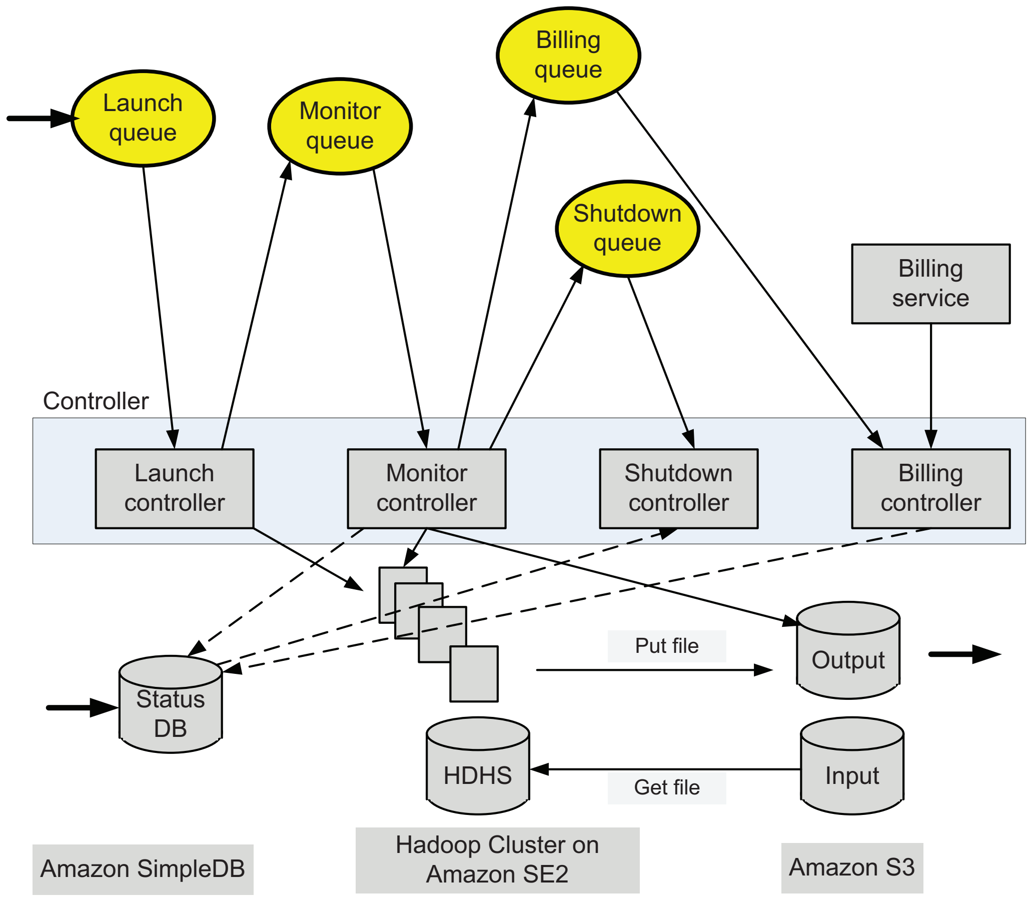
3. The monitoring phase: the monitor controller thread retrieves the message left at the beginning of the processing phase, validates the status/error in Amazon *SimpleDB* and executes the monitor task; it updates the status in the Amazon *SimpleDB* domain, enqueues messages in the shutdown and the billing queues. The monitor task checks for the Hadoop status periodically, updates the *SimpleDB* items with status/error and the Amazon *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 Amazon *SimpleDB* domain; finally, it deletes the message from the shutdown queue after processing.

   (a) The shutdown task kills the Hadoop processes, terminates the *EC2* instances after getting EC2 topology information from Amazon *SimpleDB* and disposes of the infrastructure.



(a)



(b)

Figure 26: The organization of the *GrepTheWeb* application which uses three Amazon services: *SimpleDB*, *SE2*, and *S3*, as well as the *Hadoop MapReduce* software. (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.

(b) The billing task gets the *EC2* topology information, *SimpleDB* usage, *S3* file and query input, calculates the charges, and passes the information to the billing service.

5. The cleanup phase archives the *SimpleDB* data with user info.

6. User interactions with the system to 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 Amazon *S3* after completion.

To optimize the end-to-end transfer rates in *S3* storage system multiple files were bundled up and stored as *S3* objects; another performance optimization was to run a script and sort the keys, the URL pointers, and upload them in sorted order in *S3*. Also, multiple fetch threads were started in order 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.

5.7 Clouds for science and engineering

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 [110]. Thousand of years ago the science was empirical, hundred of years ago theoretical methods based on models and generalization allowed a substantial progress, and in the last few decades we witnessed the explosion of computational science based on the simulation of complex phenomena. Today, the *eScience* unifies the experiment, theory, and simulation; data captured from measuring instruments, or generated by simulation 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: manage very large volumes of data, build and execute models, integrate data and the literature, document the experiments, share the data with others, and preserve it for a long time. All these activities can only be performed using large collections of 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[28], the astrophysics data from NASA[29], or the atmospheric data from NOAA[30] and NCAR[31]. The process of online data discovery can be viewed as an ensemble of several phases [196]: (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.

Metadata is used to describe digital data and it provides an invaluable aid for discovering useful information in a scientific data set. A recent paper, [196], describes a system for data

[28]NCBI is the National Center for Biotechnology Information, see http://www.ncbi.nlm.nih.gov/.

[29]NASA is the National Aeronautics and Space Administration, http://www.nasa.gov/

[30]NOAA is the National Oceanic and Atmospheric Administration, www.noaa.gov)

[31]NCAR is the National Center for Atmospheric Research

discovery which supports automated fine-grained metadata extraction and summarization schemes for browsing large data sets and it is extensible to different scientific domains. The system called *Glean* is designed to run on a computer cluster or on the cloud; its runtime system supports two computational models, one based on *MapReduce* and the other on graph-based orchestration.

5.8 Benchmarks for high performance computing on a cloud

A recent paper [121] 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 Laboratory and serves a diverse community of scientists; it has some 3000 users working on 400 projects and using 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[32]. 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 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 Energys 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[33]) is a code for fusion research[34]. It is a self-consistent, gyrokinetic tri-dimensional Particle-in-cell (PIC)[35] 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

[32]See http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/cam/

[33]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.

[34]See http://www.scidacreview.org/0601/html/news4.html.

[35]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.

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. Hockneys FFT algorithm is used to solve Poisson's equation with open boundary conditions. The code is sensitive to the memory bandwidth and MPI collective performance.

MAESTRO is a low Mach number hydrodynamics code for simulating astrophysical flows[36]. 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[37]. 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 gathers 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.,

[36]See http://www.astro.sunysb.edu/mzingale/Maestro/
[37]See http://physics.indiana.edu/∼sg/milc.html

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 [121] use the HPCC (High Performance Computing Challenge) benchmark to compare the performance of *EC2* with the performance of three large systems at NERSC. HPCC[38] (High Performance Computing Challenge) 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:

- DGEMM[39] - 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[40] - the benchmark measures the memory bandwidth.
- The network latency benchmark.
- The network bandwidth benchmark.
- HPL[41] - 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 9660 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

[38]For more information see http://www.novellshareware.com/info/hpc-challenge.html

[39]For more details see https://computecanada.org/?pageId=138

[40]For more details see http://www.streambench.org/

[41]For more details see http://netlib.org/benchmark/hpl/

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 (1584 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 (Compute Units) 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 [121] are summarized in Table 5.

Table 5: The results of the measurements reported in [121]

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 5 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, and lower bandwidth, more than 70 times smaller than the latency and bandwidth of *Carver*.

5.9 Cloud computing and biology research

Biology is one of the scientific fields that needs vast amounts of computing power 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 [154]. The authors carried out an "all-by-all" comparison to identify the interrelationship of the 10 million protein sequences (4.2GB size) in NCBIs non-redundant protein database using *AzureBLAST*, a version of the BLAST[42] program running on the Azure platform [154].

[42]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.

Azure offers VM with four levels of computing power depending on the number of cores: small (1 core), medium (2 cores), large (8 cores), and extra large (¿ 8 cores). The experiment used 8 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 3700 weighted instances or 475 extra-large VMs from three data centers; each data center hosted three *AzureBLAST* deployments, each with 62 extra large instances. The 10 million sequences were divided into multiple segments, each segment was submitted for execution by one *AzureBLAST* 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 few observations and conclusions useful for many scientific applications running on Azure were drawn after a post experiment analysis. A first observation is that when a task runs for more than 2 hours, a message will automatically reappear in the queue requesting the task to be scheduled thus, leading to repeated computations; a simple solution is to check if the result of a task has been generated before lunching it execution. Many applications, including *BLAST*, allow for the setting of some parameters but, the computational effort to find optimal parameters is prohibitive; a user is also expected to decide on an optimal balance between the cost and the number of instances to meet budget limitations.

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 of a job take considerably longer than others and delay the completion time of a 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.

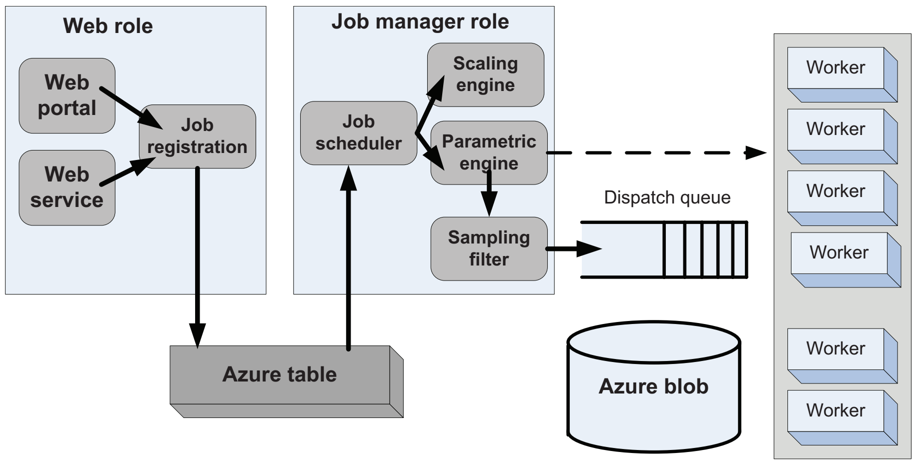


Figure 27: *Cirrus*, a general platform for executing legacy Windows applications on the cloud.

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 [146] 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 27; 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 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.

We continue our discussion of biology applications of the *Azure* infrastructure to a loosely-coupled workload for an ensemble-based simulations reported in [155]. 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 *AzureBLAST* 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 28 from [155] 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; 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 lunching 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.

5.10 Social computing, digital content, and cloud computing

In this section we discuss applications of cloud computing related to social networks. Social networks play an increasingly important role in people's lives; they have expanded in terms of the size of the population involved and in terms 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 nodes. 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; it turns out that 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.

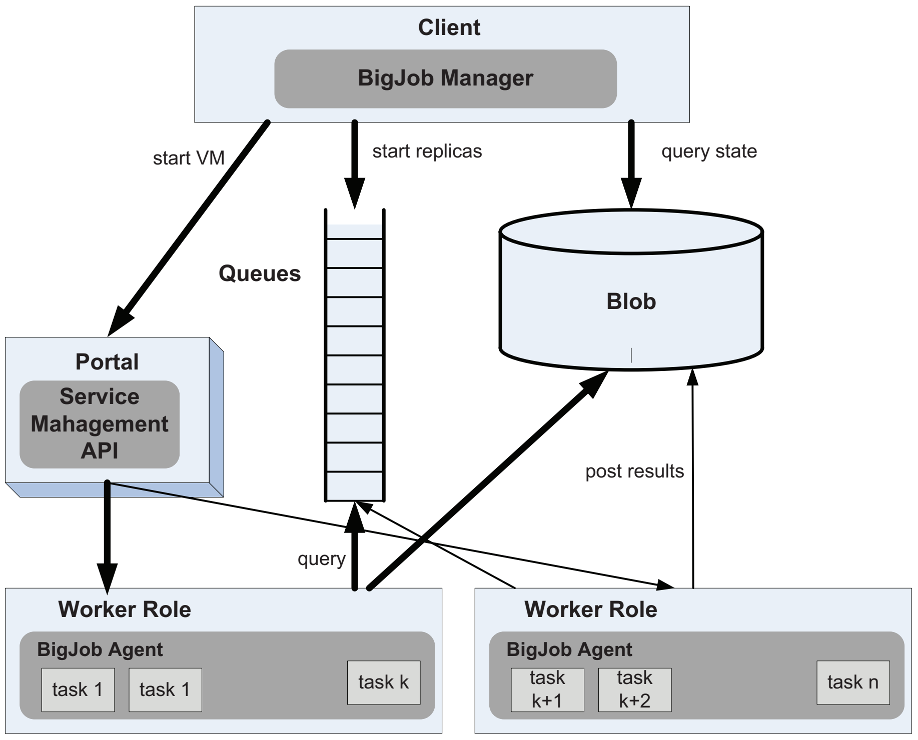


Figure 28: The execution of loosely-coupled workloads using the Azure platform.

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 as the amount of data increases.

A system based on CBR is described in [113]; the *BetterLife 2.0* system consists of a cloud layer, a case-based reasoning engine, and an API. The cloud layer uses the Hadoop Distributed File System 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 measurements between cases to retrieve the most similar ones and also stores new cases back to 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 user Id, and then a breadth 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 allows 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; for example, 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. 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.*

Such platforms cannot be used for an environment where users require some level of accountability as there are no SLAs. 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 [55]. 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 (FBML) is a subset of HTML with proprietary extensions and the Facebook JavaScript (FBJS) is a version of JavaScript parsed when a page is loaded to create a virtual application scope. 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, e.g., Facebook applications are hosted by the Amazon Web Services. Today, some organizations use the Facebook credentials of an individual for authentication.

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 [226] 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 [226] 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 sub-systems, 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 Service Level Agreement 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.

5.11 Further readings

MapReduce is discussed in [69]. The *GrepTheWeb* application is analyzed in [244]. Metadata generation for large scientific databases is presented in [196]. Cloud applications in biology are analyzed in [154] and [155]. Finally, social applications of cloud computing are presented in [55], [113], and [226].

Benchmarking of cloud services is analyzed in [59], [121], and [92].

6 Virtualization

Resource management for a community of users with a wide range of applications running under different operating systems is a very difficult problem; resource management becomes even more complex when resources are oversubscribed and users are uncooperative. The obvious solution for an organization providing utility computing is to install standard operating systems on individual systems and rely on conventional OS techniques to ensure resource sharing, application protection, and performance isolation. In this setup the system administration, accounting, and security, are very challenging for the providers of service, while application development and performance optimization are equally challenging for the users. The alternative is resource virtualization, a technique analyzed in this chapter.

Virtualization simulates the interface to a physical object by:

1. Multiplexing: create multiple virtual objects from one instance of a physical object. For example, a processor is multiplexed among a number of 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. 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 abstracts the underlaying resources and simplifies their use, isolates users from one another, and supports replication which, in turn, increases the elasticity of the system. 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:

1. System security, as it allows isolation of services running on the same hardware;

2. Performance and reliability, as it allows applications to migrate from one platform to another;

3. The development and management of services offered by a provider;

4. Performance isolation.

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 Virtual Machine architecture versus a traditional operating system.* For example, a user of the Amazon Web Services (AWS) could submit an Amazon Machine Image (AMI) containing the applications, libraries, data, and 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*. As we shall see shortly, all privileged operations of a virtual machine must be trapped and validated by the Virtual Machine Monitor which, ultimately, controls the system behavior and the increased overhead has a negative impact on the performance. The cost of the hardware for a virtual machine is higher than the cost for a system running a traditional operating system because the physical hardware is shared among a set of guest operating systems and it is typically configured with faster and/or multi-core processors, more memory, larger disks, and additional network interfaces as compared with a system running a traditional operating system.

We start our discussion with the motivation for virtualization and the interfaces, API (Application Programming Interface), ABI (Application Binary Interface), and ISA (Instruction Set Architecture) which define the properties of the system at different levels of abstraction. Next, we discuss alternatives for the implementation of virtualization in Sections 6.2 and 6.3 and analyze their impact on performance and security isolation in Section 6.4.

A critical question is the architectural support for virtualization discussed in Section 6.5. Resource sharing in a virtual machine environment requires not only ample hardware support and, in particular powerful processors, but also architectural support for multi-level control. Indeed, resources such as CPU cycles, memory, secondary storage, and I/O and communication bandwidth are shared among several virtual machines and, for each virtual machine, resources must be shared among multiple instances of an application. Traditional processor architectures were conceived for one level of control as they support two execution modes, the kernel and the user mode; in a virtualized environment all resources are under the control of a Virtual Machine Monitor (VMM) and a second level of control is exercised by the guest operating system. While a two-level scheduling for sharing CPU cycles can be easily implemented, sharing of resources such as cache, memory, and I/O bandwidth is more intricate.

Next we analyze the `Xen` VMM in Section 6.6 and discuss an optimization of its network performance in Section 6.7. High performance processors with multiple functional units such as Itanium do not provide explicit support for virtualization, as we discuss in Section 6.8.

The system functions critical for the performance of a virtual machine environment are cache and memory management, handling of privileged instructions, and I/O handling. An important source for the performance degradation in a virtual machine environment are the cache misses, as we shall see in Section 6.9. Finally, we analyze the security advantages of virtualization in Section 38 and then discuss software fault isolation in Section 6.11.

6.1 Layering and virtualization

A common approach to manage 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.

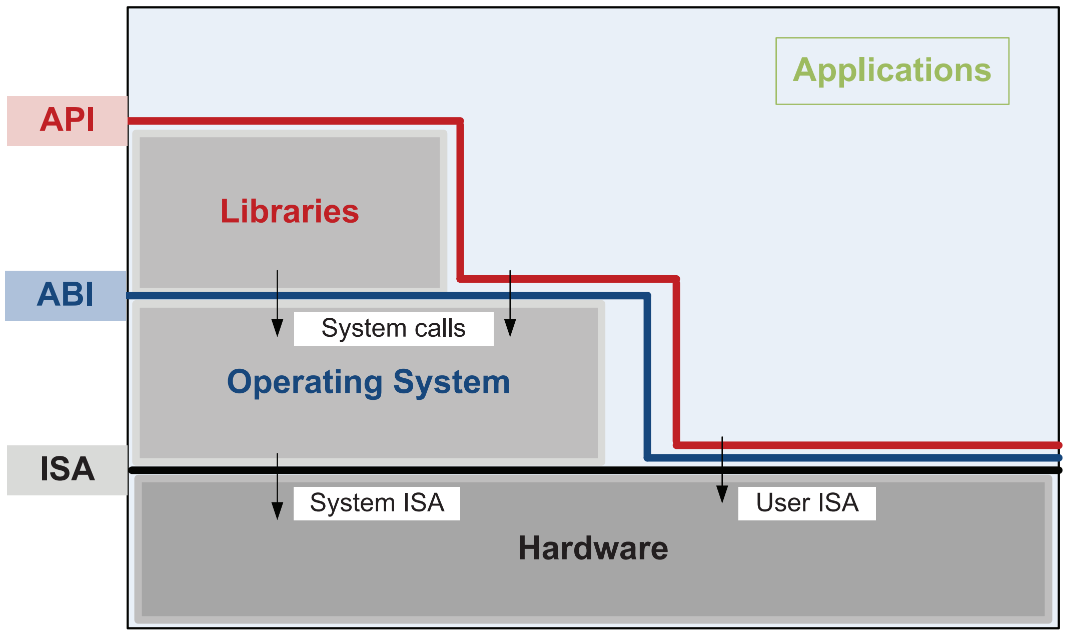


Figure 29: Layering and interfaces between layers in a computer system; the software components including applications, libraries, and operating system interact with the hardware via several interfaces: the Application Programming Interface (API), the Application Binary Interface (ABI), and the Instruction Set Architecture (ISA). The hardware consists of one or more multi-core 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.

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 disjoint sets of instructions, privileged instructions that can only be executed in kernel mode and the non-privileged instructions.

Computer systems are fairly complex and their operation is best understood when we consider a model similar with the one in Figure 29 which shows the interfaces between the software components and the hardware [223]. Applications written mostly in High Level languages (HLL) often call library modules and are compiled into *object code*; the privileged operations such as I/O requests cannot be executed in user mode; instead, application and library modules issue *system calls* and the operating systems 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 we discuss is the *Instruction Set Architecture (ISA)* at the boundary of the hardware and the software; the API defines the set of instructions the hardware was designed to execute. 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; the ABI does not include privileged system instructions, instead it invokes

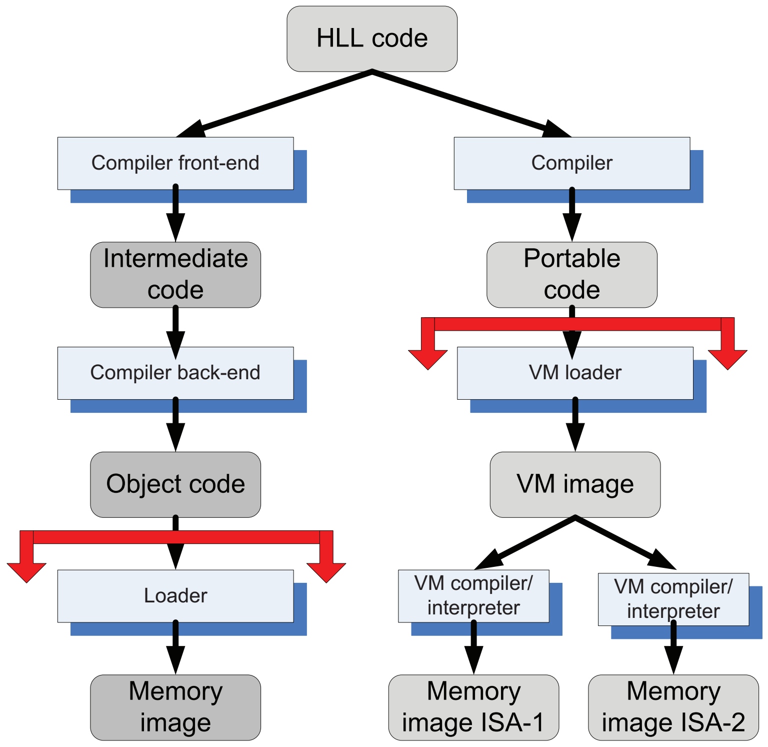


Figure 30: High Level Language (HLL) code can be translated for a specific architecture and operating system; the HLL code can also be compiled into portable code and then the portable code be translated for systems with different ISAs. The code shared is the object code in the first case and the portable code in the second case.

system calls. Finally, the *Application Program Interface (API)* gives the application access to the ISA; it includes HLL library calls which often invoke system calls. A *process* is the abstraction for the code of an application at execution time; a *thread* is a light-weight process. *The ABI is the projection of the computer system seen by the process and the API is the projection of the system from the perspective of the HLL program.*

Clearly, the binaries created by a compiler for a specific ISA and operating systems are not portable, cannot run on a computer with a different ISA or on the computer with the same ISA, but a different operating system. It is possible though to compile a HLL program for a virtual machine (VM) environment, as seen in Figure 30, where 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 instruction and leads to a significant performance improvement as such blocks are cached and reused.

6.2 Virtual machines

A *Virtual Machine (VM)* is an isolated environment that appears to be a whole computer, but actually only has access to a portion of the computer resources. 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 machines have been around since early 1970s when IBM release its `VM 370` operating system. Modern operating systems such as `Linux Vserver` [145], `OpenVZ` (Open VirtualiZation) [189], `FreeBSD Jails` [89], and `Solaris Zones` [203] based on `Linux`, `Unix,` `FreeBSD,` and `Solaris`, respectively, implement *operating system-level virtualization technologies*. They allow a physical server to run multiple isolated operating system instances, known as containers, Virtual Private Servers (VPSs), or Virtual Environments (VEs), subject to several constrains. For example, `OpenVZ` requires both the host and the guest OS to be a `Linux` distributions. These systems claim performance advantages over the systems based on a Virtual Machine Monitor such as `Xen` or `VMware`; according to [189], 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.

We distinguish two types of VMs, process and system VMs, Figure 31(a). 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. A *system VM* supports an operating systems 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 virtual machine*; e.g., Java Virtual Machine (JVM).

A literature search reveals the existence of some 60 different virtual machines many created by the large software companies; Table 6 lists a subset of them.

A *system virtual machine* provides a complete system; each VM can run its own OS, which in turn can run multiple applications. A *Virtual Machine Monitor (VMM)* allows several virtual machines to share a system. Several approaches are possible:

- *Traditional - VM also called a "bare metal" VMM* - a thin software layer that runs directly on the host machine hardware; its main advantage is performance, Figure 31(b). Examples: `VMWare ESX, ESXi` Servers, `Xen`, `OS370`, and `Denali`.

- *Hybrid* - the VMM shares the hardware with existing OS, Figure 31(c). Example: `VMWare Workstation`.

- *Hosted* - the VM runs on top of an existing OS, Figure 31(d); the main advantage of this approach is that the VM is easier to build and install. Other advantage of this solution is that the VMM could use several components 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 VMM, 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. Example: User-mode Linux.

In the next section we discuss Virtual Machine Monitors which manage the resource sharing among the virtual machines running concurrently on a physical system.

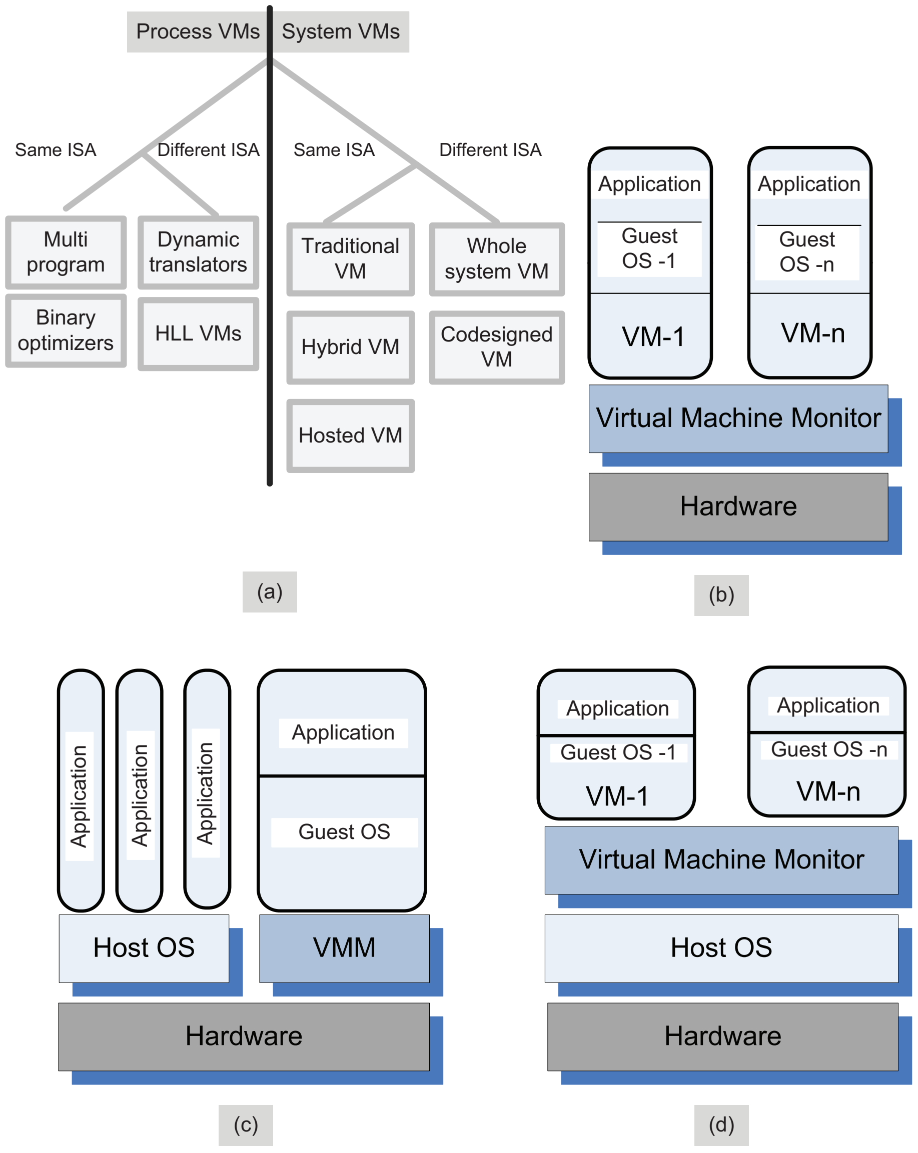


Figure 31: (a) A taxonomy of process and systems VMs for the same and for different Instruction Set Architectures (ISAs). Traditional, Hybrid, and Hosted are three classes of VMs for systems with the same ISA. (b) Traditional VMs; the VMM supports multiple virtual machines and runs directly on the hardware;. (c) Hybrid VM; the VMM shares the hardware with a host operating system and supports multiple virtual machines. (d) Hosted VM; the VMM runs under a host operating system.

6.3 Virtual machine monitors (VMMs)

A *Virtual Machine Monitor (VMM)* also called *hypervisor* is the software that securely partitions the resources of computer system into one or more virtual machines. A *guest operating system* is an operating system that runs under the control of a VMM rather than directly on the hardware. The VMM 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.

Table 6: A non-exhaustive inventory of system virtual machines. The host ISA refers to the instruction set of the hardware; the guest ISA refers to the instruction set supported by the virtual machine. The VM could run under a host OS, directly on the hardware, or under a VMM. The guest OS is the operating system running under the control of a VM which in turn may run under the control of the virtual machine 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 EXS 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

The VMMs allow several operating systems to run at the same time on a single hardware platform; at the same time, VMMs enforce isolation among these systems, hence security. A VMM controls how the guest operating systems uses the hardware resources; the events occurring in one VM do not affect any other VM running under the same VMM. At the same time, the VMM enables:

- multiple services to share the same platform;
- the movement of a server from one platform to another, the so-called live migration; and

- system modification while maintaining backward compatibility with the original system.

When a guest OS attempts to execute a privileged instruction the VMM traps the operation and enforces the correctness and safety of the operation; the VMM guarantees the isolation of the individual virtual machines and thus, ensures security and encapsulation, a major concern in cloud computing. At the same time, the VMM monitors the system performance and takes corrective actions to avoid performance degradation; for example, the VMM may swap out a Virtual Machine (copy all pages of that VM from real memory to disk and make the real memory frames available for paging by other VMs) to avoid thrashing.

A VMM virtualizes the CPU and the memory. For example, the VMM traps interrupts and dispatches then to the individual guest operating systems; if a guest OS disables interrupts the VMM buffers such interrupts until the guest OS enables them. The VMM maintains a *shadow page table* for each guest OS and replicates in its own shadow page table any modification made by the guest OS; 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 on the performance. VMMs use a range of optimization techniques; for example, VMWare systems avoid page duplication among different virtual machines, 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. VMMs 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, swap out pages of some of the processes running under that VM. Then it forces the balloon process to relinquish control of the free page frames.

6.4 Performance and security isolation

Performance isolation is a critical aspect for the Quality of Service grantees in shared environments. Indeed, 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 the completion time; moreover, it is equally difficult to optimize the application. Several operating systems including Linux/RK [186], QLinux [232], and SILK [40] support some performance isolation, but problems still exist as one has to account for all resources used and distribute the overhead for different system activities including context switching and paging to individual users, a problem often described as *QoS crosstalk* [237].

Processor virtualization presents multiple copies of the same processor or core on multicore systems; the code is executed directly by the hardware, while *processor emulation* presents a model of another hardware system; instructions are “emulated” in software much slower than virtualization. For example, Microsofts VirtualPC could run on chip sets other than the `x86` family; it was used on Mac hardware until Apple adopted Intel chips.

Traditional operating systems multiplex multiple processes or threads while a virtualization supported by a VMM (Virtual Machine Monitor) multiplexes full operating systems.

Obviously, there is a performance penalty as an OS is considerably more heavyweight than a process and the overhead of context switching is larger. A Virtual Machine Monitor executes directly on the hardware a subset of frequently used machine instruction generated by the application and emulates privileged instructions including device I/O requests. The subset of the 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. On the other hand, the software running on a virtual machine has the constrains 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. Thus, the virtualization can be used to improve security in a cloud computing environment.

A VMM is a much simpler and better specified system than a traditional operating system; for example, the `Xen` VMM discussed in Section 6.6 has approximately $60,000$ lines of code while the Denali VMM [251] has only about half, $30,000$ lines of code. The security vulnerability of VMMs is considerably reduced as the systems expose a much smaller number of privileged functions; for example, the `Xen` VMM can be accessed through 28 hypercalls while a standard Linux allows hundreds, e.g., Linux 2.6.11 allows 289 system calls. In addition to a plethora of system calls a traditional operating system supports special devices (e.g., `/dev/kmem`) and many privileged programs from a third party (e.g., `sendmail` and `sshd`).

6.5 Architectural support for virtualization; full and paravirtualization

In 1974 Gerald J. Popek and Robert P. Goldberg gave a set of sufficient conditions for a computer architecture to support virtualization and allow a Virtual Machine Monitor to operate efficiently [201]:

- A program running under the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.
- The VMM should be in complete control of the virtualized resources.
- A statistically significant fraction of machine instructions must be executed without the intervention of the VMM.

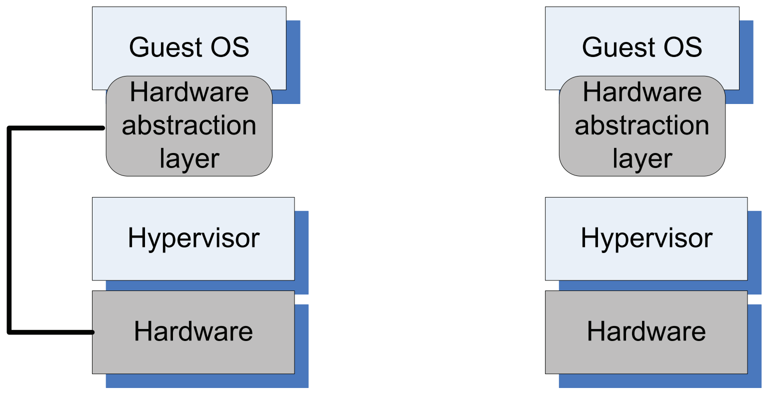
Another criteria allows us to identify an architecture suitable for a virtual machine; according to this classification we distinguish two classes of machine instructions :

- sensitive instructions of two types: *control sensitive*, instructions that attempt to change the memory allocation or the privileged mode and *mode sensitive*, instructions whose behavior is different in the privileged mode;
- innocuous instructions, that are not sensitive.

An equivalent formulation of the conditions for efficient virtualization can be based on this classification of machine instructions: *a VMM for a third or later generation computer can be constructed 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 VMM monitors the execution of guest operating systems; non-virtualizable instructions executed by a guest operating system are replaced with other instructions.
- Paravirtualization. The VMM is modified to use only instructions that can be virtualized.

There are two basic approaches to processor virtualization: *full virtualization* when each virtual machine runs on an exact copy of the actual hardware; and *paravirtualization* when each virtual machine runs on a slightly modified copy of the actual hardware. The reasons why paravirtualization is often adopted are: (i) some aspects of the hardware cannot be virtualized; (ii) to improve performance; (iii) to present a simpler interface. `VMWare` VMMs are example of full virtualization, while `Xen` [37] and `Denali` [251] are based on paravirtualization.



(a) Full virtualization (b) Paravirtualization

Figure 32: (a) The full virtualization requires the hardware abstraction layer of the guest OS to have some knowledge about the hardware. (b) The paravirtualization avoids this requirement and allows full compatibility at the Application Binary Interface (ABI).

Full virtualization requires a virtualizable architecture; the hardware is fully exposed to the guest OS which runs unchanged and this ensure that this direct execution mode is efficient. On the other hand, paravirtualization is done because some architectures such as x86 are not virtualizable. Paravirtualization demands that the guest OS be modified to run under the VMM; also the guest OS code must be ported for individual hardware platforms.

Application performance under a virtual machine are critical; generally, virtualization adds some level of overhead which affects negatively the performance. In some cases an application running under a virtual machine performs better than one running under a classical OS; this is the case of a policy called Cache Isolation. The cache is generally not partitioned equally among processes running under a classical OS, as one process may use the cache space better than the other. For example, in the case of two processes, one write-intensive and the other one read-intensive, the cache may be aggressively filled by the first. Under the Cache Isolation policy the cache is divided between the VMs and it is beneficial to run workloads competing for cache in two different VMs [222]. The application I/O performance running under a virtual machine 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. On a `Xen` platform discrepancies between the optimal choice and the default are as high as 8% to 35% [222].

6.6 Case study: `Xen`-a VMM based on paravirtualization

`Xen` is a Virtual Machine Monitor (VMM) developed by the Computing Laboratory at the University of Cambridge, United Kingdom, in the early 2000. Since 2010 `Xen` is 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` guests operating systems running on IA-32, x86-64, Itanium, and ARM architectures.

