GRID AND CLOUD COMPUTING (IV-CSE)

Unit I

SCALABLE COMPUTING OVER THE INTERNET

Over the past 60 years, computing technology has undergone a series of platform and environment changes. In this section, we assess evolutionary changes in machine architecture, operating system platform, network connectivity, and application workload. Instead of using a centralized computer to solve computational problems, a parallel and distributed computing system uses multiple computers to solve large-scale problems over the Internet. Thus, distributed computing becomes data-intensive and network-centric. This section identifies the applications of modern computer systems that practice parallel and distributed computing.
These large-scale Internet applications have significantly enhanced the quality of life and information services in society today.

1.1 The Age of Internet Computing

Billions of people use the Internet every day. As a result, supercomputer sites and large data centers must provide high-performance computing services to huge numbers of Internet users concurrently. Because of this high demand, the Linpack Benchmark for high-performance computing (HPC) applications is no longer optimal for measuring system performance. The emergence of computing clouds instead demands high-throughput computing (HTC) systems built with parallel and distributed computing technologies . We have to upgrade data centers using fast servers, storage systems, and high-bandwidth networks. The purpose is to advance network-based computing and web services with the emerging new technologies.

1.1.1 The Platform Evolution

Computer technology has gone through five generations of development, with each generation lasting from 10 to 20 years. Successive generations are overlapped in about 10 years. For instance, from 1950 to 1970, a handful of mainframes, including the IBM 360 and CDC 6400, were built to satisfy the demands of large businesses and government organizations. From 1960 to 1980, lower-cost minicomputers such as the DEC PDP 11 and VAX Series became popular among small businesses and on college campuses.From 1970 to 1990, we saw widespread use of personal computers built with VLSI microprocessors. From 1980 to 2000, massive numbers of portable computers and pervasive devices appeared in both wired and wireless applications. Since 1990, the use of both HPC and HTC systems hidden in clusters, grids, or Internet clouds has proliferated.These systems are employed by both consumers and high-end web-scale computing and information services.

The general computing trend is to leverage shared web resources and massive amounts of data over the Internet. The evolution of HPC and HTC systems. On the HPC side, supercomputers (massively parallel processors or MPPs) are gradually replaced by clusters of cooperative computers out of a desire to share computing resources. The cluster is often a collection of homogeneous compute nodes that are physically connected in close range to one another.

On the HTC side, peer-to-peer (P2P) networks are formed for distributed file sharing and content delivery applications. A P2P system is built over many client machines. Peer machines are globally distributed in nature. P2P, cloud computing, and web service platforms are more focused on HTC applications than on HPC applications. Clustering and P2P technologies lead to the development of computational grids or data grids.

1.1.2 High-Performance Computing

For many years, HPC systems emphasize the raw speed performance. The speed of HPC systems has increased from Gflops in the early 1990s to now Pflops in 2010. This improvement was driven mainly by the demands from scientific, engineering, and manufacturing communities. For example, the Top 500 most powerful computer systems in the world are measured by floating-point speed in Linpack benchmark results. However, the number of supercomputer users is limited to less than 10% of all computer users. Today, the majority of computer users are using desktop computers or large servers when they conduct Internet searches and market-driven computing tasks.

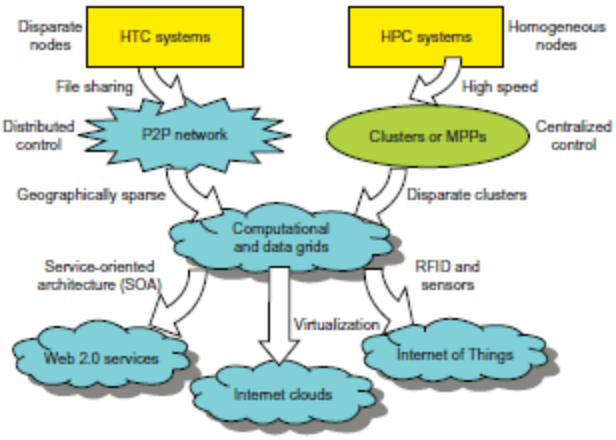


FIGURE 1.1
Evolutionary trend toward parallel, distributed, and cloud computing with clusters, MPPs, P2P networks, grids, clouds, web services, and the Internet of Things.

1.1.3 High-Throughput Computing

The development of market-oriented high-end computing systems is undergoing a strategic change from an HPC paradigm to an HTC paradigm. This HTC paradigm pays more attention to high-flux computing. The main application for high-flux computing is in Internet searches and

web services by millions or more users simultaneously. The performance goal thus shifts to measure high throughput or the number of tasks completed per unit of time. HTC technology needs to not only improve in terms of batch processing speed, but also address the acute problems of cost, energy savings, security, and reliability at many data and enterprise computing centers.

1.1.4 Three New Computing Paradigms

Advances in virtualization make it possible to see the growth of Internet clouds as a new computing paradigm. The maturity of radio-frequency identification (RFID), Global Positioning System (GPS), and sensor technologies has triggered the development of the Internet of Things (IoT). These new paradigms are only briefly introduced here. When the Internet was introduced in 1969, Leonard Klienrock of UCLA declared: ―As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of computer utilities, which like present electric and telephone utilities, will service individual homes and offices across the country.‖ Many people have redefined the term ―computer‖ since that time. In 1984, John Gage of Sun Microsystems created the slogan, ―The network is the computer.‖ In 2008,David Patterson of UC Berkeley said, ―The data center is the computer. There are dramatic differences between developing software for millions to use as a service versus distributing software to run on their PCs.‖ Recently, Rajkumar Buyya of Melbourne University simply said: ―The cloud is the computer.‖

In fact, the differences among clusters, grids, P2P systems, and clouds may blur in the future. Some people view clouds as grids or clusters with modest changes through virtualization. Others feel the changes could be major, since clouds are anticipated to process huge data sets generated by the traditional Internet, social networks, and the future IoT. In subsequent chapters, the distinctions and dependencies among all distributed and cloud systems models will become clearer and more transparent.

1.1.5 Computing Paradigm Distinctions

The high-technology community has argued for many years about the precise definitions of centralized computing, parallel computing, distributed computing, and cloud computing. In general, distributed computing is the opposite of centralized computing. The field of parallel computing overlaps with distributed computing to a great extent, and cloud computing overlaps with distributed, centralized, and parallel computing.

• Centralized computing This is a computing paradigm by which all computer resources are centralized in one physical system. All resources (processors, memory, and storage) are fully shared and tightly coupled within one integrated OS. Many data centers and supercomputers are centralized systems, but they are used in parallel, distributed, and cloud computing applications .

• Parallel computing In parallel computing, all processors are either tightly coupled with centralized shared memory or loosely coupled with distributed memory. Some authors refer to

this discipline as parallel processing . Interprocessor communication is accomplished through shared memory or via message passing. A computer system capable of parallel computing is commonly known as a parallel computer . Programs running in a parallel computer are called parallel programs.The process of writing parallel programs is often referred to as parallel programming.

• Distributed computing This is a field of computer science/engineering that studies distributed systems. A distributed system consists of multiple autonomous computers, each having its own private memory, communicating through a computer network. Information exchange in a distributed system is accomplished through message passing. A computer program that runs in a distributed system is known as a distributed program. The process of writing distributed programs is referred to as distributed programming.

• Cloud computing An Internet cloud of resources can be either a centralized or a distributed computing system. The cloud applies parallel or distributed computing, or both. Clouds can be built with physical or virtualized resources over large data centers that are centralized or distributed. Some authors consider cloud computing to be a form of utility computing or service computing .

As an alternative to the preceding terms, some in the high-tech community prefer the term concurrent computing or concurrent programming. These terms typically refer to the union of parallel computing and distributing computing, although biased practitioners may interpret them differently. Ubiquitous computing refers to computing with pervasive devices at any place and time using wired or wireless communication. The Internet of Things (IoT) is a networked connection of everyday objects including computers, sensors, humans, etc. The IoT is supported by Internet clouds to achieve ubiquitous computing with any object at any place and time.
Finally, the term Internet computing is even broader and covers all computing paradigms over the Internet.

1.1.6 Distributed System Families

Since the mid-1990s, technologies for building P2P networks and networks of clusters have been consolidated into many national projects designed to establish wide area computing infrastructures, known as computational grids or data grids. Recently, we have witnessed a surge in interest in exploring Internet cloud resources for data-intensive applications. Internet clouds are the result of moving desktop computing to service-oriented computing using server clusters and huge databases at data centers. Grids and clouds are disparity systems that place great emphasis on resource sharing in hardware, software, and data sets.

Design theory, enabling technologies, and case studies of these massively distributed systems are also covered in this book. Massively distributed systems are intended to exploit a high degree of parallelism or concurrency among many machines. In October 2010, the highest performing cluster machine was built in China with 86016 CPU processor cores and 3,211,264 GPU cores in a Tianhe-1A system. The largest computational grid connects up to hundreds of

server clusters. A typical P2P network may involve millions of client machines working simultaneously. Experimental cloud computing clusters have been built with thousands of processing nodes. In the future, both HPC and HTC systems will demand multicore or many-core processors that can handle large numbers of computing threads per core. Both HPC and HTC systems emphasize parallelism and distributed computing. Future HPC and HTC systems must be able to satisfy this huge demand in computing power in terms of throughput, efficiency, scalability, and reliability. The system efficiency is decided by speed, programming, and energy factors (i.e., throughput per watt of energy consumed).

Meeting these goals requires to yield the following design objectives:

• Efficiency measures the utilization rate of resources in an execution model by exploiting massive parallelism in HPC. For HTC, efficiency is more closely related to job throughput, data access, storage, and power efficiency.

• Dependability measures the reliability and self-management from the chip to the system and application levels. The purpose is to provide high-throughput service with Quality of Service (QoS) assurance, even under failure conditions.

• Adaptation in the programming model measures the ability to support billions of job requests over massive data sets and virtualized cloud resources under various workload and service models.

• Flexibility in application deployment measures the ability of distributed systems to run well in both HPC (science and engineering) and HTC (business) applications.

1.2 Scalable Computing Trends and New Paradigms

Several predictable trends in technology are known to drive computing applications. In fact, designers and programmers want to predict the technological capabilities of future systems. For instance, Jim Gray's paper, ‒Rules of Thumb in Data Engineering,| is an excellent example of how technology affects applications and vice versa. In addition,Moore's law indicates that processor speed doubles every 18 months. Although Moore's law has been proven valid over the last 30 years, it is difficult to say whether it will continue to be true in the future.

Gilder's law indicates that network bandwidth has doubled each year in the past. Will that trend continue in the future? The tremendous price/performance ratio of commodity hardware was driven by the desktop, notebook, and tablet computing markets. This has also driven the adoption and use of commodity technologies in large-scale computing.

For now, it's important to understand how distributed systems emphasize both resource distribution and concurrency or high degree of parallelism (DoP). Let's review the degrees of parallelism before we discuss the special requirements for distributed computing.

1.2.1 Degrees of Parallelism

Fifty years ago, when hardware was bulky and expensive, most computers were designed in a bit-serial fashion. In this scenario, bit-level parallelism (BLP) converts bit-serial processing to word-level processing gradually. Over the years, users graduated from 4-bit microprocessors to 8-, 16-, 32-, and 64-bit CPUs. This led us to the next wave of improvement, known as instruction-level parallelism (ILP), in which the processor executes multiple instructions simultaneously rather than only one instruction at a time. For the past 30 years, we have practiced ILP through pipelining, superscalar computing, VLIW (very long instruction word) architectures, and multithreading. ILP requires branch prediction, dynamic scheduling, speculation, and compiler support to work efficiently.

Data-level parallelism (DLP) was made popular through SIMD (single instruction, multiple data) and vector machines using vector or array types of instructions. DLP requires even more hardware support and compiler assistance to work properly. Ever since the introduction of multicore processors and chip multiprocessors (CMPs), we have been exploring task-level parallelism (TLP). A modern processor explores all of the aforementioned parallelism types. In fact, BLP, ILP, and DLP are well supported by advances in hardware and compilers.
However, TLP is far from being very successful due to difficulty in programming and compilation of code for efficient execution on multicore CMPs. As we move from parallel processing to distributed processing, we will see an increase in computing granularity to job-level parallelism (JLP). It is fair to say that coarse-grain parallelism is built on top of fine-grain parallelism.

1.2.2 Innovative Applications

Both HPC and HTC systems desire transparency in many application aspects. For example, data access, resource allocation, process location, concurrency in execution, job replication, and failure recovery should be made transparent to both users and system management. It highlights a few key applications that have driven the development of parallel and distributed systems over the years. These applications spread across many important domains in science, engineering, business, education, health care, traffic control, Internet and web services, military, and government applications.

Table 1.1 Applications of High-Performance and High-Throughput Systems

Domain	Specific Applications
Science and engineering	Scientific simulations, genomic analysis, etc. Earthquake prediction, global warming, weather forecasting, etc.
Business, education, services industry, and health care	Telecommunication, content delivery, e-commerce, etc. Banking, stock exchanges, transaction processing, etc. Air traffic control, electric power grids, distance education, etc. Health care, hospital automation, telemedicine, etc.
Internet and web services, and government applications	Internet search, data centers, decision-making systems, etc. Traffic monitoring, worm containment, cyber security, etc. Digital government, online tax return processing, social networking, etc.
Mission-critical applications	Military command and control, intelligent systems, crisis management, etc.

Applications of High-Performance and High-Throughput Systems Almost all applications demand computing economics, web-scale data collection, system reliability, and scalable

performance. For example, distributed transaction processing is often practiced in the banking and finance industry. Transactions represent 90 percent of the existing market for reliable banking systems. Users must deal with multiple database servers in distributed transactions. Maintaining the consistency of replicated transaction records is crucial in real-time banking services. Other complications include lack of software support, network saturation, and security threats in these applications.

1.2.3 The Trend Toward Utility Computing

It identifies major computing paradigms to facilitate the study of distributed systems and their applications. These paradigms share some common characteristics. First, they are all ubiquitous in daily life. Reliability and scalability are two major design objectives in these computing models. Second, they are aimed at autonomic operations that can be self-organized to support dynamic discovery. Finally, these paradigms are composable with QoS and SLAs (service-level agreements). These paradigms and their attributes realize the computer utility vision.

Utility computing focuses on a business model in which customers receive computing resources from a paid service provider. All grid/cloud platforms are regarded as utility service providers. However, cloud computing offers a broader concept than utility computing. Distributedcloud applications run on any available servers in some edge networks. Major technological challenges include all aspects of computer science and engineering. For example, users demand new network-efficient processors, scalable memory and storage schemes, distributed OSes, middleware for machine virtualization, new programming models, effective resource management, and application program development. These hardware and software supports are necessary to build distributed systems that explore massive parallelism at all processing levels.

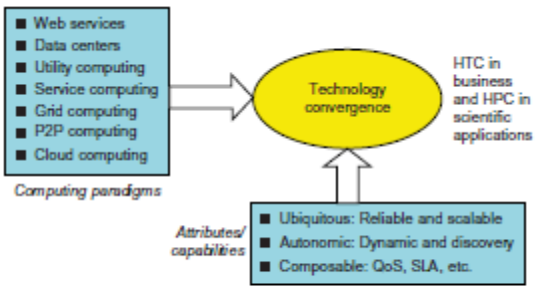


FIGURE 1.2
The vision of computer utilities in modern distributed computing systems.
(Modified from presentation slide by Raj Buyya, 2010)

1.2.4 The Hype Cycle of New Technologies

Any new and emerging computing and information technology may go through a hype cycle. This cycle shows the expectations for the technology at five different stages. The expectations rise sharply from the trigger period to a high peak of inflated expectations. Through a short period of disillusionment, the expectation may drop to a valley and then increase steadily over a long enlightenment period to a plateau of productivity. The number of years for an emerging

technology to reach a certain stage is marked by special symbols. The hollow circles indicate technologies that will reach mainstream adoption in two years. The gray circles represent technologies that will reach mainstream adoption in two to five years. The solid circles represent those that require five to 10 years to reach mainstream adoption, and the triangles denote those that require more than 10 years. The crossed circles represent technologies that will become obsolete before they reach the plateau.

Hype Cycles are graphical representations of the relative maturity of technologies, IT methodologies and management disciplines. They are intended solely as a research tool, and not as a specific guide to action. Gartner disclaims all warranties, express or implied, with respect to this research, including any warranties of merchantability or fitness for a particular purpose.

This Hype Cycle graphic was published by Gartner, Inc. as part of a larger research note and should be evaluated in the context of the entire report.

1.3 The Internet of Things and Cyber-Physical Systems

In this section, we will discuss two Internet development trends: the Internet of Things and cyber-physical systems. These evolutionary trends emphasize the extension of the Internet to everyday objects. We will only cover the basics of these concepts here;

1.3.1 The Internet of Things

The traditional Internet connects machines to machines or web pages to web pages. The concept of the IoT was introduced in 1999 at MIT . The IoT refers to the networked interconnection of everyday objects, tools, devices, or computers. One can view the IoT as a wireless network of sensors that interconnect all things in our daily life. These things can be large or small and they vary with respect to time and place. The idea is to tag every object using RFID or a related sensor or electronic technology such as GPS.

With the introduction of the IPv6 protocol, 2128 IP addresses are available to distinguish all the objects on Earth, including all computers and pervasive devices. The IoT researchers have estimated that every human being will be surrounded by 1,000 to 5,000 objects. The IoT needs to be designed to track 100 trillion static or moving objects simultaneously. The IoT demands universal addressability of all of the objects or things.

To reduce the complexity of identification, search, and storage, one can set the threshold to filter out fine-grain objects. The IoT obviously extends the Internet and is more heavily developed in Asia and European countries. In the IoT era, all objects and devices are instrumented, interconnected, and interacted with each other intelligently. This communication can be made between people and things or among the things themselves. Three communication patterns co-exist: namely H2H (human-to-human), H2T (human-to-thing), and T2T (thing-to-thing). Here things include machines such as PCs and mobile phones. The idea here is to connect things (including human and machine objects) at any time and any place intelligently with low cost.
Any place connections include at the PC, indoor (away from PC), outdoors, and on the move.

Any time connections include daytime, night, outdoors and indoors, and on the move as well.

The dynamic connections will grow exponentially into a new dynamic network of networks, called the Internet of Things (IoT). The IoT is still in its infancy stage of development. Many prototype IoTs with restricted areas of coverage are under experimentation at the time of this writing. Cloud computing researchers expect to use the cloud and future Internet technologies to support fast, efficient, and intelligent interactions among humans, machines, and any objects on Earth. A smart Earth should have intelligent cities, clean water, efficient power, convenient transportation, good food supplies, responsible banks, fast telecommunications, green IT, better schools, good health care, abundant resources, and so on. This dream living environment may take some time to reach fruition at different parts of the world.

1.3.2 Cyber-Physical Systems

A cyber-physical system (CPS) is the result of interaction between computational processes and the physical world. A CPS integrates ―cyber‖ (heterogeneous, asynchronous) with ―physical‖ (concurrent and information-dense) objects. A CPS merges the ―3C‖ technologies of computation, communication, and control into an intelligent closed feedback system between the physical world and the information world, a concept which is actively explored in the United States. The IoT emphasizes various networking connections among physical objects, while the CPS emphasizes exploration of virtual reality (VR) applications in the physical world. We may transform how we interact with the physical world just like the Internet transformed how we interact with the virtual world.

TECHNOLOGIES FOR NETWORK-BASED SYSTEMS

With the concept of scalable computing under our belt, it's time to explore hardware, software, and network technologies for distributed computing system design and applications. In particular, we will focus on viable approaches to building distributed operating systems for handling massive parallelism in a distributed environment.

Multicore CPUs and Multithreading Technologies

Consider the growth of component and network technologies over the past 30 years. They are crucial to the development of HPC and HTC systems. processor speed is measured in millions of instructions per second (MIPS) and network bandwidth is measured in megabits per second (Mbps) or gigabits per second (Gbps). The unit GE refers to 1 Gbps Ethernet bandwidth.

Advances in CPU Processors

Today, advanced CPUs or microprocessor chips assume a multicore architecture with dual, quad, six, or more processing cores. These processors exploit parallelism at ILP and TLP levels.
Processor speed growth is plotted in the upper curve across generations of microprocessors or CMPs. We see growth from 1 MIPS for the VAX 780 in 1978 to 1,800 MIPS for the Intel

Pentium 4 in 2002, up to a 22,000 MIPS peak for the Sun Niagara 2 in 2008. As the figure shows, Moore's law has proven to be pretty accurate in this case. The clock rate for these processors increased from 10 MHz for the Intel 286 to 4GHz for the Pentium 4 in 30 years.

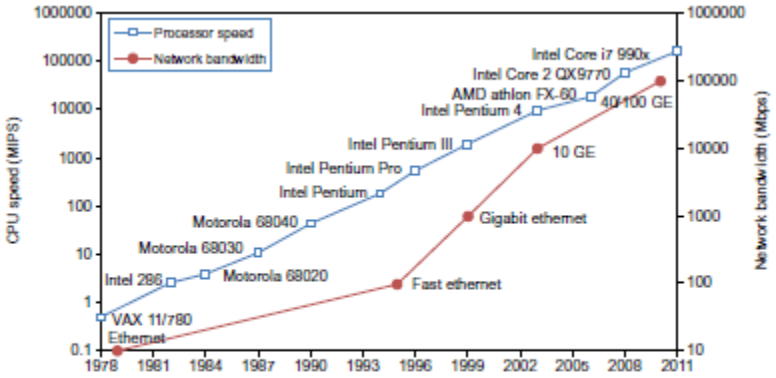


FIGURE 1.4

Improvement in processor and network technologies over 33 years.

(Courtesy of Xiaosong Lou and Lizhong Chen of University of Southern California, 2011)

However, the clock rate reached its limit on CMOS-based chips due to power limitations. At the time of this writing, very few CPU chips run with a clock rate exceeding 5 GHz. In other words, clock rate will not continue to improve unless chip technology matures. This limitation is attributed primarily to excessive heat generation with high frequency or high voltages. The ILP is highly exploited in modern CPU processors. ILP mechanisms include multiple-issue superscalar architecture, dynamic branch prediction, and speculative execution, among others. These ILP techniques demand hardware and compiler support. In addition, DLP and TLP are highly explored in graphics processing units (GPUs) that adopt a many-core architecture with hundreds to thousands of simple cores. Both multi-core CPU and many-core GPU processors can handle multiple instruction threads at different magnitudes today. The architecture of a typical multicore processor. Each core is essentially a processor with its own private cache (L1 cache). Multiple cores are housed in the same chip with an L2 cache that is shared by all cores. In the future, multiple CMPs could be built on the same CPU chip with even the L3 cache on the chip. Multicore and multithreaded CPUs are equipped with many high-end processors, including the Intel i7, Xeon, AMD Opteron, Sun Niagara, IBM Power 6, and X cell processors. Each core could be also multithreaded. For example, the Niagara II is built with eight cores with eight threads handled by each core. This implies that the maximum ILP and TLP that can be exploited in Niagara is 64 (8 × 8 = 64). In 2011, the Intel Core i7 990x has reported 159,000 MIPS execution rate as shown in the upper- most square .

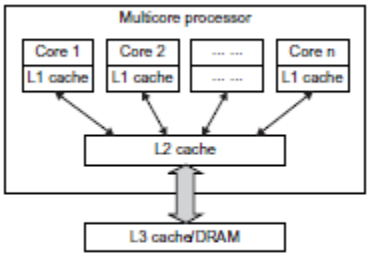


FIGURE 1.5

Schematic of a modern multicore CPU chip using a hierarchy of caches, where L1 cache is private to each core, on-chip L2 cache is shared and L3 cache or DRAM is off the chip.

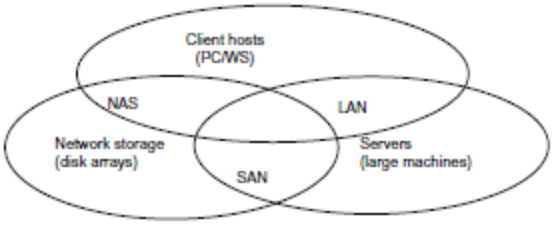


FIGURE 1.11

Three interconnection networks for connecting servers, client hosts, and storage devices; the LAN connects client hosts and servers, the SAN connects servers with disk arrays, and the NAS connects clients with large storage systems in the network environment.

Schematic of a modern multicore CPU chip using a hierarchy of caches, where L1 cache is private to each core, on-chip L2 cache is shared and L3 cache or DRAM Is off the chip.

Multicore CPU and Many-Core GPU Architectures

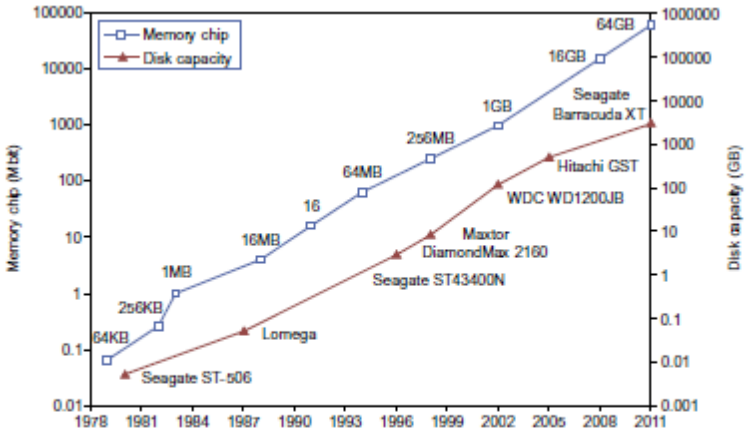


FIGURE 1.10

Improvement in memory and disk technologies over 33 years. The Seagate Barracuda XT disk has a capacity of 3 TB in 2011.

Multicore CPUs may increase from the tens of cores to hundreds or more in the future.But the CPU has reached its limit in terms of exploiting massive DLP due to the aforementioned memory wall problem. This has triggered the development of many-core GPUs with hundreds or more thin cores. Both IA-32 and IA-64 instruction set architectures are built into commercial CPUs. Now, x-86 processors have been extended to serve HPC and HTC systems in some high-end server processors. Many RISC processors have been replaced with multicore x-86

processors and many-core GPUs in the Top 500 systems. This trend indicates that x-86 upgrades will dominate in data centers and supercomputers. The GPU also has been applied in large clusters to build supercomputers in MPPs. In the future, the processor industry is also keen to develop asymmetric or heterogeneous chip multiprocessors that can house both fat CPU cores and thin GPU cores on the same chip.

Multithreading Technology

Consider the dispatch of five independent threads of instructions to four pipelined data paths (functional units) in each of the following five processor categories, from left to right: a four-issue superscalar processor, a fine-grain multithreaded processor, a coarse-grain multithreaded processor, a two-core CMP, and a simultaneous multithreaded (SMT) processor. The superscalar processor is single-threaded with four functional units. Each of the three multithreaded processors is four-way multithreaded over four functional data paths. In the dual-core processor, assume two processing cores, each a single-threaded two-way superscalar processor.

Five micro-architectures in modern CPU processors, that exploit ILP and TLP supported by multicore and multithreading technologies. Instructions from different threads are distinguished by specific shading patterns for instructions from five independent threads. Typical instruction scheduling patterns are

shown here. Only instructions from the same thread are executed in a superscalar processor. Fine-grain multithreading switches the execution of instructions from different threads per cycle. Course-grain multithreading executes many instructions from the same thread for quite a few cycles before switching to another thread. The multicore CMP executes instructions from different threads completely. The SMT allows simultaneous scheduling of instructions from different threads in the same cycle.

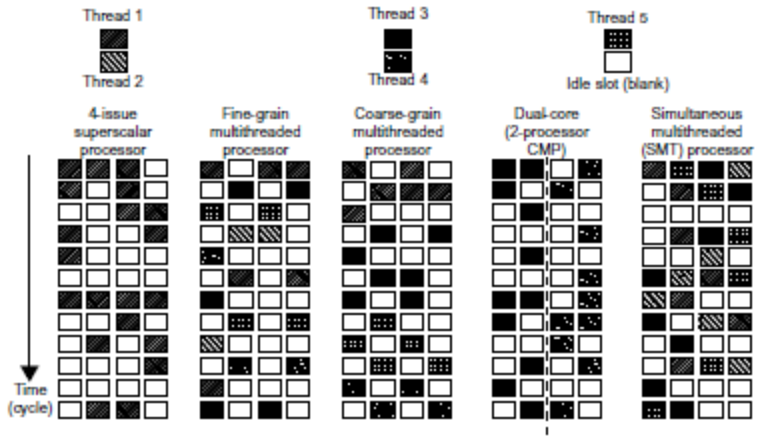


FIGURE 1.6
Five micro-architectures in modern CPU processors, that exploit ILP and TLP supported by multicore and multithreading technologies.

2 GPU Computing to Exascale and Beyond

A GPU is a graphics coprocessor or accelerator mounted on a computer's graphics card or video card. A GPU offloads the CPU from tedious graphics tasks in video editing applications. The world's first GPU, the GeForce 256, was marketed by NVIDIA in 1999. These GPU chips can process a minimum of 10 million polygons per second, and are used in nearly every computer on the market today. Some GPU features were also integrated into certain CPUs.

Traditional CPUs are structured with only a few cores. For example, the Xeon X5670 CPU has six cores. However, a modern GPU chip can be built with hundreds of processing cores. Unlike CPUs, GPUs have a throughput architecture that exploits massive parallelism by executing many concurrent threads slowly, instead of executing a single long thread in a conventional microprocessor very quickly. Lately, parallel GPUs or GPU clusters have been garnering a lot of attention against the use of CPUs with limited parallelism.General-purpose computing on GPUs, known as GPGPUs, have appeared in the HPC field.

How GPUs Work

Early GPUs functioned as coprocessors attached to the CPU. Today, the NVIDIA GPU has been upgraded to 128 cores on a single chip. Furthermore, each core on a GPU can handle eight threads of instructions. This translates to having up to 1,024 threads executed concurrently on a single GPU. This is true massive parallelism, compared to only a few threads that can be handled by a conventional CPU. The CPU is optimized for latency caches, while the GPU is optimized to deliver much higher throughput with explicit management of on-chip memory.

Modern GPUs are not restricted to accelerated graphics or video coding. They are used in HPC systems to power supercomputers with massive parallelism at multicore and multithreading levels. GPUs are designed to handle large numbers of floating-point operations in parallel. In a way, the GPU offloads the CPU from all data-intensive calculations, not just those that are related to video processing. Conventional GPUs are widely used in mobile phones, game consoles, embedded systems, PCs, and servers. The NVIDIA CUDA Tesla or Fermi is used in GPU clusters or in HPC systems for parallel processing of massive floating-pointing data.

GPU Programming Model

The interaction between a CPU and GPU in performing parallel execution of floating-point operations concurrently. The CPU is the conventional multicore processor with limited parallelism to exploit. The GPU has a many-core architecture that has hundreds of simple processing cores organized as multiprocessors. Each core can have one or more threads. Essentially, the CPU's floating-point kernel computation role is largely offloaded to the many-core GPU. The CPU instructs the GPU to perform massive data processing. The bandwidth must be matched between the on-board main memory and the on-chip GPU memory. This process is carried out in NVIDIA's CUDA programming using the GeForce 8800 or Tesla and Fermi GPUs.

The use of a GPU along with a CPU for massively parallel execution in hundreds or thousands

of processing cores.

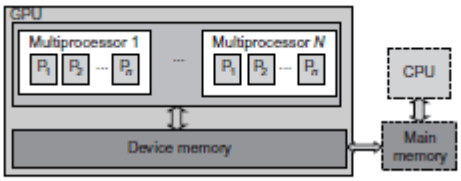


FIGURE 1.7
The use of a GPU along with a CPU for massively parallel execution in hundreds or thousands of processing cores.

(Courtesy of B. He, et al., PACT'08 [23])

Virtual Machines and Virtualization Middleware

A conventional computer has a single OS image. This offers a rigid architecture that tightly couples application software to a specific hardware platform. Some software running well on one machine may not be executable on another platform with a different instruction set under a fixed OS. Virtual machines (VMs) offer novel solutions to underutilized resources, application inflexibility, software manageability, and security concerns in existing physical machines.

Today, to build large clusters, grids, and clouds, we need to access large amounts of computing, storage, and networking resources in a virtualized manner. We need to aggregate those resources, and hopefully, offer a single system image. In particular, a cloud of provisioned resources must rely on virtualization of processors, memory, and I/O facilities dynamically.

Virtual Machines

The host machine is equipped with the physical hardware. An example is an x-86 architecture desktop running its installed Windows OS. The VM can be provisioned for any hardware system. The VM is built with virtual resources managed by a guest OS to run a specific application.
Between the VMs and the host platform, one needs to deploy a middleware layer called a virtual machine monitor (VMM). A native VM installed with the use of a VMM called a hypervisor in privileged mode. For example, the hardware has x-86 architecture running the Windows system.

The guest OS could be a Linux system and the hypervisor is the XEN system developed at Cambridge University. This hypervisor approach is also called bare-metal VM, because the hypervisor handles the bare hardware (CPU, memory, and I/O) directly. Another architecture is the host VM here the VMM runs in nonprivileged mode. The host OS need not be modified. The VM can also be implemented with a dual mode. Part of the VMM runs at the user level and another part runs at the supervisor level. In this case, the host OS may have to be modified to some extent. Multiple VMs can be ported to a given hardware system to support the virtualization process. The VM approach offers hardware independence of the OS and applications. The user application running on its dedicated OS could be bundled together

as a virtual appliance that can be ported to any hardware platform. The VM could run on an OS different from that of the host computer.

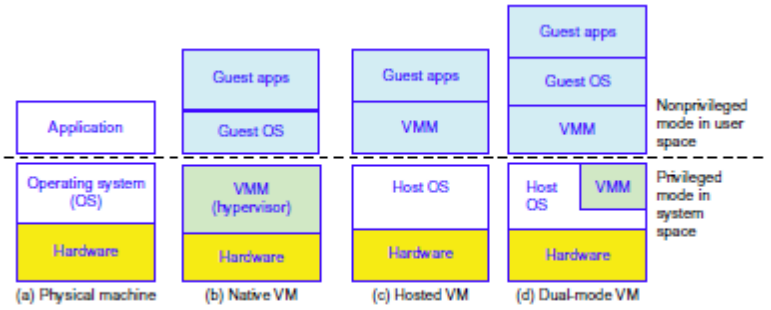


FIGURE 1.12
Three VM architectures in (b), (c), and (d), compared with the traditional physical machine shown in (a).

VM Primitive Operations

The VMM provides the VM abstraction to the guest OS. With full virtualization, the VMM exports a VM abstraction identical to the physical machine so that a standard OS such as Windows 2000 or Linux can run just as it would on the physical hardware. Low-level VMM operations are indicated by Mendel Rosenblum.

VM multiplexing, suspension, provision, and migration in a distributed computing environment.

• First, the VMs can be multiplexed between hardware machines,

• Second, a VM can be suspended and stored in stable storage,

• Third, a suspended VM can be resumed or provisioned to a new hardware platform,

• Finally, a VM can be migrated from one hardware platform to another,

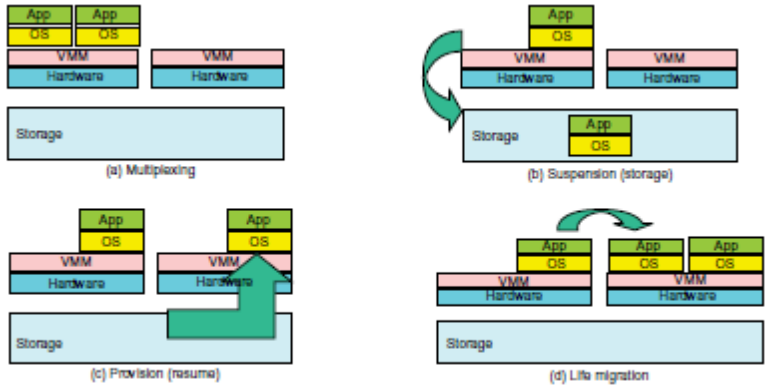


FIGURE 1.13
VM multiplexing, suspension, provision, and migration in a distributed computing environment.

These VM operations enable a VM to be provisioned to any available hardware platform. They also enable flexibility in porting distributed application executions. Furthermore, the VM approach will significantly enhance the utilization of server resources. Multiple server functions can be consolidated on the same hardware platform to achieve higher system efficiency. This will eliminate server sprawl via deployment of systems as VMs, which move transparency to the shared hardware. With this approach, VMware claimed that server utilization could be increased from its current 5–15 percent to 60–80 percent.

Virtual Infrastructures

Physical resources for compute, storage, and networking at the bottom are mapped to the needy applications embedded in various VMs at the top. Hardware and software are then separated. Virtual infrastructure is what connects resources to distributed applications. It is a dynamic mapping of system resources to specific applications. The result is decreased costs and increased efficiency and responsiveness. Virtualization for server consolidation and containment is a good example of this.

CLUSTERS OF COOPERATIVE COMPUTERS

A computing cluster consists of interconnected stand-alone computers which work cooperatively as a single integrated computing resource. In the past, clustered computer systems have demonstrated impressive results in handling heavy workloads with large data sets.

Cluster Architecture

The architecture of a typical server cluster built around a low-latency, high-bandwidth interconnection network. This network can be as simple as a SAN (e.g., Myrinet) or a LAN (e.g., Ethernet). To build a larger cluster with more nodes, the interconnection network can be built with multiple levels of Gigabit Ethernet, Myrinet, or InfiniBand switches. Through hierarchical construction using a SAN, LAN, or WAN, one can build scalable clusters with an increasing number of nodes. The cluster is connected to the Internet via a virtual private network (VPN) gateway. The gateway IP address locates the cluster. The system image of a computer is decided by the way the OS manages the shared cluster resources. Most clusters have loosely coupled node computers. All resources of a server node are managed by their own OS. Thus, most clusters have multiple system images as a result of having many autonomous nodes under different OS control.