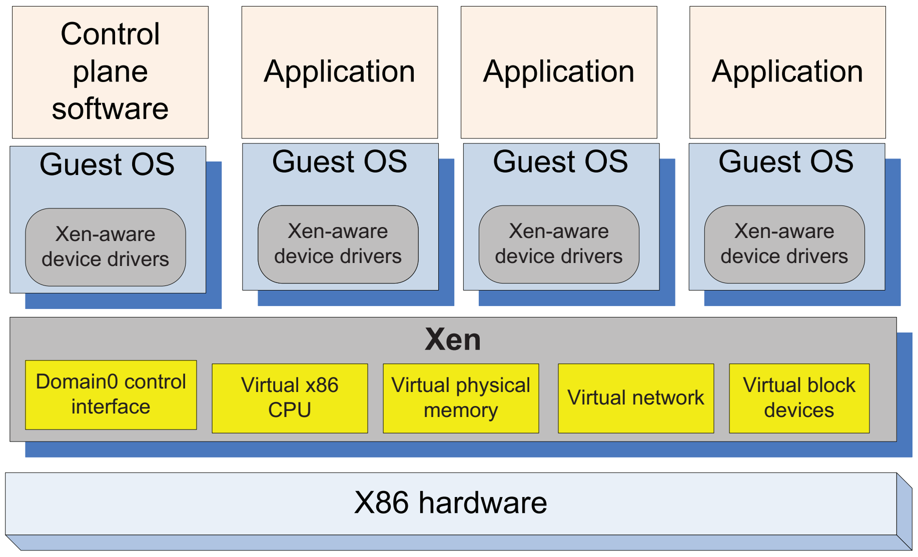


Figure 33: `Xen` for the x86 architecture; in the original `Xen` implementation [37] a guest OS could be either XenoLinix, XenoBSD, or XenoXP.

The goal of the Cambridge group led by Ian Pratt was to design a VMM capable of scaling to about 100 virtual machines running standard applications and services without any modifications to the Application Binary Interface (ABI). Fully aware that the x86 architecture does not support efficiently full virtualization, the designers of `Xen` opted for paravirtualization.

Systems such as `VMware EX Server` support full virtualization on x86 architecture. The virtualization of the MMU (Memory Management Unit) and the fact that privileged instructions executed by a guest OS fail silently pose some challenges; for example, to address the later problem one has to insert traps 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; the overhead of many operations is substantial.

Next we analyze the original implementation of `Xen` for the x86 architecture discussed in [37]. The creators of `Xen` used the concept of *domain* to refer to the ensemble an address spaces hosting a guest OS and address spaces for applications running under this guest OS; each domain runs on a virtual x86 CPU. *Domain0* is dedicated to the execution of `Xen` control functions and privileged instructions, as shown in Figure 33.

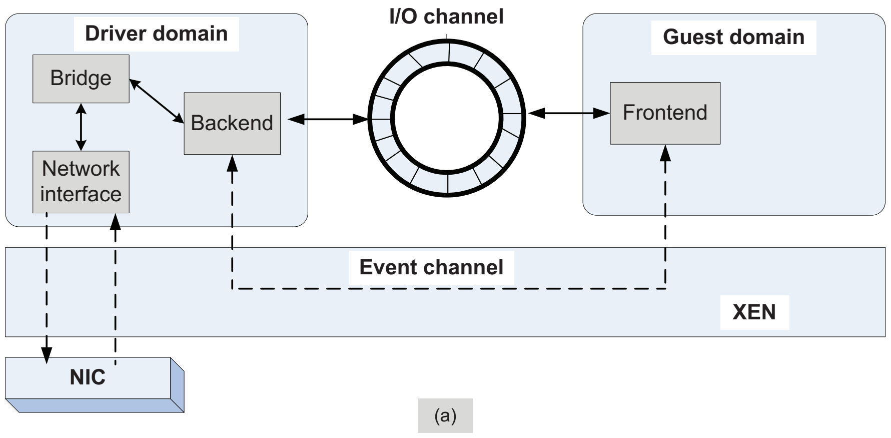
Table 7: Paravirtualization strategies for virtual memory management, CPU multiplexing, and I/O devices for the original x86 `Xen` implementation.

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; exception handlers other than the page fault handler are identical with x86 native exception handlers.
System calls	To increase efficiency, a guest OS must install a "fast" handler to allow system calls from an application to the guest OS and avoid indirection through `Xen`.
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.
Time	Each guest OS has a timer interface and is aware of "real" and "virtual" time.
Network and I/O devices	Data is transferred using asynchronous I/O rings; a ring is a circular queue of descriptors allocated by a domain and accessible within `Xen`.
Disk access	Only *Domain0* has direct access to IDE and SCSI disks; all other domains access persistent storage through the Virtual Block Device (VBD) abstraction.

The most important aspects of the `Xen` paravirtualization for virtual memory management, CPU multiplexing, and I/O device management are summarized in Table 7, [37]. 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. Unfortunately, the x86 architecture does not support either the tagging of TLB entries or the software management of the TLB; as a result, address space switching, when the VMM activates a different OS requires a complete TLB flush; this has a negative impact on the 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 VMM. Thus, a guest OS could only map pages it owns; on the other hand, it 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 VMM runs at Level 0, the guest OS at Level 1, and applications at Level 3.



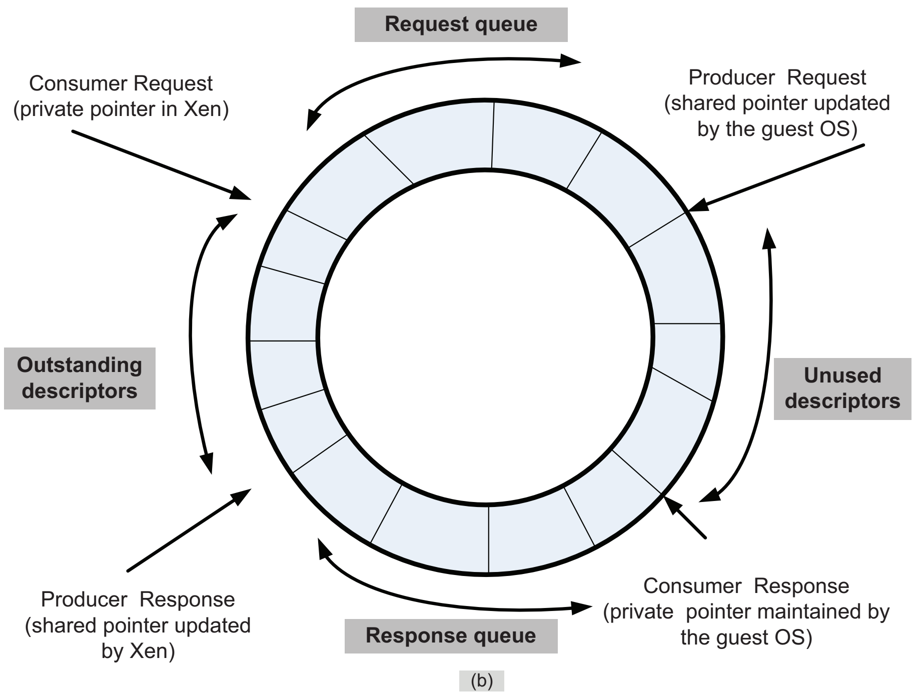

Request queue
Consumer Request (private pointer in Xen)
Producer Request (shared pointer updated by the guest OS)
Outstanding descriptors
Unused descriptors
Producer Response (shared pointer updated by Xen)
Consumer Response (private pointer maintained by the guest OS)
Response queue
(b)

Figure 34: 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. (b) the circular ring of buffers.

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.

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 exception 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.

`Xen` schedules individual domains using the Borrowed Virtual Time (BVT)[43] [75], a work conserving[44] and a low-latency wake-up scheduling algorithm. BVT uses a virtual-time warping mechanism to support low-latency dispatch to ensure timely execution when this is needed, for example, for timely delivery of TCP acknowledgments.

Data for I/O and network operations moves vertically through the system very efficiently using a set of I/O rings, see Figure 34; 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.

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` defines abstractions for networking and I/O devices; each domain has one or more Virtual Network Interfaces (VIFs) which support the functionality of a network interface card; a VIF 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 *Domain0*; 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. *Domain0* is the only one allowed to access directly the physical IDE or SCSI disks. All domains other than *Domain0* access persistent storage through a Virtual Block Device (VBD) abstraction created and managed under the control of *Domain0*.

An analysis of virtual machines performance for I/O-bound applications under `Xen` is reported in [204]. Two Apache Web servers, each under a different VM, share the same server running `Xen`; the workload generator sends requests for files of fixed size ranging from

[43]The BVT scheduling algorithm was introduced by Duda and Cheriton in 1999; it aims to reduce the wake-up latency and allows temporary violations of fair sharing to favor recently-woken threads/processes/domains.

[44]A work conserving scheduling algorithm does not allow the processor to be idle when there is work to be done.

1 KB to 100 KB. When the file size increases from 1 KB to 10 KB, and to 100 KB the CPU utilization, the accepted load, the throughput, data rate, and the response time are, respectively: $(97.5\%, 70.44\%, 44.4\%)$, $(1,900, 1,104, 112)$ requests/sec, $(2,18, 11,048, 11,208)$ KBps, and $1.52, 2.36, 2.08)$ 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 variation of the response time is quite small, it increases 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 [251]. `Denali` was designed to support one or more orders of magnitude virtual machines running network services than `Xen`; it did not target existing ABI and 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.` For Linux it is reported that a total of roughly 3,000 lines of codes or 1.36% had to be modified; for XP this figure is $4,620$ or about 0.04%, [37].

6.7 Optimization of network virtualization in `Xen` 2.0

A virtual machine monitor 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 $5-6$ times higher than that of the native system (Linux 2.2.17) in saturating a 100 Mbps network [230]. In other words, to handle the same amount of traffic as the native system to saturate the network, the VMM executes a much larger number of instructions, $5-6$ times larger.

Similar overheads are reported for other VMMs and in particular for `Xen` 2.0 [165, 166]. To understand the sources of the network overhead we examine the basic network architecture of `Xen`, see Figure 35 (a). Recall that privileged operations including I/O are executed by *Domain0* on behalf of a guest operating system; in this context we shall refer to it as the *driver domain* called in to execute networking operations on behalf of the *guest domain.* The *driver domain* uses the native Linux driver for the NIC (Network Interface Controller) which in turn communicates with the physical NIC, also called the network adapter. Recall from Section 6.6 that the *guest domain* communicates with the *driver domain* through an I/O channel; more precisely, the guest OS in the guest domain uses a virtual interface to send/receive data to/from the backend interface in the driver domain.

Recall that a *bridge* in a LAN uses broadcast to identify the MAC address of a destination system; once this address is identified, then it is added to a table. When the next packet for the same destination arrives, the bridge uses the link layer protocol to send the packet to the proper MAC address, rather than broadcast it. The bridge in the driver domain performs a multiplexing/demultiplexing function; packets received from the NIC have to be demultiplexed and sent to different VMs running under the VMM. 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 NAT (Network Address Translation).

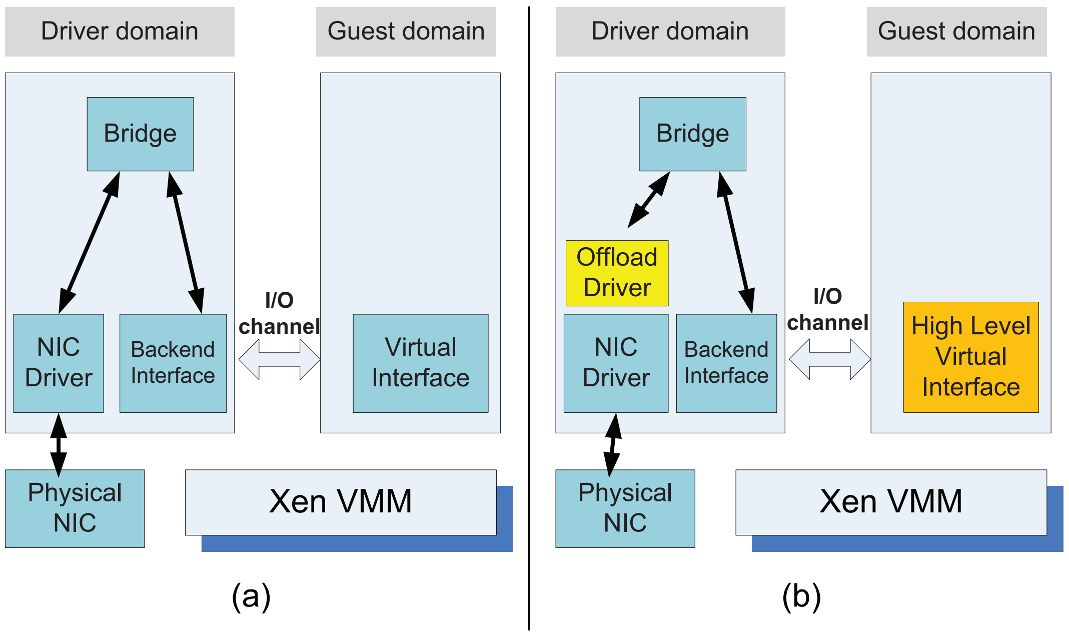


Figure 35: Xen network architecture (a) The original architecture; (b) The optimized architecture.

Table 8: 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

Table 8 from [166] shows the ultimate effect of this longer processing chain for the Xen VMM as well as the effect of optimizations; 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 multiplexing/demuultiplexing accounts for about 40% for incoming traffic and 30% for outgoing traffic of the communication overhead.

The Xen network optimization discussed in [166] 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.

Next we examine briefly each optimization area and start with the virtual interface. There is a tradeoff between generality and flexibility, on one hand, and the performance on the other hand. The original virtual network interface provides the guest domain with the abstraction of a simple low-level network interface supporting sending and receiving primitives. This 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[45], and scatter/gather DMA support[46]. These features are supported by the High Level Virtual Interface of the optimized system, Figure 35 (b).

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 in a new page and then the physical page containing the packet is re-mapped into 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 4 times increase of the send data rate compared with the non optimized version.

The third optimization covers the 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 1024 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. Obviously, 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 OS and this forces the system to use traditional page mapping, rather than the superpage mapping. The optimized version uses a special memory allocator to avoid this problem.

6.8 `vBlades` - paravirtualization targeting a x86-64 Itanium processor

To understand the impact of the computer architecture on the ability to virtualize efficiently a given architecture we discuss some of the findings of the `vBlades` project at HP-Laboratories [158]. The goal of the `vBlades` project was to create a VMM for the Itanium family of x86-64 Intel processors[47] capable of supporting the execution of multiple operating systems in isolated protection domains with security and privacy enforced by the hardware. Itanium was selected because of its multiple functional units and multi-threading support. The VMM was also expected to support optimal server utilization and allow comprehensive measurement and monitoring for detailed performance analysis.

[45]TSO stands for TCP segmentation offload; this option enables the network adapter to compute the TCP checksum on transmit and receive, and save the host CPU the overhead for computing the checksum; large packets have larger savings.

[46]Direct Memory Access is specialized hardware which allows 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, called I/O Acceleration Technology (I/OAT)

[47]Itanium is a processor developed jointly by HP and Intel based on a new architecture, explicitly parallel instruction computing (EPIC), that allows the processor to execute multiple instructions in each clock cycle. EPIC implements a form of Very Long Instruction Word (VLIW) architecture, in which a single instruction word contains multiple instructions; for more information see http://www.dig64.org/about/Itanium2_white_paper_public.pdf.

The discussion in Section 6.2 shows that to be fully virtualizable the ISA of a processor must conform to a set of requirements but, unfortunately, the x86-64 architecture does not meet these requirements and this made the `vBlades` 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 PL0, while applications run at level PL3 and can only execute non-privileged instructions; PL2 and PL4 rings are generally not used. The VMM 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); thus, a guest OS may not accept to boot or run or may itself attempt to make use of all four privilege rings.

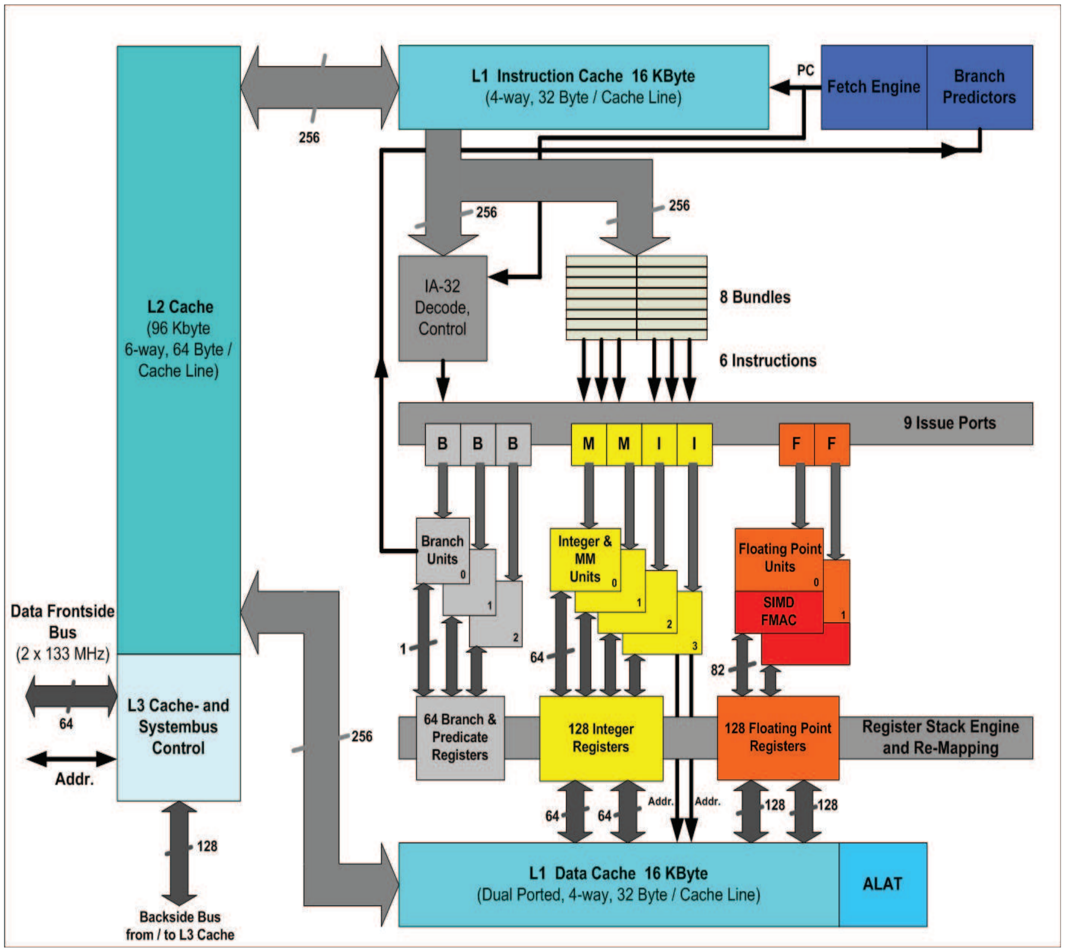


Figure 36: The Itanium processor has 30 functional units: six general-purpose ALUs, two integer units, one shift unit, four data cache units, six multimedia units, two parallel shift units, one parallel multiply, one population count, three branch units, two 82-bit floating-point multiplyaccumulate units, and two SIMD floating-point multiplyaccumulate 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. The processor has thirty functional execution units, see Figure 36. Each unit can execute a particular subset of the instruction set.

The hardware supports 64-bit addressing; it has 32 64-bit general-purpose registers numbered from R0 to R31 and 96 automatically renumbered registers R32-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 (PAL)* firmware allows the caller to set the values in the region register. The VMM intercepts the privileged instruction issued by the guest OS to its PAL 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 different interrupt handlers pointed 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.

First, we discuss *CPU virtualization.* When a guest OS attempts to execute a privileged instruction the VMM traps and emulates the instruction. For example, when the guest OS uses the `rsm psr.i` instruction to turn off delivery of a ceratin type of interrupts the VMM does not disable the interrupt, but records the fact that interrupts of that type should be delivered to the guest OS, and in this case the interrupt should be masked. There is a slight complication related to the fact that the Itanium does not have an Instruction Register (IR) and the VMM has to use state information to determine if an instruction is privileged. Another complication is caused by the *register stack engine (RSE)* which operates concurrently with the processor and may attempt to access memory (load or store) and generate a page fault. The problem is solved normally by setting up a bit indicating that the fault is due to RSE and, at the same time, the RSE operations are disabled. The handling of this problem by the VMM is more intricate.

There are a number of *privileged-sensitive* instructions which behave differently function of the privilege level; they are handled by dynamic transformations of the instruction stream and the VMM replaces each one of them with a privileged instruction. Among the instructions in this category are:

- `cover`, saves stack information into a privileged register; the VMM 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 VMM 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 --> tpa Rx=R(y+64)` and `ttag Rx=Ry --> tak Rx=R(y+64)` $0 \leq y \leq 64$.

- Access to performance data from performance data registers is controlled by a bit in the *Processor Status Register* with the `PSR.sp` instruction.

Memory virtualization is guided by the realization that a VMM should not be involved in most of memory read and write operations to prevent a significant degradation of the performance, but, at the same time, the VMM should exercise a tight control and prevent a guest OS to act maliciously. The `vBlades` VMM 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 VMM first checks if the guest OS is allowed to access that address and if so it provides the regular address translation. If the address is physical then the VMM is not involved. The hardware distinguishes between virtual and real addresses using bits in the Processor Status Register.

6.9 A performance comparison of virtual machines

Cloud computing benefits from the virtual machine architecture; user security and convenience are probably the most relevant advantages of VMs over classical operating systems. It is also evident that a VM architecture has a negative impact on system performance because it introduces additional system overhead. The topic of this section is a quantitative analysis of the impact of the VM architecture on performance.

Several papers including [37, 165, 166] analyze the performance of `Xen`. A recent paper [195] compares the performance of two virtualization techniques with a standard operating system, a plain vanilla Linux referred to as "the base" system; the two VM systems are `Xen`, based on paravirtualization, and `OpenVZ`.

`OpenVZ` is based on OS-level virtualization; it uses a single patched Linux kernel and the guest operating systems in different containers[48] may be different distributions but must use with the same Linux kernel version that the host uses. The lack of flexibility of the approach for virtualization in `OpenVZ` is compensated by a lower overhead.

The memory allocation in `OpenVZ` is more flexible than in case of paravirtualization; memory not used in one virtual environment can be used by others. The system uses a common file system; each virtual environment is a directory of files isolated using `chroot`. To start a new virtual machine one needs to copying the files from one directory to another, to create a `config` file for the virtual machine, and launch the VM.

`OpenVZ` has a two level scheduler: at the first level, the fair-share scheduler allocates CPU time slices to containers based on `cpuunits` values; the second level is a standard Linux scheduler which decides which 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 the priorities.

The discussion in [195] is focused on user's perspective thus, the performance measures analyzed are the throughput and the response time. The general question is whether consolidation of the applications and servers is a good strategy for cloud computing; the specific questions examined are:

[48]A container in `OpenVZ` 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.

- How the performance scales up with the load?
- What is the impact of a mix of applications?
- What are the implications of the load assignment on individual servers?

There is ample 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 in time clearly illustrates this fact. As we all know, this phenomenon justifies the need for CPU multiplexing among threads/processes supported by an operating system. 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; indeed, the peak resource requirements of individual applications are very unlikely to be synchronized and the aggregate load tends to lead to a better average resource utilization.

The application used in [195] is a two-tier system consisting of an Apache Web server and a MySQL database server. A client of this application starts a session as the 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 counters that allow the estimation of: (i) the CPU time used by a binary; (ii) the number of L2-cache missing; and (iii) the number of instructions executed by a binary.

The experimental setup for three different experiments are shown in Figure 37. In the first group of experiments the two tiers of the application, the Web and the DB, run on a single server for the base system, 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 it. For a given workload the Web-tier CPU consumption for the `OpenVZ` system is close to the base system and it is about half of that for the `Xen` system. The performance analysis tool shows that the `OpenVZ` execution has twice as many L2-cache misses than the base system, while the `Xen` *Domain0* has 2.5 times more and the `Xen` application domain has 9 times more. 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. For the `Xen`-based system the procedure *hypervisor_callback*, invoked when an event occurs, and the procedure *evtchn_do_upcall*, invoked to process an event, are responsible for 32% and 44%, respectively, of the L2-cache misses. 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: (i) a procedure called *do_anonimous_pages* used to allocate pages for a particular application with the percentage of cache misses 32% and 25%, respectively; (ii) the procedures called *_copy_to_user_ll* and *_copy_from_user_ll* used to copy data from user to system buffers and back with the percentage of cache misses (12+7)% and (10+1)%, respectively; the first figure refers to the copying from user to system buffers and the second to copying from system buffers to the user space.

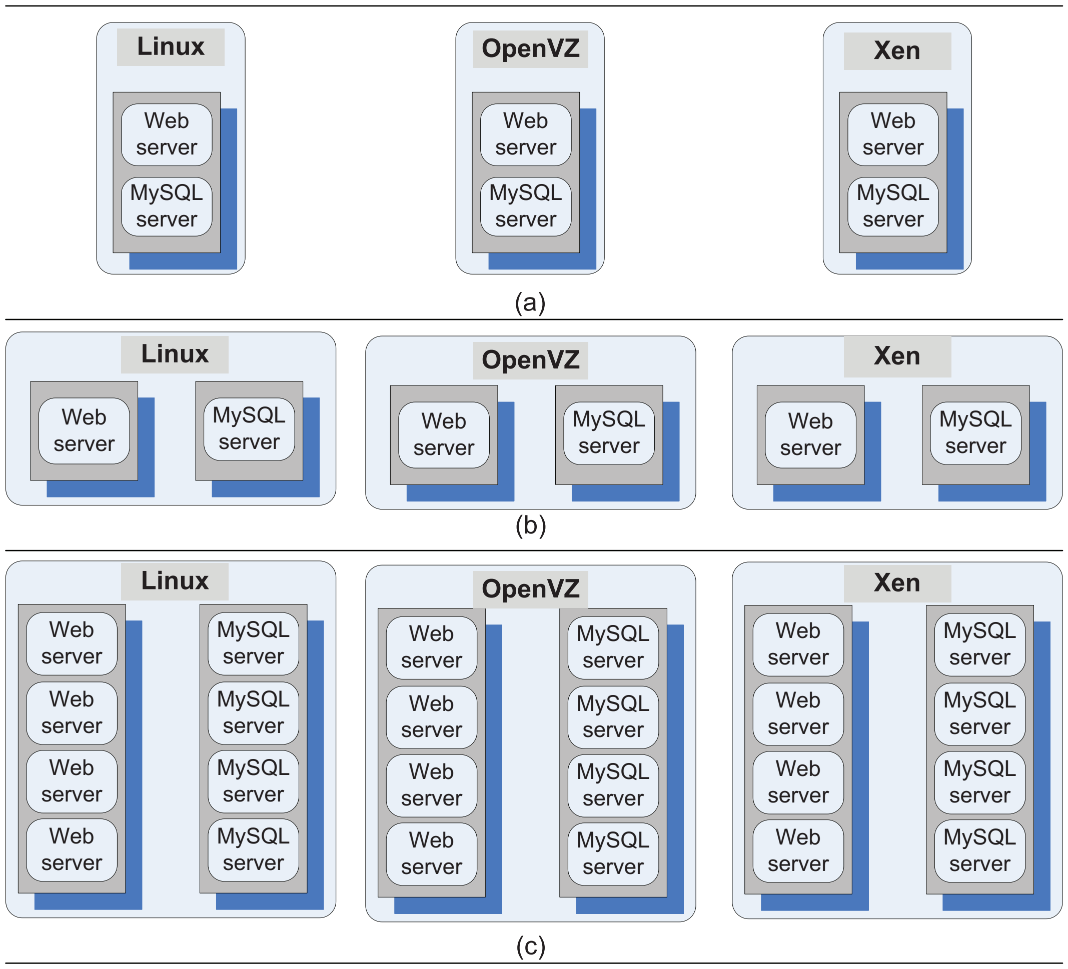


Figure 37: The setup for the performance comparison of a native Linux system with `OpenVZ`, and the `Xen` systems. The applications are a Web server and a MySQL database server. (a) The first experiment, the Web and the DB, share a single system; (b) The second experiment, the Web and the DB, run on two different systems; (c) The third experiment, the Web and the DB, run on two different systems and each has four instances.

The second group of experiments use two servers, one for the Web and the other for the DB application, for each one of the three systems. When the load increases from 500 to 800 threads the throughput increases linearly with the workload. The response time of the `Xen` system increases only 114%, compared with 600% reported for the first experiments. The CPU time of the base system, the `OpenVZ` system, the `Xen` *Domain0* and the *User Domain* are similar for the Web application; for the DB application the CPU time of the `OpenVZ` system is twice as large as that of the base system, while *Domain0* and the *User Domain* require CPU times of 1.1 and 2.5 times larger than the base system. The L2-cache misses for the Web application relative to the base system are: the same for `OpenVZ` and 1.5 larger for *Domain0* of `Xen` and 3.5 times larger for the *User Domain*; the L2-cache misses for the DB application relative to the base system are: 2 times larger for the `OpenVZ` and 3.5 larger for *Domain0* of `Xen` and 7 times larger for the *User Domain*.

The third group of experiments uses two servers, one for the Web and the other for the

DB application, for each one of the three systems but run 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. The performance degradation when the workload increases is also noticeable for `Xen`. Another important conclusion is that hosting multiple tiers of the same application on the same server is not an optimal solution.

6.10 Virtual machine security

The hybrid and the hosted VM models in Figures 31 (c) and (d), respectively, expose the entire system to the vulnerability of the host operating system thus, we will not analyze these models. Our discussion of virtual machine security is restricted to the traditional system VM model in Figure 31 (b) when the Virtual Machine Monitor controls the access to the hardware.

Virtual security services are typically provided by the VMM as shown in Figure 38 (a); another alternative is to have a dedicated security services VM as in Figure 38 (b). A secure TCB (Trusted Computing Base) is a necessary condition for security in a virtual machine environment; if the TCB is compromised then the security of the entire system is affected.

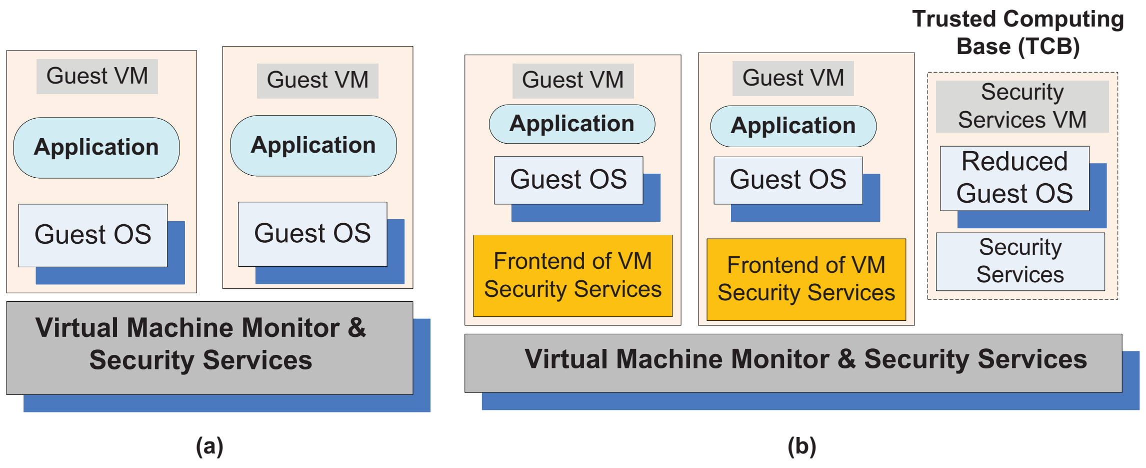

Guest VM
Application
Guest OS
Guest VM
Application
Guest OS
Virtual Machine Monitor & Security Services
(a)
Guest VM
Application
Guest OS
Frontend of VM Security Services
Guest VM
Application
Guest OS
Frontend of VM Security Services
Trusted Computing Base (TCB)
Security Services VM
Reduced Guest OS
Security Services
Virtual Machine Monitor & Security Services
(b)

Figure 38: (a) Virtual security services provided by the VMM; (b) A dedicated security VM.

Virtual machine organization is better suited for supporting security than traditional techniques used by modern operating systems; this adds to the appeal of virtualization for cloud computing. The analysis of `Xen` and `vBlades` shows that the VM technology provides a stricter isolation of virtual machines from one another than the isolation of processes in a traditional operating system. Indeed, a Virtual Machine Monitor controls the execution of privileged operations and can thus enforce memory isolation as well as disk and network access. The VMMs 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 VMM sees only raw data regarding the state of a guest operating system 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 VMM has access to the state of all virtual machines operating on the same hardware. This state of a guest virtual machine can be saved, restored, cloned, and encrypted by the VMM. 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 VM's files moved to a dedicated VM and thus, protect it from attacks [265]; this is possible because inter-VM communication is faster than communication between two physical machines.

Sophisticated attackers are able to fingerprint virtual machines and avoid virtual machine honey pots designed to study the methods of attach. 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.

There is no free lunch thus, we expect to pay some price for the better security provided by virtualization. This price includes: higher hardware costs because a virtual system requires more resources such as CPU cycles, memory, disk, network bandwidth; the cost of developing VMMs and modifying the host operating systems in case of paravirtualization; and the overhead of virtualization as the VMM is involved in privileged operations.

A recent paper [265] surveys VM-based intrusion detection systems such as `Livewire` and `Siren` which exploit the three capabilities of a virtual machine for intrusion detection, isolation, inspections, and interposition. We have examined isolation; inspection means that the VMM has the ability to review the state of the guest VMs and interposition means that VMM can trap and emulate the privileged instruction issued by the guest VMs. The paper also discusses VM-based intrusion prevention systems such as, `SVFS, NetTop`, and `IntroVirt`, and surveys Terra, a VM-based trust computing platform. `Terra` uses a *trusted virtual machine monitor* to partition resources among virtual machines.

The security group involved with the NIST project has identified the following VMM- and VM-based threats:

- VMM-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 VMM.
    2. VM side-channel attacks: malicious attack on one or more VMs by a rogue VM under the same VMM. Probable causes: (a) lack of proper isolation of inter-VM traffic due to misconfiguration of the virtual network residing in the VMM; (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 O/S without latest patches.
    3. Buffer overflow attacks.
- 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 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.

6.11 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. Insecure 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[218] discusses the application of the sandboxing technology for two modern CPU architectures, ARM and 64-bit x86. 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) and assumes an execution model where a trusted runtime shares a process with the untrusted multi-threaded plugin. The rules for binary code generation of the untrusted plugin are: (i) the code section is read-only and it is statically linked; (ii) the code is divided into 32 byte *bundles* and no instruction or pseudo-instruction crosses the bundle boundary; (iii) the disassembly starting at the bundle boundary reaches all valid instructions; (iv) 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 9 from [218]. The control flow and store sandboxing for the ARM SFI incurs less then 5% average overhead and the one for x86-64 SFI incurs less than 7% average overhead.

6.12 History notes

Virtual memory is the first application of virtualization concepts to commercial computers; it 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

Table 9: 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

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 operating system 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 VMM 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 virtual machine contexts.

In this early age, the virtualization was driven by the need to share a very expensive hardware among a large population of users and applications. The VM/370 system, released in 1972 for large IBM mainframes was very successful; it was based on a re-implementation of CP/CMS. In VM/370 a new virtual machine was created for every user and this virtual machine interacted with the applications; the VMM 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 advancements 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 [211]: “VMMs give operating system developers another opportunity to develop functionality no longer practical in today’s complex and ossified operating systems, where innovation moves at a geologic pace.”

6.13 Further readings

A good introduction to virtualization principles can be found in a recent text of Saltzer and Kaashoek [213]. Virtual machines are dissected in a paper by Smith and Nair [223]. An insightful discussion of virtual machine monitors is provided by the paper of Rosenblum and Garfinkel [211]. Several papers [37, 165, 166] discuss in depth the `Xen` VMM and analyze its performance. The `Denali` system is presented in [251]. Modern operating systems such as `Linux Vserver` [145], `OpenVZ` (Open VirtualiZation) [189], `FreeBSD Jails` [89], and `Solaris Zones` [203] implement *operating system-level virtualization technologies.*

A recent paper [195] compares the performance of two virtualization techniques with a standard operating system. The `vBlades` project at HP-Laboratories is presented in [158]. A survey of security issues in virtual systems is provided by [265].

7 Resource Management

Resources management is a core function of any man-made system thus, an important element of cloud computing. 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 is extremely challenging because of the complexity of the system which makes it impossible to have accurate global state information and because of the unpredictable interactions with the environment.

It has been argued for some time that in a cloud where changes are frequent and unpredictable, centralized control is unlikely to provide continuous service and performance guarantees. Indeed, centralized control cannot provide adequate solutions to the host of cloud management policies that have to be enforced. Such policies can be loosely grouped into five classes:

1. Admission control.
2. Capacity allocation.
3. Load balancing.
4. Energy optimization.
5. Quality of service (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 one we have already discussed. Load balancing and energy optimization are correlated and affect the cost of providing the services. Indeed, it is predicted that by 2012 up to 40% of the budget of IT enterprise infrastructure will be spent on energy [72].

The common meaning of the term "load balancing" is of distributing evenly the load to a 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 load, 50% of each server's capacity. In cloud computing a critical goal is to minimize the cost of providing the service and, in particular, to minimize the energy consumption. 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 used very little energy. In our example, 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. Quality of service is probably the most difficult to address aspect of resource management 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 target jointly the performance and the power consumption. The Dynamic Voltage and Frequency Scaling (DVFS) techniques such as Intel's SpeedStep and AMD's PowerNow lower the voltage and the frequency to decrease the power consumption[49]. 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.

As a result of lower voltages and frequencies the performance of processors decreases, but at a substantially slower rate. Table 10 shows the dependence of the normalized performance and the normalized energy 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 than the peak performance, achieved at 2.2 GHz. This seems a reasonable energy-performance tradeoff!

Table 10: The normalized performance and energy, 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

A service level agreement often specifies the rewards as well as penalties associated with specific performance metrics. As we shall see in this chapter, 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 39 shows the case when the performance metrics is R, the response time. The largest reward can be obtained when $R \leq R_0$; a slightly lower reward corresponds to $R_0 < R \leq R_1$; when $R_1 < R \leq R_2$, 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 we shall see in Section 7.2.