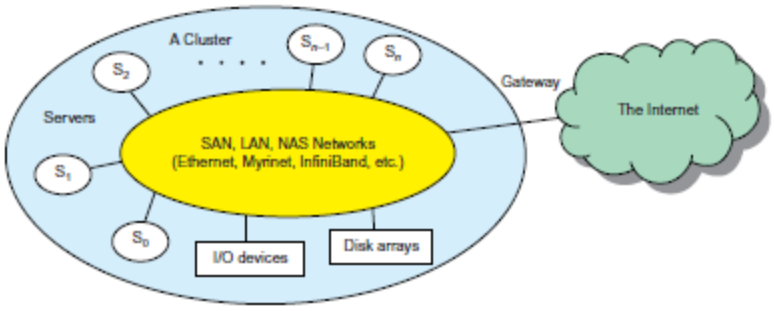


FIGURE 1.15
A cluster of servers interconnected by a high-bandwidth SAN or LAN with shared I/O devices and disk arrays; the cluster acts as a single computer attached to the Internet.

Single-System Image

Greg Pfister has indicated that an ideal cluster should merge multiple system images into a single-system image (SSI). Cluster designers desire a cluster operating system or some middleware to support SSI at various levels, including the sharing of CPUs, memory, and I/O across all cluster nodes. An SSI is an illusion created by software or hardware that presents a collection of resources as one integrated, powerful resource. SSI makes the cluster appear like a single machine to the user. A cluster with multiple system images is nothing but a collection of independent computers.

Hardware, Software, and Middleware Support

Clusters exploring massive parallelism are commonly known as MPPs. Almost all HPC clusters in the Top 500 list are also MPPs. The building blocks are computer nodes (PCs, workstations, servers, or SMP), special communication software such as PVM or MPI, and a network interface card in each computer node. Most clusters run under the Linux OS. The computer nodes are interconnected by a high-bandwidth network (such as Gigabit Ethernet, Myrinet, InfiniBand, etc.).

Special cluster middleware supports are needed to create SSI or high availability (HA). Both sequential and parallel applications can run on the cluster, and special parallel environments are needed to facilitate use of the cluster resources. For example, distributed memory has multiple images. Users may want all distributed memory to be shared by all servers by forming distributed shared memory (DSM). Many SSI features are expensive or difficult to achieve at various cluster operational levels. Instead of achieving SSI, many clusters are loosely coupled machines. Using virtualization, one can build many virtual clusters dynamically, upon user demand.

Major Cluster Design Issues

Unfortunately, a cluster-wide OS for complete resource sharing is not available yet. Middleware or OS extensions were developed at the user space to achieve SSI at selected

functional levels. Without this middleware, cluster nodes cannot work together effectively to achieve cooperative computing. The software environments and applications must rely on the middleware to achieve high performance. The cluster benefits come from scalable performance, efficient message passing, high system availability, seamless fault tolerance, and cluster-wide job management.

GRID COMPUTING INFRASTRUCTURES

In the past 30 years, users have experienced a natural growth path from Internet to web and grid computing services. Internet services such as the Telnet command enables a local computer to connect to a remote computer. A web service such as HTTP enables remote access of remote web pages. Grid computing is envisioned to allow close interaction among applications running on distant computers simultaneously. Forbes Magazine has projected the global growth of the IT-based economy from $1 trillion in 2001 to $20 trillion by 2015. The evolution from Internet to web and grid services is certainly playing a major role in this growth.

Computational Grids

Like an electric utility power grid, a computing grid offers an infrastructure that couples computers, software/middleware, special instruments, and people and sensors together. The grid is often constructed across LAN, WAN, or Internet backbone networks at a regional, national, or global scale. Enterprises or organizations present grids as integrated computing resources. They can also be viewed as virtual platforms to support virtual organizations. The computers used in a grid are primarily workstations, servers, clusters, and supercomputers. Personal computers, laptops, and PDAs can be used as access devices to a grid system.

The resource sites offer complementary computing resources, including workstations, large servers, a mesh of processors, and Linux clusters to satisfy a chain of computational needs. The grid is built across various IP broadband networks including LANs and WANs already used by enterprises or organizations over the Internet. The grid is presented to users as an integrated resource pool .

Computational grid or data grid providing computing utility, data, and information services through resource sharing and cooperation among participating organizations. Courtesy of Z. Xu, Chinese Academy of Science, 2004. Special instruments may be involved such as using the radio telescope in SETI@Home search of life in the galaxy and the austrophysics@Swineburne for pulsars. At the server end, the grid is a network. At the client end, we see wired or wireless terminal devices.The grid integrates the computing, communication, contents, and transactions as rented services. Enterprises and consumers form the user base, which then defines the usage trends and service characteristics. Many national and international grids will be reported, the NSF TeraGrid in US, EGEE in Europe, and ChinaGrid in China for various distributed scientific grid applications.

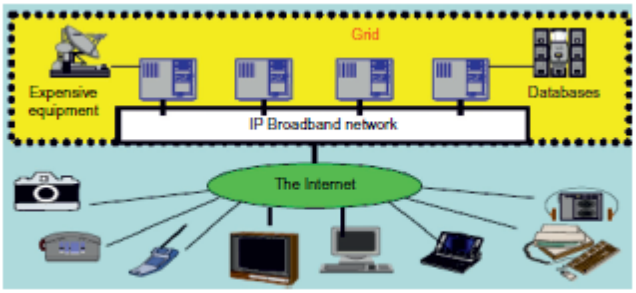


FIGURE 1.16

Computational grid or data grid providing computing utility, data, and information services through resource sharing and cooperation among participating organizations.

(Courtesy of Z. Xu, Chinese Academy of Science, 2004)

Table 1.4 Two Grid Computing Infrastructures and Representative Systems

Design Issues	Computational and Data Grids	P2P Grids
Grid Applications Reported	Distributed supercomputing, National Grid initiatives, etc.	Open grid with P2P flexibility, all resources from client machines
Representative Systems	TeraGrid built in US, ChinaGrid in China, and the e-Science grid built in UK	JXTA, FightAid@home, SETI@home
Development Lessons Learned	Restricted user groups, middleware bugs, protocols to acquire resources	Unreliable user-contributed resources, limited to a few apps

Grid Families

Grid technology demands new distributed computing models, software/middleware support, network protocols, and hardware infrastructures. National grid projects are followed by industrial grid platform development by IBM, Microsoft, Sun, HP, Dell,Cisco, EMC, Platform Computing, and others. New grid service providers (GSPs) and new grid applications have

emerged rapidly, similar to the growth of Internet and web services in the past two decades. In grid systems are classified in essentially two categories: computational or data grids and P2P grids. Computing or data grids are built primarily at the national level.

peer-to-peer network families

An example of a well-established distributed system is the client-server architecture. In this scenario, client machines (PCs and workstations) are connected to a central server for compute, e-mail, file access, and database applications. The P2P architecture offers a distributed model of networked systems. First, a P2P network is client-oriented instead of server-oriented. In this section, P2P systems are introduced at the physical level and overlay networks at the logical level.

P2P Systems

In a P2P system, every node acts as both a client and a server, providing part of the system resources. Peer machines are simply client computers connected to the Internet. All client machines act autonomously to join or leave the system freely. This implies that no master-slave relationship exists among the peers. No central coordination or central database is needed. In

other words, no peer machine has a global view of the entire P2P system. The system is self-organizing with distributed control. The architecture of a P2P network at two abstraction levels. Initially, the peers are totally unrelated. Each peer machine joins or leaves the P2P network voluntarily. Only the participating peers form the physical network at any time. Unlike the cluster or grid, a P2P network does not use a dedicated interconnection network. The physical network is simply an ad hoc network formed at various Internet domains randomly using the TCP/IP and NAI protocols. Thus, the physical network varies in size and topology dynamically due to the free membership in the P2P network.

CLOUD COMPUTING OVER THE INTERNET

Gordon Bell, Jim Gray, and Alex Szalay have advocated: ―Computational science is changing to be data-intensive. Supercomputers must be balanced systems, not just CPU farms but also petascale I/O and networking arrays.‖ In the future, working with large data sets will typically mean sending the computations (programs) to the data, rather than copying the data to the workstations. This reflects the trend in IT of moving computing and data from desktops to large data centers, where there is on-demand provision of software, hardware, and data as a service. This data explosion has promoted the idea of cloud computing.

Cloud computing has been defined differently by many users and designers. For example, IBM, a major player in cloud computing, has defined it as follows: ―A cloud is a pool of virtualized computer resources. A cloud can host a variety of different workloads,including batch-style backend jobs and interactive and user-facing applications.‖ Based on this definition, a cloud allows workloads to be deployed and scaled out quickly through

rapid provisioning of virtual or physical machines. The cloud supports redundant, self-recovering, highly scalable programming models that allow workloads to recover from many unavoidable hardware/software failures. Finally, the cloud system should be able to monitor resource use in real time to enable rebalancing of allocations when needed.

Internet Clouds

Cloud computing applies a virtualized platform with elastic resources on demand by provisioning hardware, software, and data sets dynamically . The idea is to move desktop computing to a service-oriented platform using server clusters and huge databases at data centers. Cloud computing leverages its low cost and simplicity to benefit both users and providers. Machine virtualization has enabled such cost-effectiveness. Cloud computing intends to satisfy many user applications simultaneously. The cloud ecosystem must be designed to be secure, trustworthy, and dependable. Some computer users think of the cloud as a centralized resource pool.

Others consider the cloud to be a server cluster which practices distributed computing over all the servers used.

Virtualized resources from data centers to form an Internet cloud, provisioned with

hardware,software, storage, network, and services for paid users to run their applications.

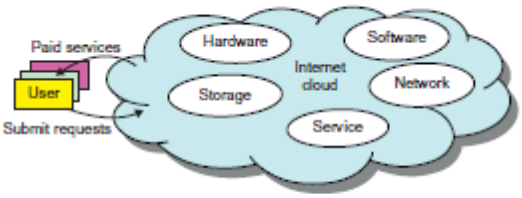


FIGURE 1.18
Virtualized resources from data centers to form an Internet cloud, provisioned with hardware, software, storage, network, and services for paid users to run their applications.

The Cloud Landscape

Traditionally, a distributed computing system tends to be owned and operated by an autonomous administrative domain (e.g., a research laboratory or company) for on-premises computing needs. However, these traditional systems have encountered several performance bottlenecks: constant system maintenance, poor utilization, and increasing costs associated with hardware/software upgrades. Cloud computing as an on-demand computing paradigm resolves or relieves us from these problems. It depicts the cloud landscape and major cloud players, based on three cloud service models.

Three cloud service models in a cloud landscape of major providers. Courtesy of Dennis Gannon,keynote address at Cloudcom2010

• Infrastructure as a Service (IaaS) This model puts together infrastructures demanded by users—namely servers, storage, networks, and the data center fabric. The user can deploy and run on multiple VMs running guest OSes on specific applications. Theuser does not manage or control the underlying cloud infrastructure, but can specify when to request and release the needed resources.

• Platform as a Service (PaaS) This model enables the user to deploy user-built applications onto a virtualized cloud platform. PaaS includes middleware, databases, development tools, and some runtime support such as Web 2.0 and Java. The platform includes both hardware and software integrated with specific programming interfaces. The provider supplies the API and software tools (e.g., Java, Python, Web 2.0, .NET). The user is freed from managing the cloud infrastructure.

• Software as a Service (SaaS) This refers to browser-initiated application software over thousands of paid cloud customers. The SaaS model applies to business processes, industry applications, consumer relationship management (CRM), enterprise resources planning (ERP), human resources (HR), and collaborative applications. On the customer side, there is no upfront investment in servers or software licensing. On the provider side, costs are rather low, compared with conventional hosting of user applications.

Internet clouds offer four deployment modes: private, public, managed, and hybrid .These modes

demand different levels of security implications. The different SLAs imply that the security responsibility is shared among all the cloud providers, the cloud resource consumers, and the third-party cloud-enabled software providers. Advantages of cloud computing have been advocated by many IT experts, industry leaders, and computer science researchers.

The following list highlights eight reasons to adapt the cloud for upgraded Internet applications and web services:

1. Desired location in areas with protected space and higher energy efficiency

2. Sharing of peak-load capacity among a large pool of users, improving overall utilization

3. Separation of infrastructure maintenance duties from domain-specific application development

4. Significant reduction in cloud computing cost, compared with traditional computing paradigms

5. Cloud computing programming and application development

6. Service and data discovery and content/service distribution

7. Privacy, security, copyright, and reliability issues

8. Service agreements, business models, and pricing policies

SERVICE-ORIENTED ARCHITECTURE (SOA)

In grids/web services, Java, and CORBA, an entity is, respectively, a service, a Java object, and a CORBA distributed object in a variety of languages. These architectures build on the traditional seven Open Systems Interconnection (OSI) layers that provide the base networking abstractions. On top of this we have a base software environment, which would be .NET or Apache Axis for web services, the Java Virtual Machine for Java, and a broker network for CORBA. On top of this base environment one would build a higher level environment reflecting the special features of the distributed computing environment. This starts with entity interfaces and inter-entity communication, which rebuild the top four OSI layers but at the entity and not the bit level.

Layered Architecture for Web Services and Grids

The entity interfaces correspond to the Web Services Description Language (WSDL), Java method, and CORBA interface definition language (IDL) specifications in these example distributed systems. These interfaces are linked with customized, high-level communication systems: SOAP, RMI, and IIOP in the three examples. These communication systems support features including particular message patterns (such as Remote Procedure Call or RPC), fault recovery, and specialized routing. Often, these communication systems are

built on message-oriented middleware (enterprise bus) infrastructure such as WebSphere MQ or Java Message Service (JMS) which provide rich functionality and support virtualization of routing, senders, and recipients.

In the case of fault tolerance, the features in the Web Services Reliable Messaging (WSRM) framework mimic the OSI layer capability (as in TCP fault tolerance) modified to match the different abstractions (such as messages versus packets, virtualized addressing) at the entity levels. Security is a critical capability that either uses or reimplements the capabilities seen in concepts such as Internet Protocol Security (IPsec) and secure sockets in the OSI layers. Entity communication is supported by higher level services for registries, metadata, and management of the entities .

Here, one might get several models with, for example, JNDI (Jini and Java Naming and Directory Interface) illustrating different approaches within the Java distributed object model. The CORBA Trading Service, UDDI (Universal Description, Discovery, and Integration), LDAP (Lightweight Directory Access Protocol), and ebXML (Electronic Business using eXtensible Markup Language) are other examples of discovery and information services . Management services include service state and lifetime support; examples include the CORBA Life Cycle and Persistent states, the different Enterprise JavaBeans models, Jini's lifetime model, and a suite of web services specifications in Chapter 5. The above language or interface terms form a collection of entity-level capabilities.

The latter can have performance advantages and offers a –shared memory‖ model allowing more convenient exchange of information. However, the distributed model has two critical advantages: namely, higher performance (from multiple CPUs when communication is unimportant) and a cleaner separation of software functions with clear software reuse and maintenance advantages. The distributed model is expected to gain popularity as the default approach to software systems. In the earlier years, CORBA and Java approaches were used in distributed systems rather than today's SOAP, XML, or REST (Representational State Transfer).

The Evolution of SOA

service-oriented architecture (SOA) has evolved over the years. SOA applies to building grids, clouds, grids of clouds, clouds of grids, clouds of clouds (also known as interclouds), and systems of systems in general. A large number of sensors provide data-collection services, denoted as SS (sensor service). A sensor can be a ZigBee device, a Bluetooth device, a WiFi access point, a personal computer, a GPA, or a wireless phone, among other things. Raw data is collected by sensor services. All the SS devices interact with large or small computers, many forms of grids, databases, the compute cloud, the storage cloud, the filter cloud, the discovery cloud, and so on. Filter services (fs in the figure) are used to eliminate unwanted raw data, in order to respond to specific requests from the web, the grid, or web services. The evolution of SOA: grids of clouds and grids, where –SS‖ refers to a sensor service and –fs‖ to a

filter or transforming service.A collection of filter services forms a filter cloud.

Processing this data will generate useful information, and subsequently, the knowledge for our daily use. In fact, wisdom or intelligence is sorted out of large knowledge bases. Finally, we make intelligent decisions based on both biological and machine wisdom. Most distributed systems require a web interface or portal. For raw data collected by a large number of sensors to be transformed into useful information or knowledge, the data stream may go through a sequence of compute, storage, filter, and discovery clouds.

Finally, the inter-service messages converge at the portal, which is accessed by all users. Two example portals, OGFCE and HUBzero, are described using both web service (portlet) and Web 2.0 (gadget) technologies. Many distributed programming models are also built on top of these basic constructs.

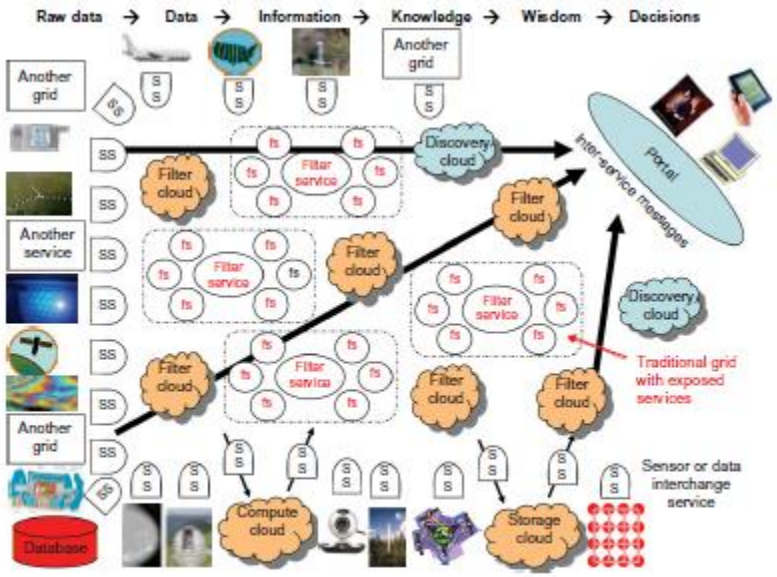


FIGURE 1.21
The evolution of SOA: grids of clouds and grids, where "SS" refers to a sensor service and "fs" to a filter or transforming service.

Grids versus Clouds

The boundary between grids and clouds are getting blurred in recent years. For webservices, workflow technologies are used to coordinate or orchestrate services with certain specifications used to define critical business process models such as two-phase transactions. Service standard, and several important workflow approaches including Pegasus, Taverna, Kepler, Trident, and Swift. In all approaches, one is building a collection of services which together tackle all or part of a distributed computing problem. In general, a grid system applies static resources, while a cloud emphasizes elastic resources. For some researchers, the differences between grids and clouds are limited only in dynamic resource allocation based on virtualization and autonomic computing.

One can build a grid out of multiple clouds. This type of grid can do a better job than a pure cloud, because it can explicitly support negotiated resource allocation. Thus one may end up building with a system of systems: such as a cloud of clouds, a grid of clouds, or a cloud of grids, or inter-clouds as a basic SOA architecture.

UNIT II

Grid Architecture and Service Modeling

The grid is a meta computing infrastructure that brings together computers (PCs,workstations, server clusters, supercomputers, laptops, notebooks, mobile computers, PDAs, etc.) to form a large collection of compute, storage, and network resources to solve large-scale computation problems or to enable fast information retrieval by registered users or user groups. The coupling between hardware and software with special user applications is achieved by leasing the hardware, software, middleware, databases, instruments, and networks as computing utilities. Good examples include the renting of expensive special-purpose application software on demand and transparent access to human genome databases.

The goal of grid computing is to explore fast solutions for large-scale computing problems. This objective is shared by computer clusters and massively parallel processor (MPP) systems . However, grid computing takes advantage of the existing computing resources scattered in a nation or internationally around the globe. In grids, resources owned by different organizations are aggregated together and shared by many users in collective applications. Grids rely heavy use of LAN/WAN resources across enterprises, organizations, and governments. The virtual organizations or virtual supercomputers are new concept derived from grid or cloud computing. These are virtual resources dynamically configured and are not under the full control of any single user or local administrator.

Grid History and Service Families

Network-based distributed computing becomes more and more popular among the Internet users. Recall that the Internet was developed in the 1980s to provide computer-to-computer connections using the telnet:// protocol. The web service was developed in the 1990s to establish direct linkage

e between web pages using the http:// protocol. Ever since the 1990s, grids became gradually available to establish large pools of shared resources. The approach is to link many Internet applications across machine platforms directly in order to eliminate isolated resource islands. We may invent upgraded protocols in the future like –grid://| and –cloud://| to realize this dream of a socialized cyberspace with greater resource sharing.

The idea of the grid was pioneered by Ian Foster, Carl Kesselman and Steve Tuecke in a 2001 paper . With is ground work, they are often recognized as the fathers of the grids. The Globus Project supported by DARPA has promoted the maturity of grid technology with a rich collection of software and middleware tools for grid computing. In 2007, the concept of cloud computing was thrown out, which in many ways was extending grid computing through virtualized data centers. In this beginning section, we introduce major grid families and review the grid service evolution over the past 15 years.

Grids differ from conventional HPC clusters. Cluster nodes are more homogeneousmachines that are better coordinated to work collectively and cooperatively. The grid nodes are heterogeneous computers that are more loosely coupled together over geographically dispersed sites. In 2001, Forbes Magazine advocated the emergence of the great global grid (GGG) as a new global infrastructure. This GGG evolved from the World Wide Web (WWW) technology we have enjoyed for many years.

Four Grid Service Families

Most of today's grid systems are called computational grids or data grids. Good examples are the NSF TeraGrid installed in the United States and the DataGrid built in the European Union. Information or knowledge grids post another grid class dedicated to knowledge management and distributed ontology processing. The Semantic web, also known as semantic grids, belongs to this faimly. Ontology platform falls into information or knowledge grids. Other information/knowledge grids include the Berkeley BOINC and NASA's Information Power Grid.

In the business world, we see a family, called business grids, built for business data/information processing. These are represented by the HP eSpeak, IBM WebSphere, Microsoft .NET, and Sun One systems. Some business grids are being transformed into Internet clouds. The last grid class includes several grid extensions such as P2P grids and parasitic grids. This will concentrate mainly in computational or data grids. Business grids are only briefly introduced.

Grid Service Protocol Stack

To put together the resources needed in a grid platform, a layered grid architecture . The top layer corresponds to user applications to run on the grid system. The user applications demand collective services including collective computing and communications. The next layer is formed by the hardware and software resources aggregated to run the user applications under the collective operations. The connectivity layer provides the interconnection among drafted resources. This connectivity could be established directly on physical networks or it could be built with virtual networking technology.

The layered grid service protocols and their relationship with the Internet service protocols. Courtesy of

Foster, Kesselman, and Tuecke .The connectivity must support the grid fabric, including the network inks and virtual private channels. The fabric layer includes all computational resources, storage systems, catalogs, network resources, sensors, and their network connections. The connectivity layer enables the exchange of data between fabric layer resources. The five-layer grid architecture is closely related to the layered Internet protocol stack . The fabric layer corresponds to the link layer in the Internet stack. The connectivity layer is supported by the network and transport layers of the Internet stack. The Internet application layer supports the top three layers.

Grid Resources

It summarizes typical resources that are required to perform grid computing.Many existing protocols (IP, TCP, HTTP, FTP, and DNS) or some new communication protocols can be used to route and transfer data. The resource layer is responsible for sharing single resources. An interface is needed to claim the static structure and dynamic status of local resources. The grid should be able to accept resource requests, negotiate the Quality of Service (QoS), and perform the operations specified in user applications.

The collective layer handles the interactions among a collection of resources. This layer implements functions such as resource discovery, co-allocation, scheduling, brokering, monitoring, and diagnostics. Other desired features include replication, grid-enabled programming, workload management, collaboration, software discovery, access authorization, and community accounting and payment. The application layer comprises mainly user applications. The applications interact with components in other layers by using well-defined APIs (application programming interfaces) and SDKs (software development kits).

CPU Scavenging and Virtual Supercomputers

The process of grid resource aggregation from local and remote sources. Then we link the grid to the concept of virtual organizations in a dynamic sense. In fact, the distinction between grids and clouds becomes blurred in recent years. Traditionally, grids were formed with allocated resources statically, while clouds were formed with provisioned resources dynamically. As virtualization is applicable to grid components, some grids involving data centers become more like clouds.

Foster, et al. [15] have compared the grid problem with the anatomy problem in biology. The application users expect grids to be designed as flexible, secure, and coordinated resources shared by individuals, institutions, and virtual organizations. The grid resources could come from two possible sources. On the one hand, large-scale HPC grids can be formed with computers from resource-rich supercomputer centers owned by government agencies and research institutions. Alternatively, one could form –virtual‖ grids, casually, out of a large number of small commodity computers owned by ordinary citizens, who volunteer to sharetheir free cycles with other users for a noble cause.

CPU Scavenging and Virtual Supercomputers

Both public and virtual grids can be built over large or small machines, that are loosely coupled together to satisfy the application need. Grids differ from the conventional supercomputers in many ways in the context of distributed computing. Supercomputers like

MPPs in the Top-500 list are more homogeneously structured with tightly coupled operations, while the grids are built with heterogeneous nodes running non-interactive workloads. These grid workloads may involve a large number of files and individual users. The geographically dispersed grids are more scalable and fault-tolerant with significantly lower operational costs than the supercomputers.The concept of creating a –grid‖ from the unused resources in a network of computers is known as CPU scavenging. In reality, virtual grids are built over large number of desktop computers by using their free cycles at night or during inactive usage periods. The donors are ordinary citizens on a voluntary participation basis. In practice, these client hosts also donate some disk space, RAM, and network bandwidth in addition to the raw CPU cycles. At present, many volunteer computing grids are built using the CPU scavenging model. The most famous example is the SETI@Home , which applied over 3 million computers to achieve 23.37 TFlpos as of Sept. 2001. More recent examples include the BOINC and Folding@Home , etc. In practice, these virtual grids can be viewed as virtual supercomputers.

Grid Resource Aggregation

During the resource aggregation process for grids or clouds, several assumptions are made. First, the compute nodes and other necessary resources for grids do not join or leave the system incidentally, except when some serious faults occur in the grid. Second, cloud resources are mostly provisioned from large data centers. Since security and reliability are very tight in these data centers, resource behavior is not predictable. Third, although resources in P2P systems are casually allocated, we can build P2P grids for distributed file sharing, content delivery, gaming, and entertainment applications. The joining or leaving of some peers has little impact on the needed functions of a P2P grid system.

We envision the grid resource aggregation process in a global setting. Hardware, software, database, and network resources are denoted by R's and are scattered all over the world. The availability and specification of these open resources is provided by Grid Information Service (GIS) agencies. The grid resource brokers assist users with fees to allocate available resources. Multiple brokers could compete to serve users. Also, multiple GISes may overlap in their resource coverage. New grid applications are enabled after the coupling of computer databases, instruments, and human operators needed in their specific applications. It should be noted that today's grid computing applications are no longer restricted to using HPC systems. HTC systems, like clouds, are even more in demand in business services.

Virtual Organization

The grid is a distributed system integrated from shared resources to form a virtual

organization(VO). The VO offers dynamic cooperation built over multiple physical organizations. The virtual resources contributed by these real organizations are managed autonomously. The grid must deal with the trust relationship in a VO. The applications in a grid vary in terms of workload and resource demand. A flexible grid system should be designed to adapt to varying workloads. In reality, physical organizations include a real company, a university, or a branch of government. These real organizations often share some common objectives.

For example, several research institutes and hospitals may undertake some joint research challenges together to explore a new cancer drug. Another concrete example is the joint venture among IBM, Apple, and Motorola to develop PowerPC processors and their supporting software in the past. The joint venture was based on the VO model. Grids definitely can promote the concept of VOs. Still, joint ventures demand resources and labor from all participants. The following example shows how two VOs or grid configurations can be formed out of three physical organizations.

Open Grid Services Architecture (OGSA)

The OGSA is an open source grid service standard jointly developed by academia and the IT industry under coordination of a working group in the Global Grid Forum (GGF). The standard was specifically developed for the emerging grid and cloud service communities. The OGSA is extended from web service concepts and technologies. The standard defines a common framework that allows businesses to build grid platforms across enterprises and business partners. The intent is to define the standards required for both open source and commercial software to support a global grid infrastructure.

OGSA Framework

The OGSA was built on two basic software technologies: the Globus Toolkit widely adopted as a grid technology solution for scientific and technical computing, and web services (WS 2.0) as a popular standards-based framework for business and network applications. The OGSA is intended to support the creation, termination, management, and invocation of stateful, transient grid services via standard interfaces and conventions . The OGSA framework specifies the physical environment, security, infrastructure profile, resource provisioning, virtual domains, and execution environment for various grid services and API access tools.

A service is an entity that provides some capability to its client by exchanging messages. We feel that greater flexibility is needed in grid service discovery and management. The service-oriented architecture (SOA) presented serves as the foundation of grid computing services. The individual and collective states of resources are specified in this service standard. The standard also specifies interactions between these services within the particular SOA for grids. An important point is that the architecture is not layered, where the implementation of one service is built upon modules that are logically dependent. One may classify this framework as

object-oriented. Many web service standards, semantics, and extensions are applied or modified in the OGSA.

OGSA Interfaces

The OGSA is centered on grid services. These services demand special well-defined application interfaces. These interfaces provide resource discovery, dynamic service creation, lifetime management, notification, and manageability. The conventions must address naming and upgradeability. The interfaces proposed by the OGSA working group. While the OGSA defines a variety of behaviors and associated interfaces, all but one of these interfaces (the grid service) is optional. Two key properties of a grid service are transience and statefulness. These properties have significant implications regarding how a grid service is named, discovered, and managed. Being transient means the service can be created and destroyed dynamically; statefulness refers to the fact that one can distinguish one service instance from another.

OGSA Grid Service Interfaces Developed by the OGSA Working Group

Grid Service Handle

A GSH is a globally unique name that distinguishes a specific grid service instance from all others. The status of a grid service instance could be that it exists now or that it will exist in the future. These instances carry no protocol or instance-specific addresses or supported protocol bindings. Instead, these information items are encapsulated along with all other instance-specific information. In order to interact with a specific service instance, a single abstraction is defined as a GSR. Unlike a GSH, which is time-invariant, the GSR for an instance can change over the lifetime of the service. The OGSA employs a –handle-resolution‖ mechanism for mapping from a GSH to a GSR. The GSH must be globally defined for a particular instance. However, the GSH may not always refer to the same network address. A service instance may be implemented in its own way, as long as it obeys the associated semantics. For example, the port type on which the service instance was implemented decides which operation to perform .

Grid Service Migration

This is a mechanism for creating new services and specifying assertions regarding the lifetime of a service. The OGSA model defines a standard interface, known as a factor, to implement this reference. Any service that is created must address the former services as the reference of later services. The factory interface is labeled as a Create Service operation . This creates a requested grid service with a specified interface and returns the GSH and initial GSR for the new service instance. It should also register the new service instance with a handle resolution service. Each dynamically created grid service instance is associated with a specified lifetime.

Grid Service Migration Using GSH and GSR shows how a service instance may migrate from one location to another during execution. A GSH resolves to a different GSR for a migrated service instance before (on the left) and after (on the right) the migration at time T. The handle resolver simply returns different GSRs before and after the migration. The initial lifetime can be extended by a specified time period by explicitly requesting the client or another grid service acting on the client's behalf.

A GSH resolving to a different GSR for a migrated service instance before (shown on the left) and after (on the right) the migration at time T. If the time period expires without having received a reaffirmed interest from a client, the service instance can be terminated on its own and release the associated resources accordingly. The lifetime management enables robust termination and failure detection. This is done by clearly defining the lifetime semantics of a service instance. Similarly, a hosting environment is guaranteed to consume bounded resources under some system failures. If the termination time of a service is reached, the hosting environment can reclaim all resources allocated.

OGSA Security Models

The OGSA supports security enforcement at various levels. The grid works in a heterogeneous distributed environment, which is essentially open to the general public. We must be able to detect intrusions or stop viruses from spreading by implementing secure conversations, single logon, access control, and auditing for nonrepudiation. At the security policy and user levels, we want to apply a service or endpoint policy, resource mapping rules, authorized access of critical resources, and privacy protection. At the Public Key Infrastructure (PKI) service level, the OGSA demands security binding with the security protocol stack and bridging of certificate authorities (CAs), use of multiple trusted intermediaries, and so on. Trust models and secure logging are often practiced in grid platforms.

The OGSA security model implemented at various protection levels. Courtesy of I. Foster, et al., http://www.ogf.org/documents/GFD.80.pdf

DATA-INTENSIVE GRID SERVICE MODELS

Applications in the grid are normally grouped into two categories: computation-intensive and data-intensive. For data-intensive applications, we may have to deal with massive amounts of data. For example, the data produced annually by a Large Hadron Collider may exceed several petabytes (1015 bytes). The grid system must be specially designed to discover, transfer, and manipulate these massive data sets. Transferring massive data sets is a time-consuming task. Efficient data management deman ds low-cost storage and high-speed data movement. Listed in the following paragraphs are several common methods for solving data movement problems.

Data Replication and Unified Namespace

This data access method is also known as caching, which is often applied to enhance data efficiency in a grid environment. By replicating the same data blocks and scattering them in multiple regions of a grid, users can access the same data with locality of references.

Furthermore, the replicas of the same data set can be a backup for one another. Some key data will not be lost in case of failures. However, data replication may demand periodic consistency checks. The increase in storage requirements and network bandwidth may cause additional problems. Replication strategies determine when and where to create a replica of the data. The factors to consider include data demand, network conditions, and transfer cost. The strategies of replication can be classified into method types: dynamic and static. For the static method, the locations and number of replicas are determined in advance and will not be modified. Although replication operations require little overhead, static strategies cannot adapt to changes in demand, bandwidth, and storage vailability. Dynamic strategies can adjust locations and number of data replicas according to changes inconditions (e.g., user behavior). However, frequent data-moving operations can result in much more overhead than in static strategies. The replication strategy must be optimized with respect to the status of data replicas. For static replication, optimization is required to determine the location and number of data replicas. For dynamic replication, optimization may be determined based on whether the data replica is being created, deleted, or moved. The most common replication strategies include preserving locality, minimizing update costs, and maximizing profits.

Grid Data Access Models

Multiple participants may want to share the same data collection. To retrieve any piece of data, we need a grid with a unique global namespace. Similarly, we desire to have unique file names. To achieve these, we have to resolve inconsistencies among multiple data objects bearing the same name. Access restrictions may be imposed to avoid confusion. Also, data needs to be protected to avoid leakage and damage. Users who want to access data have to be authenticated first and then authorized for access. In general, there are four access models for organizing a data grid, as listed here and shown in Figure 7.5.

Four architectural models for building a data grid.

Monadic model: This is a centralized data repository model. All the data is saved in a central data repository. When users want to access some data they have to submit requests directly to the central repository. No data is replicated for preserving data locality. This model is the simplest to implement for a small grid. For a large grid, this model is not efficient in terms of performance and reliability. Data replication is permitted in this model only when fault tolerance is demanded.

Hierarchical model: The hierarchical model, is suitable for building a large data grid which has only one large data access directory. The data may be transferred from the source to a second-level center. Then some data in the regional center is transferred to the third-level center. After being forwarded several times, specific data objects are accessed directly by users. Generally speaking, a higher-level data center has a wider coverage area. It provides higher bandwidth for

access than a lower-level data center. PKI security services are easier to implement in this hierarchical data access model. The European Data Grid (EDG) adopts this data access model.

Federation model: This data access model is better suited for designing a data grid with multiple sources of data supplies. Sometimes this model is also known as a mesh model. The data sources are distributed to many different locations. Although the data is shared, the data items are still owned and controlled by their original owners. According to predefined access policies, only authenticated users are authorized to request data from any data source. This mesh model may cost the most when the number of grid institutions becomes very large.

Hybrid model: This is data access model . The model combines the best features of the hierarchical and mesh models. Traditional data transfer technology, such as FTP, applies for networks with lower bandwidth. Network links in a data grid often have fairly high bandwidth, and other data transfer models are exploited by high-speed data transfer tools such as GridFTP developed with the Globus library. The cost of the hybrid model can be traded off between the two extreme models for hierarchical and mesh-connected grids.

Overview of Grid'5000 located at nine resource sites in France.

Parallel versus Striped Data Transfers

Compared with traditional FTP data transfer, parallel data transfer opens multiple data streams for passing subdivided segments of a file simultaneously. Although the speed of each stream is the same as in sequential streaming, the total time to move data in all streams can be significantly reduced compared to FTP transfer. In striped data transfer, a data object is partitioned into a number of sections, and each section is placed in an individual site in a data grid. When a user requests this piece of data, a data stream is created for each site, and all the sections of data objects are transferred simultaneously. Striped data transfer can utilize the bandwidths of multiple sites more efficiently to speed up data transfer.

Grid Projects and Grid Systems Built

Grid computing provides promising solutions to contemporary users who want to effectively share and collaborate with one another in distributed and self-governing environments. Apart from volunteer grids, most large-scale grids are national or international projects funded by public agencies. This section reviews the major grid systems developed in recent years. In particular, we describe three national grid projects that have been installed in the U.S., EU, and China.

National Grids and International Projects

Like supercomputers, national grids are mainly funded through government sources.These national grids are developed to promote research discovery, middleware products, and utility computing in grid-enabled applications.

National Grid Projects

Over the past decade, many data, information, or computational grids were built in various parts of the world. It summarizes five representative grid computing systems built in the United States, European Union, United Kingdom, France, and China. We call these national grids, because they are essentially government-funded projects pushing for grand challenge applications that demand high-performance computing and high-bandwidth communication networks. Here treat the EU countries as a single entity.

Most national grids are built by linking supercomputer centers and major computer ensembles together with Internet backbones and high-bandwidth WANs or LANs. More details can be found in the cited subsequent sections.