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 very complex computations that cannot be done effectively in the time available to respond. The performance models are very

[49]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.

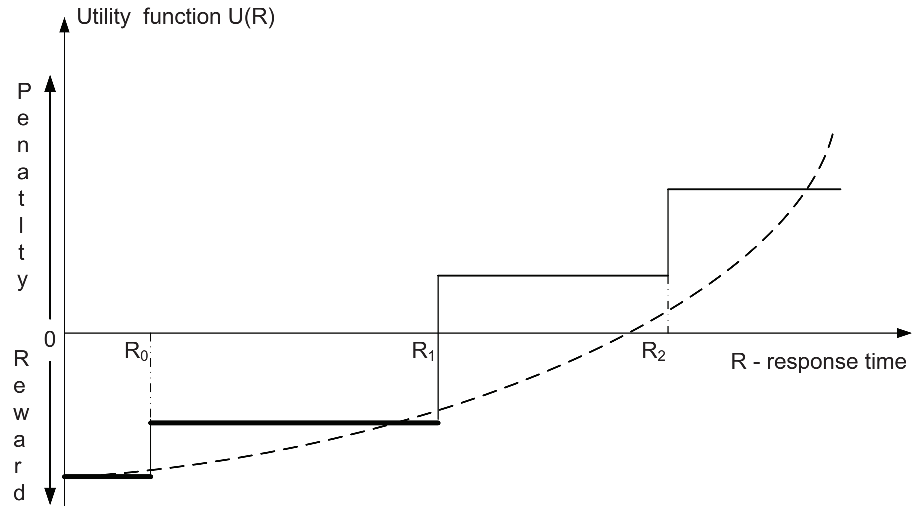


Figure 39: The utility function $U(R)$ is a series of step functions with jumps corresponding to the response time, $R = R_0|R_1|R_2$, when the reward and the penalty levels change according to the SLA. The dotted line shows a quadratic approximation of the utility function.

complex, analytical solutions are intractable, 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.

Allocation techniques in computer clouds must be based on a disciplined approach rather than ad hoc methods. The four basic mechanisms for the implementation of resource management policies are:

- *Control theory.* Control theory uses the feedback to guarantee system stability and predict transient behavior [127], [138], but can be used only to predict local rather than global behavior; Kalman filters have been used for unrealistically simplified models.

- *Machine learning.* A major advantage of machine learning techniques is that they do not need a performance model of the system [241]; this technique could be applied for coordination of several autonomic system managers as discussed in [129].

- *Utility-based.* Utility-based approaches require a performance model and a mechanism to correlate user-level performance with cost e.g., [9].

- *Market-oriented/economic mechanisms.* Such mechanisms do not require a model of the system; for example, the combinatorial auctions for bundles of resources discussed in [228].

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, and only a few, e.g., [233], to heterogeneous workloads, a combination of the two.

The strategies for resource management associated with IaaS, PaaS, and SaaS will be different. In all three types of services the providers are faced with large fluctuating loads. 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 spike loads provided that: (a) there is a pool of resources that can be released or allocated on demand and (b) there is a monitoring system which allows a control loop to decide in real time to reallocate resources. Auto-scaling is supported by PaaS services, such as Google App Engine. Auto-scaling for IaaS is complicated due to the lack of standards; OGF (Open Grid Forum) OCCI (Open Cloud Computing Interface) is involved in the definition of virtualization formats and APIs for IaaS.

Autonomic policies are of great interest due to the scale of the system, the large number of service requests, the large user population, and the unpredictability of the load; the ratio of the mean to the peak resource needs can be very large.

We start our discussion with Web services and a utility model for resource allocation in Section 7.1; we continue in Section 7.2 with a control theoretic approaches to resource allocation. Next we analyze scheduling with deadlines in Section 7.5 and then we present resource bundling and combinatorial auctions in Section 7.6.

7.1 A utility-based model for cloud-based Web services

In this section we discuss a utility-based approach for autonomic management; formulated as an optimization problem the solution discussed in [9] addresses multiple policies, including QoS, but cannot be easily extended to cloud computing. 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. The cloud model for this optimization is quite complex and requires a fair number of parameters.

We assume a cloud providing $\mid K \mid$ 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 $v_k = v_k^{max}(1 - r_k/r_k^{max})$, see Figure 40(a); call $m_k = -v_k^{max}/r_k^{max}$ the slope of the utility function.

The system is modelled 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 40(b). 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 frequency and a

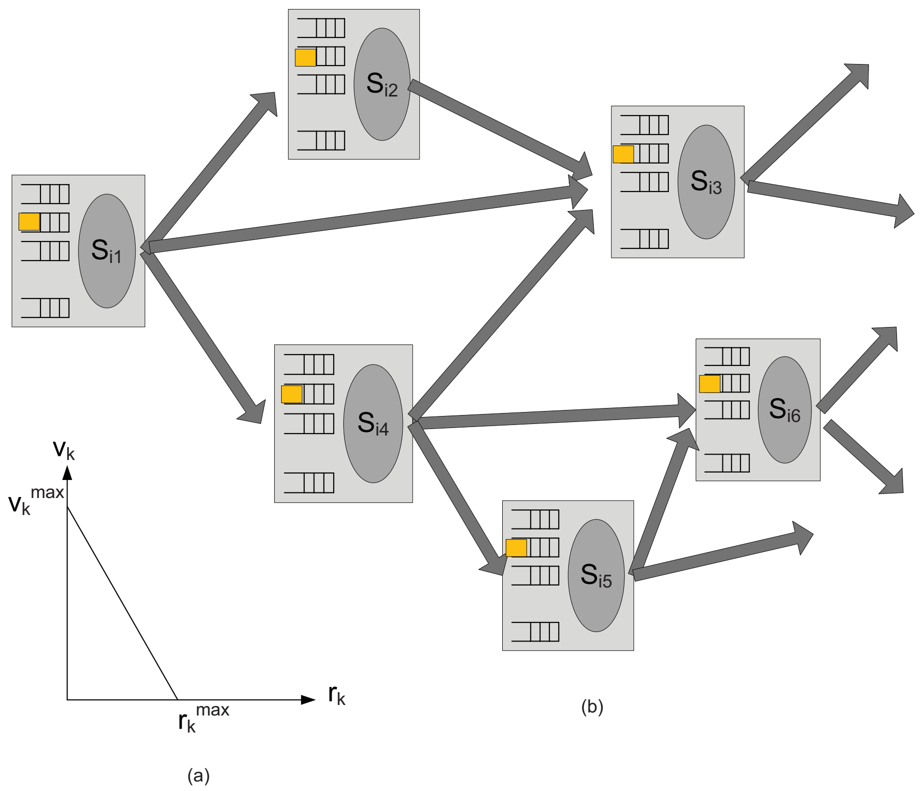


Figure 40: (a) The utility function, v_k the revenue (or the penalty) associated with a response time r_k for a request of class $k \in K$; the slope of the utility function is $m_k = -v_k^{max}/r_k^{max}$. (b) A network of multiqueues; at each server S_i there are $\mid K \mid$ queues for each one of the $k \in K$ classes of requests. A tier consists of all requests of class $k \in K$ at all servers $S_i \in I$.

discrete set of supply voltages according to a DVFS model[50]; the power consumption on a server is a function of the clock frequency. The scheduling of a server is work-conserving[51] and is modelled as a Generalized Processor Sharing (GPS) scheduling [262]. Analytical models [4], [194] are too complex for large systems.

The optimization problem formulated in [9] involves five terms: A and B reflect revenues, C the cost for servers in a low power, stand-by mode, D the cost of active servers given their operating frequency, E is the cost for switching servers from low-power, stand-by mode, to active state, and F the cost for migrating VMs from one sever 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 11 and the parameters used are shown in Table 12. The expression to be optimized is: $(A+B)-(C+D+E+F)$ with

[50]The Dynamic Voltage and Frequency Scaling (DVFS) is a method to provide variable amount of energy for a task by scaling the operating voltage/frequency.

[51]A scheduling is work-conserving if the server cannot be idle while there is work to be done.

$$A = \max \sum_{k \in K} \left(-m_k \sum_{i \in I, j \in N_k} \frac{\lambda_{i,k,j}}{\sum_{h \in H_i} (C_{i,h} \times y_{i,h}) \, \mu_{k,j} \times \phi_{i,k,j} - \lambda_{i,k,j}} \right), \quad B = \sum_{k \in K} u_k \times \Lambda_k, \tag{29}$$

$$C = \sum_{i \in I} \bar{c}_i, \quad D = \sum_{i \in I, h \in H_i} c_{i,h} \times y_{i,h}, \quad E = \sum_{i \in I} cs_i \max(0, x_i - \bar{x}_i), \tag{30}$$

and

$$F = \sum_{i \in I, k \in K, j \in N_j} cm \max(0, z_{i,j,k} - \bar{z}_{i,j,k}). \tag{31}$$

Table 11: 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 12: 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 $k \in K$, $\quad \Lambda_k = \lambda_k + \sum_{k' \in K} \Lambda_{k'} \pi_{k,k'}$
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 virtual machine 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

The nine constraints are:

(Γ_1) $\sum_{i \in I} \lambda_{i,k,j} = \Lambda_k \quad \forall k \in K, j \in N_k,$ $\quad \Rightarrow \quad$ the traffic assigned to all servers for class k requests equals the predicted load for the class.

(Γ_2) $\sum_{k \in K, j \in N_k} \phi_{i,k,j} \leq 1 \quad \forall i \in I,$ $\quad \Rightarrow \quad$ server i cannot be allocated more workload than its capacity.

(Γ_3) $\sum_{h \in H_i} y_{i,h} = x_i \quad \forall i \in I,$ $\quad \Rightarrow \quad$ 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) $z_{i,k,j} \leq x_i, \quad \forall i \in I, k \in K, j \in N_k$ $\quad \Rightarrow \quad$ requests can only be assigned to active servers.

(Γ_5) $\lambda_{i,k,j} \leq \Lambda_k \times z_{i,k,j} \quad \forall i \in I, k \in K, j \in N_k$ $\quad \Rightarrow \quad$ requests may run on server $i \in I$ only if the corresponding application tier have been assigned to server i.

(Γ_6) $\lambda_{i,k,j} \leq \left(\sum_{h \in H_i} C_{i,h} \times y_{i,h}\right) \mu_{k,j} \times \phi_{i,k,j} \quad \forall i \in I, k \in K, j \in N_k$ $\quad \Rightarrow \quad$ resources cannot be saturated.

(Γ_7) $RAM_{k,j} \times z_{i,k,j} \leq \overline{RAM}_i \quad \forall i \in I, k \in K$ $\quad \Rightarrow \quad$ the memory on server i is sufficient to support all applications running on it.

(Γ_8) $\Pi_{j=1}^{N_k} \left(1 - \Pi_{i=1}^{M}(1 - a_i^{w_{i,k}}\right) \geq A_k, \quad \forall k \in K$ $\quad \Rightarrow \quad$ the availability of all servers assigned to class k request should be at least equal to the minimum required by the SLA

(Γ_9) $\sum_{j=1}^{N_k} z_{i,k,j} \geq N_k \times w_{i,k}, \quad \forall i \in I, k \in K$
$\lambda_{i,j,k}, \phi_{i,j,k} \geq 0, \quad \forall i \in I, k \in K, j \in N_k$
$x_i, y_{i,h}, z_{i,k,j}, w_{i,k} \in \{0,1\}, \quad \forall i \in I, k \in K, j \in N_k$ $\quad \Rightarrow \quad$ constraints and relations among decision variables.

7.2 Applications of control theory to task scheduling on a cloud

Control theory has been used to design adaptive resource management for many classes of applications including: power management [129], task scheduling [153], QoS adaptation in Web servers [3], and load balancing. 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 in 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 [248]; the technique allows multiple QoS objectives and operating constraints to be expressed as a cost function and can be applied to stand alone or distributed web servers, database servers, high performance application servers, and even mobile/embedded systems. The following discussion considers a single processor serving a stream of input requests; we attempt to minimize a cost function which reflects the response time and the power consumption. Our goal is to illustrate the methodology for optimal resource management based on control theory concepts; the analysis is intricate and cannot not be easily extended to a collection of servers.

We start our discussion with a brief overview of control theory principles one could use for optimal resource allocation. 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; this cost function is minimized subject to the constraints imposed by the system dynamics. The discrete-time optimal control problem is to determine the sequence of control variables $u(i), u(i+1), \dots, u(n-1)$ to minimize the expression

$$J(i) = \Phi(n, x(n)) + \sum_{k=i}^{n-1} L^k(x(k), u(k)) \tag{32}$$

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

$$x(k+1) = f^k(x(k), u(k)). \tag{33}$$

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

$$\Lambda(x, y, \lambda) = g(x, y) + \lambda \times [h(x, y) - k]. \tag{34}$$

A necessary condition for the optimality is that (x, y, λ) is a stationary point for $\Lambda(x, y, \lambda)$, in other words

$$\nabla_{x,y,\lambda}\Lambda(x, y, \lambda) = 0 \qquad \text{or} \qquad \left(\frac{\partial \Lambda(x, y, \lambda)}{\partial x}, \frac{\partial \Lambda(x, y, \lambda)}{\partial y}, \frac{\partial \Lambda(x, y, \lambda)}{\partial \lambda} \right) = 0. \tag{35}$$

The Lagrange multiplier at time step k is λ_k and we solve Equation 35 as an unconstrained optimization problem. We define an adjoint cost function which includes the original state constrains as the Hamiltonian function H, then we construct the adjoint system consisting of the original state equation and the *costate equation* governing the Lagrange multiplier. Thus, we define a two-point boundary problem[52]; the state x_k develops forward in time while the costate occurs backward in time.

Now we 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 41 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 a FCFS (First-Come First-Served) basis. We do not assume apriori 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

[52]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.

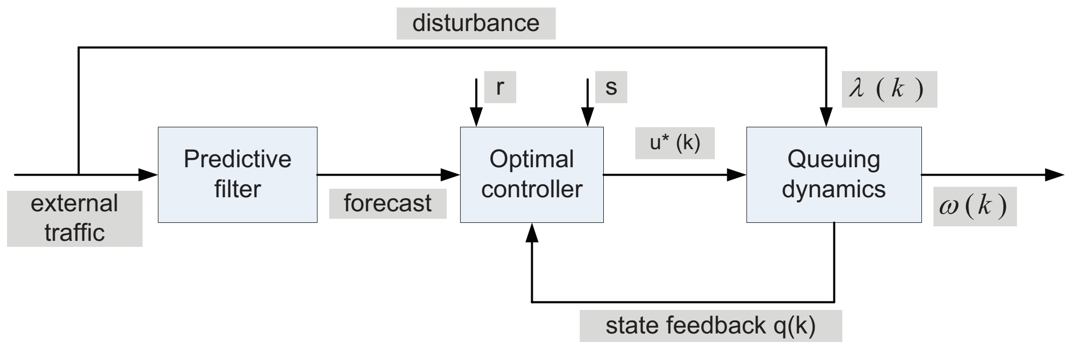


Figure 41: The structure of an optimal controller described in [248]; the controller uses the feedback regarding the current state as well as the estimation of the future disturbance due to environment to compute the optimal inputs over a finite horizon. The two parameters r and s are the weighting factors of the performance index.

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 the express an estimate of the processing rate $N(k)$ as $\alpha(k)/\hat{c}(k)$.

The behavior of a single processor is modelled 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

$$q(k+1) = \max \left\{ \left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}} \right) \times T_s \right], 0 \right\} \tag{36}$$

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

$$\omega(k) = (1 + q(k)) \times \hat{c}(k). \tag{37}$$

Indeed, the total number of requests in the systems 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 39, captures the rewards, as well as, the penalties specified by the service level agreement for the response time. In our queuing model the utility is a function of the size of the queue; it can be expressed as a quadratic function of the response time

$$S(q(k)) = 1/2 \left(s \times (\omega(k) - \omega_0)^2 \right) \tag{38}$$

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

$$R(u(k)) = 1/2\left(r \times u(k)^2\right). \tag{39}$$

The two parameters s and r are weights for the two components of the cost, the one derived from the utility function and the second from the energy consumption. We have to pay a penalty for the requests left in the queue at the end of the control horizon, a quadratic function of the queue length

$$\Phi(q(N)) = 1/2\left(v \times q(n)^2\right). \tag{40}$$

The performance measure of interest is a cost expressed as

$$J = \Phi(q(N)) + \sum_{k=1}^{N-1}\left[S(q(k)) + R(q(k))\right]. \tag{41}$$

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 41 is minimized subject to the following constraints

$$q(k+1) = \left[q(k) + \left(\hat{\Lambda}(k) - \frac{u(k)}{\hat{c}(k) \times u_{max}}\right) \times T_s\right], \quad q(k) \geq 0, \quad \text{and} \quad u_{min} \leq u(k) \leq u_{max}. \tag{42}$$

When the state trajectory $q(\cdot)$ corresponding to the control $u(\cdot)$ satisfies the constraints

$$\Gamma1: \ q(k) > 0, \qquad \Gamma2: \ u(k) \geq u_{min}, \qquad \Gamma3: \ u(k) \leq u_{max}, \tag{43}$$

then the pair $[q(\cdot), u(\cdot)]$ is called a *feasible state.* If the pair minimizes the Equation 41 then the pair is *optimal.*

The Hamiltonian H in our example is

$$\begin{aligned} H = \ & S(q(k)) + R(u(k)) + \lambda(k+1) \times \left[q(k) + \left(\Lambda(k) - \tfrac{u(k)}{c \times u_{max}}\right) T_s\right] \\ & + \mu_1(k) \times (-q(k)) + \mu_2(k) \times (-u(k) + u_{min}) + \mu_3(k) \times (u(k) - u_{max}). \end{aligned} \tag{44}$$

According to Pontryagin's minimum principle[53] 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

$$H(k, q^*, u^*, \lambda^*, \mu^*) \leq H(k, q, u^*, \lambda^*, \mu^*), \ \forall q \geq 0 \tag{45}$$

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

$$\mu_1(k) \geq 0, \ \mu_1(k)(-q(k)) = 0, \tag{46}$$

$$\mu_2(k) \geq 0, \ \mu_2(k)(-u(k) + u_{min}) = 0, \tag{47}$$

[53]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.

$$\mu_3(k) \geq 0, \;\; \mu_3(k)(u(k) - u_{max}) = 0. \tag{48}$$

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 [248]. Clearly, 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 requires the implementation of a complex workflow.

7.3 Stability of a two-level resource allocation architecture

In Section 7.2 we have seen that we can assimilate a server with a closed-loop control system and we can apply control theory principles to resource allocation. In this section we discuss a two-level resource allocation architecture based on control theory concepts for the entire cloud; the automatic resource is based on two levels of controllers, one for the service provider and one at the application level.

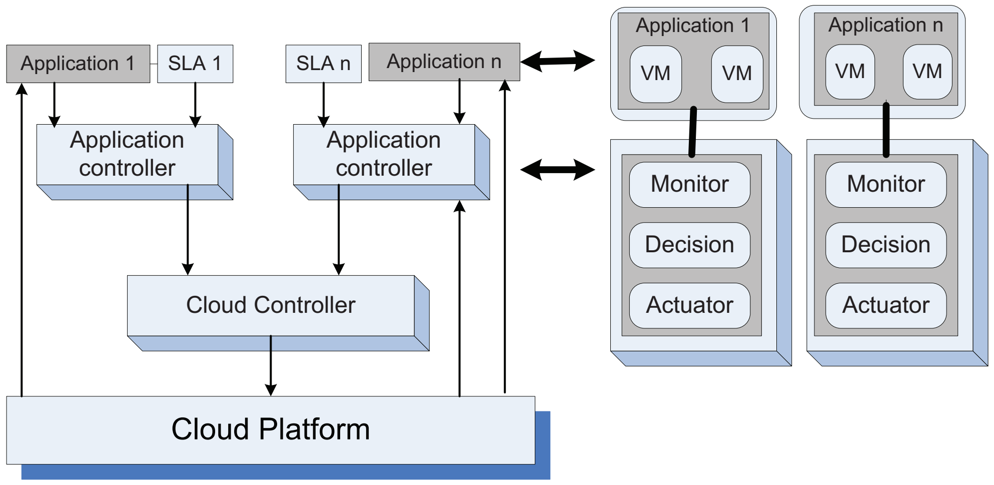


Figure 42: Cloud and applications controllers in a two-level control architecture [77]
.

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 of the output. If the change is too large then the system may become unstable. In our context the system could experience thrashing, the useful time dedicated to the execution of applications becomes increasingly smaller and most of 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 lead to very large changes of the output;
3. Oscillations, when the changes of the input are too large and the control is too weak such that the changes of 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 [77]. A first observation is that the actions of the control system should be carried in a rhythm that does not lead to instability; adjustments should only be carried after the system's performance 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 upper and a lower thresholds are set, then instability occurs when they are too close to one another if the variation of the workload are 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 virtual machines; sometimes allocation/dealocation of a single VM required by one of the threshold may cause crossing of the other, another source of instability.

7.4 Coordination of multiple 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 [129]. The paper reports on actual experiments carried out on a set of blades mounted on a chassis, see Figure 43 for 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 DVS (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 coordinate power and performance management in [129] 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

$$U_{pp}(R, P) = U(R) - \epsilon \times P \qquad \text{or} \qquad U_{pp}(R, P) = \frac{U(R)}{P}, \tag{49}$$

with $U(R)$ the utility function based on response time only and ϵ a parameter to weight the influence of the two factors, response time and power.

- 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[54] to the formware 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 form IBM Web Services Toolkit as a workload generator.

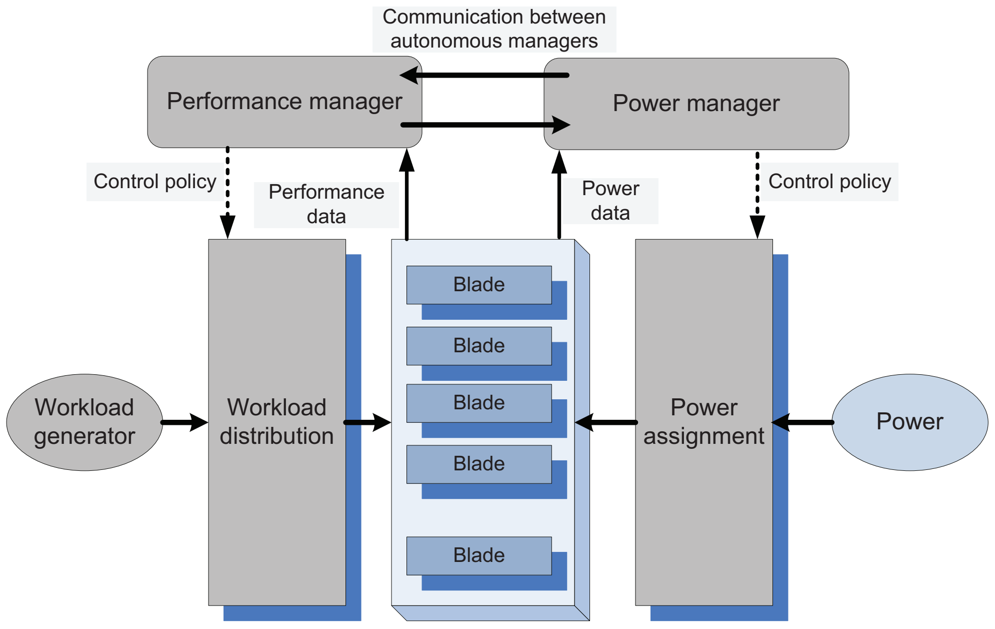


Figure 43: Autonomous performance and power managers cooperate to ensure SLA prescribed performance and energy optimization; they are fed with performance and power data and implement the performance and power management policies, respectively.

[54]IPMI (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.

For practical reasons the utility function was expressed in terms of n_c, the number of clients, and p_κ the powercap, as in

$$U'(p_\kappa, n_c) = U_{pp}(R(p_\kappa, n_c), P(p_\kappa, n_c)). \tag{50}$$

The optimal powercap, p_κ^{opt} is a function of the workload intensity expressed by the number of clients, n_c,

$$p_\kappa^{opt}(n_c) = \arg\max U'(p_\kappa, n_c). \tag{51}$$

The hardware used for these experiments were the Goldensbridge blade with an Intel Xeon processor running at 3 GHz with 1 GB of level 2 cache and 2 GB of DRAM and with hyper treading enabled. A blade could serve 30 to 40 clients with a response time at or better than $1,000$ msec limit. When p_k is lower than 80 Watts, the processor runs at its lowest frequency, 375 MHz, while for p_k 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 lead 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 wats.
- 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 [238]. 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 49 was used; for example, when $n_c = 50$ then $p_\kappa = 109$ Watts when $\epsilon = 0.05$, while $p_\kappa = 120$ for $\epsilon = 0.01$.

7.5 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.

Real-time applications involve periotic 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 task 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; the 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 composition of the workload deserves some discussion. 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 the 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. We distinguish two types of divisible workloads: *modularly divisible* when the workload partitioning is defined apriori and *arbitrarily divisible* when the workload can be partitioned into an arbitrarily large number of segments. The vast literature dedicated to Divisible Load Theory (DLT) includes hundreds of papers (see http://www.ece.sunysb.edu/∼tom/dlt.html); many realistic applications in physics, biology, and other areas of computational science and engineering obey the arbitrarily divisible load sharing model.

In our discussion 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 any 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 and
2. the workload partitioning and the task mapping to worker nodes.

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 the order of their arrival; EDF (Earliest Deadline First), the task with the earliest deadline is scheduled first; and 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

$$DC_i(n^{min}) = W_i(n_i^{min} + 1) - W_i(n_i^{min}) \tag{52}$$

with $W_i(n)$ the workload allocated to task Π_i when n nodes of the cloud are available; if $\mathcal{E}(\sigma_i, n)$ is the execution time of the task then $W_i(n) = n \times \mathcal{E}(\sigma_i, n)$. 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 and

2. the number n of nodes assigned to the application is kept to a minimum, n_i^{min}.

We discuss two workload partitioning and task mapping to worker nodes, the Optimal Partitioning Rule (OPR) and the Equal Partitioning Rule (EPR). The optimality in OPR refers to the execution time; in this case the workload is partitioned to ensure the earliest possible completion time and all tasks are required to complete at the same time. EPR, as the name suggests, means that the workload is partitioned in equal segments. In our discussion we use the derivations and some of the notations in [149]; these notations are summarized in Table 13.

Table 13: 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	cost of transferring a unit of workload from the head node S_0 to worker S_i
ρ	cost of 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, $\Delta_i = \alpha_i \times \sigma \times \rho,\ 1 \leq i \leq n$ needs to process a unit of data
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 $\mathcal{A}$

The timing diagram in Figure 44 allows us to determine the execution time $\mathcal{E}(n,\sigma)$ of the OPR as

$$\begin{aligned} \mathcal{E}(n,\sigma) &= \Gamma_1 + \Delta_1 \\ &= \Gamma_1 + \Gamma_2 + \Delta_2 \\ &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \Delta_3 \\ &\vdots \\ &= \Gamma_1 + \Gamma_2 + \Gamma_3 + \ldots + \Gamma_n + \Delta_n. \end{aligned} \tag{53}$$

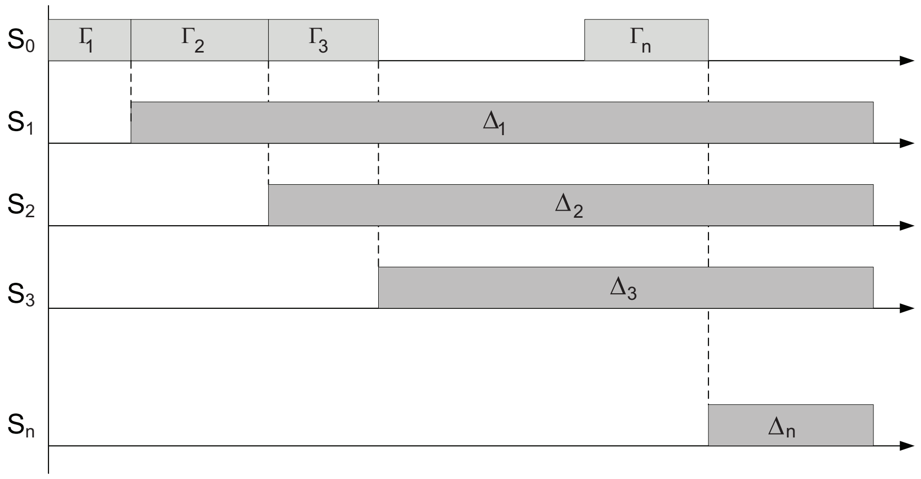


Figure 44: The timing diagram for the Optimal Partitioning Rule; the algorithm requires worker nodes to complete execution at the same time. The head node, S_0, distributes sequentially the data to individual worker nodes; the communication time is $\Gamma_i = \alpha_i \times \sigma \times \tau,\ 1 \leq i \leq n$. Worker node S_i starts processing the data as soon as the transfer is complete; the processing time is $\Delta_i = \alpha_i \times \sigma \times \rho,\ 1 \leq i \leq n$.

We substitute the expressions of $\Gamma_i, \Delta_i,\ 1 \leq i \leq n$ and rewrite these equations as

$$\begin{aligned} \mathcal{E}(n,\sigma) &= \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_2 \times \sigma \times \rho \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \alpha_3 \times \sigma \times \rho \\ &= \vdots \\ &= \alpha_1 \times \sigma \times \tau + \alpha_2 \times \sigma \times \tau + \alpha_3 \times \sigma \times \tau + \ldots + \alpha_n \times \sigma \times \tau + \alpha_n \times \sigma \times \rho. \end{aligned} \tag{54}$$

From the first two equations we find the relation between α_1 and α_2 as

$$\alpha_1 = \frac{\alpha_2}{\beta} \quad \text{with} \quad \beta = \frac{\rho}{\tau + \rho}, \quad 0 \leq \beta \leq 1. \tag{55}$$

This implies that $\alpha_2 = \beta \times \alpha_1$; it is easy to see that in the general case

$$\alpha_i = \beta \times \alpha_{i-1} = \beta^{i-1} \times \alpha_1. \tag{56}$$

But α_i are the components of the load distribution vector thus,

$$\sum_{i=1}^{n} \alpha_i = 1. \tag{57}$$

Next substitute the values of α_i and obtain the expression for α_1:

$$\alpha_1 + \beta \times \alpha_1 + \beta^2 \times \alpha_1 + \beta^3 \times \alpha_1 \ldots \beta^{n-1} \times \alpha_1 = 1 \quad \text{or} \quad \alpha_1 = \frac{1-\beta}{1-\beta^n}. \tag{58}$$

We have now determined the load distribution vector and we can now determine the execution time as

$$\mathcal{E}(n,\sigma) = \alpha_1 \times \sigma \times \tau + \alpha_1 \times \sigma \times \rho = \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho). \tag{59}$$

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

$$C^{\mathcal{A}}(n) = t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho). \tag{60}$$

The application meets its deadline if and only if

$$C^{\mathcal{A}}(n) \leq A + D \tag{61}$$

or

$$t_0 + \mathcal{E}(n,\sigma) = t_0 + \frac{1-\beta}{1-\beta^n}\sigma(\tau+\rho) \leq A + D. \tag{62}$$

But $0 < \beta < 1$ thus, $1 - \beta^n > 0$ and it follows that

$$(1-\beta)\sigma(\tau+\rho) \leq (1-\beta^n)(A + D - t_0). \tag{63}$$

The application can only meet its deadline if $(A + D - t_0) > 0$ and under this condition this inequality becomes

$$\beta^n \leq \gamma \quad \text{with} \quad \gamma = 1 - \frac{\sigma \times \tau}{A + D - t_0}. \tag{64}$$

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

$$n^{min} = \lceil \frac{\ln \gamma}{\ln \beta} \rceil. \tag{65}$$

The Equal Partitioning Rule (EPR) assigns an equal workload to individual worker nodes, $\alpha_i = 1/n$. From the diagram in Figure 45 we see that

$$\mathcal{E}(n,\sigma) = \sum_{i=1}^{n} \Gamma_i + \Delta_n = n \times \frac{\sigma}{n} \times \tau + \frac{\sigma}{n} \times \rho = \sigma \times \tau + \frac{\sigma}{n} \times \rho. \tag{66}$$

The condition for meeting the deadline, $C^{\mathcal{A}}(n) \leq A + D$ leads to

$$t_0 + \sigma \times \tau + \frac{\sigma}{n} \times \rho \leq A + D \quad \text{or} \quad n \geq \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau}. \tag{67}$$

Thus,

$$n^{min} = \lceil \frac{\sigma \times \rho}{A + D - t_0 - \sigma \times \tau} \rceil. \tag{68}$$

The pseudocode for a general schedulability test for FIFO, EDF, and MDF scheduling policies, for two node allocation policies, MN (minimum number of nodes) and AN (all nodes), and for OPR and EPR partitioning rules are given in reference [149]. The same paper

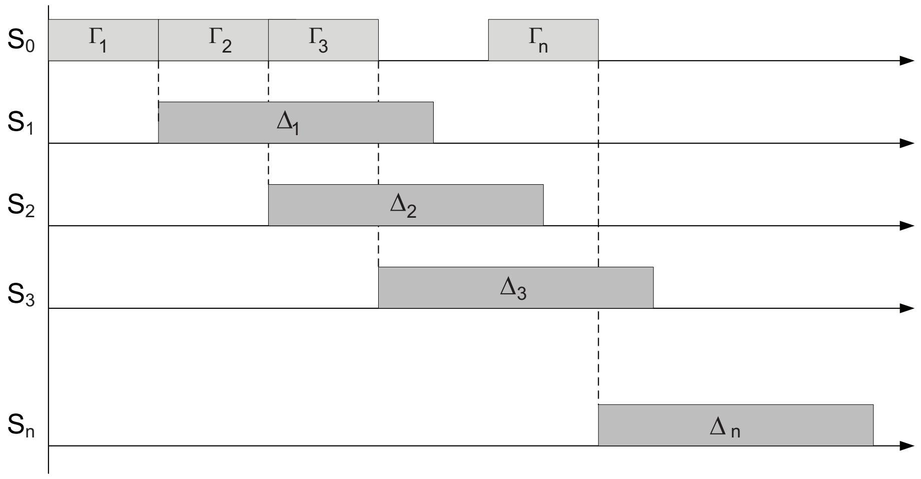


Figure 45: The timing diagram for the Equal Partitioning Rule; the algorithm assigns an equal workload to individual worker nodes , $\alpha_i = 1/n$. The head node, S_0, distributes sequentially the data to individual worker nodes; the communication time is $\Gamma_i = (\sigma/n) \times \tau,\ 1 \leq i \leq n$. Worker node S_i starts processing the data as soon as the transfer is complete; the processing time is $\Delta_i = (\sigma/n) \times \rho,\ 1 \leq i \leq n$.

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/MDF, No=MN/AN, and Pa=OPR/EPR. For example, MDF-MN-OPR uses MDF 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.

Now we turn our attention to applications of the analysis in this section and discuss scheduling of MapReduce applications on the cloud subject to deadlines. Current options for scheduling Apache Hadoop, an open source implementation of Google's MapReduce algorithm are: the default FIFO schedular, the FairScheduler [261], the Capacity Scheduler, and the Dynamic Proportional Scheduler [215]. A recent paper [128] applies the deadline scheduling framework analyzed in this section to Hadoop tasks.

Table 14 summarizes the notations used for the analysis of Hadoop; the term "slots" is equivalent with "nodes" and means the number of instances. We make several assumptions for our initial derivation; first, we assume that the system is homogeneous, ρ_m and ρ_r, the cost of processing a unit data in map task and the reduce task, respectively are the same for all servers. Second, we assume an equipartition of the workload. Under these conditions the duration of the job J with input of size σ is

$$\mathcal{E}(n_m, n_r, \sigma) = \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right] \tag{69}$$

thus, the condition that the query $Q = (A, \sigma, D)$ with arrival time A meets the deadline D can be expressed as

$$t_m^0 + \sigma \left[\frac{\rho_m}{n_m} + \phi \left(\frac{\rho_r}{n_r} + \tau \right) \right] \leq A + D. \tag{70}$$

Table 14: The parameters used for 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, $1 \leq i \leq u$
Π_r^j	a reduce task, $1 \leq j \leq v$
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	cost of processing a unit data in map task
ρ_r	cost of processing a unit data in reduce task
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 $\alpha_i = 1/u$
ϕ	filter ratio, the fraction of the input produced as output by the map process

It follows immediately that the maximum value for the startup time of the reduce task is

$$t_r^{max} = A + D - \sigma\phi\left(\frac{\rho_r}{n_r} + \tau\right). \tag{71}$$

We now plug the expression of the maximum value for the startup time of the reduce task in the condition to meet the deadline

$$t_m^0 + \sigma\frac{\rho_m}{n_m} \leq t_r^{max}. \tag{72}$$

It follows immediately that n_m^{min}, the minimum number of slots for the map task, satisfies the condition

$$n_m^{min} \geq \frac{\sigma\rho_m}{t_r^{max} - t_m^0}, \quad \text{thus,} \quad n_m^{min} = \lceil\frac{\sigma\rho_m}{t_r^{max} - t_m^0}\rceil. \tag{73}$$

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 [128].

7.6 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 and 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.

Auctions in which participants can bid on combinations of items or *packages* are called *combinatorial auctions* [65]; such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the *Simultaneous Clock Auction* [27] and the *Clock Proxy Auction* [28]; the algorithm discussed in this paper and introduced in [228] 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, $u = \{1, 2, \dots, U\}$, and R resources, $r = \{1, 2, \dots, R\}$. The bid of user u is $\mathcal{B}_u = \{\mathcal{Q}_u, \pi_u\}$ with $\mathcal{Q}_i = (q_u^1, q_u^2, q_u^3, \dots)$ 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 component of a vector q_u^i is a positive quantity and encodes the quantity of a resource desired, or if negative, the quantity of the resource offered. A user expresses her desires as an *indifference set* $\mathcal{I} = (q_u^1 \text{ XOR } q_u^2 \text{ XOR } q_u^3 \text{ XOR } \dots)$.

The final auction prices for individual resources are given by the vector $p = (p^1, p^2, \dots, p^R)$ and the amounts of resources allocated to user u are $x_u = (x_u^1, x_u^2, \dots, x_u^R)$; 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. The scalar $[\min_{q \in \mathcal{Q}_u}(q^T p)]$ is the final price established through the bidding process.

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.

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 she is a looser.

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 optimization algorithm and the constraints imposed in it are

$\max_{x,p} \ f(x,p)$	$\Leftarrow$	the function to be maximized
constraints		
$x_u \in \{0 \cup \mathcal{Q}_u\} \ \forall u$	$\Leftarrow$	a user gets all resources or nothing
$\sum_u x_u \leq 0$	$\Leftarrow$	final allocation leads to a net surplus of resources
$\pi_u \geq (x_u)^T p, \ \forall u \in \mathcal{W}$	$\Leftarrow$	auction winners are willing to pay the final price
$(x_u)^T p = \min_{q \in \mathcal{Q}_u}(q^T p), \ \forall u \in \mathcal{W}$	$\Leftarrow$	winners get the cheapest bundle in $\mathcal{I}$
$\pi_u < \min_{q \in \mathcal{Q}_u}(q^T p), \ \forall u \in \mathcal{L}$	$\Leftarrow$	the bids of the losers are below the final price
$p \geq 0$	$\Leftarrow$	prices must be non-negative

The constraints are self-explanatory: the first one states that a user either gets one of the bundles it has opted for or nothing, no partial allocation is acceptable; the second one expresses the fact that the system awards only available resources, only offered resources can be allocated; the third one is that the bid of the winners exceeds the final price; the fourth one states that the winers get the least expensive bundles in their indifference set; the fifth one states that losers bid below the final price; finally, the last one states that all prices are positive numbers.

Informally, in the ASCA algorithm [228] the participants at the auction specify the resource and the quantities of that resource offered or desired at the price listed for that time slot. Then the excess vector, $z(t) = \sum_u x_u(t)$ is computed and if all its components are negative then the auction stops; negative components means that the demand does not exceed the offer. If the demand is larger than the offer, $z(t) \geq 0$ than the auctioneer increases the price for items with a positive excess demand and solicits bids at the new price. Note that the algorithm satisfies conditions 1 through 6, 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 46. These proxies can be modelled as functions which compute the "best bundle" from each $\mathcal{Q}_u$ set given the current price

$$\mathcal{Q}_u = \begin{cases} \hat{q}_u & \text{if } \hat{q}_u^T p \leq \pi_u \quad \text{with } \hat{q}_u \in \arg\min(q_u^T p) \\ 0 & \text{otherwise} \end{cases}$$

The pseudo code of the ASCA algorithm is

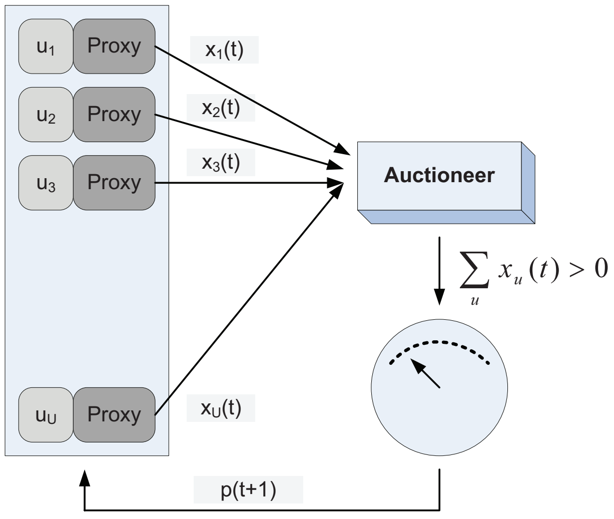


Figure 46: The schematics of the ASCA algorithm; to allow for a single round auction users are represented by proxies which place the bids $x_u(t)$. The auctioneer determines if there is an excess demand and in that case it raises the price of resources for which the demand exceeds the supply and requests new bids.

```
Input: $U$ users, $R$ resources, starting price $\bar{p}$, update increment function $g : (x,p) \mapsto \mathbb{R}^R$
1 :  set $t = 0, p(0) = \bar{p}$
2 :  **loop**
3 :      collect bids $x_u(t) = \mathcal{G}_u(p(t)), \quad \forall u$
4 :      calculate excess demand $z(t) = \sum_u x_u(t)$
5 :      **if** $z(t) < 0$ **then**
6 :          break
7 :      **else**
8 :          update prices $p(t+1) = p(t) + g(x(t), p(t))$
9 :          $t \leftarrow t + 1$
10 :     **end if**
11 : **end loop**
```

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 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 convergence of the optimization problem is guaranteed *only if* all participants at the auction are either providers of resources or consumer 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 [228] have implemented the algorithm within Google organization and 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 Ascending Clock Auction algorithm.

7.7 Further readings

A fair number of papers are dedicated to resource management in cloud computing including [54], [214], [21]. Several papers are concerned with Service Level agreements and QoS, e.g., [4] covers SLA-driven capacity management and [22] covers SLA-based resource allocation policies. Dynamic request scheduling of applications subject to SLA requirements is presented in [44]. The QoS in clouds is analyzed in [44].

Autonomic computing is the subject of papers such as: [23] which covers energy-aware resource allocation in autonomic computing; [130] which analyzes policies for autonomic computing based on utility functions; [129] which discusses coordination of multiple autonomic managers and power-performance tradeoffs; and [9] which presents autonomic management of cloud services subject to availability guarantees.

Auctions in which participants can bid on combinations of items or *packages* are called *combinatorial auctions* [65]; such auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation. Two recent combinatorial auction algorithms are the *Simultaneous Clock Auction* [27] and the *Clock Proxy Auction* [28]; the algorithm discussed in this paper and introduced in [228] is called *Ascending Clock Auction, (ASCA).*

An authoritative reference on fault-tolerance is [29]; applications of control theory to resource allocation are discussed in [77]. [51] covers resource multiplexing in data centers. Admission control policies are discussed in [103]. Power and performance management are the subject of [138] and performance management for cluster based Web services is covered in [194]. Autonomic management of heterogeneous workloads is discussed in [233] and application placement controllers is the topic of [235].

Scheduling and resource allocation are also covered by numerous papers: a batch queuing system on clouds with Hadoop and HBase is presented in [263]. The paper [74] is focused on data flow driven scheduling for business applications. Scalable thread scheduling is the topic of [253]. Scheduling of realtime services in cloud computing is presented in [151].

8 Networking support for cloud computing

Cloud computing and delivery of content stored on a cloud are feasible only due to the interconnectivity supported by a continually evolving Internet and by the access access to remote resources provided by the World Wide Web. The cloud itself is built around a high-performance interconnect; the servers in a cloud communicate through high-bandwidth and low-latency specialized networks. It is thus obvious why networking will continue to play a crucial role in the evolution of cloud computing and content-centric world.

This chapter starts with a brief review of basic concepts related to packet switching and some transformational changes suffered by the Internet under the pressure of applications and the need to accommodate a very large address space. Then we overview models for network resource management. Our presentation continues with some statistics regarding Web metrics and the arguments for increasing the initial congestion control window in TCP. Next we discuss efficient topologies for computer clusters and storage area networks.

8.1 Packet-switched networks and the Internet

A *packet-switched network* transports data units called *packets* through a maze of switches where packets are queued and routed towards their destination thus, subject to variable delay, loss, and possibly arriving at their final destination out of order. 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 *network architecture* describes the protocol stack used for communication; a *protocol* is a discipline for communication, it specifies the actions taken by the sender and the receiver of a data unit. We use the term *host* to describe a system located at the network edge and capable to initiate and to receive communication, be it a computer, a mobile device such as an phone, or a sensor.

This very concise description hints that a packet-switched network is a complex system consisting of a very large number of autonomous components and subject to complex and sometimes contradictory requirements. Basic strategies for implementing a complex system are *layering* and *modularization*; this means decomposing a complex function into components that interact through well defined channels, e.g., a layer can only communicate with its two adjacent layers. All network architectures are based on layering; this justifies the term protocol stack, Figure 47. In the Internet architecture the network layer is responsible for routing packets through the packet switched network from the source to the destination, while 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.

Physically, at the sender site the data flows down the protocol stack from the application layer to the transport, network, and data link layer and the streams of bits are pushed through a physical communication link encoded either as electrical, optical, or electromagnetic signals. Once they reach a router the bits are passed to the data link and then to the network layer which decides where the packet should be sent, either to another router or to a destination host connected to a local area network connected to the router; then the data link layer encapsulates the packet for the link of the next hop and then the bit stream is passed to the next physical channel, Figure 47. At the receiving end the data flows upwards from the data link to the application layer.

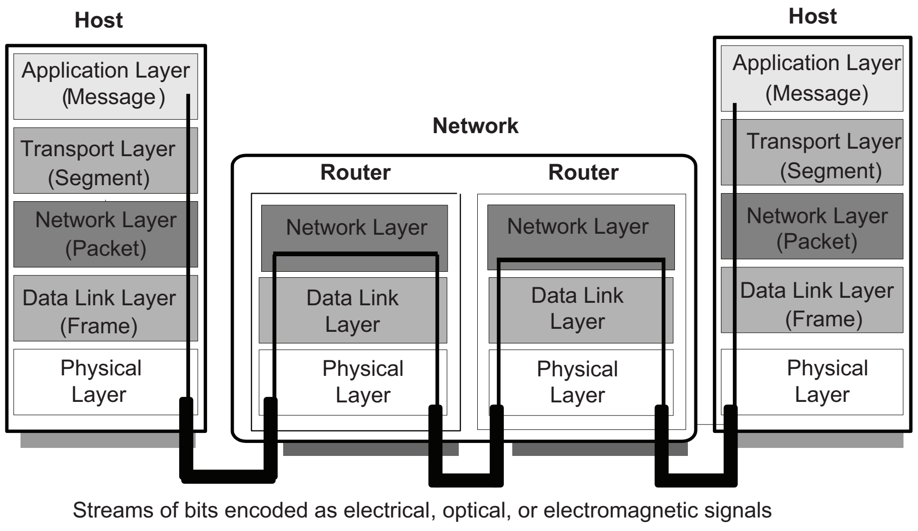


Figure 47: Protocol stacks. Applications running on hosts at the edge of the network communicate using application layer protocols; the transport layer deals with end-to-end delivery, the network layer is responsible for routing a packet through the network, the data link layer ensures reliable communication 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). The corresponding data units for the five layer architecture are: messages, segments, packets, frames and encoded bits, respectively.

A protocol on one system *communicates logically* with its peer on an other system; for example the application protocol on the sender, host A, communicates with the application protocol on the receiver, host B, the sending side encapsulates the data and adds control information as headers that can only be understood by its peer; when the peer receives the data unit, it carries out a decapsulation, it retrieves the control information, removes the headers then passes the payload to the next layer up.

The Internet is a collection of separate and distinct networks, each one operating under a common framework consisting of globally unique IP addressing and using IP routing and global Border Gateway Routing (BGP) protocols[55]. An *IP address* is a unique string of integers uniquely identifying every host connected to the Internet; the IP address allows the network to identify first the destination network and then the host in that network where a datagram should be delivered.

The Internet is based on a hourglass network architecture and the TCP/IP protocol family, see Figure 48. The hourglass model captures the fact that all packets transported through the Internet use the IP (Internet Protocol) to reach the destination. IP ensures datagrams delivery from the source host to the destination host based on their IP addresses. A host may have multiple IP addresses as 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. IP only provides *best effort delivery*

[55]The Border Gateway Protocol (BGP) is a path vector reachability protocol which makes the core routing decisions on the Internet; it maintains a table of IP networks which designate network reachability among autonomous systems. BGP makes routing decisions based on path, network policies and/or rule sets.

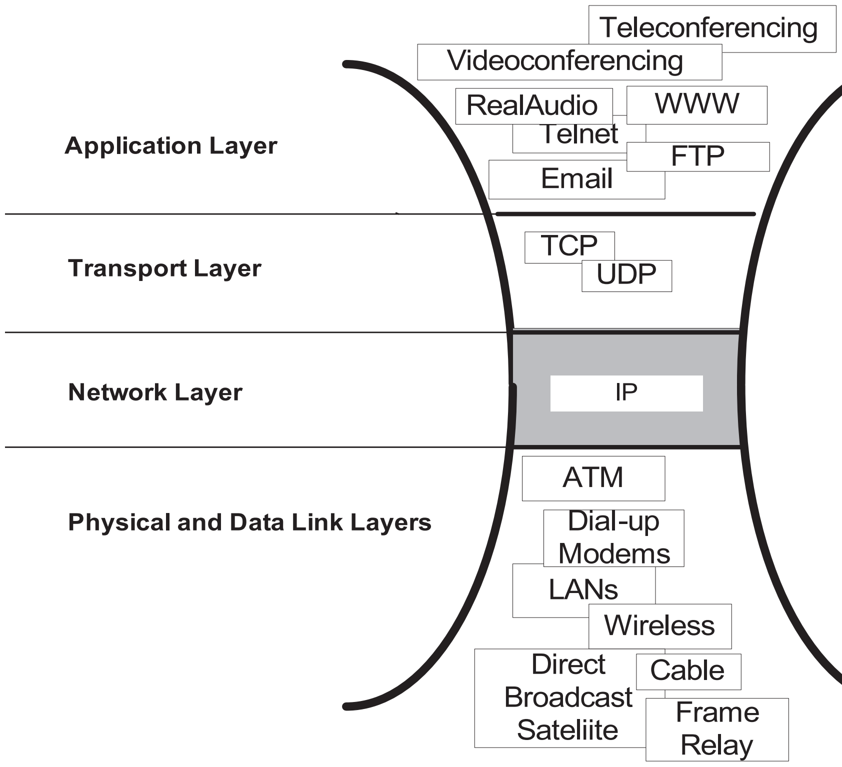


Figure 48: The hourglass network architecture of the Internet. Regardless of the application, the transport protocol, and the physical network all packets are routed through the Internet from the source to the destination by the IP protocol; routing is based on the destination IP address.

and its service is characterized as unreliable; best effort delivery means that any router on the path from the source to the destination may drop a packet when it is overloaded.

The Internet uses two transport protocols, a connectionless datagram protocol, UDP (User Datagram Protocol) and a connection-oriented protocol, TCP (Transport Control Protocol). A datagram is a basic transfer unit in a packet-switched network; it consists of a header and data. The header 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; the datagrams may arrive out of order, duplicated, or may not arrive at all. Applications using UDP include: the DNS (Domain Name System), VoIP (Voice over IP), 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.

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 new traffic is allowed to enter the highway and this is precisely what the TCP emulates.

TCP uses several mechanisms to control the rate of data entering the network, keeping the data flow below a rate that would lead to a network collapse and to enforce a max-min fair allocation between flows. Acknowledgments coupled to timers are used to infer network conditions between the sender and receiver. The four algorithms, slow-start, congestion avoidance, fast retransmit, and fast recovery 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. Obviously, since UDP is a connectionless protocol there are no means to control the UDP traffic.

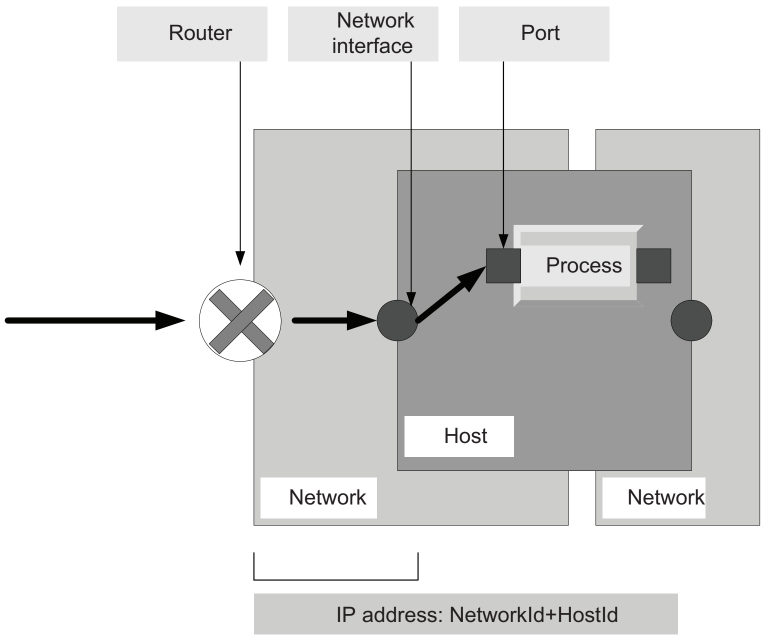


Figure 49: Packet delivery to processes and threads; the packet is first routed by the IP protocol to the destination network and then to the host specified by the IP address. Each application listens to an abstraction for an end point of a logical communication channel called a *port*.

In addition to the logical address, each network interface, the hardware connecting a host with a network, has a unique *physical* or *MAC address*. While the IP address may be dynamically assigned and changes depending on the network, the MAC address is permanently assigned to a network interface of the device. For example, a laptop will get a different IP addresses when it is connected to a different networks, a phone will get different IP addresses in different WiFi networks. A host may have multiple network interfaces and consequently multiple IP addresses when it is connected to multiple networks.

Once a packet reaches the destination host it is delivered to the proper transport protocol demon which, in turn, delivers it to the application which listens to an abstraction for an end point of a logical communication channel called a *port*, Figure 49. The processes or threads running an application use an abstraction called *socket* to send and receive data through the network. A socket manages a queue of incoming messages and one for outgoing messages.

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[56], 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.

8.2 The transformation of the Internet

While the decades long evolution of microprocessor and storage technologies, of the computer architecture and of software systems, of parallel algorithms and distributed control strategies paved the way to cloud computing, at the heart of this new technology are the fascinating developments in networking. In this section we only discuss those features of the Internet affecting cloud computing and focus on the transformation of the Internet core, on migration to IPv6, and on the efforts to update the Internet infrastructure within the US.

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 Internet 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 addresses in an IPv6 multicast group address which 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

[56]NTT (Nippon Telegraph and Telephone) achieved a speed of 69.1 Tbps in 2010; it used wavelength division multiplex (WDM) of 432 wavelengths with a capacity of 171 Gbps over a single 240 km-long optical fiber. Note that 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, this explains why the term channel "bandwidth" is also used to measure the channel capacity, or the maximum data rate.

for its configuration parameters; if configured suitably, 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; it is optional for IPv4.

Unfortunately, migration to IPv6 is a very challenging and costly proposition. 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.

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 $1,000$ phones for an organization using a 3 digit extension. Analogously, NAT (Network Address Translation) 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 Networking). Nowadays Skype and STUN VoIP applications work well with NAT and NAT-T and SSLVPN supports 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 needs 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.

A group from Google have investigated in 2009 the adoption of IPv6 [63] and has concluded that: "...the IPv6 deployment is small but growing steadily, and that adoption is still heavily influenced by a small number of large deployments. While we see IPv6 adoption in research and education networks, IPv6 deployment is, with one notable exception, largely absent from consumer access networks."

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 supporting only the transport of data files, it has morphed into today's network for data with real-time constraints for multi-media applications. To understand the architectural consequences of this evolution we examine first the relations between two networks:

- Peering, the two networks exchange traffic between each other's customers freely.
- Transit, a network pays to another to access the Internet.
- Customer, a network is payed money to allow Internet access.

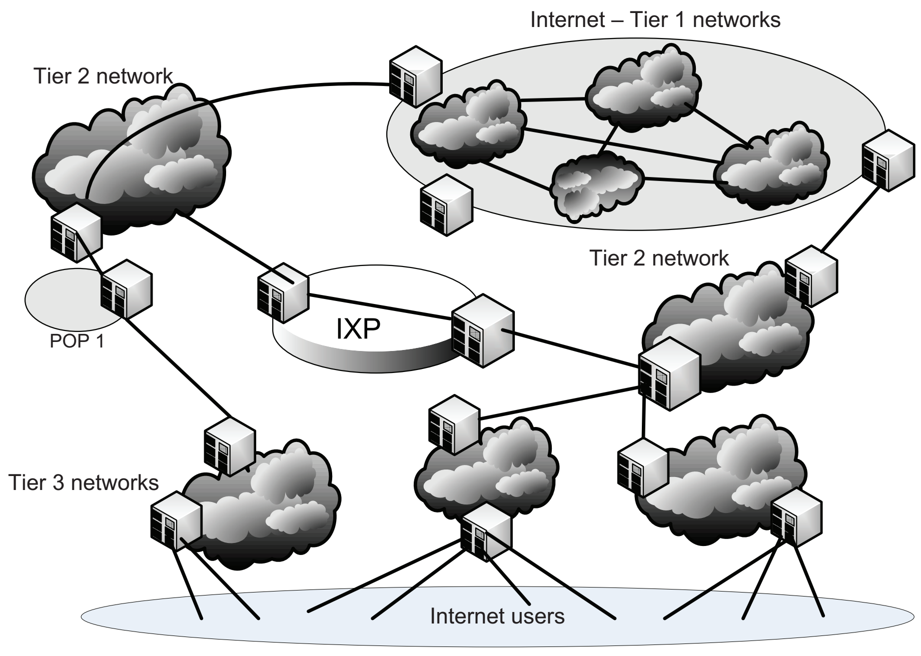


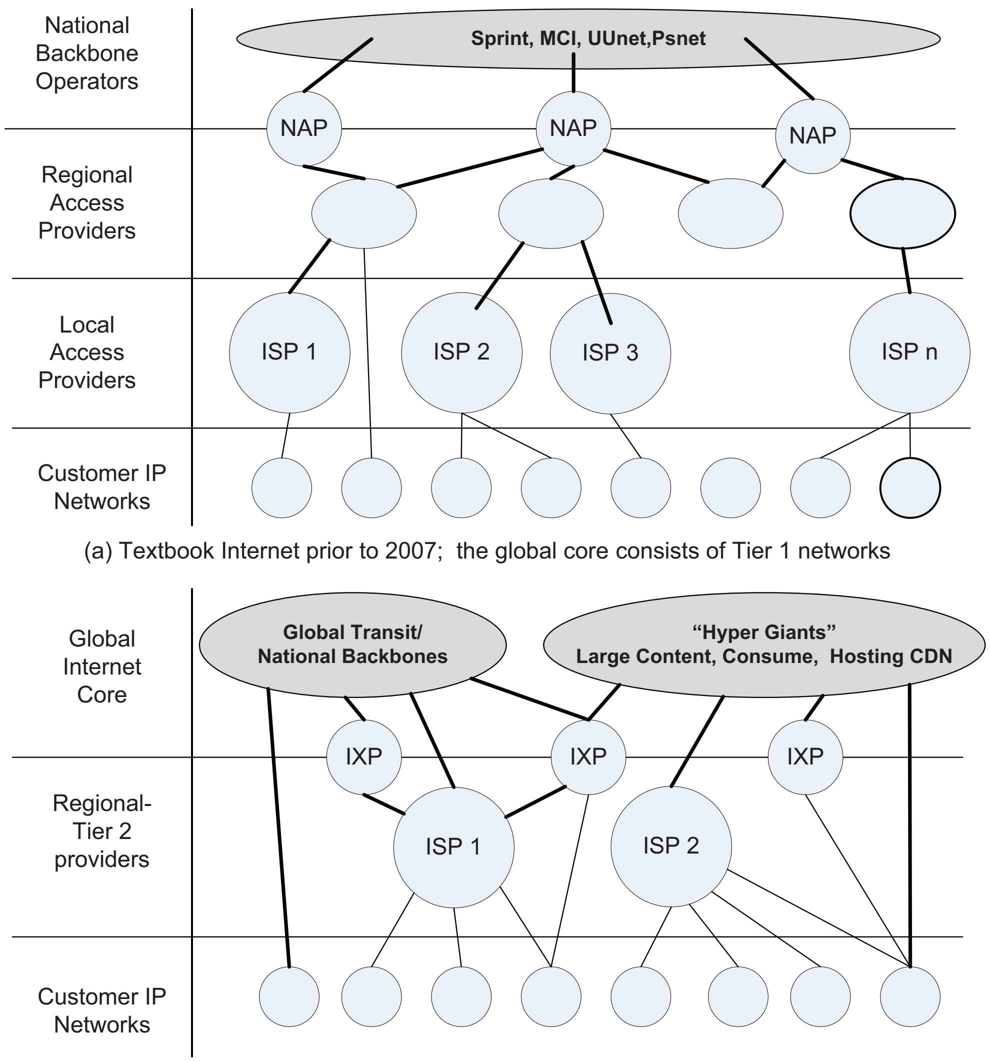
Figure 50: The relation of networks in the Internet.

Based on these relations the networks are commonly classified as Tier 1, 2 and 3. A *Tier 1 networks* is 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. 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. The primary purpose of an IXP is to allow networks to interconnect 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 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 learned 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 connect.

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 51 from [139]. The Web, Gaming, and Entertainment are merging and more computer applications are moving to the cloud. Data steaming consumes an increasingly larger fraction of the available bandwidth as high definition TV sets become less expensive and content providers such as


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
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

(a) Textbook Internet prior to 2007; the global core consists of Tier 1 networks

(b) The 2009 Internet reflects the effect of comoditization of IP hosting and of content-delivery networks (CDNs)

Figure 51: The transformation of the Internet; the traffic carried by Tier 3 providers almost doubled from 2007 to 2009, from about 5.8% to 9.4% while Goggle applications account for 5.2% of the traffic in 2009 [139].

Netflix and Hulu offer customers services that require a significant increase of the network bandwidth.

Does the network infrastructure keep up with the demand for bandwidth? As we can see in Figure 52 the Internet infrastructure in the US is falling behind in terms of the network bandwidth. 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 1Gb/s access speed to individual households through FTTH[57].

[57]The fiber-to-the-home (FTTH) is a broadband network architecture that uses optical fiber to replace the copper-based local loop used for last mile network access to home.

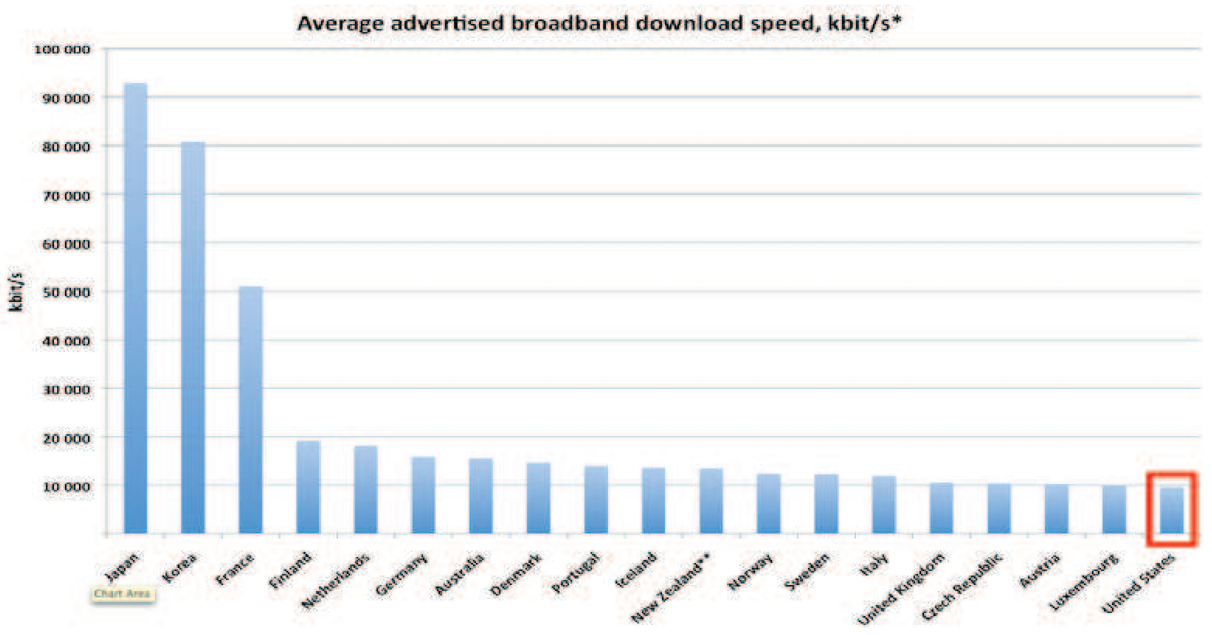


Figure 52: The broadband access, the average download speed advertised by several countries.

8.3 Network resource management

As mentioned repeatedly, cloud computing is intrinsically dependent on communication thus, network resource management is a very important aspect of the management of computer clouds. A critical aspect of resource management in cloud computing is to guarantee the communication bandwidth required by an application as specified by a Service Level Agreement. The solutions to this problem are based on the strategies used for some time in the Internet to support the Quality of Service (QoS) requirements of data streaming.

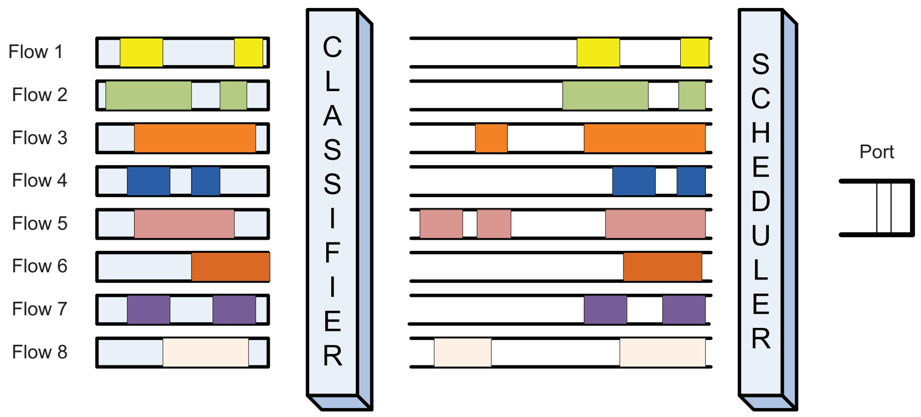


Figure 53: Fair Queuing (FQ) - packets are first classified into flows by the system and then assigned to a queue dedicated to the flow; queues are serviced one packet at a time in roundrobin order and empty queues are skipped.

First, we discuss the Stochastic Fairness Queuing (SFQ) algorithm [164] which takes into account data packet sizes to ensure that each flow has the opportunity to transmit an equal amount of data. SFQ is a simpler and less accurate implementation of fair queueing algorithms and thus, requires less calculations. The Fair Queuing (FQ) algorithm was developed by Nagle [172]; it 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; queues are serviced one packet at a time in roundrobin order, Figure 53. FQ's objective is *max-min* fairness; this means that first it maximizes the minimum data rate of any of the data flows, then it maximize the second minimum data rate, etc. Starvation of expensive flows is avoided but the throughput is lower.

Next, we review a widely used strategy for link sharing, the Class-Based Queuing (CBQ) proposed by Sally Floyd and Van Jacobson in 1995 [90]; 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 CQB 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, which identifies 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.

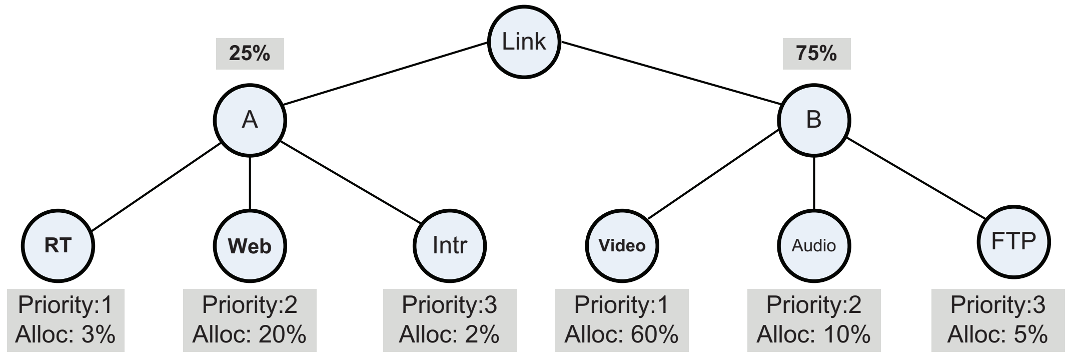


Figure 54: CBQ link sharing for two groups A, short-lived traffic, and B, long-lived traffic, allocated 25% and 75% of the link capacity, respectively. There are three classes with priorities 1, 2, and 3; the RT (real-time) and the video streaming have priority one and are allocated 3% and 60%, respectively, Web transactions and audio streaming have priority 2 and are allocated 20% and 10%, respectively, and Intr (interactive applications) and FTP (file transfer protocols) have priority 3 and are allocated 2% and 5%, respectively, of the link capacity.

The classes are organized in a tree-like hierarchy; for example in Figure 54 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 considered Level 1 include six classes of traffic: real-time, Web, interactive, video streaming, audio streaming, and file transfer; at Level 2 are the two classes of traffic, A and B, and the root; at Level 3 is the link itself.

The link sharing policy aims to ensure that if sufficient demand exits 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 unsatidfied 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 55 for two examples.

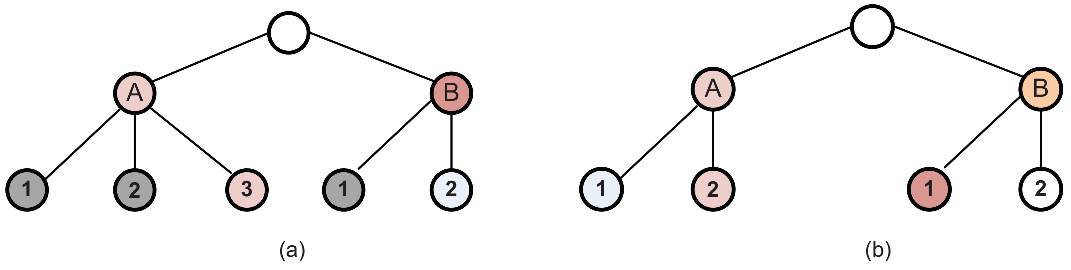


Figure 55: There are two groups A and B and three types of traffic, e.g., Web, real-time, and interactive, denoted as 1, 2, and 3. (a) Group A and class $A.3$ traffic are underlimit and unsatisfied; classes $A.1, A.2$ and $B.1$ are overlimit, unsatisfied and with persistent backlog and have to be regulated; type $A.3$ is underlimit and unsatisfied; group B is over-limit. (b) Group A is underlimit and unsatisfied; Group B is overlimit and needs to be regulated; class $A.1$ traffic is under-limit; class $A.2$ is overlimit and with persistent backlog; class $B.1$ traffic is overlimit and with persistent backlog and needs to be regulated.

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 56. 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.

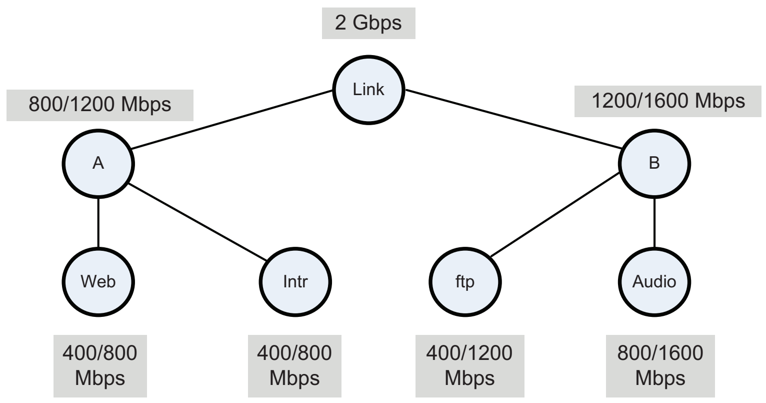


Figure 56: HTB packet scheduling uses for every node a ceil rate in addition to the allowed rate.

8.4 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 thus 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; Table 15 summarizes these metrics. The statistics are collected from a sample of several billion pages detected during Google's crawl and indexing pipeline.

Table 15: Web statistics.

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

Such statistics are useful to tune the transport protocols to deliver optimal performance in terms of latency and throughput; `HTTP`, the application protocol for Web browsers uses the TCP transport protocol which supports mechanisms to limit the amount of data transported over the network to avoid congestion. 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.

Another example illustrating the need of the networking infrastructure to adapt to new requirements is the TCP initial congestion window. Before we analyze in depth this problem we review two important mechanisms to control the data transport, called flow control and congestion control. TCP seeks to achieve high channel utilization, avoid congestion, and, at the same time, ensure a fair sharing of the network bandwidth.

TCP uses a sliding window flow control protocol; if W is the window size, then the data rate S of the sender is:

$$S = \frac{W \times MSS}{RTT} \text{ bps} = \frac{W}{RTT} \text{ packets/second}$$

where MSS and RTT denote the Maximum Segment Size and the Round Trip Time, respectively. If S is too small the transmission rate is smaller than the channel capacity, while a large S leads to congestion. The channel capacity in the case of communication over the Internet is not a fixed quantity but, as different physical channels are shared among many flows, it depends on the load of the network.