International Grid Projects

Grid applications cannot be restricted to geographical boundaries. As summarized , several global-scale grid projects were launched or are still active in use today. These projects promote volunteer computing, utility computing, and specific software applications that utilizes grid infrastructure. International grids involve both government and industrial funding. The European Union has been a major player in grid computing. The most famous EU grid projects are the EGEE, DataGrid, and BEinGrid. In the industrial sector, we have seen grid providers including Sun Microsystems, IBM, HP, etc. International grids are built with fix-term projects. Some of them are no longer active to provide public services at the end of funding.

UNIT III

CLOUD COMPUTING AND SERVICE MODELS

Over the past two decades, the world economy has rapidly moved from manufacturing to more service-oriented. In 2010, 80 percent of the U.S. economy was driven by the service industry, leaving only 15 percent in manufacturing and 5 percent in agriculture and other areas. Cloud computing benefits the service industry most and advances business computing with a new paradigm. In 2009, the global cloud service marketplace reached $17.4 billion. IDC predicted in 2010 that the cloud-based economy may increase to $44.2 billion by 2013. Developers of innovative cloud applications no longer acquire large capital equipment in advance. They just rent the resources from some large data centers that have been automated for this purpose.

Users can access and deploy cloud applications from anywhere in the world at very competitive costs. Virtualized cloud platforms are often built on top of large data centers. With that in mind, we examine first the server cluster in a data center and its interconnection issues. In other words, clouds aim to power the next generation of data centers by architecting them as virtual resources over automated hardware, databases, user interfaces, and application environments. In this sense, clouds grow out of the desire to build better data centers through automated resource provisioning.

Public, Private, and Hybrid Clouds

The concept of cloud computing has evolved from cluster, grid, and utility computing. Cluster and grid computing leverage the use of many computers in parallel to solve problems of any size. Utility and Software as a Service (SaaS) provide computing resources as a service with the notion of pay per use. Cloud computing leverages dynamic resources to deliver large numbers of services to end users. Cloud computing is a high-throughput computing (HTC) paradigm whereby the infrastructure provides the services through a large data center or server farms. The cloud computing model enables users to share access to resources from anywhere at any time through their connected devices.

In this scenario, the computations (programs) are sent to where the data is located, rather than copying the data to millions of desktops as in the traditional approach. Cloud computing avoids large data movement, resulting in much better network bandwidth utilization. Furthermore, machine virtualization has enhanced resource utilization, increased application flexibility, and reduced the total cost of using virtualized data-center resources. The cloud offers significant benefit to IT companies by freeing them from the low-level task of setting up the hardware (servers) and managing the system software. Cloud computing applies a virtual platform with elastic resources put together by on-demand provisioning of hardware, software, and data sets, dynamically. The main idea is to move desktop computing to a service-oriented platform

using server clusters and huge databases at data centers. Cloud computing leverages its low cost and simplicity to both providers and users. According to Ian Foster, cloud computing intends to leverage multitasking to achieve higher throughput by serving many heterogeneous applications, large or small, simultaneously.

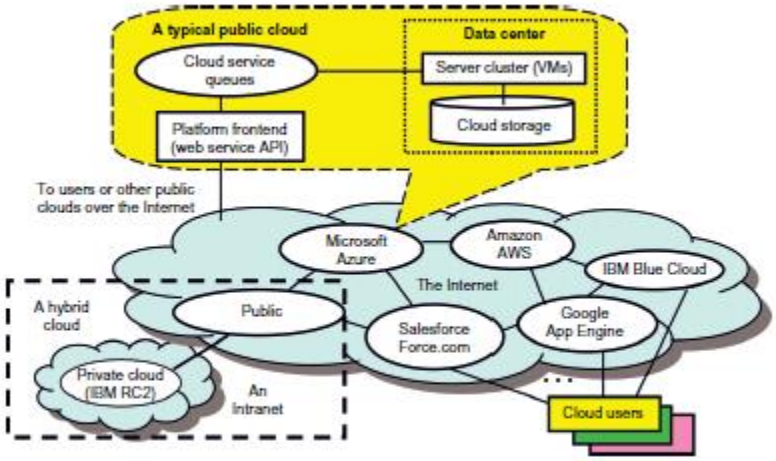


FIGURE 4.1
Public, private, and hybrid clouds illustrated by functional architecture and connectivity of representative clouds available by 2011.

Public Clouds

A public cloud is built over the Internet and can be accessed by any user who has paid for the service. Public clouds are owned by service providers and are accessible through a subscription. The callout box in top of the architecture of a typical public cloud. Many public clouds are available, including Google App Engine (GAE), Amazon Web Services (AWS), Microsoft Azure, IBM Blue Cloud, and Salesforce.com's Force.com. The providers of the aforementioned clouds are commercial providers that offer a publicly accessible remote interface for creating and managing VM instances within their proprietary infrastructure. A public cloud delivers a selected set of business processes. The application and infrastructure services are offered on a flexible price-per-use basis.

Private Clouds

A private cloud is built within the domain of an intranet owned by a single organization. Therefore, it is client owned and managed, and its access is limited to the owning clients and their partners. Its deployment was not meant to sell capacity over the Internet through publicly accessible interfaces. Private clouds give local users a flexible and agile private infrastructure to run service workloads within their administrative domains. A private cloud is supposed to deliver more efficient and convenient cloud services. It may impact the cloud standardization, while retaining greater customization and organizational control.

Hybrid Clouds

A hybrid cloud is built with both public and private clouds, Private clouds can also support a hybrid cloud model by supplementing local infrastructure with computing capacity from an external public cloud. For example, the Research Compute Cloud (RC2) is a private cloud, built by IBM, that interconnects the computing and IT resources at eight IBM Research Centers scattered throughout the United States, Europe, and Asia. A hybrid cloud provides access to clients, the partner network, and third parties. In summary, public clouds promote standardization, preserve capital investment, and offer application flexibility. Private clouds attempt to achieve customization and offer higher efficiency, resiliency, security, and privacy. Hybrid clouds operate in the middle, with many compromises in terms of resource sharing.

Infrastructure-as-a-Service (IaaS)

Cloud computing delivers infrastructure, platform, and software (application) as services, which are made available as subscription-based services in a pay-as-you-go model to consumers. The services provided over the cloud can be generally categorized into three different service models: namely IaaS, Platform as a Service (PaaS), and Software as a Service (SaaS). These form the three pillars on top of which cloud computing solutions are delivered to end users. All three models allow users to access services over the Internet, relying entirely on the infrastructures of cloud service providers.

These models are offered based on various SLAs between providers and users. In a broad sense, the SLA for cloud computing is addressed in terms of service availability, performance, and data protection and security. The three cloud models at different service levels of the cloud. SaaS is applied at the application end using special interfaces by users or clients. At the PaaS layer, the cloud platform must perform billing services and handle job queuing, launching, and monitoring services. At the bottom layer of the IaaS services, databases, compute instances, the file system, and storage must be provisioned to satisfy user demands. The IaaS, PaaS, and SaaS cloud service models at different service levels.

Infrastructure as a Service

This model allows users to use virtualized IT resources for computing, storage, and networking. In short, the service is performed by rented cloud infrastructure. The user can deploy and run his applications over his chosen OS environment. The user does not manage or control the underlying cloud infrastructure, but has control over the OS, storage, deployed applications, and possibly select networking components. This IaaS model encompasses storage as a service, compute instances as a service, and communication as a service. The Virtual Private Cloud

(VPC) in Example 4.1 shows how to provide Amazon EC2 clusters and S3 storage to multiple users. Many startup cloud providers have appeared in recent years. GoGrid, FlexiScale, and Aneka are good examples. It summarizes the IaaS offerings by five public cloud providers. Interested readers can visit the companies' web sites for updated information.

Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS)

In this section, we will introduce the PaaS and SaaS models for cloud computing. SaaS is often built on top of the PaaS, which is in turn built on top of the IaaS.

Platform as a Service (PaaS)

To be able to develop, deploy, and manage the execution of applications using provisioned resources demands a cloud platform with the proper software environment. Such a platform includes operating system and runtime library support. This has triggered the creation of the PaaS model to enable users to develop and deploy their user applications. It highlights cloud platform services offered by five PaaS services.

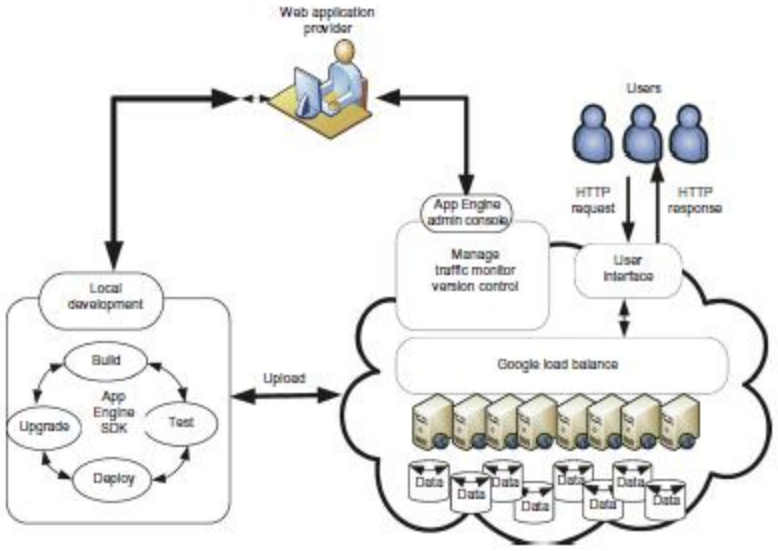


FIGURE 4.7
Google App Engine platform for PaaS operations.
(Courtesy of Yangting Wu, USC)

Software as a Service (SaaS)

This refers to browser-initiated application software over thousands of cloud customers. Services and tools offered by PaaS are utilized in construction of applications and management of their deployment on resources offered by IaaS providers. The SaaS model provides software applications as a service. As a result, on the customer side, there is no upfront investment in servers or software licensing. On the provider side, costs are kept rather low, compared with conventional hosting of user applications. Customer data is stored in the cloud that is either vendor proprietary or publicly hosted to support PaaS and IaaS.The best examples of SaaS services include Google Gmail and docs, Microsoft SharePoint, and the

CRM software from Salesforce.com. They are all very successful in promoting their own business or are used by thousands of small businesses in their day-to-day operations. Providers such as Google and Microsoft offer integrated IaaS and PaaS services, whereas others such as Amazon and GoGrid offer pure IaaS services and expect third-party PaaS providers such as Manjrasoft to offer application development and deployment services on top of their infrastructure services. To identify important cloud applications in enterprises, the success stories of three real-life cloud applications for HTC, news media, and business transactions. The benefits of using cloud services are evident in these SaaS applications.

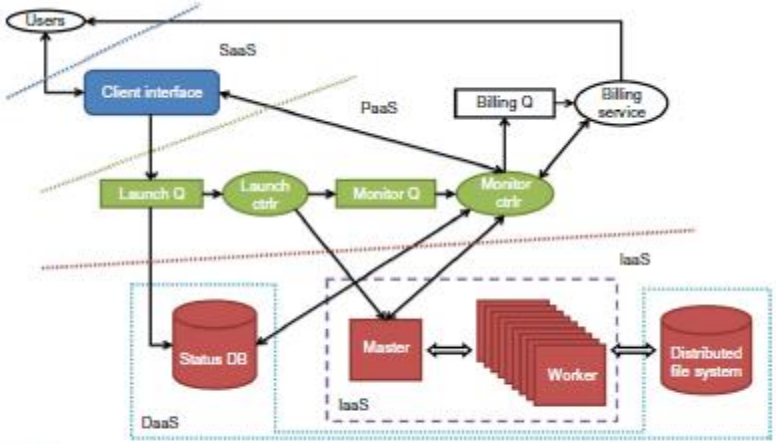


FIGURE 4.5
The IaaS, PaaS, and SaaS cloud service models at different service levels.

IMPLEMENTATION LEVELS OF VIRTUALIZATION

Virtualization is a computer architecture technology by which multiple virtual machines (VMs) are multiplexed in the same hardware machine. The idea of VMs can be dated back to the 1960s [53]. The purpose of a VM is to enhance resource sharing by many users and improve computer performance in terms of resource utilization and application flexibility. Hardware resources

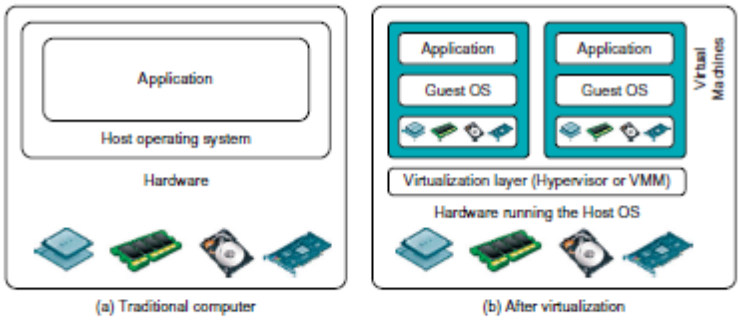


FIGURE 3.1

The architecture of a computer system before and after virtualization, where VMM stands for virtual machine monitor.

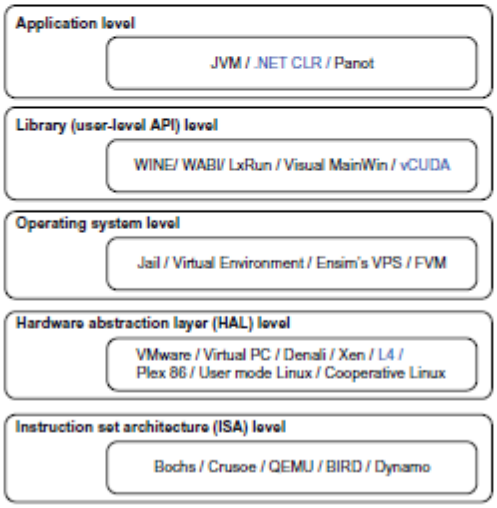


FIGURE 3.2

Virtualization ranging from hardware to applications in five abstraction levels.

(CPU, memory, I/O devices, etc.) or software resources (operating system and software libraries) can be virtualized in various functional layers. This virtualization technology has been revitalized as the demand for distributed and cloud computing increased sharply in recent years .

The idea is to separate the hardware from the software to yield better system efficiency. For example, computer users gained access to much enlarged memory space when the concept of virtual memory was introduced. Similarly, virtualization techniques can be applied to enhance the use of compute engines, networks, and storage. According to a 2009 Gartner Report, virtualization was the top strategic technology poised to change the computer industry. With sufficient storage, any computer platform can be installed in another host computer, even if they use processors with different instruction sets and run with distinct operating systems on the same hardware.

Levels of Virtualization Implementation

A traditional computer runs with a host operating system specially tailored for its hardware architecture. After virtualization, different user applications managed by their own operating systems (guest OS) can run on the same hardware, independent of the host OS. This is often done by adding additional software, called a virtualization layer . This virtualization layer is known as hypervisor or virtual machine monitor (VMM) . The VMs are shown in the upper boxes, where applications run with their own guest OS over the virtualized CPU, memory, and I/O resources. The architecture of a computer system before and after virtualization, where VMM stands for virtual machine monitor.

The main function of the software layer for virtualization is to virtualize the physical hardware of a host machine into virtual resources to be used by the VMs, exclusively. This can be implemented at various operational levels, as we will discuss shortly. The virtualization software creates the abstraction of VMs by interposing a virtualization layer at various levels of a computer system. Common virtualization layers include the instruction set architecture (ISA) level, hardware level, operating system level, library support level, and application level.
Virtualization ranging from hardware to applications in five abstraction levels.

Instruction Set Architecture Level

At the ISA level, virtualization is performed by emulating a given ISA by the ISA of the host machine. For example, MIPS binary code can run on an x86-based host machine with the help of ISA emulation. With this approach, it is possible to run a large amount of legacy binary code written for various processors on any given new hardware host machine. Instruction set emulation leads to virtual ISAs created on any hardware machine. The basic emulation method is through code interpretation. An interpreter program interprets the source instructions to target instructions one by one. One source instruction may require tens or hundreds of native target instructions to perform its function. Obviously, this process is relatively slow. For better performance, dynamic binary translation is desired. This approach

translates basic blocks of dynamic source instructions to target instructions. The basic blocks can also be extended to program traces or super blocks to increase translation efficiency.
Instruction set emulation requires binary translation and optimization. A virtual instruction set architecture (V-ISA) thus requires adding a processor-specific software translation layer to the compiler.

Hardware Abstraction Level

Hardware-level virtualization is performed right on top of the bare hardware. On the one hand, this approach generates a virtual hardware environment for a VM. On the other hand, the process manages the underlying hardware through virtualization. The idea is to virtualize a computer's resources, such as its processors, memory, and I/O devices. The intention is to upgrade the hardware utilization rate by multiple users concurrently. The idea was implemented in the IBM

VM/370 in the 1960s. More recently, the Xen hypervisor has been applied to virtualize x86-based machines to run Linux or other guest OS applications.

Operating System Level

This refers to an abstraction layer between traditional OS and user applications. OS-level virtualization creates isolated containers on a single physical server and the OS instances to utilize the hardware and software in data centers. The containers behave like real servers. OS-level virtualization is commonly used in creating virtual hosting environments to allocate hardware resources among a large number of mutually distrusting users. It is also used, to a lesser extent, in consolidating server hardware by moving services on separate hosts into containers or VMs on one server. OS-level virtualization is depicted .

Library Support Level

Most applications use APIs exported by user-level libraries rather than using lengthy system calls by the OS. Since most systems provide well-documented APIs, such an interface becomes another candidate for virtualization. Virtualization with library interfaces is possible by controlling the communication link between applications and the rest of a system through API hooks. The software tool WINE has implemented this approach to support Windows applications on top of UNIX hosts. Another example is

the vCUDA which allows applications executing within VMs to leverage GPU hardware acceleration. This approach is detailed .

User-Application Level

Virtualization at the application level virtualizes an application as a VM. On a traditional OS, an application often runs as a process. Therefore, application-level virtualization is also known as process-level virtualization. The most popular approach is to deploy high level language (HLL) VMs. In this scenario, the virtualization layer sits as an application program on top of the

operating system, and the layer exports an abstraction of a VM that can run programs written and compiled to a particular abstract machine definition. Any program written in the HLL and compiled for this VM will be able to run on it. The Microsoft .NET CLR and Java Virtual Machine (JVM) are two good examples of this class of VM.

Other forms of application-level virtualization are known as application isolation, application sandboxing, or application streaming. The process involves wrapping the application in a layer that is isolated from the host OS and other applications. The result is an application that is much easier to distribute and remove from user workstations. An example is the LANDesk application virtualization platform which deploys software applications as self-contained, executable files in an isolated environment without requiring installation, system modifications, or elevated security privileges.

Relative Merits of Different Approaches

Compares the relative merits of implementing virtualization at various levels.The column headings correspond to four technical merits. –Higher Performance‖ and –Application Flexibility‖ are self-explanatory. –Implementation Complexity‖ implies the cost to implement that particular virtualization level. –Application Isolation‖ refers to the effort required to isolate resources committed to different VMs. Each row corresponds to a particular level of virtualization.