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 receive 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 acknowledgement 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 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:

$$W = \min(cwnd, awnd).$$

Several algorithms are used to calculate *cwnd* including Tahoe and Reno, developed by Jacobson in 1988 and 1990. Tahoe was based on slow start (SS), congestion avoidance (CA), and fast retransmismit (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 congestion avoidance 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 then made:

$$flightsize = \min(awnd, cwnd) \quad \text{and} \quad ssthresh \leftarrow \max(flightsize/2, 2)$$

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 [76] show that in 2009 the average bandwidth of an Internet connection was 1.7 Mbps; more than 50% of the traffic can be considered broadband as it requires more than 2 Mbps, while only about 5% of the flows are narrowband and require less that 256 Kbps. 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 streaming 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 overcame 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 Chrom open up 6 TCP connections per domain to increase the parallelism and to boost start-up performance when downloading a Web page, while the Internet Explorer 8 open 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 [76] are:

- The TCP latency is dominated by the number of RTT's during the slow start phase[58]; 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 allow faster recovery after losses recovered through Fast Retransmission.

In the experiments reported in [76] the TCP latency was reduced from about 490 msec when $init\_cwnd = 3$ to about 466 msec for $init\_cwnd = 16$.

[58] It can be shown that the latency of a transfer completing during the slow start without losses is given by the expression

$$\lceil \log_\gamma \left(\frac{L(\gamma - 1)}{init\_cwnd} + 1 \right) \rceil \times RTT + \frac{L}{C}$$

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; $L/init\_cwnd \geq 1$.

8.5 Interconnects for warehouse-scale computers

A warehouse-scale computer (WSC) is an infrastructure consisting of a very large number of servers interconnected by high-speed networks; a WSC is homogeneous in terms of the hardware and the software running on individual servers. While processor and memory technology have followed Moore's law, the interconnection networks have evolved at a slower pace and have become a major factor in determining the overall performance and cost of the system. The speed of the Ethernet has increased from 1 Gbps in 1997 to 100 Gbps in 2010; this increase is slightly slower than the Moore law for traffic [171] which would require, 1 Tbps Ethernet by 2013.

The networking infrastructure of a WSC 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, every 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.

An important element of the interconnection fabric are routers and switches. Routers are switches with a very specific function, joining multiple networks, LANs and WANs; they receive IP packets, look inside each packet to identify the source and target IP addresses, then forward these packets as needed to ensure that data reaches its final destination.

Typically, the networking infrastructure is organized hierarchically. The servers are packed into racks and interconnected by a top of the rack router; then rack routers are connected to cluster routers which in turn are interconnected by a local communication fabric; finally, inter-datacenter networks connects multiple WSCs [135]. The switching fabric must have sufficient bi-directional bandwidth for cloud computing. Clearly, in a hierarchical organization true location transparency is not feasible and cost considerations ultimately decide the actual organization and performance of the communication fabric.

The cost of routers and the number of cables interconnecting the routers are major components of the overall cost of the interconnection network. We should note that the 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 archived with innovative router architecture. This motivates us to take a closer look at the actual design of routers.

The number of ports of a router distinguishes *low-radix* routers with a small number of ports and *high-radix* routers with a large number of ports. High-radix chips divide the bandwidth into larger number of narrow ports while low-radix chips divide the bandwidth into a smaller number of wide ports. The number of intermediate routers in high-radix networks is greatly reduced and such networks enjoy a lower latency and reduced power consumption. As a result of the increase in the signaling rate and in the number of signals, the pin bandwidth of the chips used for switching has increased by approximately an order of magnitude every 5 years during the past two decades.

The topology of an interconnection network determines the network diameter and its bisection bandwidth, as well as the cost and the power consumption [133]. First, we introduce informally the *Clos* and the *flattened butterfly* topologies. 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

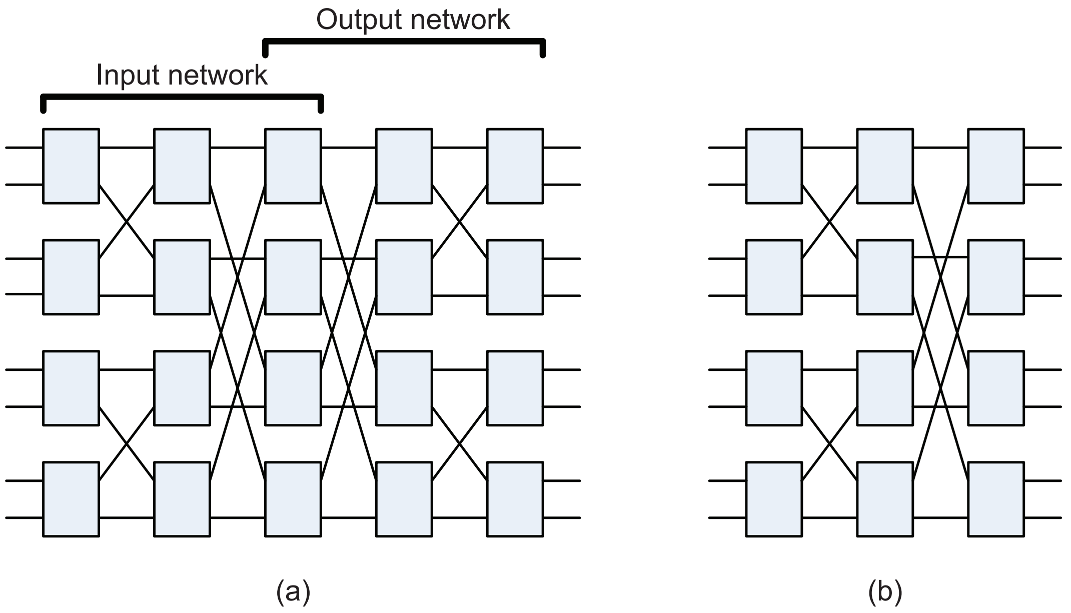


Figure 57: (a) A 5-stage Clos network with radix-2 routers and unidirectional channels; the network is equivalent to two back-to-back butterfly networks. (b) The corresponding folded-Clos network with bidirectional channels; the input and the output networks share switch modules.

is blocking, it cannot handle a conflict between two packets attempting to reach the same port at the same time.

A *Clos network* is a multistage nonblocking network with an odd number of stages, see Figure 57(a); the network consists of two butterfly networks and the last stage of the input is fused with the first stage of the output. 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, InfniBand, and Quadrics implement a fat-tree topology. Some folded Clos networks use low-radix routers, e.g., the Cray XD1 which 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 [217]. The Black Widow topology is used in Cray computers.

The flattened butterfly topology [132] is similar to the generalized hypercube that was proposed in the early 80s but the wiring complexity is reduced and this topology is able to exploit high-radix routers. To construct 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. 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 58(a) 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

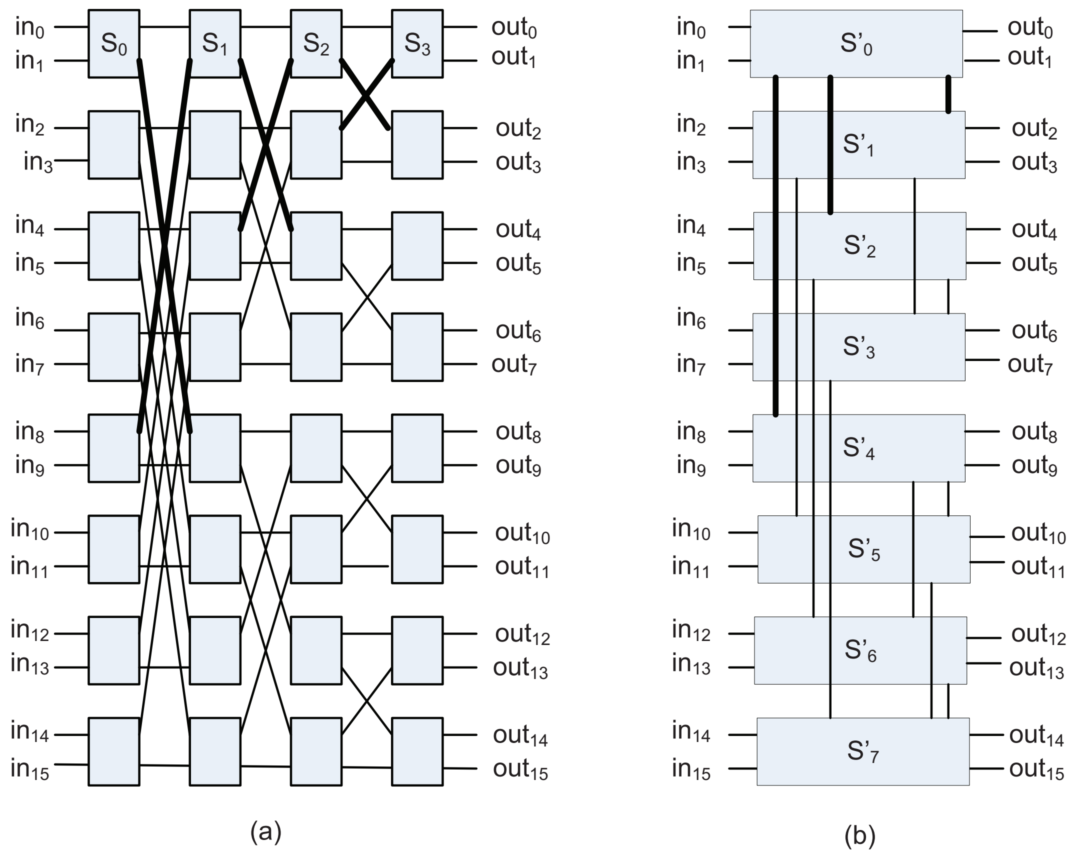


Figure 58: (a) A 2-ary 4-fly butterfly with unidirectional links. (b) The corresponding 2-ary 4-flat flattened butterfly obtained by combining the four switches S_0, S_1, S_2 and S_3 in the first row of the traditional butterfly into a single switch S'_0 and adding some additional connections between switches [132].

S'_0. The flattened butterfly adaptively senses congestion and overshoots only when it needs to. On adversarial traffic pattern, the flattened butterfly provides similar performance as the folded Clos, but provides over an order of magnitude increase in performance compared to the conventional butterfly.

The authors of [133] argue that the cost of the networks in Storage Area Networks and 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. 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%.

8.6 Storage area networks

A storage area network (SAN) is a specialized, high-speed network for data block transfers between computer systems and storage elements, Figure 59; it consists of a communication infrastructure and a management layer. The Fibre Channel (FC) is the dominant architecture of SANs.

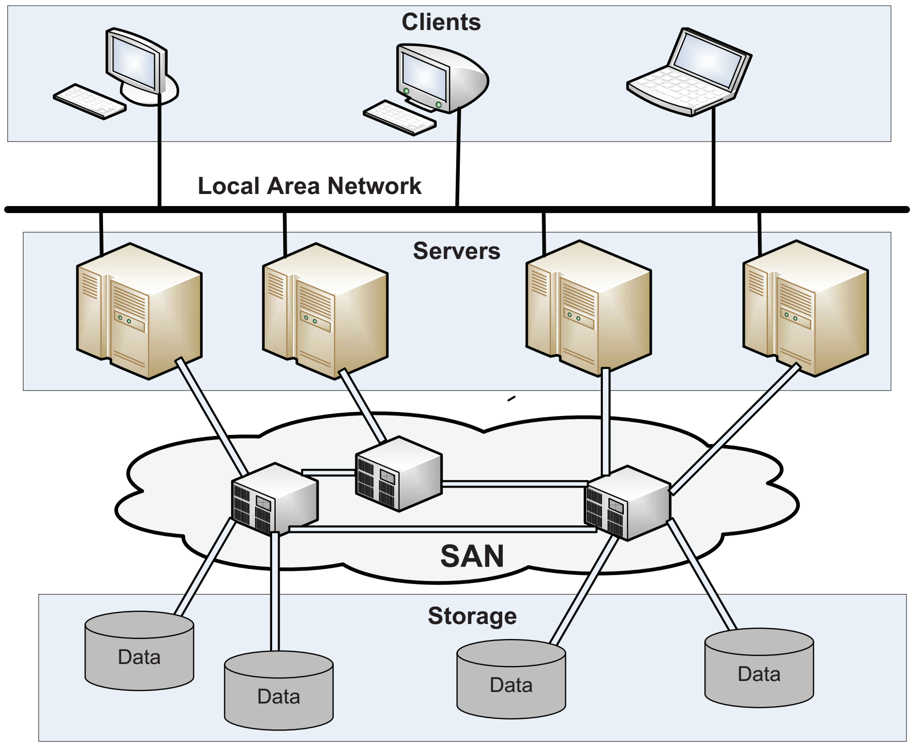


Figure 59: A storage area network interconnects servers to servers, servers to storage devices, and storage devices to storage devices. Typically it uses fiber optics and the FC protocol.

FC it is a layered protocol with:

- Three physical 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, it 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.

- 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 frames 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 nodes.

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, guarantees in-order delivery and acknowledgment of delivered frames, but the fabric is responsible for multiplexing frames of different VCs. Guaranteed Quality of Service (QoS), including bandwidth and latency; 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 F: similar with class 2 but used for the control and management of the fabric; a connectionless service with notification of non-delivery.

While each device connected to a LAN has a unique physical address also called MAC (Media Access Control) address, each FC device has a unique id called the WWN (World Wide Name), a 64 bit address. Each port in the switched fabric has its 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 logs into the switch on a port, the switch assigns the port address to that port and maintains the correlation between the port address and the WWN address of the device using the Name Server, a component of the fabric operating system, which runs inside the switch.

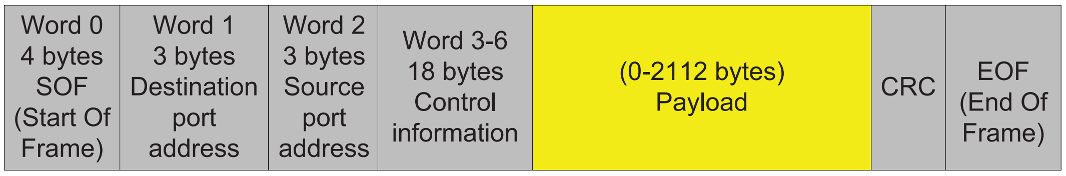


Figure 60: The format of an FC frame; the payload can be at most 2112 bytes, larger data units are carried by multiple frames.

The format of an FC frame is shown in Figure 60. 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 tunnelling[59]. 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.

8.7 Content delivery networks

Content Delivery Networks (CDNs) offer fast and reliable content delivery and reduce the 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.

CDS 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 crouds[60] 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. 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 life or on-demand streaming media. Static content refers to media that can be maintained using traditional caching technologies becuase 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

[59]Tunnelling 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. Tunnelling allows a protocol that a firewall would normally block 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. and most often with applications that lack native support for communication in such conditions of restricted connectivity. Restricted connectivity is a commonly used method to lock down a network to secure it against internal and external threats.

[60]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 increases dramatically.

of audio and/or video streams, movie files and music clips provided to the end-users content encoded and then stored on media servers. Virtually all CDN providers support static content delivery, while life or on-demand streaming media is considerably more challenging.

The first CDN was setup by *Akamai*, a company evolved from an MIT project to optimize the network traffic; since its inception *Akamai* has placed some $20,000$ servers in $1,000$ networks in 71 countries and has some 85% of the market [198]. There are several other active commercial CDNs including *EdgeStream*, which provides video streaming and *Limelight Networks*, which provides 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.

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. There are two strategies for CDN organization; in the so called *overlay* when 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 network core elements redirect content requests to local caches or redirect data centers incoming traffic to servers optimized for specific content type access. Some CDNs including *Akamai* use both strategies.

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 [198]. For example, caches exchange ICP queries and replies to locate the best sites to retrieve an object; HTCP is used to discover HTTP caches, cached data, managing sets of HTTP caches and monitoring cache activity.

Important design and policy decisions for a CDN are: (a) the placement of the edge servers, (b) the content selection and delivery, (c) the content management, and (d) the 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 replicated 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 clients 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 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.

8.8 Overlay networks for cloud computing

An overlay network, or a virtual network, is a network built on top of a physical network; the nodes of an overlay network are connected by virtual links which could traverse multiple physical links. Overlay networks are widely used in many distributed systems such as peer-to-peer systems, content-delivery systems, and client-server systems; in all these cases the distributed systems communicate through the Internet.

An overlay network could support quality of service guarantees for data streaming applications through improved routing in the Internet. It could also support routing of messages to destinations which which are not specified by an IP address; in this case distributed hash tables can be used to route messages based on their logical address. For example, Akamai is a company which manages an overlay network to provide a reliable and efficient content delivery; it mirroring 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; the Akamai server is automatically picked depending on the type of content and the user's network location.

Virtualization is a central concept in cloud computing; we have discussed extensively the virtualization of processors and it makes sense to consider also the virtualization of the cloud interconnect. Indeed, communication is a critical function of a cloud and overlay networks can be used to support a more efficient resource management. In this section we discuss several possible candidates as virtual cloud interconnects and algorithms to construct such networks. Such networks are modelled as graphs and we start out discussion with a few concepts from graph theory.

The topology of a network used to model the interactions in complex biological, social, economic and computing systems is described by means of graphs where vertices represent the entities and the edges represent their interactions. The number of edges incident upon a vertex is called the *degree of the vertex*.

Several types of graphs have been investigated starting with the Erdös-Rény model [43, 82] where the number of vertices is fixed and the edges connecting vertices are created randomly; this model produces a homogeneous network with an exponential tail, and connectivity follows a Poisson distribution peaked at the the average degree $\bar{k}$ and decaying exponentially for $k >> \bar{k}$. An evolving network, where the number of vertices increases linearly and a newly introduced vertex is connected to m existing vertices according to a preferential attachment rule, is described by Barabási and Albert in [10, 11, 12, 35].

Regular graphs where a fraction of edges are rewired with a probability p have been proposed by Watts and Strogatz and called small-worlds networks [249]. Networks whose degree distribution follows a power law, $p(k) \sim k^{-\gamma}$, are called scale-free networks. The four models are sometimes referred to as ER (Erdös-Rény), BA (Barabási - Albert), WS (Watts-Strogatz), and SF (Scale-free) models, respectively [97].

Small worlds networks. Traditionally, the connection topology of a network was assumed to be either completely regular, or completely random. Regular graphs are highly clustered and have large characteristic path length, while random graphs exhibit low clustering and have small characteristic path length.

The *characteristic path length, L*, is the number of edges in the shortest path between two vertices averaged over all pairs of vertices. The *clustering coefficient C* is defined as follows: if vertex a has m_a neighbors, then a fully connected network of its neighbors could have at most $E_a = m_a(m_a - 1)/2$ edges. Call C_a the fraction of E_a of edges that actually exist; C is the average of C_a over all vertices. Clearly, C measures the degree of clusterings of the network.

In 1998, D. Watts and S. H. Strogatz studied the graphs combining the two desirable features, high clustering and small path length, and introduced the Watts-Strogatz graphs [249]. They proposed the following procedure to interpolate between regular and random graphs: starting from a ring lattice with n vertices and m edges per node rewire each edge at random with probability $0 \leq p \leq 1$; when $p = 0$ the graph is regular and when $p = 1$ the graph is random.

When $0 < p < 1$ the structural properties of the graph are quantified by:

1. The characteristic path length, $L(p)$;

2. the clustering coefficient, $C(p)$.

If the condition

$$n >> m >> \ln(n) >> 1 \tag{74}$$

is satisfied then

$$p \to 0 \quad \Rightarrow \quad L_{regular} \approx n/2m >> 1 \;\text{ and }\; C_{regular} \approx 3/4, \tag{75}$$

while

$$p \to 1 \quad \Rightarrow \quad L_{random} \approx \ln(n)/\ln(m) \;\text{ and }\; C_{random} \approx m/n << 1. \tag{76}$$

The small-worlds networks have many vertices with sparse connections, but are not in danger of getting disconnected; moreover, there is a broad range of the probability p such that $L(p) \approx L_{random}$ and, at the same time, $C(p) >> C_{random}$. The significant drop of $L(p)$ is caused by the introduction of a few shortcuts which connect vertices that otherwise would be much further apart. For small p, the addition of a shortcut has a highly nonlinear effect; it affects not only the distance between the pair of vertices it connects, but also the distance between their neighbors. If the shortcut replaces an edge in a clustered neighborhood, $C(p)$ remains practically unchanged, as it is a linear function of m.

Scale-free networks. The degree distribution of scale-free networks follows a power law; we only consider the discrete case when the probability density function is $p(k) = af(k)$ with $f(k) = k^{-\gamma}$ and the constant a is $a = 1/\zeta(\gamma, k_{min})$ thus,

$$p(k) = \frac{1}{\zeta(\gamma, k_{min})} k^{-\gamma}. \tag{77}$$

In this expression k_{min} is the smallest degree of any vertex, and for the applications we discuss in this paper $k_{min} = 1$; ζ is the Hurvitz zeta function[61]

$$\zeta(\gamma, k_{min}) = \sum_{n=0}^{\infty} \frac{1}{(k_{min} + n)^{\gamma}} = \sum_{n=0}^{\infty} \frac{1}{(1 + n)^{\gamma}}. \tag{78}$$

Many physical and social systems are interconnected by a scale-free network. Indeed, empirical data available for power grids, the Web, the citation of scientific papers, or social networks confirm this trend: the power grid of the Western US has some $5,000$ vertices representing power generating stations and in this case $\gamma \approx 4$; for the World Wide Web the probability that m pages point to one page is $p(k) \approx k^{-2.1}$ [36]; recent studies indicate that $\gamma \approx 3$ for the citation of scientific papers; the collaborative graph of movie actors where links are present if two actors were ever cast in the same movie follows the power low with $\gamma \approx 2.3$. The larger the network, the closer a power law with $\gamma \approx 3$ approximates the distribution [35].

A scale-free network is *non-homogeneous*; the majority of the vertices have a low degree and only a few vertices are connected to a large number of edges, Figure 61. On the other hand, an exponential network is homogeneous as most of the vertices have the same degree.

[61]The Hurvitz zeta function $\zeta(s, q) = \sum_{n=0}^{\infty} \frac{1}{(q+n)^s}$ for $s, q \in \mathbb{C}$ and $\mathfrak{Re}(s) > 1$ and $\mathfrak{Re}(q) > 0$. The Riemann zeta function is $\zeta(s, 1)$.

The average distance d between the N vertices, also referred to as the diameter of the scale-free network, scales as $\ln N$; in fact it has been shown that when $k_{min} > 2$ a lower bound on the diameter of a network with $2 < \gamma < 3$ is $\ln \ln N$ [62]. A number of studies have shown that scale-free networks have remarkable properties such as: robustness against random failures [36], favorable scaling [10, 11], resilience to congestion [97], tolerance to attacks [240], small diameter [62] and small average path length [35]. The moments of a power law distribution play an important role in the behavior of a network. It has been shown that the giant connected component (GCC) of networks with a finite average vertex degree and divergent variance can only be destroyed if all vertices are removed; thus, such networks are highly resilient against faulty constituents [170]. These properties make scale-free networks very attractive for interconnection networks in many applications including social systems [173], peer-to-peer systems [220], sensor networks [159], and cloud computing.

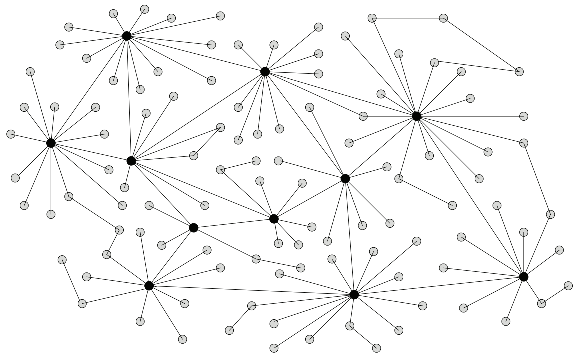


Figure 61: A scale-free network is non-homogeneous; the majority of the vertices have a low degree and only a few vertices are connected to a large number of edges; the majority of the vertices are directly connected with the vertices with the highest degree.

As an example, consider the case $\gamma = 2.5$ and the minimum vertex degree, $x_{min} = 1$; we first determine the value of the zeta function $\zeta(\gamma, x_{min})$ and approximate $\zeta(2.5, 1) = 1.341$ thus, the distribution function is $p(k) = k^{-2.5}/1.341 = 0.745 \times (1/k^{2.5})$, where k is the degree of each vertex. The probability of vertices with degree $k > 10$ is $\text{Prob}(k > 10) = 1 - \text{Prob}(k \leq 10) = 0.015$. This means that at most 1.5% of the total number of vertices will have more than 10 edges connected to them; we also see that 92.5% of the vertices have degree $1, 2$ or 3. Table 16 shows the number of vertices of degrees one to ten for a very large network, $N = 10^8$.

Another important property is that the majority of the vertices of a scale-free network are directly connected with the vertices with the highest degree; for example, in a network with $N = 130$ vertices and $m = 215$ edges 60% of the nodes are directly connected with the five vertices with the highest degree, while in an exponential network fewer than half, 27%, have this property [11].

Thus, the nodes of a scale-free network with a degree larger than a given threshold, e.g., $k = 4$ in our example, could assume the role of control nodes and the remaining 92.5% of the nodes could be servers and this partition is autonomic; moreover, most of the server nodes are at distance one two, or three from a control node which could gather more accurate state information from these nodes and with minimal communication overhead.

Table 16: A power-law distribution with degree $\gamma = 2.5$; the probability, $p(k)$, and N_k, the number of vertices with degree k, when the total number of vertices is $N = 10^8$.

k	$p(k)$	N_k	k	$p(k)$	N_k
1	0.745	74.5×10^6	6	0.009	0.9×10^6
2	0.131	13.1×10^6	7	0.006	0.6×10^6
3	0.049	4.9×10^6	8	0.004	0.4×10^6
4	0.023	2.3×10^6	9	0.003	0.3×10^6
5	0.013	1.3×10^6	10	0.002	0.2×10^6

We conclude that a scale-free network is an ideal interconnect for a cloud. It is not practical to construct a scale-free physical interconnect for a cloud, but we can generate instead a virtual interconnect with the desired topology; we pay a small penalty in terms of latency and possibly bandwidth when the nodes communicate through the virtual interconnect, but this penalty is likely to become smaller and smaller as new networking technologies for cloud computing emerge.

An algorithm for the construction of graphs with power law degree distribution. Consider an Erdös-Renyi (ER) graph G_{ER} with N vertices; vertex i has a unique label from a compact set $i \in \{1, N\}$. We wish to rewire this graph and produce a new graph G_{SF} where the degrees of the vertices follow a power-law distribution. The procedure we discuss consists of the following steps [144]:

1. We assign to each node i a probability
$$p_i = \frac{i^{-\alpha}}{\sum_{j=1}^{N} j^{-\alpha}} = \frac{i^{-\alpha}}{\zeta_N(\alpha)} \quad \text{with} \quad 0 < \alpha < 1 \quad \text{and} \quad \zeta_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha}. \tag{79}$$

2. We select a pair of vertices i and j and create an edge between them with probability
$$p_{ij} = p_i p_j = \frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)} \tag{80}$$
and repeat this process n times.

Then the probability that a given pair of vertices i and j is not connected by an edge h_{ij} is
$$p_{ij}^{NC} = (1 - p_{ij})^n \approx e^{-2np_{ij}} \tag{81}$$
and the probability that they are connected is
$$p_{ij}^{C} = (1 - p_{ij}^{NC}) = 1 - e^{-2np_{ij}}. \tag{82}$$
Call k_i the degree of vertex i; then the moment generating function of k_i is
$$g_i(t) = \prod_{j \neq i} [p_{ij}^{NC} + t p_{ij}^{C}]. \tag{83}$$

The average degree of vertex i is

$$\bar{k}_i = t\frac{d}{dt}g_i(t)|_{t=1} = \sum_{j\neq i} p_{ij}^C. \tag{84}$$

Thus,

$$\bar{k}_i = \sum_{j\neq i}(1-e^{-2np_{ij}}) = \sum_{j\neq i}\left(1-e^{-2n\frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)}}\right) \approx \sum_{j\neq i} 2n\frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)} = \frac{2n}{\zeta_N^2(\alpha)}\sum_{j\neq i}(ij)^{-\alpha}. \tag{85}$$

This expression can be transformed as

$$\bar{k}_i = \frac{2n}{\zeta_N^2(\alpha)}\sum_{j\neq i}(ij)^{-\alpha} = \frac{2ni^{-\alpha}\sum_{j\neq i} j^{-\alpha}}{\zeta_N^2(\alpha)} = \frac{2ni^{-\alpha}\left(\zeta_N(\alpha) - i^{-\alpha}\right)}{\zeta_N^2(\alpha)} \tag{86}$$

The moment generating function of k_i can be written as

$$g_i(t) = \prod_{j\neq i}[p_{ij}^{NC} + tp_{ij}^C] = e^{(1-t)\bar{k}_i} = \prod_{j\neq i} e^{-(1-t)p_{ij}^C} \approx \prod_{j\neq i}[1-(1-t)p_{ij}^C = e^{(1-t)\sum_{j\neq i}p_{ij}^C} = e^{(1-t)\bar{k}_i} \tag{87}$$

Then we conclude that the probability that $k_i = k$ is given by

$$p_{d,i}(k) = \frac{1}{k!}\frac{d^k}{dt^k}g_i(t)|_{t=0} \approx \frac{\bar{k}_i}{k!}e^{-\bar{k}_i}. \tag{88}$$

When $N \to \infty$ then $\zeta_N(\alpha) = \sum_{i=1}^N i^{-\alpha}$ converges to the Riemann zeta function $\zeta(\alpha)$ for $\alpha > 1$ and diverges as $\frac{N^{1-\alpha}}{1-\alpha}$ if $0 < \alpha < 1$. For $0 < \alpha < 1$ equation (79) becomes

$$p_i = \frac{i^{-\alpha}}{\zeta_N(\alpha)} = \frac{1-\alpha}{N^{1-\alpha}}i^{-\alpha}. \tag{89}$$

When $N \to \infty$, $0 < \alpha < 1$, and the average degree of the vertices is $2m$, then the degree of vertex i is

$$k = p_i \times mN = 2mN\frac{1-\alpha}{N^{1-\alpha}}i^{-\alpha} = 2m(1-\alpha)\left(\frac{i}{N}\right)^{-\alpha} \tag{90}$$

Indeed, the total number of edges in graph is mN and the graph has a power law distribution. Then

$$i = N\left(\frac{k}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} \tag{91}$$

From this expression we see that there is a one-to-many correspondence between the unique label of the node i and the degree k; this reflects the fact that multiple vertices may have the same degree k. The number of vertices of degree k is

$$n(k) = N\left(\frac{k}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} - N\left(\frac{k-1}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} = N\left(\frac{k-1}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}}\left(\left(1+\frac{1}{k}\right)^{-\frac{1}{\alpha}} - 1\right). \tag{92}$$

We denote $\gamma = 1 + \frac{1}{\alpha}$ and observe that

$$\left(1+\frac{1}{k}\right)^{-\frac{1}{\alpha}} = 1 + \left(-\frac{1}{\alpha}\right)\left(\frac{1}{k}\right)^{-\frac{1}{\alpha}} + \frac{1}{2}\left(-\frac{1}{\alpha}\right)\left(-\frac{1}{\alpha}-1\right)\left(\frac{1}{k}\right)^{-\frac{1}{\alpha}-1} + ... \tag{93}$$

We see that

$$n(k) = N\left(\frac{(k-1)(\gamma-1)}{2m(\gamma-2)}\right)^{-\gamma+1}\left((1-\gamma)\left(\frac{1}{k}\right)^{-\gamma+1} - \frac{\gamma(1-\gamma)}{2}\left(\frac{1}{k}\right)^{-\gamma} + ...\right) \tag{94}$$

We conclude that to reach the value predicted by the theoretical model for the number of vertices of degree k, the number of iterations is a function of N, of the average degree $2m$, and of γ, the degree of the power law. Next we discuss an algorithm for construction of a scale-free networks using biased random walks.

Biased random walks. A strategy used successfully to locate systems satisfying a set of conditions in applications such as peer-to-peer systems is based on biased random walks. Random walks are reported to be more efficient in searching for nodes with desirable properties than other methods such as flooding [93].

Unfortunately, the application of random walks in a large network with an irregular topology is unfeasible because a central authority could not maintain accurate information about a dynamic set of members. A solution is to exploit the fact that sampling with a given probability distribution can be simulated by a discrete-time Markov chain. Indeed, consider an irreducible Markov chain with states $(i, j) \in \{0, 1, \ldots, S\}$ and let $P = [p_{ij}]$ denote its probability transition matrix where

$$p_{ij} = \text{Prob}[X(t+1) = j \mid X(t) = i], \tag{95}$$

with $X(t)$ the state at time t. Let $\pi = (\pi_0, \pi_1, \ldots, \pi_S)$ be a probability distribution with nonzero probability for every state, $\pi_i > 0,\ 0 \leq i \leq S$. The transition matrix P is chosen so that π is its unique stationary distribution thus, the reversibility condition $\pi = \pi P$ holds. When $g(.)$ is a function defined on the states of the Markov chain and we wish to estimate

$$E = \sum_{i=0}^{S} g(i)\pi_i \tag{96}$$

we can simulate the Markov chain at times $t = 1, 2, \ldots, N$ and the quantity

$$\hat{E} = \sum_{i=1}^{N} \frac{f(X(t))}{N} \tag{97}$$

is a good estimate of E, more precisely $\hat{E} \mapsto E$ when $N \mapsto \infty$. Hastings [108] generalizes the sampling method of Metropolis [167] to construct the transition matrix given the distribution

π. He starts by imposing the reversibility condition $\pi_i p_{ij} = \pi_j p_{ji}$. If $Q = [q_{ij}]$ is the transition matrix of an arbitrary Markov chain on the states $\{0, 1, \ldots, S\}$, it is assumed that

$$p_{ij} = q_{ij}\alpha_{ij} \ \text{ if } \ i \neq j \quad \text{ and } \quad p_{ii} = 1 - \sum_{j \neq i} p_{ij}. \tag{98}$$

Two versions of sampling are discussed in [108], that of Metropolis and one proposed by Baker[33]; the quantities α_{ij} are, respectively,

$$\alpha_{ij}^{M} = \begin{cases} 1 & \text{if } \frac{\pi_j}{\pi_i} \geq 1 \\ \frac{\pi_j}{\pi_i} & \text{if } \frac{\pi_j}{\pi_i} < 1 \end{cases} \qquad \text{and} \qquad \alpha_{ij}^{B} = \frac{\pi_j}{\pi_i + \pi_j} \tag{99}$$

For example, consider a Poisson distribution $\pi_i = \lambda^i e^{-\lambda}/i!$; we choose $q_{ij} = 1/2$ if $j = i-1, i \neq 0$ or $j = i+1, i \neq 0$ and $q_{00} = q_{01} = 1/2$. Then using Baker's approach we have

$$p_{ij} = \begin{cases} \lambda/(\lambda + i + 1) & \text{if } \ j = i+1, i \neq 0 \\ i/(i + \lambda) & \text{if } \ j = i-1, i \neq 0 \end{cases} \tag{100}$$

and $p_{00} = 1/2$ and $p_{01} = \lambda e^{-\lambda}/(1 + \lambda e^{-\lambda})$.

The algorithm to construct scale-free overlay topologies with an adjustable exponent presented in [221] adopts the equilibrium model discussed in [97]. The algorithm is based on random walks in a connected overlay network $G(V, E)$ viewed as a Markov chain with state space V and a stationary distribution with a random walk bias configured according to a Metropolis-Hastings chain [108]. Recall that in this case they assign a weight $p_i = i^{-\alpha}, \ 1 \leq i \leq N, \alpha \in [0, 1)$ to each vertex and add an edge between two vertices a and b with probability $p_a/\sum_{i=1}^{N} p_i \times p_b/\sum_{i=1}^{N} p_i$ if none exists; they repeat the process until mN edges are created and the mean degree is $2m$. Then the degree distribution is

$$p(k) \sim k^{-\gamma}, \quad \text{ with } \quad \gamma = (1 + \alpha)/\alpha. \tag{101}$$

The elements of the transition matrix $P = [p_{ij}]$ are

$$p_{ij} = \begin{cases} \frac{1}{k_i} \min\left\{ \left(\frac{1}{j}\right)^{\frac{1}{\gamma - 1}} \frac{k_i}{k_j}, 1 \right\} & (i, j) \in E \\ 1 - \frac{1}{k_i} \sum_{(l,i) \in E} c_{il} & i = j \\ 0 & (i, j) \notin E \end{cases} \tag{102}$$

with k_i the degree of vertex i. An upper bound for the number of random walk steps can be determined from a lower bound for the second smallest eigenvalue of the transition matrix, a non-trivial problem.

A distributed rewiring scheme to construct scale-free overlay topologies with an adjustable exponent is presented in [221]. An alternative method of creating the scale-free overlay network could be based on the gossip-based peer-sampling discussed in [123]. The distributed algorithm for the construction of a scale-free network in [221] is based on the method to construct a random graph with a power law distribution sketched in [97] and [144].

Estimation of the degree of a power-law network. The question we address next is how to estimate the degree distribution of any scheme for the construction of a power-law network. The estimation of the degree distribution from empirical data is analyzed in [61]; according to this study a good approximation for γ for a discrete power law distribution for a network with P vertices and $k_{min} = 1$ is

$$\hat{\gamma} \approx 1 + P \left[\sum_{i=1}^{P} \ln \frac{k_i}{k_{min} - 1/2} \right]^{-1} = 1 + \frac{P}{\sum_{i=1}^{P} 2k_i} \tag{103}$$

Several measures exist for the similarity/dissimilarity of two probability density functions of discrete random variables including the trace distance, fidelity, mutual information, and relative entropy [64, 136]. The *trace distance* (also called Kolmogorov or L1 distance) of two probability density functions, $p_X(x)$ and $p_Y(y)$, and their *fidelity* are defined as

$$D\left(p_X(x), p_Y(x)\right) = \frac{1}{2} \sum_x \mid p_X(x) - p_Y(x) \mid \tag{104}$$

and

$$F\left(p_X(x), p_Y(x)\right) = \sum_x \sqrt{p_X(x) p_Y(x)}. \tag{105}$$

The trace distance is a metric: it is easy to prove non-negativity, symmetry, the identity of indiscernibles, and the triangle inequality. On the other hand, the fidelity is not a metric, as it fails to satisfy the identity of indiscernibles,

$$F\left(p_X(x), p_X(x)\right) = \sum_x \sqrt{p_X(x) p_X(x)} = 1 \neq 0. \tag{106}$$

Determining either the $L1$ distance between the distribution calculated based on equation (77) and the one produced by the algorithm discussed earlier requires information about the degree of all vertices. From Table 16 we see that the degree one vertices represent a very large fraction of the vertices of a power-law network and may provide a reasonable approximation of the actual degree distribution.