Relative Merits of Virtualization at Various Levels (More –X‖'s Means Higher Merit, with a Maximum of 5 X's)

The number of X's in the table cells reflects the advantage points of each implementation level. Five X's implies the best case and one X implies the worst case. Overall, hardware and OS support will yield the highest performance. However, the hardware and application levels are also the most expensive to implement. User isolation is the most difficult to achieve. ISA implementation offers the best application flexibility.

VMM Design Requirements and Providers

As mentioned earlier, hardware-level virtualization inserts a layer between real hardware and traditional operating systems. This layer is commonly called the Virtual Machine Monitor (VMM) and it manages the hardware resources of a computing system. Each time programs access the hardware the VMM captures the process. In this sense, the VMM acts as a traditional OS. One hardware component, such as the CPU, can be virtualized as several virtual copies. Therefore, several traditional operating systems which are the same or different can sit on the same set of hardware simultaneously.

There are three requirements for a VMM. First, a VMM should provide an environment for programs which is essentially identical to the original machine. Second, programs run in this environment should show, at worst, only minor decreases in speed. Third, a VMM should be in complete control of the system resources. Any program run under a VMM should exhibit a function identical to that which it runs on the original machine directly. Two possible exceptions in terms of differences are permitted with this requirement: differences caused by the availability of system resources and differences caused by timing dependencies. The former arises when more than one VM is running on the same machine.

The hardware resource requirements, such as memory, of each VM are reduced, but the sum of them is greater than that of the real machine installed. The latter qualification is required because of the intervening level of software and the effect of any other VMs concurrently existing on the same hardware. Obviously, these two differences pertain to performance, while the function a VMM provides stays the same as that of a real machine. However, the identical environment requirement excludes the behavior of the usual time-sharing operating system from

being classed as a VMM.

A VMM should demonstrate efficiency in using the VMs. Compared with a physical machine, no one prefers a VMM if its efficiency is too low. Traditional emulators and complete software interpreters (simulators) emulate each instruction by means of functions or macros. Such a method provides the most flexible solutions for VMMs.

However, emulators or simulators are too slow to be used as real machines. To guarantee the efficiency of a VMM, a statistically dominant subset of the virtual processor's instructions needs to be executed directly by the real processor, with no software intervention by the VMM , compares four hypervisors and VMMs that are in use today.

Comparison of Four VMM and Hypervisor Software Packages Complete control of these resources by a VMM includes the following aspects:

(1) The VMM is responsible for allocating hardware resources for programs;

(2) it is not possible for a program to access any resource not explicitly allocated to it; and

(3) it is possible under certain circumstances for a VMM to regain control of resources already allocated. Not all processors satisfy these requirements for a VMM. A VMM is tightly related to the architectures of processors. It is difficult to implement a VMM for some types of processors, such as the x86. Specific limitations include the inability to trap on some privileged instructions. If a processor is not designed to support virtualization primarily, it is necessary to modify the hardware to satisfy the three requirements for a VMM. This is known as hardware-assisted virtualization.

Virtualization Support at the OS Level

With the help of VM technology, a new computing mode known as cloud computing is emerging. Cloud computing is transforming the computing landscape by shifting the hardware and staffing costs of managing a computational center to third parties, just like banks. However, cloud computing has at least two challenges. The first is the ability to use a variable number of physical machines and VM instances depending on the needs of a problem. For example, a task may need only a single CPU during some phases of execution but may need hundreds of CPUs at other times. The second challenge concerns the slow operation of instantiating new VMs.
Currently, new VMs originate either as fresh boots or as replicates of a template VM, unaware of the current application state. Therefore, to better support cloud computing, a large amount of research and development should be done.

Why OS-Level Virtualization?

As mentioned earlier, it is slow to initialize a hardware-level VM because each VM creates its own image from scratch. In a cloud computing environment, perhaps thousands of VMs need to be initialized simultaneously. Besides slow operation, storing the VM

images also becomes an issue. As a matter of fact, there is considerable repeated content among VM images. Moreover, full virtualization at the hardware level also has the disadvantages of slow performance and low density, and the need for para-virtualization to modify the guest OS. To reduce the performance overhead of hardware-level virtualization, even hardware modification is needed. OS-level virtualization provides a feasible solution for these hardware-level virtualization issues. Operating system virtualization inserts a virtualization layer inside an operating system to partition a machine's physical resources. It enables multiple isolated VMs within a single operating system kernel. This kind of VM is often called a virtual execution environment (VE), Virtual Private System (VPS), or simply container. From the user's point of view, VEs look like real servers. This means a VE has its own set of processes, file system, user accounts, network interfaces with IP addresses, routing tables, firewall rules, and other personal settings. Although VEs can be customized for different people, they share the same operating system kernel. Therefore, OS-level virtualization is also called single-OS image virtualization which illustrates operating system virtualization from the point of view of a machine stack.

The OpenVZ virtualization layer inside the host OS, which provides some OS images to create VMs quickly. Courtesy of OpenVZ User's Guide

Advantages of OS Extensions

Compared to hardware-level virtualization, the benefits of OS extensions are twofold:

(1) VMs at the operating system level have minimal startup/shutdown costs, low resource requirements, and high scalability; and

(2) for an OS-level VM, it is possible for a VM and its host environment to synchronize state changes when necessary. These benefits can be achieved via two mechanisms of OS-level virtualization:

(1) All OS-level VMs on the same physical machine share a single operating system kernel; and

(2) the virtualization layer can be designed in a way that allows processes in VMs to access as many resources of the host machine as possible, but never to modify them. In cloud computing, the first and second benefits can be used to overcome the defects of slow initialization of VMs at the hardware level, and being unaware of the current application state, respectively.

Disadvantages of OS Extensions

The main disadvantage of OS extensions is that all the VMs at operating system level on a single container must have the same kind of guest operating system. That is, although different OS-level VMs may have different operating system distributions, they must pertain to the same operating system family. For example, a Windows distribution such as Windows XP cannot run on a Linux-based container. However, users of cloud computing have various preferences. Some prefer Windows and others prefer Linux or other operating systems.

Therefore, there is a challenge for OS-level virtualization in such cases illustrates the concept of OS-level virtualization. The virtualization layer is inserted inside the OS to partition the hardware resources for multiple VMs to run their applications in multiple virtual environments. To implement OS-level virtualization, isolated execution environments (VMs) should be created based on a single OS kernel. Furthermore, the access requests from a VM need to be redirected to the VM's local resource partition on the physical machine. For example, the chroot command in a UNIX system can create several virtual root directories within a host OS. These virtual root directories are the root directories of all VMs created.

There are two ways to implement virtual root directories: duplicating common resources to each VM partition; or sharing most resources with the host environment and only creating private resource copies on the VM on demand. The first way incurs significant resource costs and overhead on a physical machine. This issue neutralizes the benefits of OS-level virtualization, compared with hardware-assisted virtualization.

Therefore, OS-level virtualization is often a second choice.

Virtualization on Linux or Windows Platforms

By far, most reported OS-level virtualization systems are Linux-based. Virtualization support on the Windows-based platform is still in the research stage. The Linux kernel offers an abstraction layer to allow software processes to work with and operate on resources without knowing the hardware details. New hardware may need a new Linux kernel to support.
Therefore, different Linux platforms use patched kernels to provide special support for extended functionality.

However, most Linux platforms are not tied to a special kernel. In such a case, a host can run several VMs simultaneously on the same hardware, summarizes several examples of OS-levelvirtualization tools that have been developed in recent years. Two OS tools (Linux vServer and OpenVZ) support Linux platforms to run other platform-based applications through virtualization. These two OS-level tools are illustrated in

The third tool, FVM, is an attempt specifically developed for virtualization on the Windows NT platform. Virtualization Support for Linux and Windows NT Platforms Uses system call interfaces to create VMs at the NY kernel space; multiple VMs are supported by virtualized namespace and copy-on-write.

Middleware Support for Virtualization

Library-level virtualization is also known as user-level Application Binary Interface (ABI) or API emulation. This type of virtualization can create execution environments for running alien programs on a platform rather than creating a VM to run the entire operating system. API call interception and remapping are the key functions performed. This section provides an overview of several library-level virtualization systems: namely the Windows Application

Binary Interface (WABI), lxrun, WINE, Visual MainWin, and vCUDA.

VIRTUALIZATION STRUCTURES/TOOLS AND MECHANISMS

In general, there are three typical classes of VM architecture showed the architectures of a machine before and after virtualization. Before virtualization, the operating system manages the hardware. After virtualization, a virtualization layer is inserted between the hardware and the operating system. In such a case, the virtualization layer is responsible for converting portions of the real hardware into virtual hardware. Therefore, different operating systems such as Linux and Windows can run on the same physical machine, simultaneously. Depending on the position of the virtualization layer, there are several classes of VM architectures, namely the hypervisor architecture, para-virtualization, and host-based virtualization. The hypervisor is also known as the VMM (Virtual Machine Monitor). They both perform the same virtualization operations.

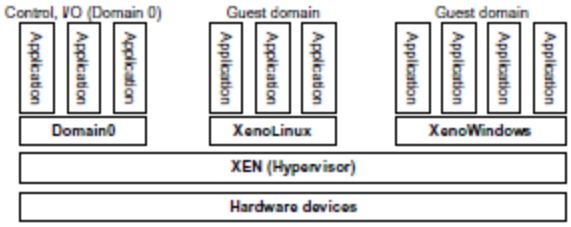


FIGURE 3.5
The Xen architecture's special domain 0 for control and I/O, and several guest domains for user applications.

Hypervisor and Xen Architecture

The hypervisor supports hardware-level virtualization on bare metal devices like CPU, memory, disk and network interfaces. The hypervisor software sits directly between the physical hardware and its OS. This virtualization layer is referred to as either the VMM or the hypervisor. The hypervisor provides hypercalls for the guest OSes and applications. Depending on the functionality, a hypervisor can assume a micro-kernel architecture like the Microsoft Hyper-V. Or it can assume a monolithic hypervisor architecture like the VMware ESX for server virtualization.

A micro-kernel hypervisor includes only the basic and unchanging functions (such as physical memory management and processor scheduling). The device drivers and other changeable components are outside the hypervisor. A monolithic hypervisor implements all the aforementioned functions, including those of the device drivers. Therefore, the size of the hypervisor code of a micro-kernel hypervisor is smaller than that of a monolithic hypervisor. Essentially, a hypervisor must be able to convert physical devices into virtual resources dedicated for the deployed VM to use.

The Xen Architecture

Xen is an open source hypervisor program developed by Cambridge University. Xen is a micro-

kernel hypervisor, which separates the policy from the mechanism. The Xen hypervisor implements all the mechanisms, leaving the policy to be handled by Domain 0. Xen does not include any device drivers natively . It just provides a mechanism by which a guest OS can have direct access to the physical devices. As a result, the size of the Xen hypervisor is kept rather small. Xen provides a virtual environment located between the hardware and the OS. A number of vendors are in the process of developing commercial Xen hypervisors, among them are Citrix XenServer and Oracle VM .

The core components of a Xen system are the hypervisor, kernel, and applications. The organization of the three components is important. Like other virtualization systems, many guest OSes can run on top of the hypervisor. However, not all guest OSes are created equal, and one in particular controls the others. The guest OS, which has control ability, is called Domain 0, and the others are called Domain U. Domain 0 is a privileged guest OS of Xen. It is first loaded when Xen boots without any file system drivers being available. Domain 0 is designed to access hardware directly and manage devices.

Therefore, one of the responsibilities of Domain 0 is to allocate and map hardware resources for the guest domains (the Domain U domains).

For example, Xen is based on Linux and its security level is C2. Its management VM is named Domain 0, which has the privilege to manage other VMs implemented on the same host. If Domain 0 is compromised, the hacker can control the entire system. So, in the VM system, security policies are needed to improve the security of Domain 0. Domain 0, behaving as a VMM, allows users to create, copy, save, read, modify, share, migrate, and roll back VMs as easily as manipulating a file, which flexibly provides tremendous benefits for users.
Unfortunately, it also brings a series of security problems during the software life cycle and data lifetime.

Traditionally, a machine's lifetime can be envisioned as a straight line where the current state of the machine is a point that progresses monotonically as the software executes. During this time, configuration changes are made, software is installed, and patches are applied. In such an environment, the VM state is akin to a tree: At any point, execution can go into N different branches where multiple instances of a VM can exist at any point in this tree at any given time. VMs are allowed to roll back to previous states in their execution (e.g., to fix configuration errors) or rerun from the same point many times (e.g., as a means of distributing dynamic content or circulating a –live‖ system image).

Binary Translation with Full Virtualization

Depending on implementation technologies, hardware virtualization can be classified into two categories: full virtualization and host-based virtualization. Full virtualization does not need to modify the host OS. It relies on binary translation to trap and to virtualize the execution of certain sensitive, nonvirtualizable instructions. The guest OSes and their applications consist of

noncritical and critical instructions. In a host-based system, both a host OS and a guest OS are used. A virtualization software layer is built between the host OS and guest OS. These two classes of VM architecture are introduced next.

Full Virtualization

With full virtualization, noncritical instructions run on the hardware directly while critical instructions are discovered and replaced with traps into the VMM to be emulated by software. Both the hypervisor and VMM approaches are considered full virtualization. Why are only critical instructions trapped into the VMM? This is because binary translation can incur a large performance overhead. Noncritical instructions do not control hardware or threaten the security of the system, but critical instructions do. Therefore, running noncritical instructions on hardware not only can promote efficiency, but also can ensure system security.

Binary Translation of Guest OS Requests Using a VMM

This approach was implemented by VMware and many other software companies. VMware puts the VMM at Ring 0 and the guest OS at Ring 1. The VMM scans the instruction stream and identifies the privileged, control- and behavior-sensitive instructions. When these instructions are identified, they are trapped into the VMM, which emulates the behavior of these instructions. The method used in this emulation is called binary translation. Therefore, full virtualization combines binary translation and direct execution. The guest OS is completely decoupled from the underlying hardware. Consequently, the guest OS is unaware that it is being virtualized. Indirect execution of complex instructions via binary translation of guest OS requests using the VMM plus direct execution of simple instructions on the same host. Courtesy of VM Ware The performance of full virtualization may not be ideal, because it involves binary translation which is rather time-consuming. In particular, the full virtualization of I/O-intensive applications is a really a big challenge. Binary translation employs a code cache to store translated hot instructions to improve performance, but it increases the cost of memory usage. At the time of this writing, the performance of full virtualization on the x86 architecture is typically 80 percent to 97 percent that of the host machine.

Host-Based Virtualization

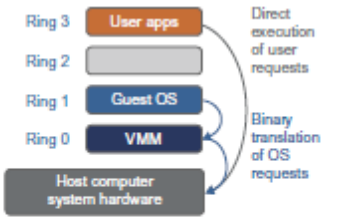


FIGURE 3.6
Indirect execution of complex instructions via binary translation of guest OS requests using the VMM plus direct execution of simple instructions on the same host.

An alternative VM architecture is to install a virtualization layer on top of the host OS. This host

OS is still responsible for managing the hardware. The guest OSes are installed and run on top of the virtualization layer. Dedicated applications may run on the VMs. Certainly, some other applications can also run with the host OS directly. This host-based architecture has some distinct advantages, as enumerated next. First, the user can install this VM architecture without modifying the host OS. The virtualizing software can rely on the host OS to provide device drivers and other low-level services. This will simplify the VM design and ease its deployment. Second, the host-based approach appeals to many host machine configurations. Compared to the hypervisor/VMM architecture, the performance of the host-based architecture may also be low. When an application requests hardware access, it involves four layers of mapping which downgrades performance significantly. When the ISA of a guest OS is different from the ISA of the underlying hardware, binary translation must be adopted. Although the host-based architecture has flexibility, the performance is too low to be useful in practice.

Para-Virtualization with Compiler Support

Para-virtualization needs to modify the guest operating systems. A para-virtualized VM provides special APIs requiring substantial OS modifications in user applications. Performance degradation is a critical issue of a virtualized system. No one wants to use a VM if it is much slower than using a physical machine. The virtualization layer can be inserted at different positions in a machine software stack. However, para-virtualization attempts to reduce the virtualization overhead, and thus improve performance by modifying only the guest OS kernel, illustrates the concept of a para-virtualized VM architecture. The guest operating systems are para-virtualized. They are assisted by an intelligent compiler to replace the nonvirtualizable OS instructions by hypercalls . The traditional x86 processor offers four instruction execution rings: Rings 0, 1, 2, and 3. The lower the ring number, the higher the privilege of instruction being executed. The OS is responsible for managing the hardware and the privileged instructions to execute at Ring 0, while user-level applications run at Ring 3. The best example of para-virtualization is the KVM to be described below.

Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization process

The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls. Courtesy of VMWare

Para-Virtualization Architecture

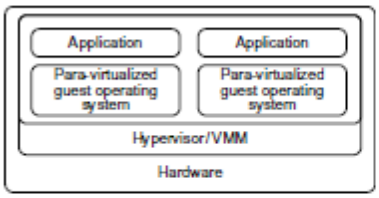


FIGURE 3.7
Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization process (See Figure 3.8 for more details.)

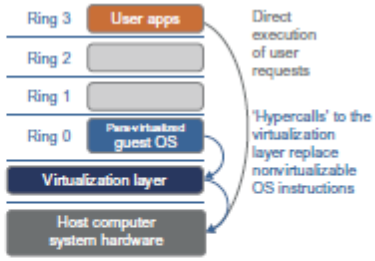


FIGURE 3.8
The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls.
(Courtesy of VMWare [71])

When the x86 processor is virtualized, a virtualization layer is inserted between the hardware and the OS. According to the x86 ring definition, the virtualization layer should also be installed at Ring 0. Different instructions at Ring 0 may cause some problems. Para-virtualization replaces nonvirtualizable instructions with hypercalls that communicate directly with the hypervisor orVMM. However, when the guest OS kernel is modified for virtualization, it can no longer run on the hardware directly.

Although para-virtualization reduces the overhead, it has incurred other problems. First, its compatibility and portability may be in doubt, because it must support the unmodified OS as well. Second, the cost of maintaining para-virtualized OSes is high, because they may require deep OS kernel modifications. Finally, the performance advantage of para-virtualization varies greatly due to workload variations. Compared with full virtualization, para-virtualization is relatively easy and more practical. The main problem in full virtualization is its low performance in binary translation. To speed up binary translation is difficult. Therefore, many virtualization products employ the para-virtualization architecture. The popular Xen, KVM, and VMware ESX are good examples.

KVM (Kernel-Based VM)

This is a Linux para-virtualization system—a part of the Linux version 2.6.20 kernel.Memory management and scheduling activities are carried out by the existing Linux kernel. The KVM does the rest, which makes it simpler than the hypervisor that controls the entire machine. KVM is a hardware-assisted para-virtualization tool, which improves performance and supports unmodified guest OSes such as Windows, Linux, Solaris, and other UNIX variants.

Para-Virtualization with Compiler Support

Unlike the full virtualization architecture which intercepts and emulates privileged and sensitive instructions at runtime, para-virtualization handles these instructions at compile time. The guest OS kernel is modified to replace the privileged and sensitive instructions with hypercalls to the hypervisor or VMM. Xen assumes such a para-virtualization architecture. The guest

OS running in a guest domain may run at Ring 1 instead of at Ring 0. This implies that the guest OS may not be able to execute some privileged and sensitive instructions. The privileged instructions are implemented by hypercalls to the hypervisor. After replacing the instructions with hypercalls, the modified guest OS emulates the behavior of the original guest OS. On an UNIX system, a system call involves an interrupt or service routine. The hypercalls apply a dedicated service routine in Xen.