8.9 Further readings

An extensive treatment of Storage Area Networks can be found in [116]

9 Storage Systems

9.1 GPFS, IBM's General Parallel File System

9.2 Calypso, a distributed file system

9.3 Lustre, a massively parallel distributed file system

9.4 GFS, the Google File System

9.5 A storage abstraction layer for portable cloud applications

9.6 Megastore

9.7 Performance of large-scale distributed storage systems

10 Cloud Security

10.1 A taxonomy for attacks on cloud services

10.2 Security checklist for cloud models

10.3 Privacy impact assessment for cloud computing

10.4 Risk controls in cloud computing

10.5 Security services life cycle management

10.6 Secure collaboration in cloud computing

10.7 Secure VM execution under an un-trusted OS

10.8 Anonymous access control and accountability for cloud computing

10.9 Inter-cloud security

10.10 Social impact of privacy in cloud computing

****

10.11 Further readings

A 2010 paper, [101] presents a taxonomy of attaks on computer clouds and [70] covers the management of security services lifecycle. A 2009 paper by the cloud security alliance [68] analyzes the critical areas of security focus in cloud computing. Security issues vary depending on the cloud model as discussed in [188]. The privacy impact in cloud computing is the topic of [234]. A 2011 book [252] gives a comprehensive look at cloud security. Privacy and protection of personal data in the European Community is discussed in a document avalable at http://ec.europa.eu/justice/policies/privacy.

The paper [26] analyzes the inadequacies of current risk controls for the cloud. Intercloud security is the theme of [41]. Secure collaborations are discussed in [42]. The paper [147] presents an approach for secure VM execution under untrusted management OS. The social impact of privacy in cloud computing is analyzed in [84]. An anonymous access control scheme is presented in [124].

11 Complex Systems and Self-Organization

The very practical question of how to design, maintain, and use complex computing and communication systems cannot be answered without some understanding of the general characteristics of a complex system, the metrics that allow us to assess the complexity of a system, and some insight into how complex systems behave in nature. Some of these considerations lead to very abstract questions that have preoccupied the minds of humans for millennia. For example, Aristotle stated that "...the whole is something over and above its parts, and not just the sum of them all.." In "The Republic" Plato, introduces the concept of "level of knowledge" ranging from total ignorance to total knowledge. "True knowledge" exists only if a foundation of axioms or a priori knowledge exists [114] and this cannot be the case for complex systems.

While we have an intuitive notion of what complexity means, no rigorous definition allowing us to quantify and measure the complexity of a system is universally accepted. Certainly, the scale of a system, the dynamics of the system behavior, the unpredictability of the next state, the length of time a system has been in existence may affect its complexity, but as we shall see none of these elements by itself allows us to conclude that a system is complex or not.

11.1 Quantifying complexity.

Thermodynamic entropy, von Neumann entropy, and Shannon entropy are related to the number of states of a system, thus they reflect to some extent the system complexity [64]. A measure of complexity introduced by Crutchfield is the *relative predictive efficiency*, $e = E/C$ with E the excess entropy and C the statistical complexity [67]. The *excess entropy* measures the complexity of the stochastic process and can be regarded as the fraction of historical information about the process that allows us to predict the future behavior of the process. The *statistical complexity* reflects the size of the model of the system at a certain level of abstraction. The scale of organization considered by an external observer plays a critical role in assessing the relative predictive efficiency. For example, at the microscopic level the calculation of e for a volume of gas requires very complex molecular dynamics computations in order to accurately predict the excess entropy; both E and C are very high and the predictive efficiency is low. On the other hand, at the macroscopic level the relationship between the pressure P, the volume V, and the temperature T is very simple $PV = nRT$ with n the number of moles of gas and R the universal gas constant. In this case E maintains a high value, but now C is low and the predictive efficiency E/C is large.

We could use the complexity of a program that simulates the system as a measure of complexity of the system; this will reflect not only the number of states but also the transitions among states. This proposal has its own limitations, as we generally simulate approximate models of a system, rather than exact ones.

The proposal is consistent with the concept of *depth* defined as the number of computational steps needed to simulate a system's state [156]. Machta argues that the emergence of complexity requires a long history, but one needs a measure stricter than physical time to reflect this history [156]. The depth reflects not how long the system remains in equilibrium, but *how many steps are necessary to reach equilibrium following some efficient process.* The rate of change of the system state and the communication time do not reflect the complexity of a system. Indeed, two rotating structures involving very different physical processes, a hurricane and a spiral galaxy are at the limit of today's realistic computer simulation thus, of similar depth and, consequently, of similar complexity. Yet, galaxy formation occurs at a scale of millions of light years and is bounded by communication at the speed of light, while the time for hurricane formation is measured in days, the atmospheric disturbances propagate more slowly, but the scale of hurricane formation is only hundreds of kilometers.

Physical systems in equilibrium display their most complex behavior at *critical points* (in thermodynamics a critical point specifies the conditions of temperature and pressure, at which a phase boundary, e.g., between liquid and gas, ceases to exist). The time to reach equilibrium becomes very high at critical points, a phenomena called *critical slowing.* Wolpert and Macready [256] argue that *self-similarity* can be used to quantify complexity; the patterns exhibited by complex systems at different scales are very different, while the patterns exhibited by simple systems such as gases and crystals do not vary significantly from one scale to another.

11.2 Emergence and self-organization

The 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 [105].

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* [35, 36]. 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 [43] 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, $P(m) \approx m^{-\gamma}$, 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 $\gamma \approx 4$. For the World Wide Web the exponent is $\gamma \approx 2.1$; this means that the probability that m pages point to one page is $P(m) \approx m^{-2.1}$ [36]. 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 [35].

Table 17: Attributes associated with self-organization and complexity

Simple systems lacking self-organization	Complex systems exhibiting 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	Very different patterns at different scales
Do not require a long history	Require a long history
Simple emergence	Complex emergence
Limited scalability	Scale-free

11.3 Complexity of computing and communication systems

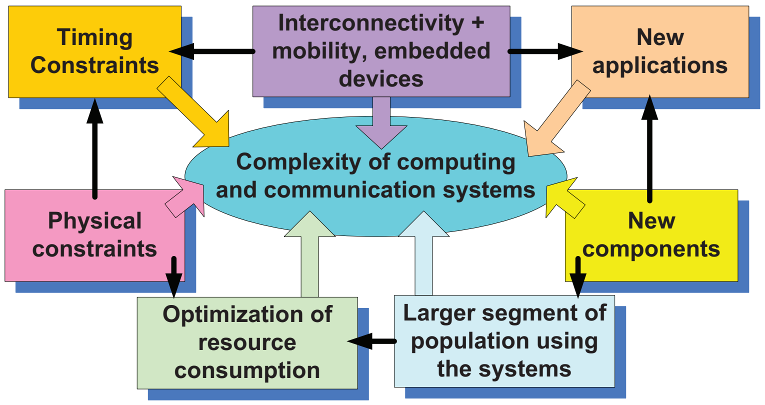

Timing Constraints
Interconnectivity + mobility, embedded devices
New applications
Complexity of computing and communication systems
Physical constraints
New components
Optimization of resource consumption
Larger segment of population using the systems

Figure 62: Factors contributing to the complexity of modern computing and communication systems.

Our limited understanding of system complexity and the highly abstract concepts developed in the context of natural sciences do not lend themselves to straightforward development of design principles for modern computing and communication systems. Nevertheless, the generic attributes of complex systems exhibiting self-organization, summarized in Table 11.2, allow us to identify some of the specific factors affecting the complexity of these systems [2]:

- The physical nature and the physical properties of computing and communication systems must be well understood and the system design must obey the laws of physics.
- The behavior of the systems is controlled by phenomena that occur at multiple scales/levels. As levels form or disintegrate, phase transitions and/or chaotic phenomena may occur.
- Systems have no predefined bottom level; it is never known when a lower level phenomena will affect how the system works.

- Abstractions of the system useful for a particular aspect of the design may have unwanted consequences at another level.
- Systems are entangled with their environment. A system depends on its environment for its persistence, therefore it is far from equilibrium. The environment is man made and the selection required by the evolution can either result in innovation, generate unintended consequences, or both.
- Systems are expected to function simultaneously as individual systems and as groups of systems (systems of systems).
- Typically, computing and communication systems are both deployed and under development at the same time.

Some of the factors contributing to the complexity of modern computing and communication systems are summarized in Figure 62

12 Cloud Application Development

In the previous chapters our discussion was focused on research issues in cloud computing; now we examine the clouds from the perspective of an application developer. This chapter presents a few recipes useful to assemble the cloud computing environment on a local system and to use basic cloud functions. Then we discuss several applications; we present first application related to mobile wireless devices. Mobile wireless applications are likely to benefit from cloud computing, as discussed in Chapter 5; 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 in the near future a major concern; thus, it is seems reasonable to delegate compute- and data-intensive tasks to an external entity, e.g., a cloud.

It is fair to assume that the population of application developers and cloud users is and will continue to be very diverse. Some have developed and run parallel applications for many years on clusters or other types of systems and expect a smooth transitions to the cloud; others, are less experienced and view cloud computing as an opportunity to develop a new business with minimum investment in computing equipment and human resources and expect a smooth learning curve.

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 to port an existing application to the cloud? How easy is to develop a new cloud application?

The answers to these questions are different for the three cloud computing paradigms; the level of difficulty increases as we move towards the base of the cloud service pyramid in Figure 63. Recall that *SaaS* applications are designed for the end-users and are accessed over the Web and 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 storage management software. The infrastructure as a service poses the most challenges thus, we restrict our discussion to the *IaaS* cloud computing paradigm and we concentrate on the most popular services offered at this time, the Amazon Web Services (AWS).

To access AWS one must first create an account at *http://aws.amazon.com/*; once the account is created the Amazon Management Console (AWC) allows the user to select one of the service, e.g., *EC2*, *S3*, and so on. Several operating systems are supported by *AWS* including: *Amazon Linux, Cent OS, Debian, Fedora, Open Solaris, Open Suse, Red Hat, Ubuntu, Windows*, and *SUSE Linux*

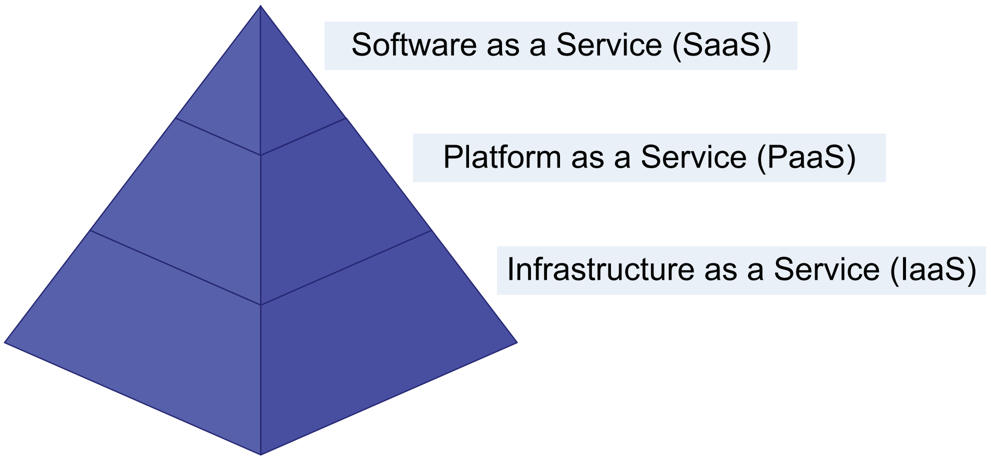
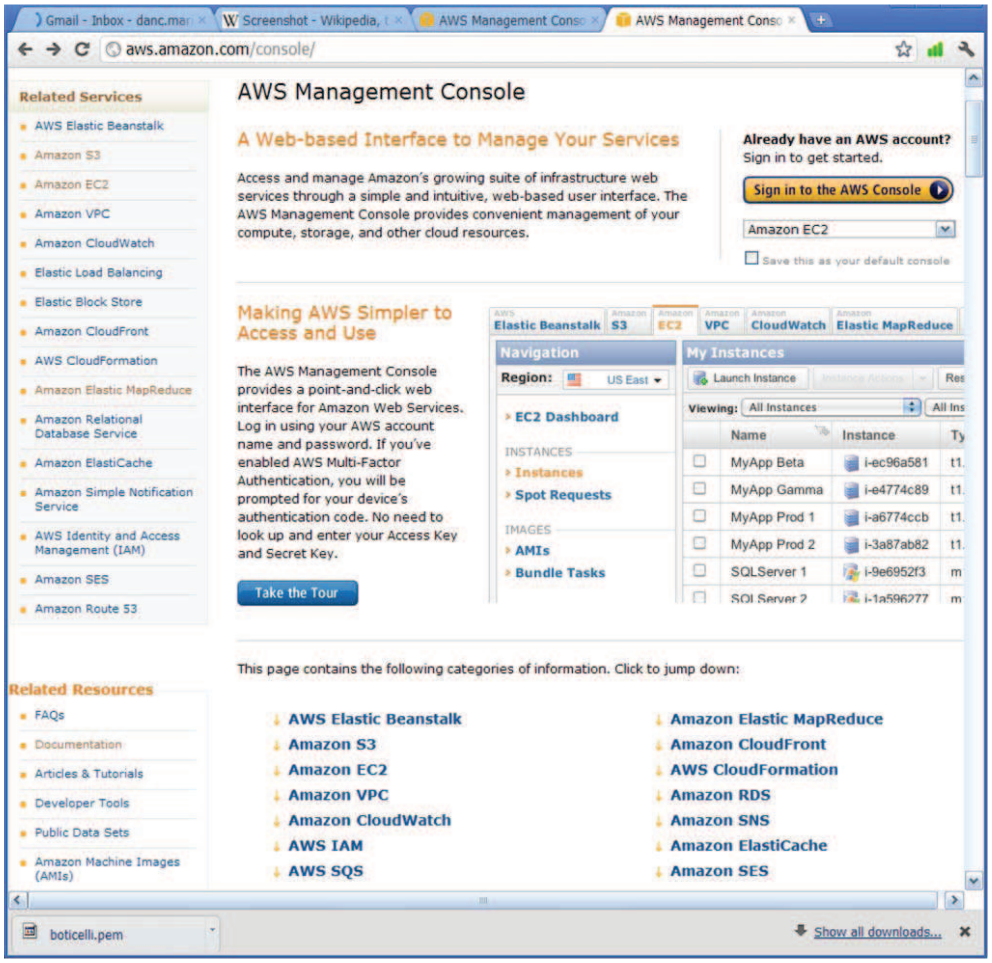

Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a Service (IaaS)

Figure 63: 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.



AWS Management Console

A Web-based Interface to Manage Your Services

Access and manage Amazon's growing suite of infrastructure web services through a simple and intuitive, web-based user interface. The AWS Management Console provides convenient management of your compute, storage, and other cloud resources.

Already have an AWS account? Sign in to get started.

Making AWS Simpler to Access and Use

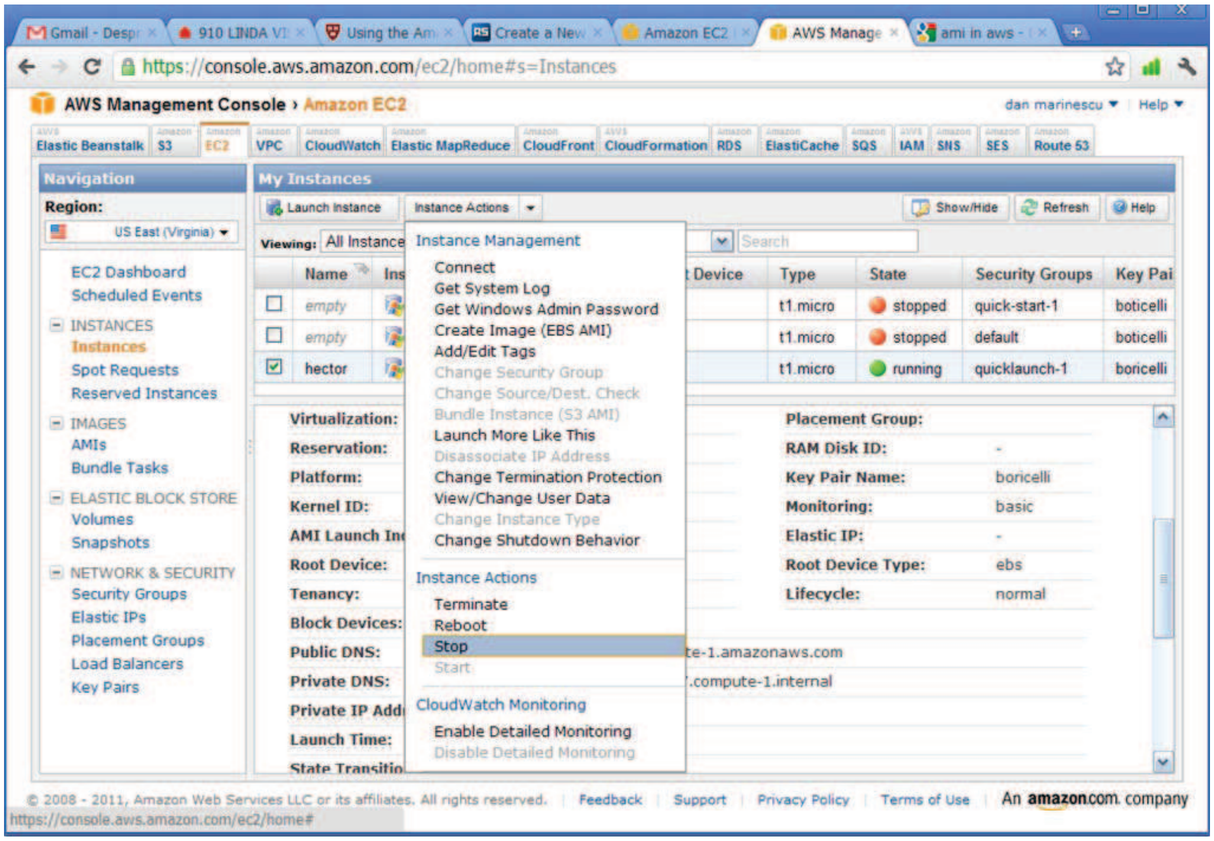
The AWS Management Console provides a point-and-click web interface for Amazon Web Services. Log in using your AWS account name and password. If you've enabled AWS Multi-Factor Authentication, you will be prompted for your device's authentication code. No need to look up and enter your Access Key and Secret Key.

Name	Instance	Ty
MyApp Beta	i-ec96a581	t1.
MyApp Gamma	i-e4774c89	t1.
MyApp Prod 1	i-a6774ccb	t1.
MyApp Prod 2	i-3a87ab82	t1.
SQLServer 1	i-9e6952f3	m
SQLServer 2	i-1a596277	m

This page contains the following categories of information. Click to jump down:

Figure 64: Sign in screen to Amazon's *EC2* using the AWS Management Console.

The next step is to create an AMI (Amazon Machine Image)[62] on one of the platforms supported by AWS and start an instance using the *RunInstance* API; if the application needs more than 20 instances then a special form must be filled out. The local instance store persists only for the duration of an instance; the data will persist if an instance is started using the Amazon EBS (Elastic Block Storage) and then the instance can be restated at a later time.



	Name	Ins	Device	Type	State	Security Groups	Key Pai
☐	empty			t1.micro	stopped	quick-start-1	boticelli
☐	empty			t1.micro	stopped	default	boticelli
☑	hector			t1.micro	running	quicklaunch-1	boricelli

Virtualization:
Reservation:
Platform:
Kernel ID:
AMI Launch Inc
Root Device:
Tenancy:
Block Devices:
Public DNS: te-1.amazonaws.com
Private DNS: .compute-1.internal
Private IP Addi
Launch Time:
State Transitio

Placement Group:
RAM Disk ID: -
Key Pair Name: boricelli
Monitoring: basic
Elastic IP: -
Root Device Type: ebs
Lifecycle: normal

Figure 65: 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.

Once an instance was created the user can perform several actions; for example, one can connect to the instance, launch more instances identical to the current one, or create an EBS AMI; one can also terminate, reboot, or stop the instance, see Figure 65. 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 12.2, while the EBS panel allows the specification of volumes and the creation of snapshots.

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

[62]An AMI is a unit of deployment, it is an environment including all information necessary to set up and boot an instance.

was asked to develop a cloud application for a problem of interest in her 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.

12.1 Connecting clients with 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*, rather stream 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 worked up to the transport layer and maintained the state of all connections passing through it. Unfortunately, this traffic filtering solution opened the possibility of *denial of service attacks*; a denial of service attack targets a widely used network service and forces the operating system of the host to fill the connection tables with illegitimate entries thus, preventing 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* (IDS).

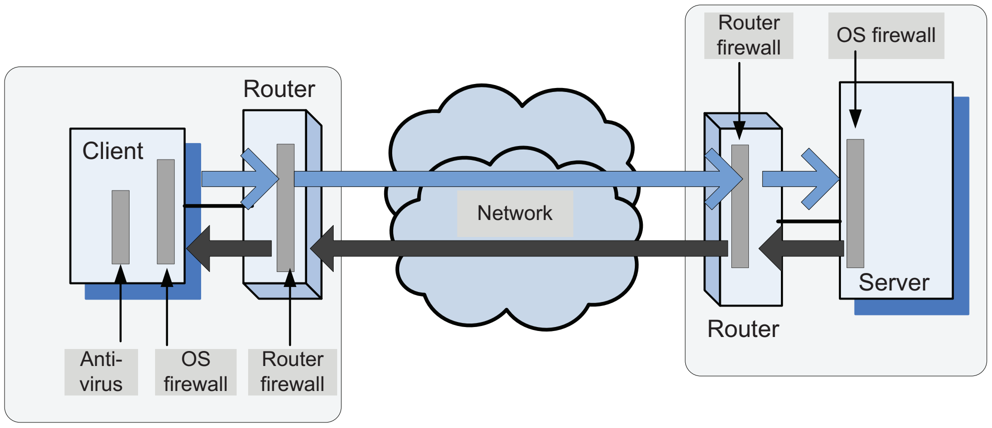


Figure 66: Firewalls screen incoming and sometimes outgoing traffic. The first obstacle encountered by the inbound or outbound traffic is a router firewall, the next one is the firewall provided by the host operating system; sometimes, the antivirus software provides a third line of defense.

Firewalls screen incoming traffic and sometimes filer outgoing traffic as well. The first obstacle encountered by the incoming traffic in a network is a the firewall supported by the operating system of the router, the next one is the firewall provided by the operating system running on the local computer, see Figure 66.

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[63].

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 18 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; the MAC layer when filtering is based on the destination/source MAC address.

In *Linux* or *Unix* systems the firewall can be configured only as a *root* using the *sudo* command. The kernel data structure which control the firewall is the *iptables*. The *iptables* command is used to set up, to maintain, and to 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; *INPUT* controls all incoming connections, *FORWARD* controls all packets passing through this host, and *OUTPUT* 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

```
 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 (the 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:

[63]The the 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 is also called a *pinhole*.

Table 18: 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 7	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Windows Vista	Yes	Yes	Yes	Yes	Yes	No	No	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

```
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; if a user accesses an *EC2* virtual machine using *ssh*, the *Secure Shell protocol* and issues the following command

```
sudo iptables -P INPUT DROP
```

the ability to access that virtual machine will be permanently lost.

The access to the *Windows 7* firewall is provide by a GUI accessed as follws:

$ControlPanel \rightarrow System\&Security \rightarrow WindowsFirewall \rightarrow AdvancedSettings$

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 provide 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.

12.2 Security rules for application- and transport-layer protocols in *EC2*

To be able to connect to a virtual machine in a cloud a client must know its IP address. For security reasons public IP addresses are mapped internally to private IP addresses. For example, a virtual machine running under Amazon's *EC2* has several IP addresses:

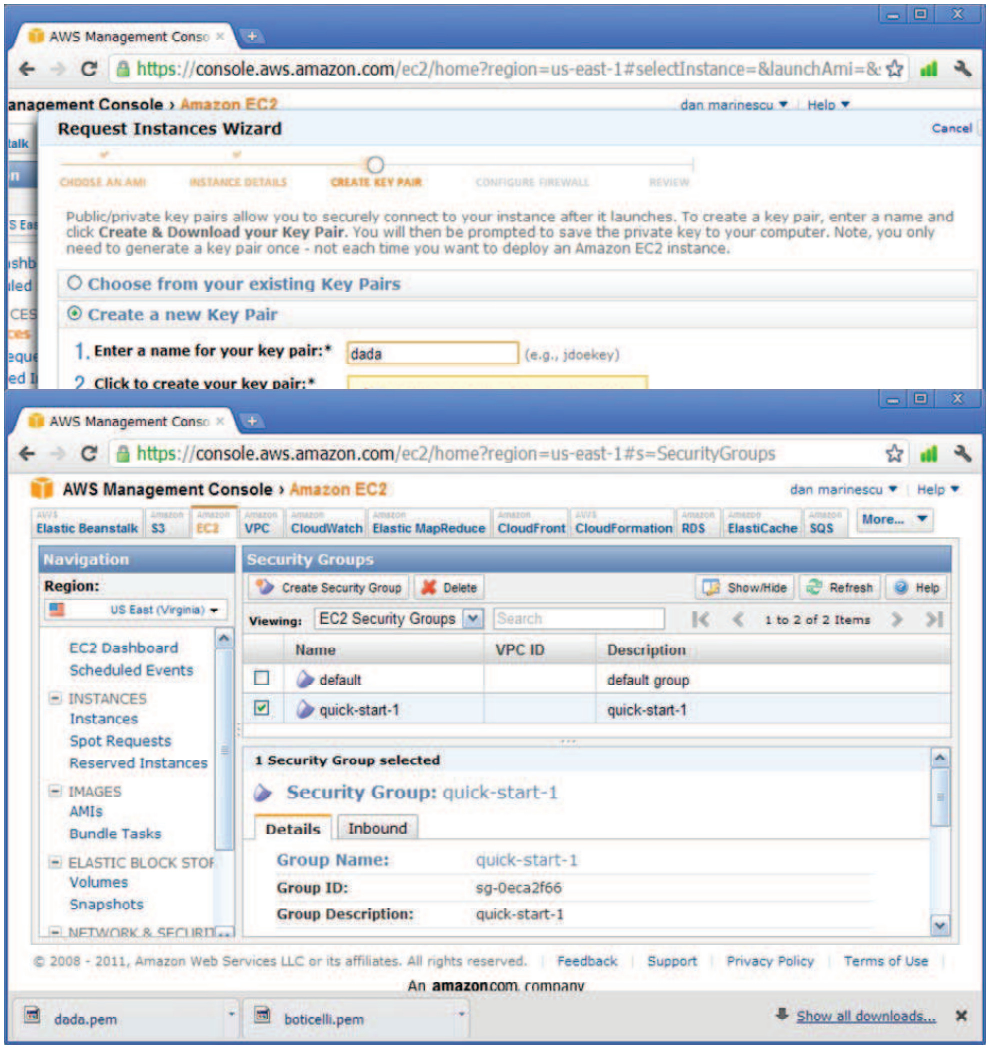
1. *EC2 Private IP Address:* The internal address of an instance; it is only used for routing within the EC2 Cloud.

2. *EC2 Public IP Address:* Network traffic originating outside the EC2 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 AWS EC2 account and used by traffic originated outside the EC2 cloud. 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 Web Services use *security groups* to control access to user's virtual machines. A virtual machine 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. All inbound traffic to an instance, originated 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



Public/private key pairs allow you to securely connect to your instance after it launches. To create a key pair, enter a name and click **Create & Download your Key Pair**. You will then be prompted to save the private key to your computer. Note, you only need to generate a key pair once - not each time you want to deploy an Amazon EC2 instance.

○ Choose from your existing Key Pairs
◉ Create a new Key Pair
1. Enter a name for your key pair:* dada (e.g., jdoekey)
2. Click to create your key pair:*

	Name	VPC ID	Description
☐	default		default group
☑	quick-start-1		quick-start-1

1 Security Group selected

Security Group: quick-start-1

Group Name:	quick-start-1
Group ID:	sg-0eca2f66
Group Description:	quick-start-1

Figure 67: AWS Security. (a) Provide a name for the key par; (b) 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.

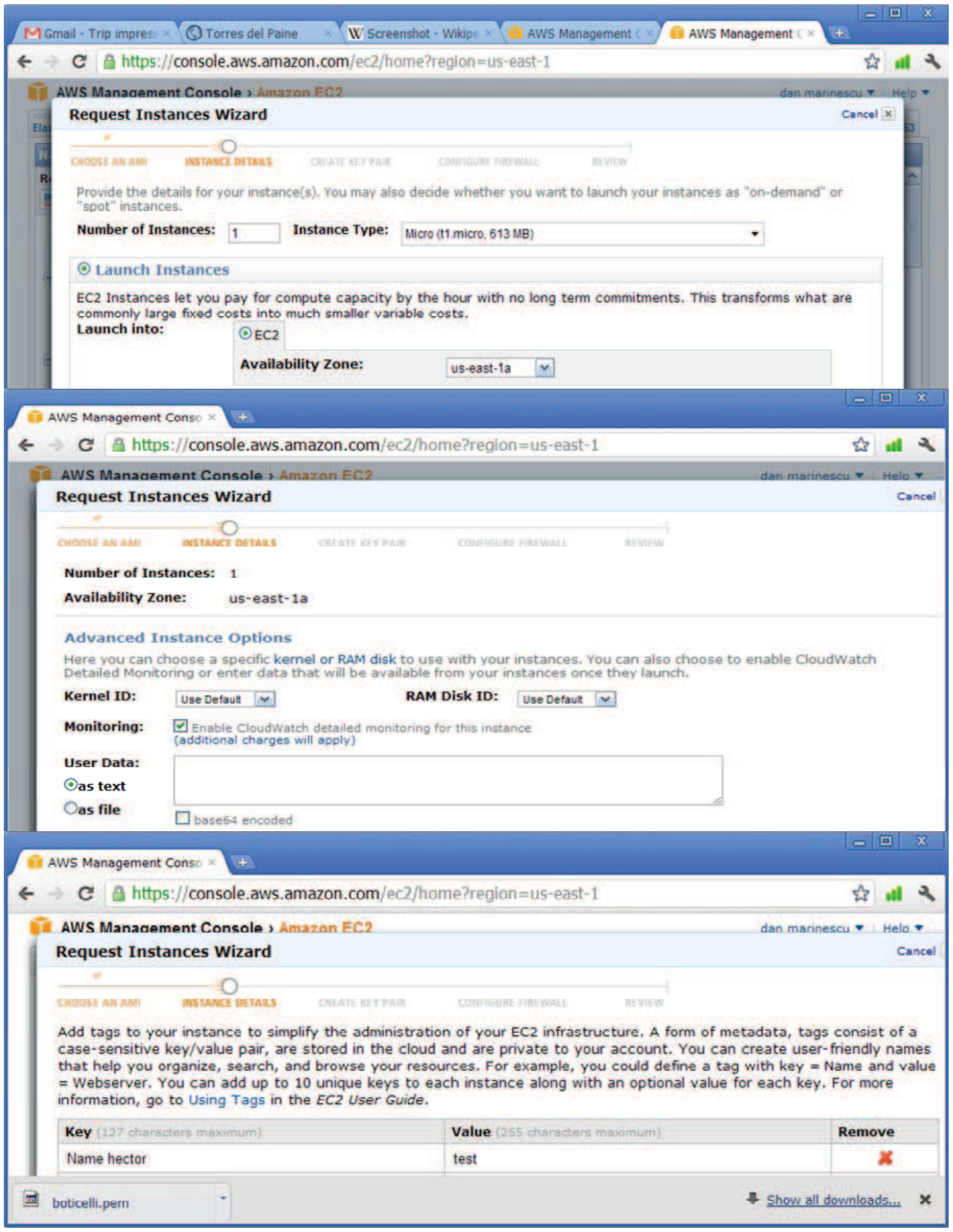
B listening on TCP port P, where B is in security group Σ_B. A new rule must be added to security group Σ_B to allow connections to port P; to accept responses from server B a new rule must be added to security group Σ_A.

The following steps allows 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 as in Figure 64.

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 68.

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



Request Instances Wizard

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

Request Instances Wizard

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

Request Instances Wizard

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 68: *EC2 Request Instance Wizard* is used to: (a) specify the number and type of instances and the zone; (b) specify the kerned Id, the RAM disk Id, and to enable the *CloudWatch* to monitor the 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.

Inbound tab to enter the desired rule, see Figure 67. In this case the private security key was stored in the file `dada.pem`.

To allocate an elastic IP address use the *Elastic IPs* tab of the *Network $ Security* left hand side panel.

On *Linux* or *Unix* systems port numbers below 1024 can only be assigned by *root*; services is a plain ASCII file providing a mapping between friendly textual names for internet services, and their underlying assigned port numbers and protocol types. A sample *services* filer may look like:

```
netstat 15/tcp
ftp     21/udp
ssh     22/tcp
telnet  23/tcp
http    80/tcp
```

12.3 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. Enter the command

   ```
   uname -m
   ```

   in the command line to determine the processor architecture and choose an appropriate *Amazon Linux AMI* by pressing *Select.*
3. Choose *Instance Details* to control the number and size and other settings for instances.
4. To learn how the system works press *Continue* to select the default settings.
5. Define the instances security, as discussed in Section 12.2: 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*

   ```
   cd awcdir
   chmod 400 dada.pem
   ```

7. Configure the firewall; go to the *Configure firewall* page, 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 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. Sometimes the application may require an elastic IP address.

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*; without this step either *access denied* or *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, s(he) cannot install any software. Once connected to *EC2* use

```
sudo -I
```

to gain root rights; 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

```
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 &
```

12.4 How to use *S3* in Java

Assuming that

Create an S3 client. *S3* access are handled by the class *AmazonS3Client* which is instantiated with user's AWS account credentials

```
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 12.2.

Buckets. An *S3 bucket* is analogous to a file folder or directory and it is used to store *S3 Objects*; bucket names must be unique and it is advisable to check first if the name exits:

```
s3.doesBucketExist("bucket_name");
```

This function return "true" is the name exists and "false" otherwise. Buckets can be created and deleted wither 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 is indexed by a key string. A single key points to only one S3 object in one bucket thus, key names do not have to be globally unique. To upload an object in a bucket one can use the *AWS Management Console,* or programmatically. To upload the file $local\_file_name$ from the local machine to the bucket *bucket_name* under the key *key*

```
File f = new File("local_file_name");
s3.putObject("bucket_name", "key", f);
```

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();
```

Once you have the InputStream you can read it in any m,manner you prefer (Scanner, BufferedReader, etc.). Amazon recommends closing the stream as early as possible, as the content is not buffered and it is streamed directly from the *S3*; and 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");
```

ObjectListing supports several useful methods including *getObjectSummaries(); S3ObjectSummary* encapsulates most of an S3Object's 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; 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 there is a very large number of objects then *s3.listObjects()* will return a truncated list. To test if the list is truncated one could use *listing.isTruncated()*; to get the next batch of objects use

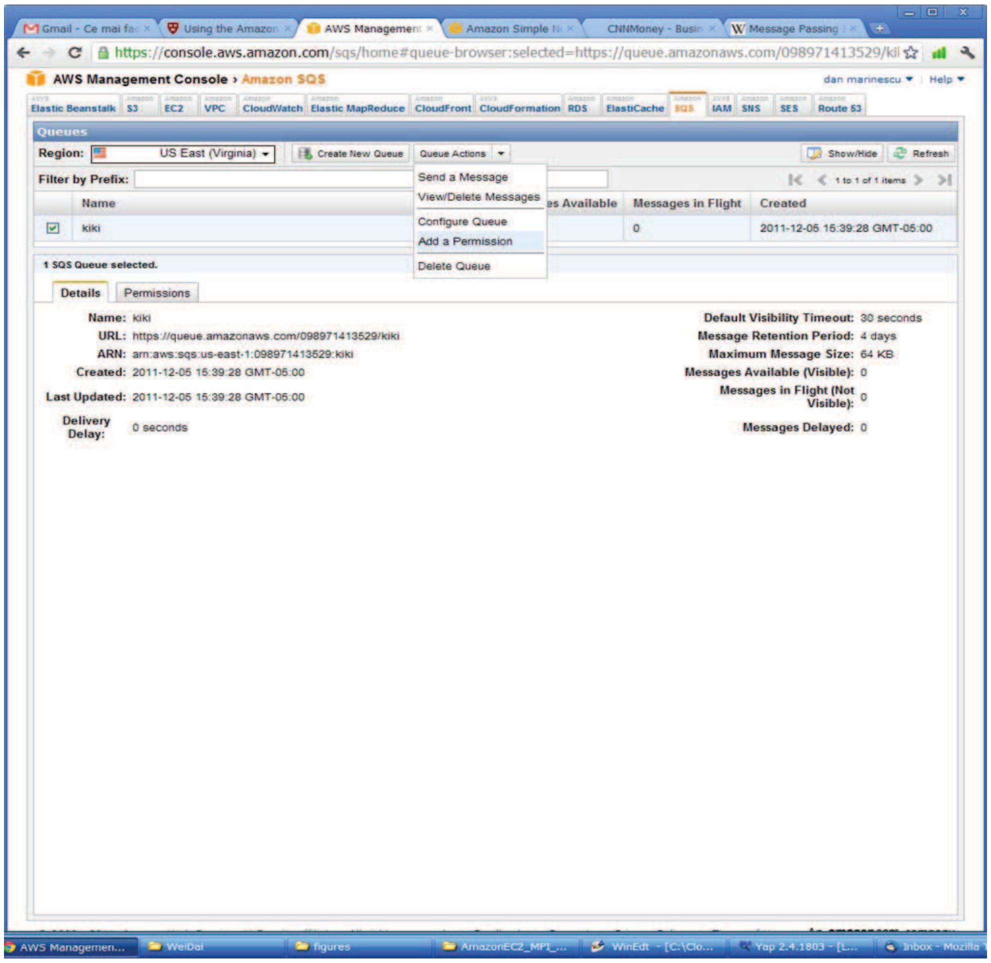
```
s3.listNextBatchOfObjects(listing)}.
```

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);
}
```

12.5 How to manage SQS services in C#

Recall from Section 4.1 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 5.6. Figure 69 shows the actions available for a given queue in *SQS*.



Name	Messages in Flight	Created
kiki	0	2011-12-05 15:39:28 GMT-05:00

1 SQS Queue selected.

Name: kiki
URL: https://queue.amazonaws.com/098971413529/kiki
ARN: arn:aws:sqs:us-east-1:098971413529:kiki
Created: 2011-12-05 15:39:28 GMT-05:00
Last Updated: 2011-12-05 15:39:28 GMT-05:00
Delivery Delay: 0 seconds

Default Visibility Timeout: 30 seconds
Message Retention Period: 4 days
Maximum Message Size: 64 KB
Messages Available (Visible): 0
Messages in Flight (Not Visible): 0
Messages Delayed: 0

Figure 69: Queue actions in *SQS*.

The following steps can be used to create a queue, send a message, receive a message, delete 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);
```

12.6 How to install the *Simple Notification Service* on Ubuntu 10.04

The *Simple Notification Service (SNS)* 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 users to define topics, the transport protocol used (HTTP/HTTPS, Email/Email, SMS, SQS) and the end-point (URL, Email address, phone number, SQS queue) for notifications to be delivered. It supports the following actions:

- Add/Remove Permission
- Confirm Subscription
- Create/Delete Topic
- Get/Set Topic Attributes
- List Subscriptions/Topics/SubscriptionsByTopic
- Publish/Subscribe/Unsubscribe

The site *http://awsdocs.s3.amazonaws.com/SNS/latest/sns-qrc.pdf* provides detailed information about each one of these actions.

Ubuntu[64] is an open source operating system for personal computers based on Debian Linux distribution; the desktop version of *Ubuntu* supports the Intel *x86* 32-bit and 64-bit architectures.

To install the SNS client the following steps must be taken:

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
   ```

[64]Ubuntu is an African humanist philosophy; "ubuntu" is a word in the Bantu language of South Africa meaning "humanity towards others."

```
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/
```

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.

12.7 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) images. 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" [100]. *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 [100] 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 BUFFSIZE 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,&numprocessors);
  MPI_Comm_rank(MPI_COM_PROCESS_GROUP,&my_process_Id);
```

Once started each process other than the coordinator, the process with $rank = 0$, sends a message to the entire group and then receives a message from all other members of the process group. 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 - datatype of each send buffer element (handle)
   dest - rank of destination (integer)
   tag - message tag (integer)
   comm - communicator (handle)

   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 [85]. An example of MPI use on *EC2* is at *http://rc.fas.harvard.edu/faq/amazonec2*.

12.8 How to install *hadoop* on *eclipse* on a Windows system

The software packages used are:

- *Eclipse* (http://www.eclipse.org/) is a software development environment; it 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.

- Apache *Hadoop* is a software framework that supports data-intensive distributed applications under a free license. The software was inspired by Google's *MapReduce*; see Section 5.5 for a discussion of *MapReduce* and Section 5.6 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 *Helios* might have some issues with *hadoop* plugin).

B. <u>SSH Installation</u>

1. Install *cygwin* using the installer downloaded from http://www.cygwin.com. Select the following packages under Net *openssh* and *openssl* from the *Select Packages* window. Note: Create a desktop icon when it is asked during installation

2. Edit the *Path* variable by following the

   $Computer \rightarrow SystemProperties \rightarrow AdvancedSystemSettings \rightarrow EnvironmentVariables.$

   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 deamon* 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 the command again:

    ```
    ssh localhost
    ```

    this time no prompt should appear.

C. <u>Download *hadoop*</u>

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 .
    ```

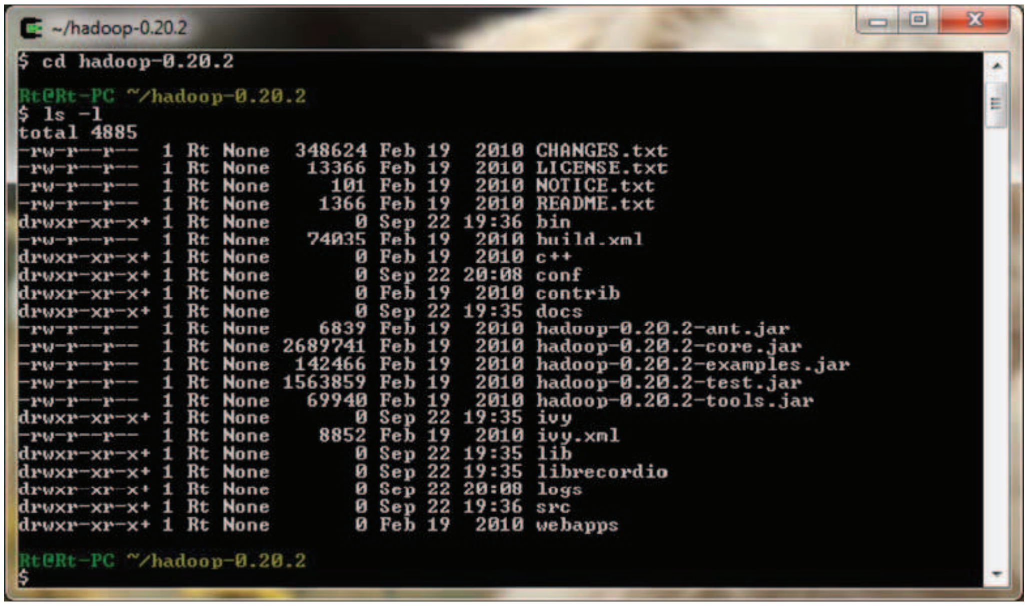
4. Open another Windows Explorer window and navigate to the folder that contains the downloaded *hadoop* archive.

5. Copy the *hadoop* archive into the home directory folder.

D. <u>Unpack *hadoop*</u>

1. Open a new *cygwin* window and execute

    ```
    tar -xzf hadoop-0.20.1.tar.gz
    ```



```
$ 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 70: The result of unpacking *hadoop*.

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 70 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.