VIRTUALIZATION OF CPU, MEMORY, AND I/O DEVICES

To support virtualization, processors such as the x86 employ a special running mode and instructions, known as hardware-assisted virtualization. In this way, the VMM and guest OS run in different modes and all sensitive instructions of the guest OS and its applications are trapped in the VMM. To save processor states, mode switching is completed by hardware. For the x86 architecture, Intel and AMD have proprietary technologies for hardware-assisted virtualization.

Hardware Support for Virtualization

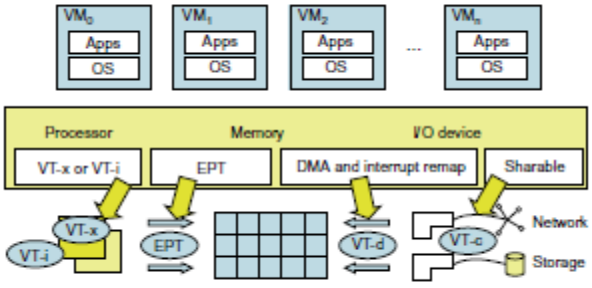


FIGURE 3.10
Intel hardware support for virtualization of processor, memory, and I/O devices.

Modern operating systems and processors permit multiple processes to run simultaneously. If there is no protection mechanism in a processor, all instructions from different processes will access the hardware directly and cause a system crash. Therefore, all processors have at least two modes, user mode and supervisor mode, to ensure controlled access of critical hardware. Instructions running in supervisor mode are called privileged instructions. Other instructions are unprivileged instructions. In a virtualized environment, it is more difficult to make OSes and applications run correctly because there are more layers in the machine stack. Intel's hardware support approach.

At the time of this writing, many hardware virtualization products were available. The VMware Workstation is a VM software suite for x86 and x86-64 computers. This software suite allows users to set up multiple x86 and x86-64 virtual computers and to use one or more of these VMs simultaneously with the host operating system. The VMware Workstation assumes the host-based virtualization. Xen is a hypervisor for use in IA-32, x86-64, Itanium, and PowerPC 970 hosts. Actually, Xen modifies Linux as the lowest and most privileged layer, or a hypervisor.
One or more guest OS can run on top of the hypervisor. KVM (Kernel-based Virtual Machine) is a Linux kernel virtualization infrastructure. KVM can support hardware-assisted virtualization and paravirtualization by using the Intel VT-x or AMD-v and VirtIO framework, respectively. The VirtIO framework includes a paravirtual Ethernet card, a disk I/O controller, a balloon device for adjusting guest memory usage, and a VGA graphics interface using VMware drivers.

CPU Virtualization

A VM is a duplicate of an existing computer system in which a majority of the VM instructions are executed on the host processor in native mode. Thus, unprivileged instructions of VMs run directly on the host machine for higher efficiency. Other critical instructions should be handled carefully for correctness and stability. The critical instructions are divided into three categories: privileged instructions, control-sensitive instructions, and behavior-sensitive instructions. Privileged instructions execute in a privileged mode and will be trapped if

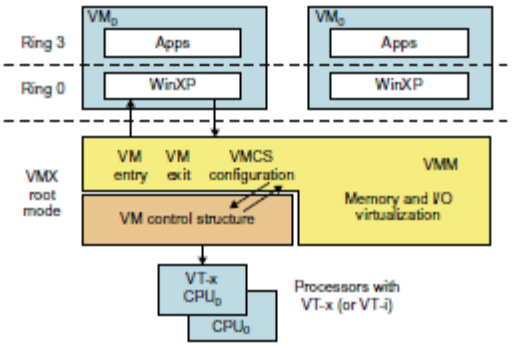


FIGURE 3.11
Intel hardware-assisted CPU virtualization.

executed outside this mode. Control-sensitive instructions attempt to change the configuration of resources used. Behavior-sensitive instructions have different behaviors depending on the configuration of resources, including the load and store operations over the virtual memory. A CPU architecture is virtualizable if it supports the ability to run the VM's privileged and unprivileged instructions in the CPU's user mode while the VMM runs in supervisor mode. When the privileged instructions including control- and behavior-sensitive instructions of a VM are executed, they are trapped in the VMM. In this case, the VMM acts as a unified mediator for hardware access from different VMs to guarantee the correctness and stability of the whole system. However, not

all CPU architectures are virtualizable. RISC CPU architectures can be naturally virtualized because all control-and behavior-sensitive instructions are privileged instructions. On the contrary, x86 CPU architectures are not primarily designed to support virtualization. This is because about 10 sensitive instructions, such as SGDT and SMSW, are not privileged instructions. When these instructions execute in virtualization, they cannot be trapped in the VMM. On a native UNIX-like system, a system call triggers the 80h interrupt and passes control to the OS kernel. The interrupt handler in the kernel is then invoked to process the system call. On a paravirtualization system such as Xen, a system call in the guest OS first triggers the 80h interrupt normally. Almost at the same time, the 82h interrupt in the hypervisor is triggered. Incidentally, control is passed on to the hypervisor as well. When the hypervisor completes its task for the guest OS system call, it passes control back to the guest OS kernel. Certainly, the guest OS kernel may also invoke the hypercall while it's running. Although paravirtualization of a CPU lets unmodified applications run in the VM, it causes a small performance penalty.

Hardware-Assisted CPU Virtualization

This technique attempts to simplify virtualization because full or paravirtualization is complicated. Intel and AMD add an additional mode called privilege mode level (some people call it Ring-1) to x86 processors. Therefore, operating systems can still run at Ring 0 and the hypervisor can run at Ring -1. All the privileged and sensitive instructions are trapped in the hypervisor automatically. This technique removes the difficulty of implementing binary translation of full virtualization. It also lets the operating system run in VMs without modification.

Memory Virtualization

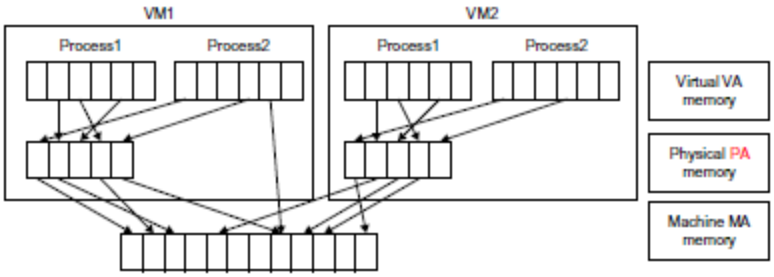


FIGURE 3.12
Two-level memory mapping procedure.

Virtual memory virtualization is similar to the virtual memory support provided by modern operating systems. In a traditional execution environment, the operating system maintains mappings of virtual memory to machine memory using page tables, which is a one-stage mapping from virtual memory to machine memory. All modern x86 CPUs include a memory management unit (MMU) and a translation lookaside buffer (TLB) to optimize virtual memory performance. However, in a virtual execution environment, virtual memory virtualization involves sharing the physical system memory in RAM and dynamically allocating it to the physical memory of the VMs.That means a two-stage mapping process should be maintained by the guest OS and the VMM, respectively: virtual memory to physical memory and physical memory to machine memory. Furthermore, MMU virtualization should be supported, which is transparent to the guest OS. The guest OS continues to control the mapping of virtual addresses to the physical memory addresses of VMs. But the guest OS cannot directly access the actual machine memory. The VMM is responsible for mapping the guest physical memory to the actual machine memory, shows the two-level memory mapping procedure. Two-level memory mapping procedure. Courtesy of R. Rblig, et al. Since each page table of the guest OSes has a separate page table in the VMM corresponding to it, the VMM page table is called the

shadow page table. Nested page tables add another layer of indirection to virtual memory. The MMU already handles virtual-to-physical translations as defined by the OS. Then the physical memory addresses are translated to machine addresses using another set of page tables defined by the hypervisor. Since modern operating systems maintain a set of page tables for every process, the shadow page tables will get flooded. Consequently, the performance overhead and cost of memory will be very high. VMware uses shadow page tables to perform virtual-memory-to-machine-memory address translation. Processors use TLB hardware to map the virtual memory directly to the machine memory to avoid the two levels of translation on every access. When the guest OS changes the virtual memory to a physical memory mapping, the VMM updates the shadow page tables to enable a direct lookup. The AMD Barcelona processor has featured hardware-assisted memory virtualization since 2007. It provides hardware assistance to the two-stage address translation in a virtual execution environment by using a technology called nested paging.

I/O Virtualization

I/O virtualization involves managing the routing of I/O requests between virtual devices and the shared physical hardware. At the time of this writing, there are three ways to implement I/O virtualization: full device emulation, para-virtualization, and direct I/O.Full device emulation is the first approach for I/O virtualization. Generally, this approach emulates well-known, real-world devices. All the functions of a device or bus infrastructure, such as device enumeration, identification, interrupts, and DMA, are replicated in software. This software is located in the VMM and acts as a virtual device. The I/O access requests of the guest OS are trapped in the VMM which interacts with the I/O devices. Device emulation for I/O virtualization implemented inside the middle layer that maps real I/O devices into the virtual devices for the guest device driver to use. Courtesy of V. Chadha, et al. and Y. Dong, et al. A single hardware device can be shared by multiple VMs that run concurrently. However, software emulation runs much slower than the hardware it emulates.

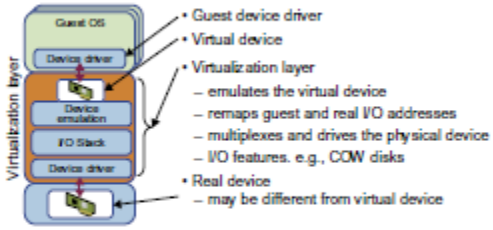


FIGURE 3.14
Device emulation for I/O virtualization implemented inside the middle layer that maps real I/O devices into the virtual devices for the guest device driver to use.

The para-virtualization method of I/O virtualization is typically used in Xen. It is also known as the split driver model consisting of a frontend driver and a backend driver. The frontend driver is running in Domain U and the backend driver is running in Domain 0. They interact with each other via a block of shared memory. The frontend driver manages the I/O requests of the guest OSes and the backend driver is responsible for managing the real I/O devices and multiplexing the I/O data of different VMs. Although para-I/O-virtualization achieves better device performance than full device emulation, it comes with a higher CPU overhead.

Direct I/O virtualization lets the VM access devices directly. It can achieve close-to-native performance without high CPU costs. However, current direct I/O virtualization implementations focus on networking for mainframes. There are a lot of challenges for commodity hardware devices. For example, when a physical device is reclaimed (required by workload migration) for later reassignment, it may have been set to an arbitrary state (e.g., DMA to some arbitrary

memory locations) that can function incorrectly or even crash the whole system. Since software-based I/O virtualization requires a very high overhead of device emulation, hardware-assisted I/O virtualization is critical. Intel VT-d supports the remapping of I/O DMA transfers and device-generated interrupts. The architecture of VT-d provides the flexibility to support multiple usage models that may run unmodified, special-purpose, or –virtualization-aware‖ guest OSes.

Another way to help I/O virtualization is via self-virtualized I/O (SV-IO) . The key idea of SV-IO is to harness the rich resources of a multicore processor. All tasks associated with virtualizing an I/O device are encapsulated in SV-IO. It provides virtual devices and an associated access API to VMs and a management API to the VMM. SV-IO defines one virtual interface (VIF) for every kind of virtualized I/O device, such as virtual network interfaces, virtual block devices (disk), virtual camera devices, and others. The guest OS interacts with the VIFs via VIF device drivers. Each VIF consists of two message queues. One is for outgoing messages to the devices and the other is for incoming messages from the devices. In addition, each VIF has a unique ID for identifying it in SV-IO.

Virtual Clusters and Resource Management

A physical cluster is a collection of servers (physical machines) interconnected by a physical network such as a LAN. various clustering techniques on physical machines. Here, we introduce virtual clusters and study its properties as well as explore their potential applications. In this section, we will study three critical design issues of virtual clusters: live migration of VMs, memory and file migrations, and dynamic deployment of virtual clusters. When a traditional VM is initialized, the administrator needs to manually write configuration information or specify the configuration sources. When more VMs join a network, an inefficient configuration always causes problems with overloading or underutilization.
Amazon's Elastic Compute Cloud (EC2) is a good example of a web service that provides elastic computing power in a cloud. EC2 permits customers to create VMs and to manage user accounts over the time of their use. Most virtualization platforms, including XenServer and VMware ESX Server, support a bridging mode which allows all domains to appear on the network as individual hosts. By using this mode, VMs can communicate with one another freely through the virtual network interface card and configure the network automatically.

Physical versus Virtual Clusters

Virtual clusters are built with VMs installed at distributed servers from one or more physical clusters. The VMs in a virtual cluster are interconnected logically by a virtual network across several physical networks. Figure 3.18 illustrates the concepts of virtual clusters and physical clusters. Each virtual cluster is formed with physical machines or a VM hosted by multiple physical clusters. The virtual cluster boundaries are shown as distinct boundaries. A cloud platform with four virtual clusters over three physical clusters shaded differently. Courtesy of Fan Zhang, Tsinghua UniversityThe provisioning of VMs to a virtual cluster is done dynamically to have the following interesting properties:

• The virtual cluster nodes can be either physical or virtual machines. Multiple VMs running with different OSes can be deployed on the same physical node.

• A VM runs with a guest OS, which is often different from the host OS, that manages the resources in the physical machine, where the VM is implemented.

• The purpose of using VMs is to consolidate multiple functionalities on the same server. This will greatly enhance server utilization and application flexibility.

• VMs can be colonized (replicated) in multiple servers for the purpose of promoting distributed parallelism, fault tolerance, and disaster recovery.

• The size (number of nodes) of a virtual cluster can grow or shrink dynamically, similar to the way an overlay network varies in size in a peer-to-peer (P2P) network.

• The failure of any physical nodes may disable some VMs installed on the failing nodes. But the failure of VMs will not pull down the host system.

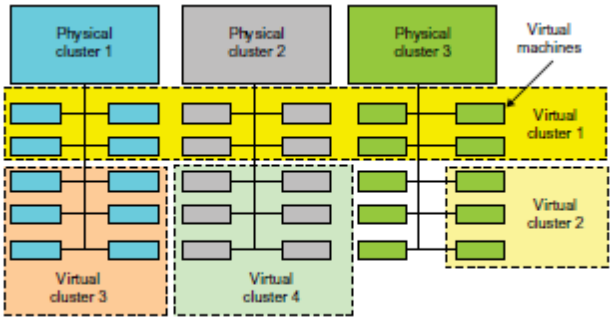


FIGURE 3.18
A cloud platform with four virtual clusters over three physical clusters shaded differently.

Since system virtualization has been widely used, it is necessary to effectively manage VMs running on a mass of physical computing nodes (also called virtual clusters) and consequently build a high-performance virtualized computing environment. This involves virtual cluster deployment, monitoring and management over large-scale clusters, as well as resource scheduling, load balancing, server consolidation, fault tolerance, and other techniques. The different node colors refer to different virtual clusters. In a virtual cluster system, it is quite important to store the large number of VM images efficiently.The concept of a virtual cluster based on application partitioning or customization. The different colors in the figure represent the nodes in different virtual clusters. As a large number of VM images might be present, the most important thing is to determine how to store those images in the system efficiently. There are common installations for most users or applications, such as operating systems or user-level programming libraries. These software packages can be preinstalled as templates (called template VMs). With these templates, users can build their own software stacks. New OS instances can be copied from the template VM. User-specific components such as programming libraries and applications can be installed to those instances. The concept of a

virtual cluster based on application partitioning. Courtesy of Kang Chen, TsinghuaUniversity 2008Three physical clusters and Four virtual clusters are created on the right, over the physical clusters. The physical machines are also called host systems. In contrast, the VMs are guest systems. The host and guest systems may run with different operating systems. Each VM can be installed on a remote server or replicated on multiple servers belonging to the same or different physical clusters. The boundary of a virtual cluster can change as VM nodes are added, removed, or migrated dynamically over time.

Fast Deployment and Effective Scheduling

The system should have the capability of fast deployment. Here, deployment means two things: to construct and distribute software stacks (OS, libraries, applications) to a physical node inside clusters as fast as possible, and to quickly switch runtime environments from one user's virtual cluster to another user's virtual cluster. If one user finishes using his system, the corresponding virtual cluster should shut down or suspend quickly to save the resources to run other VMs for other users.

The concept of ―green computing‖ has attracted much attention recently. However,previous approaches have focused on saving the energy cost of components in a single workstation without a global vision. Consequently, they do not necessarily reduce the power consumption of the whole cluster. Other cluster-wide energy-efficient techniques can only be applied to homogeneous workstations and specific applications. The live migration of VMs allows workloads of one node to transfer to another node. However, it does not guarantee that VMs can randomly migrate among themselves. In fact, the potential overhead caused by live migrations of VMs cannot be ignored.

The overhead may have serious negative effects on cluster utilization, throughput, and QoS issues. Therefore, the challenge is to determine how to design migration strategies to implement green computing without influencing the performance of clusters. Another advantage of virtualization is load balancing of applications in a virtual cluster. Load balancing can be achieved using the load index and frequency of user logins. The automatic scale-up and scale-down mechanism of a virtual cluster can be implemented based on this model.
Consequently, we can increase the resource utilization of nodes and shorten the response time of systems. Mapping VMs onto the most appropriate physical node should promote performance. Dynamically adjusting loads among nodes by live migration of VMs is desired, when the loads on cluster nodes become quite unbalanced.

High-Performance Virtual Storage

The template VM can be distributed to several physical hosts in the cluster to customize the VMs. In addition, existing software packages reduce the time for customization as well as switching virtual environments. It is important to efficiently manage the disk spaces occupied by template software packages. Some storage architecture design can be applied to reduce

duplicated blocks in a distributed file system of virtual clusters. Hash values are used to compare the contents of data blocks. Users have their own profiles which store the identification of the data blocks for corresponding VMs in a user-specific virtual cluster. New blocks are created when users modify the corresponding data. Newly created blocks are identified in the users' profiles. Basically, there are four steps to deploy a group of VMs onto a target cluster: preparing

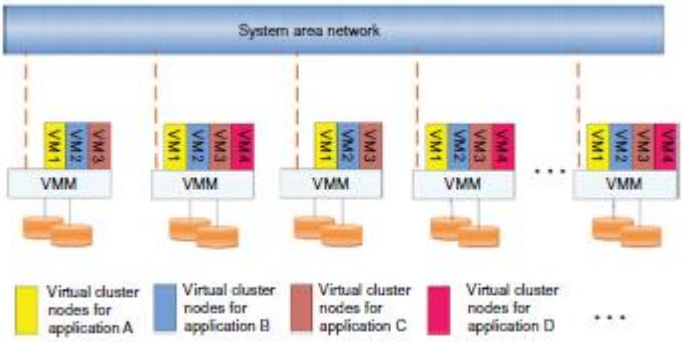


FIGURE 3.19
The concept of a virtual cluster based on application partitioning.

the disk image, configuring the VMs, choosing the destination nodes, and executing the VM deployment command on every host. Many systems use templates to simplify the disk image preparation process. A template is a disk image that includes a preinstalled operating system with or without certain application software. Users choose a proper template according to their requirements and make a duplicate of it as their own disk image. Templates could implement the COW (Copy on Write) format. A new COW backup file is very small and easy to create and transfer. Therefore, it definitely reduces disk space consumption. In addition, VM deployment time is much shorter than that of copying the whole raw image file. Every VM is configured with a name, disk image, network setting, and allocated CPU and memory. One needs to record each VM configuration into a file. However, this method is inefficient when managing a large group of VMs. VMs with the same configurations could use preedited profiles to simplify the process. In this scenario, the system configures the VMs according to the chosen profile. Most configuration items use the same settings, while some of them, such as UUID, VM name, and IP address, are assigned with automatically calculated values. Normally, users do not care which host is running their VM. A strategy to choose the proper destination host for any VM is needed. The deployment principle is to fulfill the VM requirement and to balance workloads among the whole host network.