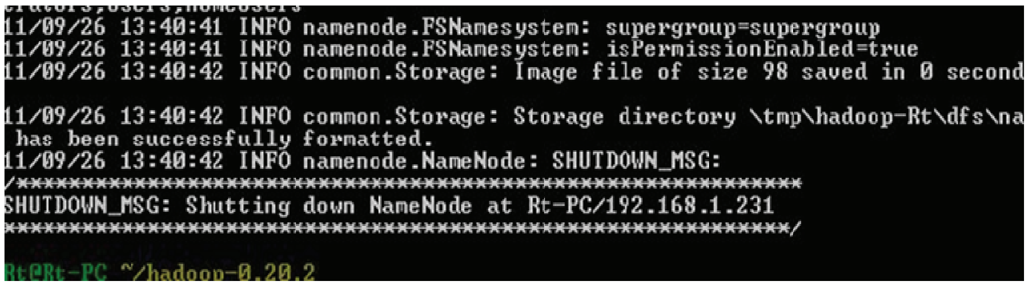
3. Launch Eclipse / Notepad ++ and navigate to the *conf* directory and open the file *hadoop*-site to insert the following lines between $< configuration >$ and $< /configuration >$ tags.

```
<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
```



```
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 71: The creation of HDFS.

When the formatting of the *namenode* is finished the message in Figure 71 appears.

12.9 Cloud applications for cognitive radio networks

The first application we discuss is related to mobile wireless networks and mobile devices such as smart phones and tablets; such mobile devices are able to communicate using two networks: (i) a cellular wireless network; and (ii) a Wifi network. The communication channels for the two networks operate in different regions of the spectrum[65]. In our discussion we assume 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 motivation for such applications is to delegate compute- and data-intensive tasks to the cloud to reduce the power consumption of the mobile devices. The algorithms allow mobile devices to minimize the cost for communication by using a Wifi network instead of a cellular; indeed, if a subscriber to a cellular network exceeds the amount of data allowed by its data plan it must pay additional fees.

Transferring computations related to CR 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.

12.9.1 Motivation and basic concepts

Mobile phones are also called cellular phones because they use a cellular radio network to communicate. The coverage area of a *cellular radio network* is divided into cells, usually of regular shape, e.g., hexagonal. The base station in each cell is assigned a communication channel in such a manner that neighboring cells do not interfere with each other. This allows the cellular network to reuse the communication channels; the carrier frequency of a channel can be reused in a non-adjacent cell at a distance D provided that

$$D \geq R\sqrt{3N} \qquad (107)$$

with R the radius of a cell and N the number of cells in the cluster. In a city a cell may have a range of approximately one kilometer, while in an open area the range could be 40 times larger.

Cellular networks use either Frequency Division Multiple Access (FDMA) or Code Division Multiple Access (CDMA). For example, `4G` is the fourth generation cellular wireless standard, with peak speed of 100 Mbps for highly mobile communication (e.g., cars and trains) and 1 Gbps for low mobile or stationary users (e.g., pedestrians); `4G` is expected to support secure services not only for cellular telephony, but also for gaming and streamed multimedia and to ensure smooth handover across heterogeneous networks. Any mobile device with a data plan can connect to the Internet via a `3G` or `4G` cellular network.

[65]Spectrum allocation is a matter of international agreements. In the US the spectrum management is a function of the FCC (Federal Communication Commission). Different regions of the spectrum are allocated to: cell phones, broadcast television and audio, mobile radio, GPS systems, satellite TV broadcast reception, Wifi, RFID devices such as product tags, passports, and active badges, radar for air-traffic control, vehicle-speed control, weather monitoring, Citizen's band radio, Bluetooth, and so on. See also http://transition.fcc.gov/connectglobe/sec7.html).

The sets of frequency allocated for cellular phone use changed over the years; initially, the phones were able to operate only in regions supporting the standard they were built for. Now, the GSM standard allows a cellular phobe to operate on several continents; some GSM phones are tri-band, i.e., support three bands, 0.9 or 0.85, 1.8, and 1.9 GHz others are quad-bands, i.e., support communication in four bands, 0.85, 0.9, 1.8, and 1.9 GHz. The coverage area and the number of devices that can communicate simultaneously depend on the frequency used; lower frequencies cover a larger area, while higher frequencies allow carriers to provide service to more customers in a smaller area.

Wifi is a technology that allows devices equipped with wireless network interface to connect to a LAN (Local Area Network) via an *access point* also called a *hotspot*; Wifi communication is described by the IEEE 802.11 standards. *Wireless ad-hoc networks* do not use an access point; individual nodes act as virtual hotspots. The range of Wifi networks is limited to less than 30 meters indoor and 90 meters outdoors function of their antennas and the frequency they operate at. The Wifi networks using the 5.0 GHz frequency bloc have a smaller range than the ones using the 2.4 frequency block; the Bluetooth wireless technology has a range of less than 10 meters.

While a GSM cellular phone can only operate in areas covered by the provider of service, standard Wifi devices can operate anywhere in the world. Though there are serious concerns regarding the security of Wifi networks, they are very popular in airports, malls, stadiums, and other public places; some cities such as Sunnyvale, California offer city-wide free Wifi access. Public access is supported in London using wireless mesh networks.

A network is a complex system with multiple components; in traditional networks each component attempts to optimize its own performance without concern for the end-to-end effect; for example, a router may drop packets to avoid overflowing its buffers without the concern that this leads to retransmissions thus, increases network congestion and end-to-end delays. A *cognitive network (CN)* is one where the end-to-end performance is optimized; its components dynamically adapt to network conditions to ensure optimal performance. To adapt to network conditions individual components use a range of techniques borrowed from several areas including machine learning, knowledge representation, trust management, and network management. Increased network complexity, the rapid development of wireless networks, and QoS (Quality of Service) requirements justify the increased research effort dedicated to CNs.

The communication bandwidth is a critical resource for wireless networks; periodically, a regulatory agency, the FCC in the US, auctions bandwidth regions and wireless network providers acquire licenses to use these channels. Licence violations have legal consequences and wireless service providers have developed technologies to avoid such situations.

A *cognitive radio (CR)* is a two-way wireless communication network where the individual nodes use the communication bandwidth effectively, while attempting to avoid interference with licensed users. We recognize two types of devices connected to a CR network, primary and secondary; *primary* nodes/devices have exclusive rights to specific regions of the spectrum, while *secondary* nodes/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.* There is another mode of operation based on the idea that a secondary operates at a much lower power lever than a primary. 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 (S/N), the signal-to-noise ratio, is large enough.

We are only concerned with the overlay mode; in this mode a secondary node maintains an *occupancy report*, 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 nodes continually sense the channels available to them to gather accurate information about available channels.

Computing an accurate occupancy report requires cooperation among all the secondary nodes. Indeed, a secondary node has a limited transmission and reception range; moreover, node mobility adds to the difficulties of gathering accurate information. A secondary node must use information provided by its neighbors, nodes in its transmission and reception range to construct the occupancy map, the same way nodes in a wired network use information from their neighbors to update the routing tables.

Unfortunately, all secondary nodes compete for free channels and the information one node may provide to its neighbors could be deliberately distorted. To *deny the service* a node will send false information, the occupancy report sent to its neighbors showing 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 in fact free, a strategy called *secondary spectrum data falsification (SSDF).*

12.9.2 A distributed algorithm to compute the trust in a CR

The algorithm computes the trust of node $1 \leq i \leq n$ in each node in its vicinity, $j \in V_i$ 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:

$$S_i(t) = \{s_{i,1}(t), s_{i,2}(t), \ldots, s_{i,K}(t)\} \tag{108}$$

   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 node $j \in V_i(t)$ using the algorithm described in this section.

4. Infer the power as measured by each node $j \in V_i(t)$ 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 $s_{i,k,j}^{infer}(t), 1 \leq k \leq K,\ j \in V_i(t)$ a secondary node j should have determined: 0 if the channel is free for use, 1 if the primary is active, and X if it cannot be determined.

$$s_{i,k,j}^{infer}(t) = \begin{cases} 0 & \text{if secondary node j should decide that channel k is free} \\ 1 & \text{if secondary node j should decide that channel k used by the primary} \\ X & \text{if no inference can be made} \end{cases} \tag{109}$$

6. Receive the information provided by neighbor $j \in V_i(t)$, $S_{i,k,j}^{recv}(t)$.

7. Compare the information provided by neighbor $j \in V_i(t)$

$$S_{i,k,j}^{recv}(t) = \{s_{i,1,j}^{recvd}(t), s_{i,2,j}^{recv}(t), \ldots, s_{i,K,j}^{recv}(t)\} \tag{110}$$

with the information inferred by node i about node j

$$S_{i,k,j}^{infer}(t) = \{s_{i,1,j}^{infer}(t), s_{i,2,j}^{infer}(t), \ldots, s_{i,K,j}^{infer}(t)\} \tag{111}$$

8. Compute the number of matches, mismatches, and cases when no inference is possible

$$\alpha_{i,j}(t) = \mathcal{M}\left[S_{i,k,j}^{infer}(t), S_{i,k,j}^{recv}(t)\right] \tag{112}$$

with $\mathcal{M}$ the number of matches between the two vectors,

$$\beta_{i,j}(t) = \mathcal{N}\left[S_{i,k,j}^{infer}(t), S_{i,k,j}^{recv}(t)\right] \tag{113}$$

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

$$\zeta_{i,j}(t) = [1 + X_{i,j}(t))] \frac{\alpha_{i,j}(t)}{\alpha_{i,j}(t) + \beta_{i,j}(t)} \tag{114}$$

12.9.3 Simulation of a distributed trust computing algorithm

The cloud application is a simulation of a CR network 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 72. The secondary nodes are identified by an instance Id, *iId*, as well as a global Id, *gId*. The simulation assumes that the primary node(s) cover the entire area thus their position is immaterial.

The simulation involves of 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

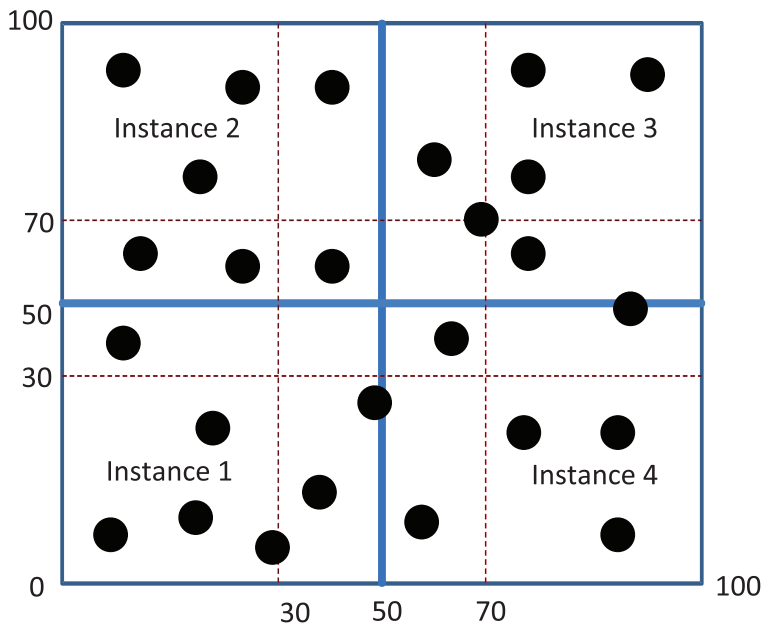


Figure 72: Data partitioning for the simulation of a trust algorithm; the area covered is of size 100×100 units. The nodes in the four sub-areas of size 50×50 units are processed by an instance of the cloud application. The sub-areas allocated to an instance overlap to allow an instance to have all the information about a node in its coverage area.

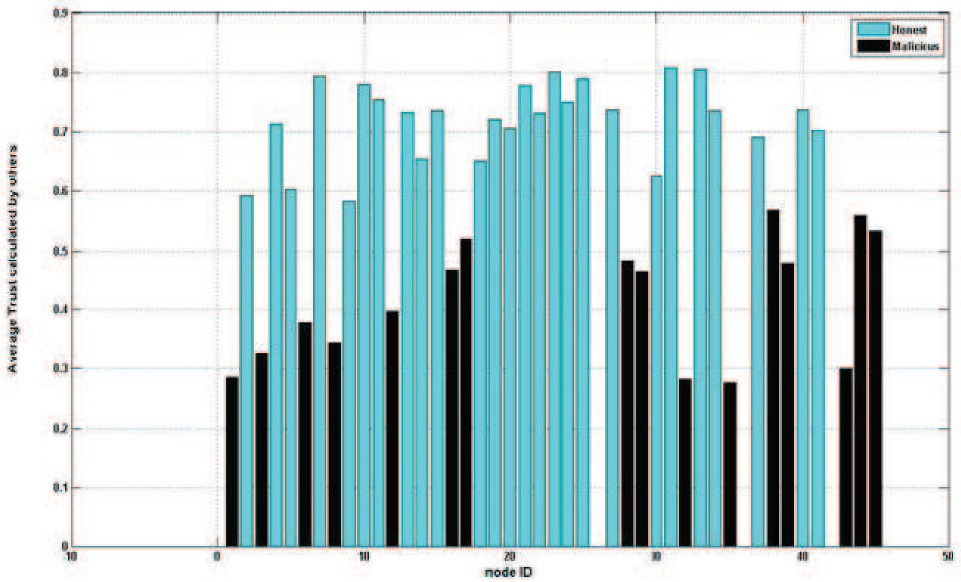


Figure 73: The trust values computed using the distributed trust algorithm. The secondary nodes programmed to act maliciously have a trust value less than 0.6 and many less than 0.5, lower than that of the honest nodes

a neighbor; the results are then processed by an `awk`[66] 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 runs also on the cloud.

In the simulation discussed here the nodes with

$$gId = \{1, 3, 6, 8, 12, 16, 17, 28, 29, 32, 35, 38, 39, 43, 44, 45\} \quad (115)$$

were programmed to be dishonest. The results of the simulation are summarized in Figure 73; they show that the nodes programmed to act maliciously have a trust value lower than that of the honest nodes, always lower than 0.6 and many instances lower than 0.5. 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 indicated that when the trust is computed using the information provided by all secondary nodes we can expect a higher accuracy of the trust determination.

12.9.4 A history-based cloud service for cognitive radio networks

The distributed algorithms for trust determination reported in the literature allow a secondary node to infer the occupancy report, but require a fair number of assumptions that may, or may not, be reasonable in practice. A common assumption is that all secondary nodes transmit with the same power level; another assumption is that all secondary nodes use the same threshold to normalize the power vectors; there are assumptions regarding the method used to estimate the noise. Yet, another assumption is that the links connecting the mobile nodes are symmetric, the transmission range is the same as the reception range.

We discuss next a history-based algorithm for bandwidth management in CR networks. The algorithm is based on several reasonable assumptions regarding the secondary nodes; we assume that the secondary nodes:

- Are mobile devices; some are slow-moving, while others are fast-moving.
- Do not have a GPS system thus, they cannot report their position.
- 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.

The algorithm is intended for a cloud service for CR resource management; it has several advantages over a distributed system based on the algorithm discussed earlier:

- Reduces drastically the computations a mobile device is required to carry out to identify free channels and to avoid penalties associated with interference with primary transmitters.

[66]The AWK utility is based on a scripting language and used to produce formatted reports.

- Allows a secondary node 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.

- Is 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.

- It 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.

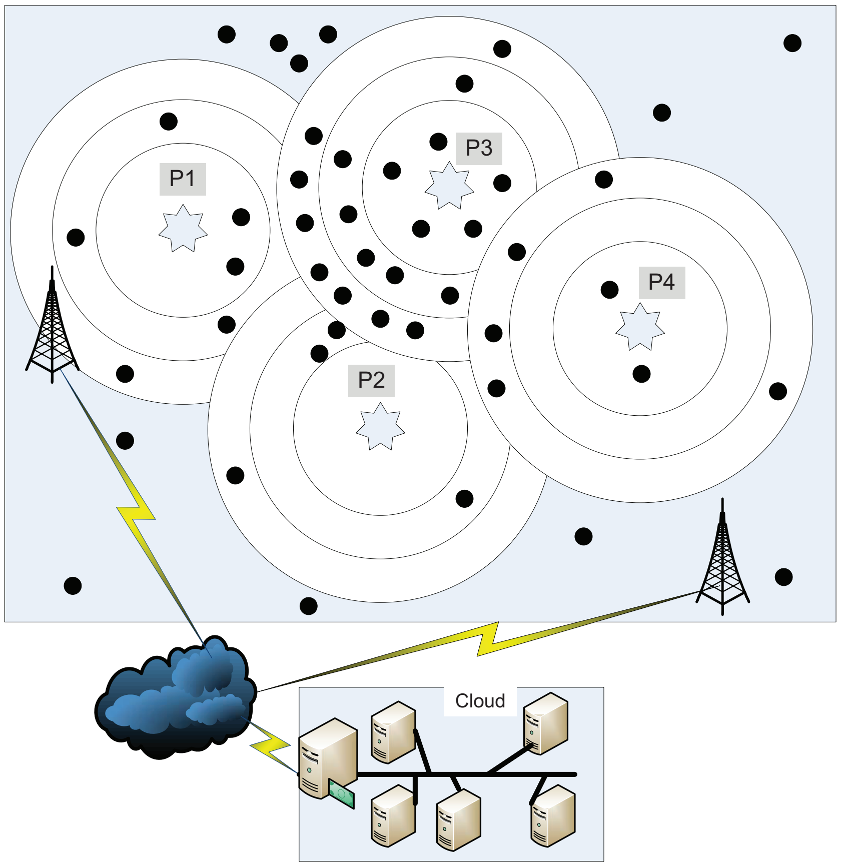


Figure 74: Schematic representation of a CR layout; four primary nodes, P1-P4, a number of mobile devices, two towers for a cellular cellular network and a cloud are shown. Not shown are the hotspots for the Wifi network.

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.

The nodes which do not participate in the trust computation will not register in this cloud-based version of the resource management algorithm thus, 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 addresses and the cloud responds with the tuple (Δ, δ_s); 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 only transmit 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.

The cloud computes the probable distance d_i of a secondary i from the known location of a primary transmitter, P^k; based on this distance it defines N circular rings centered at the primary; modes at distance $d \leq d_1^k$ are included in the ring $\mathcal{R}_1^k$, those at distance $d_1^k > d < d_2^k$ 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.

After each report cycle the cloud computes the occupancy report for channel $1 \leq k \leq K$ used by primary transmitter P^k; the status of channel reported by the $n_i^k \in \mathcal{R}_i^k$ secondary nodes considered trustworthy is

$$\sigma_i^k(t) = \frac{1}{n_i^k} \Sigma_{j=1}^{n_i} s_j^k(t) \tag{116}$$

with $s_j^k(t)$ the status reported by node $j \in \mathcal{R}_i^k$; $s_j^k(t) = 0$ if the secondary $j \in \mathcal{R}_i^k$ determines that channel is free, $s_j^k(t) = 1$ if it determines that the channel is in use by the primary, and $s_j^k(t) = X$.

Call w_i^k the weight of each ring around primary transmitter, P^k; $\Sigma_{i=1}^N w_i^k = 1$. The weight w_i^k is a function of the distance of the ring $\mathcal{R}_i^k$ from the primary transmitter and the number n_i^k of nodes in that ring. Then the cloud algorithm determines the average

$$s_{avg}^k(t) = \frac{1}{N^k} \Sigma_{i=1}^N w_i \sigma_i^k(t) \tag{117}$$

with

$$N^k = \sum n_i^k \ \text{ for all rings } \ \mathcal{R}_i^k \ \text{ at distance } \ d_i^k \leq d^k \tag{118}$$

with d^k the distance that allows a good reception of the primary P^k; this distance is a function of the power of the primary transmitter. Finally, the occupancy report for the K channels is

$$S(t) = \{s^1(t), s^2(t), \ldots, s^K(t)\} \tag{119}$$

with

$$s^k(t) = \begin{cases} 0 & \text{if} \quad s^k_{avg}(t) \geq \lambda \\ 1 & \text{if} \quad s^k_{avg}(t) < \lambda \end{cases} \tag{120}$$

with λ is a parameters of the algorithm.

The trust in each node in 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.

We compute the number of matches and mismatches

$$\alpha_j(t) = \mathcal{M}\left[S(t), S_j(t)\right] \tag{121}$$

with $\mathcal{M}$ the number of matches,

$$\beta_j(t) = \mathcal{N}\left[S(t), S_j(t)\right] \tag{122}$$

with $\mathcal{N}$ the number of mismatches.

Then we use the quantities $\alpha_j(t)$ and $\beta_j(t)$, to assess the trust in node j. For example, we compute the trust in node j at time t as

$$\zeta_j(t) = \frac{\alpha_j(t)}{\alpha_j(t) + \beta_j(t)} \tag{123}$$

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 5% the 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 trustworthy of all all nodes and and may accept the information from node j when its behavior changes.

12.10 Adaptive data streaming to and from a cloud

Data streaming is the name given to the transfer of data with real-time constraints at a high-rate. Multi-media applications such as music streaming, or 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 an application running on a computer cloud. The stream could originate from the cloud as is the case of the iCloud service provided by Apple or be directed towards a service running on the cloud, as is 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. Adaptive data streaming is possible only if the application permits tradeoffs between quantity and quality; lower data rates imply lower quality, but reduce the demands for system resources. Such tradeoffs are feasible for audio and video streaming which allow lossy compression, but are not acceptable for 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 convert to a lower quality and thus, reduce the data rate; data conversion can be done on the fly and this means that the data flow has to be changed. A CDN (Content Delivery Network) such as the one provided by Akamai is often used to deliver media streaming using the Internet; the CDN receives the stream from an *Origin* server, then replicates it to its *Edge* cache servers; the stream is delivered to an end-user from the "closest" Edge server.

Adaptive video streaming monitors in real time the available bandwidth, as well as the the CPU capacity of the the player and adjusts the quality of a video stream accordingly; an encoder encodes a single source video at multiple bit rates while the player client switches between different encodings depending on available resources. This strategy reduces the amount of buffering and allows a fast-up start time. Today the adaptive video streaming technologies are almost exclusively based on the HTTP protocol, while in the past the communication protocols used for delivering audio and video over IP networks were exclusively RTP (Real-time Transport Protocol) and RTCP (RTP Control Protocol). RTP transports the media while RTCP monitors the transmission and aids synchronization of multiple streams; RTP is originated and received on even port numbers on the sending and receiving hosts and the associated RTCP communication uses the next higher odd port number. The use of HTTP-based adaptive video streaming allows the Edge server to run a simple HTTP server software; for example *HTTP Live Streaming* is a protocol implemented by Apple as part of their QuickTime X and and iPhone software.

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 [229].

12.10.1 Digital music streaming

Cloud-based music streaming to mobile phones is increasingly more popular among the very large base (some 80 million in the early 2011) of smart phone users. Several companies including Sony's Music Unlimited (http://www.sony.co.uk), Pandora (http://www.pandora.com), and Spotify (http://www.spotify.com) have provided digital music services for some time. Apple has introduced the iCloud service in June 2011, while Google has revealed plans for digital music streaming to Android handsets and it is likely to announce its own cloud-based digital music services. Spotify provides a premium service over Wifi and *3G* networks.

The data rates for digital music services increase function of the quality; for example, 96 Kbps for radio quality, 160 Kbps for medium, *G3* quality, and 320 Kbps for CD quality. MP3 is a widely used standard for digital music; it is based on a lossy data compression algorithm that reduces the amount of data necessary to store and transmit a digital audio recording. There are significant incentives for a user to use a Wifi network rather than cellular network; indeed a 2 GB data plan will permit an iCloud user to download 45.76 hours of radio quality music, or 27.52 hours of medium quality, or 13.79 hours of CD-quality music.

Different formats achieve distinct sound quality at similar data rates; for example the AAC (Advanced Audio Coding), a standardized lossy compression and encoding scheme produces a better sound quality than MP3 at similar data rates. AAC is the default or the standard audio format for iPhone, iPod, as well as a host of other mobile devices used for communication or for games (e.g., PlayStaion 3, Nintendo, etc.). Some of the advantages of AAC over MP3 are

- Sample frequencies from 8 to 96 kHz versus 16 to 48 kHz for than MP3.
- Up to 48 channels while MP3 supports up to 5.1 channels in MPEG-2 mode.
- Arbitrary bit-rates and variable frame length.
- Higher coding efficiency for stationary signals, i.e., a blocksize of 1024 or 960 samples versus 576 for MP3.
- Higher coding accuracy for transient signals, i.e., a blocksize of 128 or 120 samples versus 192 for MP3.
- Much better handling of audio frequencies above 16 kHz.

References

[1] W. M. P. van der Aalst and A. H. ter Hofstede and B. Kiepuszewski and A.P. Barros. “Workflow patterns.” *Technical Report, Eindhoven University of Technology,* 2000.

[2] R. Abbott. “Complex systems engineering: putting complex systems to work,” *Complexity,* **13**(2):10–11, 2007.

[3] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, “Performance guarantees for Web server end-system: a control theoretic approach.” *IEEE Trans. Parallel & Distributed Systems,* **13**(1):80-96, 2002.

[4] B. Abrahao, V. Almeida, J. Almeida, A. Zhang, D.Beyer, and F. Safai. “Self-adaptive SLA-driven capacity management for Internet srvices.” *Proc. IEEE/IFIP Network Operations & Management Symposium (NOMS06)*, pp. 557–568, 2006.

[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. “RACS: A case for cloud storage diversity,” *Proc. ACM Symp. on Cloud Computing (SOCC),* (CD Proceedings) ISBN: 978-1-4503-0036-0.

[6] D. Abts. “The Cray XT4 and Seastar 3-D torus interconnect.” *Encyclopedia of Parallel Computing,* Ed. David Padua, Springer, 2011 (to appear).

[7] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. “Energy proportional datacenter networks.” *ACM IEEE Int. Symp. on Comp. Arch. (ISCA’10)*, pp. 338–347, 2010.

[8] L. A. Adamic, R. M. Lukose, A. R. Puniyami, and B. A. Huberman. “Search in power-law networks.” *Phys. Rev. E,* **64**(4):46135–46143, 2001.

[9] B. Addis, D. Ardagna, B. Panicucci, and L. Zhang. “Autonomic management of cloud service centers with availability guarantees.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 220–227, 2010.

[10] R. Albert, H. Jeong, and A.-L. Barabási. “The diameter of the world wide web.” *Nature,* **401**:130, 1999.

[11] R. Albert, H. Jeong, and A.-L. Barabási. “Error and attack tolerance of complex networks.” *Nature,* **406**:378382, 2000.

[12] R. Albert and A-L. Barabási. “Statistical mechanics of complex networks.” *Reviews of Modern Physics,* **72**(1):48–97, 2002.

[13] “Amazon elastic compute cloud.” *http://aws.amazon.com/ec2/*

[14] “Amazon virtual private cloud.” *http://aws.amazon.com/vpc/*

[15] “Amazon web services: Overview of security processes.” *http://s3.amazonaws.com.*

[16] “Amazon CloudWatch.” *http://aws.amazon.com/cloudwatch*

[17] “Amazon elastic block store (EBS).” *http://aws.amazon.com/ebs/*

[18] “AWS management console.” *http://aws.amazon.com/console/*

[19] T. Andrei. “Cloud computing challenges and related security issues.” *http://www1.cse.wustl.edu/jain/cse571-09/ftp/cloud/index.html*

[20] 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.

[21] R. Aoun, E. A. Doumith, and M. Gagnaire. “Resource provisioning for enriched services in cloud environment.” *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 296–303, 2010.

[22] D. Ardagna, M. Trubian, and L. Zhang. “SLA based resource allocation policies in autonomic environments.” *J. Parallel Distrib. Comp.*, **67**(3):259–270, 2007.

[23] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. “Energy-aware autonomic resource allocation in multi-tier virtualized environments.” *IEEE Trans. on Services Computing,* (to appear).

[24] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz,, A Konwinski, G. Lee, D. Paterson, A. Rabkin, I. Stoica, 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*

[25] M. Asay. “An application war is brewing in the cloud.” *http://news.cnet.com/8301-13505_3-10422861-16.html*

[26] M. Auty, S. Creese, M. Goldsmith, and P. Hopkins. “Inadequacies of current risk controls for the cloud.” *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 659–666, 2010.

[27] L. Ausubel and P. Cramton. “Auctioning many divisible goods.” *Journal European Economic Assoc.*, **2**(2-3):480-493, 2004.

[28] L. Ausubel, P. Cramton, and P. Milgrom. “The clock-proxy auction: a practical combinatorial auction design.” *Chapter 5,* in *Combinatorial Auctions*, P. Cramton, Y. Shoham, and R. Steinberg, Eds. MIT Press, 2006.

[29] A. Avisienis, J. C. Laprie, B. Randell,and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing.” *IEEE Trans. Dependable and Secure Computing,* **1**(1):11–33, 2004.

[30] M. Azambuja, R. Pereira, K. Breitman, and M. Endler. “An architecture for public and open submission systems in the cloud.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing,* pp. 513–517, 2010.

[31] Y. Azar, A.Z. Broder, A.R. Karlin, N. Linial, and S. Phillips. "Biased random walks." *Proc STOC92, 24th Annual Symp on Theory of Computing,* pp. 1-9, 1992.

[32] Ö Babaoğlu and K. Marzullo. "Consistent global states." *In Sape Mullender, Ed., Distributed Systems, Addison Wesley, Reading, Mass.,* pp. 55–96, 1993.

[33] A. A. Baker. "Monte Carlo simulations of radial distribution functions for a proton-electron plasma." *Aust. J. Phys.* **18**:119-133, 1965.

[34] 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. on Innovative Data Systems Research (CIDR'11)*, pp. 223–234, 2011.

[35] A-L. Barabási and R. Albert. "Emergence of scaling in random networks," *Science*, *286*:509–512, 1999.

[36] 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.

[37] 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." *Peoc. 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, pps. 14, 2003.

[38] L. A. Barroso and U. Hözle. "The case for energy-proportional computing." *IEEE Computer,* **40**(12):33–37, 2007.

[39] H. Bauke. "Parameter estimation for power-law distributions by maximum likelyhood methods." *The European Physical Journal B,* **58**(2):167–173, 2007.

[40] 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.

[41] D. Bernstein and D. Vij. "Intercloud security considerations." *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 537–544, 2010.

[42] 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. on Cloud Computing*, pp. 518–525, 2010.

[43] B. Bollobás. *Random Graphs,* Academic Press, London, 1985.

[44] 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. on Cloud Computing Technology and Science,* pp. 464–472, 2010.

[45] 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. on Cloud Computing*, pp. 244–251, 2010.

[46] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura. "An evaluation of distributed datatstores using the AppScale cloud platform." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 305–312, 2010.

[47] 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. on Cloud Computing Technology and Science,* pp. 67–74, 2010.

[48] A. G. Carlyle, S. L. Harrell, and P. M. Smith. "Cost-effective HPC: The community or the cloud?" *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 169–176, 2010.

[49] 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. on Cloud Computing Technology and Science,* pp. 456–463, 2010.

[50] A. Celesti, F. Tusa, M. Villari, and A. Puliafito. "How to enhance cloud architectures to enable cross-federation." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 337–345, 2010.

[51] A. Chandra, P. Goyal, and P. Shenoy. "Quantifying the benefits of resource multiplexing in on-demand data centers." *Proc. 1st Workshop on Algorithms and Architecture for Self-Managing Systems,* 2003.

[52] K. M. Chandy and L. Lamport. "Distributed snapshots: determining global states of distributed systems." *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[53] V. Chang, G. Wills, and D. De Roure. "A review of cloud business models and sustainability." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 43–50, 2010.

[54] F. Chang, J. Ren, and R. Viswanathan. "Optimal resource allocation in clouds." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 418–425, 2010.

[55] K. Chard, S. Caton, O. Rana, and K. Bubendorfer. "Social cloud: Cloud computing in social networks." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 99–106, 2010.

[56] J. Chase. "Orca technical note: guests and guest controllers." *http://www.cs.duke.edu/nicl /pub/papers/control.pdf*, 2008.

[57] A. Chazalet. "Service level checking in the cloud computing context." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 297–304, 2010.

[58] 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*, pp.1–9, 2007. Also, *http://ppi.fudan.edu.cn/system/publications/paper/OVM-ppi-tr.pdf.*

[59] D. Chiu and G. Agarwal. " Evaluating cache and storage options on the Amazon Web services cloud." *Proc. CCGRID 2011,* pp. 362 –371, 2011.

[60] B. Clark, T. Deshane, E. Dow, S. Evabchik, M. Finlayson, J. Herne, and J. Neefe Matthews. "Xen abd the art of repeated research." *Proc. USENIX Annual Technical Conf.(ATEC'04)*, pp. 135–144, 2004. Also, *http://web2.clarkson.edu/class/cs644/xen/files/repeatedxen-usenix04.pdf.*

[61] A. Clauset, C. R. Shalizi, and M. E. J. Newman. "Power-law distributions in empirical data." *SIAM Reviwes,* **51**:661-704, 2007.

[62] R. Cohen and S. Havlin. "Scale-free networks are ultrasmall." *Phys. Rev. Lett.,* **90**(5):058701, 2003.

[63] L. Colitti, S. H. Gunderson, E. Kline, and T. Refice. "Evaluating IPv6 adoption in the internet." *Proc. Passive and Active Measurement Conference, PAM '2010*, pp.141–150, 2010.

[64] T. M. Cover and J. A. Thomas. *Elements of Information Theory,* Wiley, New York, NY, 1991.

[65] P. Cramton, Y. Shoham, and R. Steinberg, Eds., *Combinatorial Auctions,* MIT Press, 2006.

[66] F. Cristian, H. Aghili, R. Strong, and D. Dolev. "Atomic broadcast from simple message diffusion to byzantine agreement." *Proc. Int. Conf. on Fault Tolerant Computing*, IEEE Press, pp. 200–206, 1985.

[67] J. P. Crutchfield and J. P. Shalizi. "Thermodynamic depth of causal states: objective complexity via minimal representation," *Phys. Rev. E*, **59**:275–283, 1999.

[68] Cloud Security Alliance. "Security guidance for critical areas of focus in cloud computing (v2.1)." *https://cloudsecurityalliance.org/csaguide.pdf*, December 2009.

[69] J. Dean and S. Ghernawat. "MapReduce: Simplified Data Processing on Large Clusters." *Proc. 6th Symp. on Operating Systems Design and Implementation, OSDI04.*, 2004.

[70] Y. Demchenko, C. de Laat, and D. R. Lopes. "Security services lifecucle management in on-demand infrastructure services provisioning." *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 644–650, 2010.

[71] M. Devarakonda, B. Kish, and A. Mohindra. "Recovery inthe Calypso file system." *ACM Trans. Comput. Syst.,* **14**(3):287–310, 1996. Also *http://www.cc.gatech.edu/classes/AY2008/cs6210a_fall/recovery.pdf.*

[72] 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.

[73] P. Donnelly, P. Bui, and D. Thain. "Attaching cloud storage to a campus grid using Parrot, Chirp, and Hadoop." *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 488–495, 2010.

[74] 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. on Cloud Computing*, pp. 196–203, 2010.

[75] K. J. Duda and R. R. Cheriton. "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler." *Proc. 17th Symp. on OS Principles,* pp. 261–276, 1999.

[76] 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.

[77] 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. on Cloud Computing*, pp. 410–417, 2010.

[78] D. Ebneter, S. Gatziu Grivas, T. U. Kumar, and H. Wache. "Enterprise architecture frameworks for enabling cloud computing." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 542–543, 2010.

[79] J. Ejarque, R. Sirvent, and R. M. Badia. "A multi-agent approach for semantic resource allocation." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science.* pp. 335–342, 2010.

[80] M. Elhawary and Z. J. Haas. "Energy-efficient protocol foe cooperative networks." *IEEE/ACM Trans. on Networking,* **19**(2):561–574, 2011.

[81] 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/*

[82] P. Erdös and A. Rényi. "On random graphs." *Publicationes Mathematicae*, **6**: 290297, 1959.

[83] P. Erdös and T. Gallai. "Graphs with points of prescribed degree (Gráfok elört fokú pontokkal)." *Mat. Lapok* **11**: 264–274; *Zentralblatt Math.* 103.39701, 1961

[84] R. M. Esteves and C. Rong. "Social impact of privacy in cloud computing." *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 593–596, 2010.

[85] C. Evanghelinos and C. N. Hill. "Cloud computing for parallel scientic HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazons EC2" *Cloud Computing and Its Applications, CCA-08*, http://www.cca08.org/papers/Paper34-Chris-Hill.pdf, 2008.

[86] A. D. H. Farwell, M. J. Sergot, m. Salle, C. Bartolini, D. Tresour, A. Christodoulou. "Performance monitoring of service-level agreements for utility computin." *Proc IEEE. Int. Workshop on Electronic Contracting (WEC04)*, 2004.

[87] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini. “QoS-aware clouds.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 321–328, 2010.

[88] M. Fischer, N. A. Lynch, and M. S. Paterson. “Impossibility of distributed consensus with one faulty process.” *Journal of the ACM*, **32**(2):374–382, 1985.

[89] *http://www.freebsd.org/doc/handbook/jails.html*

[90] S. Floyd and Van Jacobson. “Link-sharing and resource management models for packet networks.” *IEEE/ACM Trans. on Networking*, **3**(4):365–386, 1995.

[91] A. G. Ganek and T. A. Corbi. “ The dawning of the autonomic computing era.” *IBM Systems Journal*, **42**(1):5-18, 2003.Also, *https://www.cs.drexel.edu/ jsalvage/Winter2010/CS576/autonomic.pdf*

[92] S. Garfinkel. “An evaluation of Amazon’s grid computing services: EC2, S3, and SQS.“ *Technical Report, TR-08-07,* Harvard University, 2007.

[93] C. Gkantsidis, M. Mihail, A. Saberi. “Random walks in peer-to-peer networks.” *Performance Evaluation,* **63**(3): 241–263, 2006.

[94] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google file system.” *Proc. 19th ACM Symp. on Operating Systems Principles (SOSP’03)*, pps. 15, 2003.

[95] D. Gmach, S. Kompass, A. Scholz, M. Wimmer, and A. Kemper. “Adaptive quality of service management for entreprize services.” *ACM Trans. on the Web (TWEB),* **2**(1):243–253, 2009.

[96] D. Gmach, J. Rolia, and L. Cerkasova. “Satisfying service-level objectives in a self-managed resource pool.” *Proc. 3rd. Int. Conf. on Self-Adaptive and Self-Organizing Systems,* pp. 243–253, 2009.

[97] K. I Goh, B. Kahang, and D. Kim. “Universal behavior of load distribution in scale-free networks.” *Physical Review Letters,* **87**:278701, 2001.

[98] R. P. Goldberg. “Architectural principles for virtual computer systems.” *Thesis, Harvard University*, 1973.

[99] J. Gray and D. Patterson. “A conversation with Jim Gray.” *ACM Queue* **1**(4):8–17, 2003.

[100] W. Gropp, E. Lusk, and A. Skjellum. “Using MPI.” *MIT Press*, 1994.

[101] N. Gruschka and M. Jensen. “Attack surfaces: A taxonomy for attacks on cloud services.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 276–279, 2010.

[102] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox. “MapReduce in the clouds for science.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 565–572, 2010.

[103] V. Gupta and M. Harchol-Balter. “Self-adaptive control policies for resource-sharing systems.” *Proc. 11th Int. Joint Conf. Measurement and Modeling Computer Systems (SIGMETRICS’09)*, pp. 311–322, 2009.

[104] J. O. Gutierrez-Garcia and K.- M. Sim. “Self-organizing agents for service composition in cloud computing.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 59–66, 2010.

[105] J. D. Halley and D. A. Winkler. “Classification of emegence and its relation to self-organization,” *Complexity,* **13**(5):10–15, 2008.

[106] K. Hasebe, T. Niwa, A. Sugiki, and K. Kato. “Power-saving in large-scale storage systems with data migration.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 266–273, 2010.

[107] 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.

[108] W. K. Hastings. “Monte Carlo sampling methods using Markov chains and their applications.” *Biometrika,* **57**:97109, 1970.

[109] 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 (FeBID 2010)* http://eurosys2010-dev.sigops-france.fr/workshops/FeBID2010/hellerstein.pdf

[110] T. Hey, S. Tansley, and K. Tolle. “Jim Gray on eScience: a transformed scientific method.” In **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*

[111] Z. Hill and M. Humphrey. “CSAL: A cloud storage abstraction layer to enable portable cloud applications.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 504–511, 2010.

[112] C. A. R. Hoare. “Communicating sequential processes.” *Comm. of the ACM*, 21(8):666–677, 1978.

[113] D. H. Hu, Y. Wang, and C.-L. Wang. “BetterLife 2.0: Large-scale social intelligence reasoning on cloud.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 529–536, 2010.

[114] A. W. Hübler. “Understanding complex systems,” *Complexity,* **12**(5):9–11, 2007.

[115] K. Hwang, G. Fox, and J. Dongarra. *Distributed and Cloud Computing,* Morgan-Kaufmann Publishers, 2011.

[116] J. Tate, F. Luchese, and R. Moore. “Introduction to storage area networks.” *ibm.com/redbooks,* 2006.

[117] IBM. “Tivoli performance analyzer.” *www.ibm.com/software/tivoli/products/performance-analyzer,* 2008.

[118] IBM. “Dispelling the vapor around the cloud computing. Drivers, barriers and considerations for public and private cloud adoption.” *IBM Smart Business. Thought Leadership White Paper, 2010. ftp.software.ibm.com/common/ssi/ecm/en/ciw03062usen/CIW03062USEN.PDF.*

[119] IBM. “General parallel file systems (version 3, update 4). Documentation Updates.” *http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.gpfs.doc.*

[120] IBM. “The evolution of storage ststems.” *IBM Research. Almaden Research Center Publications. http://www.almaden.ibm.com/storagesystems/pubs.*

[121] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, N. J. Wright. “Performance analysis of vhigh performance computing applications on the Amazon Web services cloud.” *Proc. IEEE Second Int. Conf. on Cloud Computing Technology and Science,* pp. 159–168, 2010.

[122] M. Jelasity, A. Montresor, and O. Babaoglu. “Gossip-based aggregation in large dynamic networks.” *ACM Transactions on Computer Systems,* 23(3):219252, 2005.

[123] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. “Gossip-based peer sampling.” *ACM Trans. Comput. Syst.,* **25**(3):8, 2007.

[124] M. Jensen, S. Schäge, and J. Schwenk. “Towards an anonymous access control and accountability scheme for cloud computing.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 540–541, 2010.

[125] H. Jin, X.-H. Sun, Y. Chen, and T. Ke. “REMEM: REmote MEMory as checkpointing storage.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 319–326, 2010.

[126] E. Kalyvianaki, T. Charalambous, and S. Hand. “Self-adaptive and self-configured CPU resource provisionong for virtualized servers using Kalman filters.” *Proc. 6th Int. Conf. Autonomic Comp. (ICAC2009),* pp. 117–126, 2009.

[127] 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 (FeBid),* 2008.

[128] K. Kc and K. Anyanwu. “Scheduling Hadoop jobs to meet deadlines.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 388–392, 2010.

[129] 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 (ICAC2007),* pp. 100-109, 2007.

[130] J. Kephart. “The utility of utility.” *Proc. Policy 2011*, 2011.

[131] 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. on Cloud Computing*, pp. 450–457, 2010.

[132] J. Kim, W.J. Dally, and D. Abts. “Flattened butterfly: a cost-efficient topology for high-radix networks.” *Proc. Int. Symp. on Comp. Arch. (ISCA),* pp. 126–137, 2007.

[133] 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.(OFC/NFOEC),* pp. 1–3, 2010.

[134] 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. on Cloud Computing Technology and Science,* pp. 687–692, 2010.

[135] B. Koley, V. Vusirikala, C. Lam, and V. Gill. “100Gb Ethernet and beyong for warehouse scale computing.” *Proc. 15th OptoElectronics and Com. Conf, (OECC2010),* pp. 106–107, 2010.

[136] A. N. Kolmogorov. “Three approaches to the quantitative definition of information.” *Problemy Peredachy Informatzii*, **1**:4-7, 1965.

[137] 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. on Cloud Computing Technology and Science,* pp. 49–58, 2010.

[138] 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 Comp. (ICAC2008)*, pp. 3–12, 2008.

[139] C. Labovitz et. al. “ATLAS Internet Observatory 2009 Annual Report.” *http://www.nanog.org/meetings/nanog47/presentations*

[140] L. Lamport and P. M. Melliar-Smith. “Synchronizing clocks in the presence of faults.” *Journal of the ACM*, 32(1):52–78, 1985.

[141] L. Lamport. “The part-time parliament.” *ACM Trans. on Computer Systems* **2**:133–169, 1998.

[142] L. Lamport. “Paxos made simple.” *ACM SIGACT News* **32**(4):51–58, 2001.

[143] C. F.Lam. “FTTH look ahead - Technologies and architectures.” *Proc. 36th European Conf. on Optical Communications (ECOC’10)*, pp. 1–18, 2010.

[144] D. S. Lee, K. I. Goh, B. Kahng, and D. Kim. “Evolution of scale-free random graphs: Potts model formulation.” *Nuclear Physics B.* 696:351–380, 2004.

[145] *http://linux-vserver.org*

[146] Z. Li, N.-H. Yu, and Z. Hao. “A novel parallel traffic control mechanism for cloud computing.” *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 376–382, 2010.

[147] C. Li, A. Raghunathan, and N. K.Jha. "Secure virtual machine execution under an untrusted management OS." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 172–179, 2010.

[148] H C. Lim, S. Babu, J. S. Chase, S. S. Parekh. "Automated control in cloud computing: challenges and opportunities." *Proc. First Workshop on Automated Control for Datacenters and Clouds,*, ACM Press, pp. 13-18, 2009.

[149] 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.

[150] C. Lin and D. C. Marinescu. "Stochastic high level Petri Nets and applications." *IEEE Transactions on Computers*, C-37, **7**:815–825, 1988.

[151] S. Liu, G. Quan, and S. Ren. "On-line scheduling of real-time services for cloud computing." *Proc. SERVICES'2010*, pp.459–464, 2010.

[152] 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. on Cloud Computing Technology and Science,* pp. 143–150, 2010.

[153] C. Lu, J. Stankovic, G.Tao, and S. Son. "Feedback control real-time schedul-ing: framework, modeling and algorithms." *Journal of Real-time Systems*, **23**(1-2):85-126, 2002.

[154] 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. on Cloud Computing Technology and Science,* pp. 209–217, 2010.

[155] A. Luckow and S. Jha. "Abstractions for loosely-coupled and ensemble-based simulations on Azure." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 550–556, 2010.

[156] J. Machta. "Complexity, parallel computation, and statistical physics," *Complexity,* **11**(5):46–64, 2006.

[157] 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.*

[158] 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.

[159] D. C. Marinescu, C. Yu, and G. M. Marinescu. "Scale-free, self-organizing very large sensor networks." Journal of Parallel and Distributed Computing (JPDC), **50**(5):612-622, 2010.

[160] von der Marlsburg, C. "Network Self-organization." In *An Introduction to Neural and Electronic Networks.* S. Zonetzer, J. L. Davis, and C.Lau (Eds.), 421-432, Academic Press, San Diego, CA, 1995.

[161] M. Rodriguez-Martinez, J. Seguel, and M. Greer. "Open source cloud computing tools: A case study with a weather application." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 443–449, 2010.

[162] F. Mattern. "Virtual time and global states of distributed systems." *Proc. Int. Workshop on Parallel & Distributed Algorithms*, Elsevier, New York, pp. 215–226, 1989.

[163] M. Mazzucco, D. Dyachuk, and R. Deters. "Maximizing cloud providers revenues via energy aware allocation policies." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 131–138, 2010.

[164] P. McKenney. "On the efficient implementation of fair queuing." *Internetworking: Research and Experience*, **2**: 113-131, 1991.

[165] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. "Diagnosing performance overheads in Xen virtual machine environments." *Proc. First ACM/USENIX Conf. on Virtual Execution Environments,* 2005.

[166] A. Menon, A. L. Cox, and W. Zwaenepoel. "Optimizing network virtualization in Xen." *Proc. 2006 USENIX Annual Technical Conf.,* pp. 15–28, 2006. Also, *http://www.usenix.org/event/usenix06/tech/menon/menon_html/paper.html.*

[167] N. Metropolis, A. W. Rosenbluth, A. Teller, and E.Teller. "Equation of state calculations by fast computing machines." *J. of Chemical Physics,* **21**(6):1097–1092, 1953.

[168] R. Milner. "Lectures on a calculus for communicating systems." *Lecture Notes in Computer Science*, Vol. 197, Springer Verlag, Heidelberg, 1984.

[169] T. Miyamoto, M. Hayashi, and K. Nishimura. "Sustainable network resource management system for virtula private clouds." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 512–520, 2010.

[170] 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. Database Systems for Advanced Applications, DAS-FAA* 703–714, 2007.

[171] R. J. T. Morris and B. J. Truskowski. "The evolution of storage systems." *IBM Systems Journal*, **42**(2):205–217, 2003.

[172] J. Nagle. "On packet switches with infinite storage." *IEEE Transactions on Communications,* **35**(4):435-438, 1987.

[173] M. E. J. Newman. "The structure of scientific collaboration networks." *Proc. Nat. Academy of Science,* **98**(2):404–409, 2001.

[174] A. J. Nicholson, S. Wolchok, and B. D. Noble. "Juggler: virtual networks for fun and profit." *IEEE Trans. Mobile Computing*, **9**(1):31–43, 2010.

[175] NIST. "Cloud architecture reference models: A survey." Document *NIST CCRATWG 004*, v2, pps.32, 2011. Also, *http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[176] NIST–Reference Architecture Analysis Team. "Cloud computing reference architecture - Straw man model V2." Document *NIST CCRATWG 0028*, pps. 8, 2011.

[177] NIST–ITLCCP. "NIST cloud computing reference architecture. (version 1)" *NIST – Information Technology Laboratory Cloud Computing Program,* 2011.

[178] NIST. "Cloud specific terms and definitions." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/..*

[179] NIST. "Basic security functional areas." *NIST Cloud Computing Collaboration Site. NIST Reference Architecture - Strawman Model, http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[180] NIST. "Cloud security services architecture." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[181] NIST. "Threat sources by cloud architecture component." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[182] NIST. "General cloud environments - SWG." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[183] NIST. "Threat analysis of cloud services (initial thoughts for discussion)." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[184] NIST. "Top 10 cloud security concerns (Working list)." *NIST Cloud Computing Collaboration Site. http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing.*

[185] 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. on Cluster Computing and the Grid,* 124-131, 2009.

[186] S. Oikawa and R. Rajkumar. "Portable RK: A portable resource kernel for guaranteed and enfoced timing behavior." In *Proc. IEEE Real Time Technology and Applications Symp.*, pp. 111–120, June 1999.

[187] T. Okuda, E. Kawai, and S. Yamaguchi. "A mechanism of flexible memory exchange in cloud computing environments." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 75–80, 2010.

[188] 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*

[189] *http://wiki.openvz.org*

[190] A. M. Oprescu and T. Kielmann. "Bag-of-tasks scheduling under budget constraints." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 351–359, 2010.

[191] Oracle Corporation. "Lustre (file system)." *http://en.wikipedia.org/wiki/Lustre _(file_system)*

[192] OSA. "SP-011: Cloud computing pattern." *http://www.opensecurityarchitecture.org/ cms/library/patternlandscape/251-pattern-cloud-computing*

[193] 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. on Cloud Computing Technology and Science,* pp. 621–628, 2010.

[194] G. Pacifici, M. Spreitzer, A. N. Tantawi, and A. Youssef. "Peerformance management for cluster-based Web services." *IEEE J. Selected Areas in Communications,* **23**(12):2333–2343, 2005.

[195] 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*, 2007. Also, *http://www.hpl.hp.com/techreports/2007/HPL-2007-59R1.pdf*

[196] S. L. Pallickara, S. Pallickara, M. Zupanski, and S. Sullivan. "Efficient metadata generation to enable interactive data discovery over large-scale scientific data collections." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 573–580, 2010.

[197] Y. Pan, S. Maini, and E. Blevis. "Framing the issues of cloud computing and sustaunability: A design perspective." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 603–608, 2010.

[198] AMK Pathan and R. Buya. "A taxonomy of content delivery networks." *http://cloudbus.org/reports/CDN-Taxonomy.pdf*

[199] C. A. Petri. "Kommunikation mit Automaten." *Schriften des Institutes fur Instrumentelle Mathematik,* Bonn, 1962.

[200] C. A. Petri. "Concurrency theory." *Lecture Notes in Computer Science*, Vol. 254, pp. 4–24. Springer–Verlag, Heidelberg, 1987.

[201] G. j. Popek and R. P. Golberg. "Formal requirements for virtualizable third generation architecture." *Communications of the ACM*, **17**(7):412-421, 1974.

[202] C. Preist and P. Shabajee. "Energy use in the media cloud." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 581–586, 2010.

[203] D. Price and A. Tucker. "Solaris Zones: operating systems support for consolidating commercial workloads." *Proc. Large Installation System Administration*, USENIX, pp. 241–254, 2004.

[204] 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. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 51–58, 2010.

[205] 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).

[206] A. Ranabahu and A. Sheth. "Semantics centric solutions for application and data portability in cloud computing." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 234–241, 2010.

[207] N. Regola and J.-C. Ducom. "Recommendations for virtualization technologies in high performance computing." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 409–416, 2010.

[208] 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,* pp.65–79, July/August 2010. *http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/36575.pdf*

[209] L. M. Riungu, O. Taipale, and K. Smolander. "Research issues for software testing in the cloud." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 557–564, 2010.

[210] J. Rolia, L. Cerkasova, M. Arlit, and A. Andrzejak. "A capacity management service for resource pools." *Proc. 2nd Symp. on Software and Performance,* pp. 224–237, 2005.

[211] M. Rosenblum and T. Garfinkel. "Virtual machine monitors: Current technology and future trends." *Computer,* **38**(5): 39–47, 2005.

[212] T. L. Ruthkoski. "Exploratory project: State of the cloud, from University of Michigan and beyond." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 427–432, 2010.

[213] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design.* Morgan Kaufamnn, N.Y., 2009.

[214] 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. on Networked SystemsDesign and Implementation (NSDI'11)*, pps. 14, 2011.

[215] T. Sandholm and K. Lai. "Dynamic proportional share scheduling in Hadoop." *Proc. JSSPP 10, 15th Workshop on Job Scheduling Strategies for Parallel Processing,* 2010.

[216] F. Schmuck and R. Haskin. "GFPS: A shared-disk file system for large computing clusters." *Proc. Conf. on File and Storage Technologies (FAST'02)*, pp. 231–244, 2002.

[217] S. Scott, D. Abts, J. Kim, and W. J. Dally. "The Blackwidow highradix Clos network." *Proc. Int. Symp. on Computer Architecture (ISCA),* pp. 16-28, 2006.

[218] 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 Conf. on Security (USENIX Security'10),* pp. 1-11, 2010.

[219] P. Sempolinski and D. Thain. "A comparison and critique of Eucalyptu, OpenNebula and Nimbus." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 417–426, 2010.

[220] I. Sholtes, J. Botev, A. Höhfeld, H. Schloss, and M. Esch. "Awareness-driven phase transitions in very large scale distributed systems," *Proc. SASO-08, Second IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems,* IEEE Press, pp. 25–34, 2008.

[221] I. Scholtes. " Distributed creation and adaptation of random scale-free overlay networks." *Proc. Fourth IEEE Int. Conf. of Self-Adaptive and Self-Organizing Systems, SASO-10*, 2010.

[222] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu. "Storage management in virtualized cloud environment." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 204–211, 2010.

[223] J. E. Smith and R. Nair. "The architecture of virtual machines." *Computer,* **38**(5): 32–38, 2005.

[224] 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.

[225] T. Stanley, T. Close, and M. S. Miller. "Causeway: A message-orientd distributed debugger." *Technical Report HPL-2009-78*, pps. 14, 2009. Also, *http://www.hpl.hp.com/techreports*

[226] M. Stecca and M. Maresca. "An architecture for a mashup container in vizualized environments." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 386–393, 2010.

[227] P. Stingley. "Cloud architecture." *http://sites.google.com/site/cloudarchitecture/*

[228] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. "Using a market economy to provision compute resources across planet-wide clusters." *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS 2009)*, pp. 1–8, 2009.

[229] "Only 18% using adaptive streaming, says Skyfire report." *http://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=79393*

[230] J. Sugerman, G. Venkitachalam, and B. Lim. "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor." *Proc. USENIX Conf.*, pp. 70–85 2001.

[231] SUN Microsystems. "Introduction to cloud computing architecture." (White Paper, June 2009). *http://docs.google.com*

[232] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. M. Vin. "Application performance in the QLinux multimedia operating system." In *Proc. 8th ACM Conf. on Multimedia*, Nov. 2000.

[233] M. Steinder, I. Walley, and D. Chess. "Server virtualization in autonomic management of heterogeneous workloads." *SIGOPS Oper. Sys. Rev.,* **42**(1):94–95, 2008.

[234] D. Tancock, S. Pearson, and A. Charlesworth. "A privacy impact assessment tool for cloud computing." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 667–674, 2010.

[235] L. Tang, J. Dong, Y. Zhao, and L.-J. Zhang. "Enterprise cloud service architecture." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 27–34, 2010.

[236] J. Tate, F. Lucchese, and R. Moore. *Introduction to Storage Area Networks.* IBM Redbooks, 2006. *http://www.redbooks.ibm.com/redbooks/pdfs/sg245470.pdf*

[237] D. Tennenhouse. "Layered multiplexing considered harmful." In *Protocols for High-Speed Networks*, H. Rudin and R. C. Williamson (Eds.), pp. 143–148, North Holland, 1989.

[238] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. " A hybrid reinforcement learning approach to autonomic resource allocation." *Proc. Int Conf. on Autonomic Computing, ICAC-06*, pp. 65-73, 2006.

[239] J. Timmermans, V. Ikonen, B. C. Stahl, and E. Bozdag. "The ethics of cloud computing. A conceptual review" *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 614–620, 2010.

[240] Z. Toroczkai and K. E. Bassler. "Jamming is limited in scale-free systems." *Nature,* **428**:716, 2004.

[241] C. Tung, M. Steinder, M.!Spreitzer, and G. Pacifici. "A scalable application placement controller for enterprise data centers." *Proc. 16th Int. Conf. World Wide Web (WWW2007),* 2007.

[242] A. M. Turing. "On computable numbers, with an application to the Entscheidungsproblem." *Proc. London Math. Soc.*, **2**(42): 230-265, 193637 and "On computable numbers, with an application to the Entscheidungsproblem: A correction," *Proc. London Math. Soc.*, **43**: 544-546. 1937.

[243] H. N. Van, F. D. Tran, and J.-M. Menaud. "Performance and power management for cloud infrastructures." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 329–336, 2010.

[244] J. Varia. "Cloud architectures." *http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf*

[245] 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.

[246] S. V. Vrbsky, M. Lei, K. Smith, and J. Byrd. "Data replication and power consumption in data grids." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 288–295, 2010.

[247] 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 3rd Int. Conf. on Cloud Computing*, pp. 164–171, 2010.

[248] 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 Intl. Conf. on Autonomic Computing,* pp. 165–174, 2006.

[249] D. J. Watts and S. H. Strogatz. "Collective-dynamics of small-world networks," *Nature,* **393**:440–442, 1998.

[250] J. Webster. "Evaluating IBM's SVC and TPC for server virtualization." IBM Evaluator Group, 2010. *ftp://ftp.boulder.ibm.com/software/at/tivoli/analyst_paper_ibm_svc_tpc.pdf*

[251] 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.

[252] V. Winkler. *Securing the cloud: cloud computer security techniques and tactics.* Elsevier Science and Technologies Books, 2011.

[253] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker. "Scalable thread scheduling and global power management for heterogeneous many-core architectures." *9th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'10)*, pp. 29–40, 2010.

[254] E. C. Withana and B. Plale. "Usage patterns to provosion for scientific experimentation in clouds." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 226–233, 2010.

[255] M. Witkowski, P. Brenner, R. Jansen, D. B. Go, and E. Ward. "Enabling sustainable clouds via environmentally opportunistic computing." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 587–592, 2010.

[256] D. H. Wolpert and W. Macready. "Using self-dissimilarity to qunatify complexity," *Complexity,* **12**(3):77–85, 2007.

[257] Xen Wiki. http://wiki.xensource.com/xenwiki/CreditScheduler, 2007.

[258] Z. Xiao and D. Cao. "A policy-based framework for automated SLA negotiation for internet-based virtual computing environment." *Proc. IEEE 16th Int. Conf. on Parallel and Distributed Systems,* pp. 694–699, 2010.

[259] S. Yi, D. Kondo, and A. Andrzejak. "Reducing costs of spot instances via checkpointing in the Amazon Elastic Compute Cloud." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 236–243, 2010.

[260] M. Zapf and A. Heinzl. "Evaluation of process design patterns: an experimental study." In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management*, *Lecture Notes on Computer Science*, Vol. 1806, pp. 83–98, Springer–Verlag, Heidelberg, 2000.

[261] 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 cheduling." *Proc. EuroSys 10 5th European Conf. Computer Systems,* pp. 265-278, 2010.

[262] 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.

[263] C. Zhang and H. D. Sterck. "CloudBATCH: A batch job queuing system on clouds with Hadoop and HBase." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 368–375, 2010.

[264] W. Zhao, P. M. Melliar-Smith, and L. E. Moser. "Fault tolerance middleware for cloud computing." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 67–74, 2010.

[265] X. Zhao, K. Borders, and A. Prakash. "Virtual machine security systems." In *Advances in Computer Science and Engineering*, pp. 339–365, (to appear). *http://www.eecs.umich.edu/ aprakash/eecs588/handouts/virtualmachinesecurity.pdf*

13 Glossary

13.1 General

Service consumer - person/organization that maintains a business relationship with, and uses service from, Service Providers.

Service provider - entity responsible for making a service available to service consumers.

Software as a Service (SaaS) - The capability to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does 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.

Platform as a Service (PaaS) - the capability to deploy onto the cloud infrastructure 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 and possibly application hosting environment configurations.

Infrastructure as a Service (IaaS) - the capability to provision processing, storage, networks, and other fundamental computing resources where 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 select networking components (e.g., host firewalls).

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 premise or off premise of the organization.

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 premise or off premise.

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).

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 Cloud Providers and Cloud Consumers.

Auditor - a party that can conduct independent assessment of cloud services, information system operations, performance and security of the cloud implementation.

Audits - 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-impact, performance audit if the criteria is performance.

Service Level Agreement (SLA) - A document explaining expected quality of service and legal guarantees.

SLA management - includes SLA contract definition (basic schema with the quality of service parameters), SLA monitoring, and SLA enforcement, according to the defined policies.

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.

Physical data container - a storage device suitable for transferring data between cloud-subscribers and clouds e.g., a hard disk; there has to be a standard format that the Provider supports (e.g., EIDE, IDE, SCSI). The physical data container must be formatted with a standard logical organization, such as FAT32, ufs, etc.

13.2 Virtualization

Virtualization - an abstraction or simulation of hardware resources. e.g., virtual memory.

Virtual machine (VM) - an isolated environment that appears to be a whole computer, but actually only has access to a portion of the computers resources.

Virtual machine monitor (VMM) - the software layer that supports one or more virtual machines.

Hypervisor or VMM - software that securely partitions the computers resources 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 machine.

Guest operating system - an operating system that runs on a VMM rather than directly on the hardware.

System virtual machine/hardware virtual machine - provide a complete system. Each VM can run its own OS, which in turn can run multiple applications. Several approaches:

- *Traditional - VM also called a "bare metal" VMM* - a thin software layer that runs directly on the host machine hardware; its main advantage is performance. Examples: VMWare ESX, ESXi Servers, Xen, OS370, Denali
- *Hosted* - run on top of an existing OS; main advantage: easier to build and easier to install. Examples: User-mode Linux
- *Hybrid* - shares the hardware with existing OS. Example: VMWare Workstation

Process or application virtual machine - VM runs under the control of a normal OS and provides a platform-independent host for a single application, e.g., Java Virtual Machine (JVM).

Full virtualization - each virtual machine runs on an exact copy of the actual hardware.

Paravirtualization - each virtual machine runs on a slightly modified copy of the actual hardware; the reasons for paravitualization: (i) some aspects of the hardware cannot be virtualized; (ii) to improve performance; (iii) to present a simpler interface.

13.3 Cloud Services

Service Deployment - all of the activities and organization needed to make a cloud service available

Service Orchestration - the arrangement, coordination and management of cloud infrastructure to provide different cloud services to meet IT and business requirements.

Service Management - all service-related functions necessary for the management and operations of those services required by customers.

Service intermediation - an intermediation broker provides a service that directly enhances a given service delivered to one or more service consumers.

Service aggregation - an aggregation brokerage service combines multiple services into one or more new services.

Service arbitrage - similar to cloud service aggregation; the difference is that the services being aggregated aren't 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.

Cloud distribution - the process of transporting cloud data between Providers and Cloud.

Mobile/Fixed endpoints - a physical device that provided a man/machine interface to cloud services and applications.

13.4 Layers and functions

Service layer - defines the basic services provided by cloud providers

Physical resource layer - includes all physical resources used to provide cloud services

Resource abstraction and control layer - software elements used to realize the infrastructure upon which a cloud service can be established, e.g., hypervisor, virtual machines, virtual data storage.

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.

Application layer - deployed software applications targeted towards end-user software clients or other programs, and made available via the cloud.

Platform architecture layer - entails compilers, libraries, utilities, and other software tools and development environments needed to implement applications.

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.

Facility layer - heating, ventilation, air conditioning (HVAC), power, communications, and other aspects of the physical plant comprise the lowest layer, the facility layer.

Provisioning/Configuration - preparing a cloud to allow it to provide (new) services to its users

Resource change - adjust configuration/resource assignment for repairs, upgrades, and joining new nodes into the cloud

Monitoring and reporting - discover and monitor the virtual resources, monitor cloud operations and events, and generate performance reports.

Metering - provide a measurement capability at some level of abstraction appropriate to the type of service (e.g, storage, processing, bandwidth, and active user accounts)

13.5 Desirable characteristics/attributes

Resilience - the ability to reduce the magnitude and/or duration of disruptive events to critical infrastructure. The effectiveness of a resilient infrastructure or enterprize depends upon its ability to anticipate, absorb, adapt to, and/or rapidly recover from a potentially disruptive event

Reliability - A measure of the ability of a functional unit to perform a required function under given conditions for a given time interval.

Maintainability - A measure of the ease with which maintenance of a functional unit can be performed using prescribed procedures and resources. Synonymous with serviceability.

Usability - The 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.

Privacy - the assured, proper, and consistent collection, processing, communication, use and disposition of disposition of personal information and personally-identifiable information throughout its life cycle.

Data portability - the ability to transfer data from one system to another without being required to recreate or reenter data descriptions or to modify significantly the application being transported.

System portability - the ability of a service to run on more than one type or size of cloud.

Service interoperability - the capability to communicate, execute programs, or transfer data among various cloud services under specified conditions.

Rapid provisioning - automatically deploying cloud system based on the requested service, resources, and capabilities.

Interoperability - capability to communicate, execute programs, or transfer data among various functional units under specified conditions.

13.6 Cloud security

Authentication credential - something that an entity is, has, or knows that allows an entity to prove its identity to a system.

Cloud-subscriber - an authenticated person that accesses a cloud system over a network. A cloud-subscriber may possess administrative privileges, such as the ability to manage virtual machines, or the ability to regulate access by users to cloud resources the cloud-subscriber controls.

Security assessment - assess the management, operational, and technical controls of the cloud system with frequency depending on risk, but no less than annually.

Security certification - a security certification is conducted for accrediting the cloud system; the security certification is a key factor in all security accreditation (i.e., authorization) decisions and is integrated into and spans the system development life cycle.

Security accreditation - the organization authorizes (i.e., accredits) the cloud system for processing before operations and updates the authorization or when there is a significant change to the system.

FISMA compliant environment - an environment that meets the requirements of the Federal Information Security Management Act of 2002. Specifically, 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.

Moderate impact - moderate impact refers to the impact levels defined in FIPS 199. A potential impact is " moderate if the loss of confidentiality, integrity, and availability could be expected to have a serious adverse effect on organizational operations, organizational assets, or individuals."

FedRAMP - FedRAMP allows 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 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.