Live VM Migration Steps and Performance Effects

In a cluster built with mixed nodes of host and guest systems, the normal method of operation is to run everything on the physical machine. When a VM fails, its role could be replaced by another VM on a different node, as long as they both run with the same guest OS. In other words, a physical node can fail over to a VM on another host. This is different from physical-to-physical failover in a traditional physical cluster. The advantage is enhanced failover flexibility. The potential drawback is that a VM must stop playing its role if its residing host node fails. However, this problem can be mitigated with VM life migration. Figure 3.20 shows the process of life migration of a VM from host A to host B. The migration copies the VM state file from the storage area to the host machine. Live migration process of a VM from one host to

another. Courtesy of C. Clark, et al. are four ways to manage a virtual cluster. First, you can use a guest-based manager, by which the cluster manager resides on a guest system. In this case, multiple VMs form a virtual cluster. For example, openMosix is an open source Linux cluster running different guest systems on top of the Xen hypervisor. Another example is Sun's cluster Oasis, an experimental Solaris cluster of VMs supported by a VMware VMM.

Second, you can build a cluster manager on the host systems. The host-based manager supervises the guest systems and can restart the guest system on another physical machine. A good example is the VMware HA system that can restart a guest system after failure.

These two cluster management systems are either guest-only or host-only, but they do not mix. A third way to manage a virtual cluster is to use an independent cluster manager on both the host and guest systems. This will make infrastructure management more complex, however. Finally, you can use an integrated cluster on the guest and host systems. This means the manager must be designed to distinguish between virtualized resources and physical resources. Various cluster management schemes can be greatly enhanced when VM life migration is enabled with minimal overhead.

VMs can be live-migrated from one physical machine to another; in case of failure, one VM can be replaced by another VM. Virtual clusters can be applied in computational grids, cloud platforms, and high-performance computing (HPC) systems. The major attraction of this scenario is that virtual clustering provides dynamic resources that can be quickly put together upon user demand or after a node failure. In particular, virtual clustering plays a key role in cloud computing. When a VM runs a live service, it is necessary to make a trade-off to ensure that the migration occurs in a manner that minimizes all three metrics. The motivation is to design a live VM migration scheme with negligible downtime, the lowest network bandwidth consumption possible, and a reasonable total migration time.

Furthermore, we should ensure that the migration will not disrupt other active services residing in the same host through resource contention (e.g., CPU, network bandwidth). A VM can be in one of the following four states. An inactive state is defined by the virtualization platform, under which the VM is not enabled. An active state refers to a VM that has been instantiated at the virtualization platform to perform a real task. A paused state corresponds to a VM that has been instantiated but disabled to process a task or paused in a waiting state. A VM enters the suspended state if its machine file and virtual resources are stored back to the disk, live migration of a VM consists of the following six steps:

Steps 0 and 1: Start migration. This step makes preparations for the migration,including determining the migrating VM and the destination host. Although users could manually make a VM migrate to an appointed host, in most circumstances, the migration is automatically started by strategies such as load balancing and serverconsolidation.

Steps 2: Transfer memory. Since the whole execution state of the VM is stored in memory,

sending the VM's memory to the destination node ensures continuity of the service provided by the VM. All of the memory data is transferred in the first round, and then the migration controller recopies the memory data which is changed in the last round. These steps keep iterating until the dirty portion of the memory is small enough to handle the final copy. Although precopying memory is performed iteratively, the execution of programs is not obviously interrupted.

Step 3: Suspend the VM and copy the last portion of the data. The migrating VM's execution is suspended when the last round's memory data is transferred. Other nonmemory data such as CPU and network states should be sent as well. During this step, the VM is stopped and its applications will no longer run. This ‐service unavailable‐ time is called the ‐downtime‐ of migration, which should be as short as possible so that it can be negligible to users.

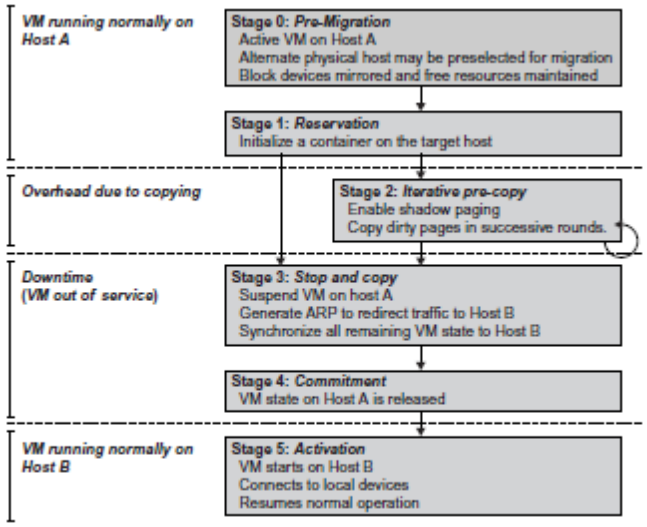


FIGURE 3.20
Live migration process of a VM from one host to another.

Steps 4 and 5: Commit and activate the new host. After all the needed data is copied, on the destination host, the VM reloads the states and recovers the execution of programs in it, and the service provided by this VM continues. Then the network connection is redirected to the new VM and the dependency to the source host is cleared. The whole migration process finishes by removing the original VM from the source host.

The effect on the data transmission rate (Mbit/second) of live migration of a VM from one host to another. Before copying the VM with 512 KB files for 100 clients, the data throughput was 870 MB/second. The first precopy takes 63 seconds, during which the rate is reduced to 765 MB/second. Then the data rate reduces to 694 MB/second in 9.8 seconds for more iterations of the copying process. The system experiences only 165 ms of downtime, before the VM is restored at the destination host. This experimental result shows a very small migration overhead in live transfer of a VMbetween host nodes. This is critical to achieve dynamic cluster reconfiguration and disaster recovery as needed in cloud computing.

Effect on data transmission rate of a VM migrated from one failing web server to another. Courtesy ofC. Clark, et al. With the emergence of widespread cluster computing more than a

decade ago, manycluster configuration and management systems have been developed to achieve a range of goals. These goals naturally influence individual approaches to cluster management. VM technology has become a popular method for simplifying management and sharing of physical computing resources. Platforms such as VMware and Xen allow multiple VMs with different operating systems and configurations to coexist on the same physical host in mutual isolation. Clustering inexpensive computers is an effective way to obtain reliable, scalable computing power for network services and compute-intensive applications

Migration of Memory, Files, and Network Resources

Since clusters have a high initial cost of ownership, including space, power conditioning, and cooling equipment, leasing or sharing access to a common cluster is an attractive solution when demands vary over time. Shared clusters offer economies of scale and more effective utilization of resources by multiplexing. Early configuration and management systems focus on expressive and scalable mechanisms for defining clusters for specific types of service, and physically partition cluster nodes among those types. When one system migrates to another physical node, we should consider the following issues.

Memory Migration

This is one of the most important aspects of VM migration. Moving the memory instance of a VM from one physical host to another can be approached in any number of ways. But

traditionally, the concepts behind the techniques tend to share common implementation paradigms. The techniques employed for this purpose depend upon the characteristics of application/workloads supported by the guest OS.

Memory migration can be in a range of hundreds of megabytes to a few gigabytes in a typical system today, and it needs to be done in an efficient manner. The Internet Suspend-Resume (ISR) technique exploits temporal locality as memory states are likely to have considerable overlap in the suspended and the resumed instances of a VM. Temporal locality refers to the fact that the memory states differ only by the amount of work done since a VM was last suspended before being initiated for migration.

To exploit temporal locality, each file in the file system is represented as a tree of small subfiles. A copy of this tree exists in both the suspended and resumed VM instances. The advantage of using a tree-based representation of files is that the caching ensures the transmission of only those files which have been changed. The ISR technique deals with situations where the migration of live machines is not a necessity. Predictably, the downtime (the period during which the service is unavailable due to there being no currently executing instance of a VM) is high, compared to some of the other techniques

File System Migration

To support VM migration, a system must provide each VM with a consistent, location-

independent view of the file system that is available on all hosts. A simple way to achieve this is to provide each VM with its own virtual disk which the file system is mapped to and transport the contents of this virtual disk along with the other states of the VM. However, due to the current trend of high-capacity disks, migration of the contents of an entire disk over a network is not a viable solution. Another way is to have a global file system across all machines where a VM could be located. This way removes the need to copy files from one machine to another because all files are network-accessible.

A distributed file system is used in ISR serving as a transport mechanism for propagating a suspended VM state. The actual file systems themselves are not mapped onto the distributed file system. Instead, the VMM only accesses its local file system. The relevant VM files are explicitly copied into the local file system for a resume operation and taken out of the local file system for a suspend operation. This approach relieves developers from the complexities of implementing several different file system calls for different distributed file systems. It also essentially disassociates the VMM from any particular distributed file system semantics.
However, this decoupling means that the VMM has to store the contents of each VM's virtual disks in its local files, which have to be moved around with the other state information of that VM.

In smart copying, the VMM exploits spatial locality. Typically, people often move between the same small number of locations, such as their home and office. In these conditions,it is possible to transmit only the difference between the two file systems at suspending and resuming locations. This technique significantly reduces the amount of actual physical data that has to be moved. In situations where there is no locality to exploit, a different approach is to synthesize much of the state at the resuming site. On many systems, user files only form a small fraction of the actual data on disk. Operating system and application software account for the majority of storage space. The proactive state transfer solution works in those cases where the resuming site can be predicted with reasonable confidence.

Network Migration

A migrating VM should maintain all open network connections without relying on forwarding mechanisms on the original host or on support from mobility or redirection mechanisms. To enable remote systems to locate and communicate with a VM, each VM must be assigned a virtual IP address known to other entities. This address can be distinct from the IP address of the host machine where the VM is currently located. Each VM can also have its own distinct virtual MAC address. The VMM maintains a mapping of the virtual IP and MAC addresses to their corresponding VMs. In general, a migrating VM includes all the protocol states and carries its IP address with it.

If the source and destination machines of a VM migration are typically connected to a single switched LAN, an unsolicited ARP reply from the migrating host is provided advertising that the IP has moved to a new location. This solves the open network connection problem by

reconfiguring all the peers to send future packets to a new location. Although a few packets that have already been transmitted might be lost, there are no other problems with this mechanism. Alternatively, on a switched network, the migrating OS can keep its original Ethernet MAC address and rely on the network switch to detect its move to a new port.

Live migration means moving a VM from one physical node to another while keeping its OS environment and applications unbroken. This capability is being increasingly utilized in today's enterprise environments to provide efficient online system maintenance, reconfiguration, load balancing, and proactive fault tolerance. It provides desirable features to satisfy requirements for computing resources in modern computing systems, including server consolidation, performance isolation, and ease of management. As a result, many implementations are available which support the feature using disparate functionalities. Traditional migration suspends VMs before the transportation and then resumes them at the end of the process. By importing the precopy mechanism, a VM could be live-migrated without stopping the VM and keep the applications running during the migration. Live migration is a key feature of system virtualization technologies. Here, we focus on VM migration within a cluster environment where a network-accessible storage system, such as storage area network (SAN) or network attached storage (NAS), is employed. Only memory and CPU status needs to be transferred from the source node to the target node. Live migration techniques mainly use the precopy approach, which first transfers all memory pages, and then only copies modified pages during the last round

iteratively. The VM service downtime is expected to be minimal by using iterative copy operations.When applications' writable working set becomes small, the VM is suspended and only the CPU state and dirty pages in the last round are sent out to the destination. In the precopy phase, although a VM service is still available, much performance degradation will occur because the migration daemon continually consumes network bandwidth to transfer dirty pages in each round. An adaptive rate limiting approach is employed to mitigate this issue, but total migration time is prolonged by nearly 10 times. Moreover, the maximum number of iterations must be set because not all applications' dirty pages are ensured to converge to a small writable working set over multiple rounds. In fact, these issues with the precopy approach are caused by the large amount of transferred data during the whole migration process.

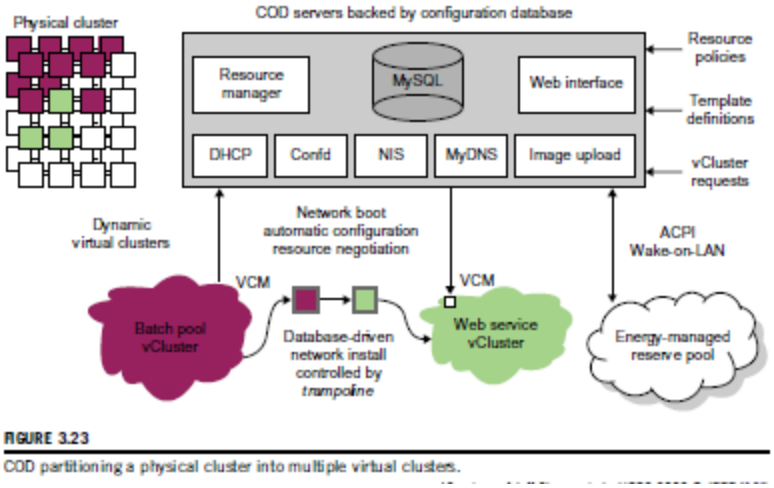


FIGURE 3.23
COD partitioning a physical cluster into multiple virtual clusters.

A checkpointing/recovery and trace/replay approach (CR/TR-Motion) is proposed to provide fast VM migration. This approach transfers the execution trace file in iterations rather than dirty pages, which is logged by a trace daemon. Apparently, the total size of all log files is much less than that of dirty pages. So, total migration time and downtime of migration are drastically reduced. However, CR/TR-Motion is valid only when the log replay rate is larger than the log growth rate. The inequality between source and target nodes limits the application scope of live migration in clusters. Another strategy of postcopy is introduced for live migration of VMs. Here, all memory pages are transferred only once during the whole migration process and the baseline total migration time is reduced. But the downtime is much higher than that of precopy due to the latency of fetching pages from the source node before the VM can be resumed on the target. With the advent of multicore or many-core machines, abundant CPU resources are available. Even if several VMs reside on a same multicore machine, CPU resources are still rich because physical CPUs are frequently amenable to multiplexing. We can exploit these copious CPU resources to compress page frames and the amount of transferred data can be significantly reduced. Memory compression algorithms typically have little memory overhead. Decompression is simple and very fast and requires no memory for decompression.

VIRTUALIZATION FOR DATA-CENTER AUTOMATION

Data centers have grown rapidly in recent years, and all major IT companies are pouring their resources into building new data centers. In addition, Google, Yahoo!, Amazon, Microsoft, HP, Apple, and IBM are all in the game. All these companies have invested billions of dollars in data-center construction and automation. Data-center automation means that huge volumes of hardware, software, and database resources in these data centers can be allocated dynamically to millions of Internet users simultaneously, with guaranteed QoS and cost-effectiveness.

This automation process is triggered by the growth of virtualization products and cloud computing services. From 2006 to 2011, according to an IDC 2007 report on the growth of

virtualization and its market distribution in major IT sectors. In 2006, virtualization has a market share of $1,044 million in business and enterprise opportunities. The majority was dominated by production consolidation and software development. Virtualization is moving towards enhancing mobility, reducing planned downtime (for maintenance), and increasing the number of virtual clients.

The latest virtualization development highlights high availability (HA), backup services, workload balancing, and further increases in client bases. IDC projected that automation, service orientation, policy-based, and variable costs in the virtualization market. The total business opportunities may increase to $3.2 billion by 2011. The major market share moves to the areas of HA, utility computing, production consolidation, and client bases. In what follows, we will discuss server consolidation, virtual storage, OS support, and trust management in automated data-center designs.

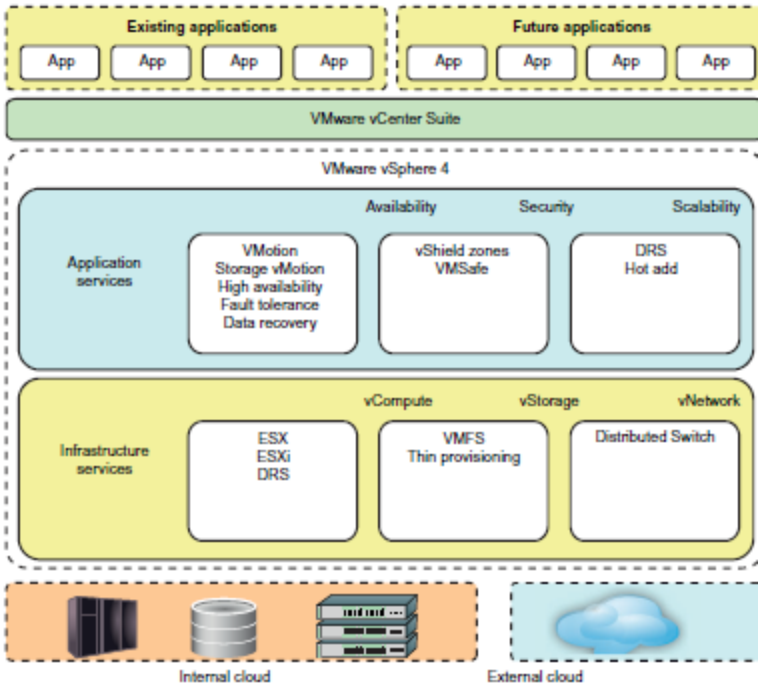


FIGURE 3.28

vSphere/4, a cloud operating system that manages compute, storage, and network resources over virtualized data centers.

Server Consolidation in Data Centers

In data centers, a large number of heterogeneous workloads can run on servers at various times. These heterogeneous workloads can be roughly divided into two categories: chatty workloads and noninteractive workloads. Chatty workloads may burst at some point and return to a silent state at some other point. A web video service is an example of this, whereby a lot of people use it at night and few people use it during the day. Noninteractive workloads do not require people's efforts to make progress after they are submitted. High-performance computing is a typical example of this. At various stages, the requirements for resources of these workloads

are dramatically different. However, to guarantee that a workload will always be able to cope with all demand levels, theworkload is statically allocated enough resources so that peak demand is satisfied.

In this case, the granularity of resource optimization is focused on the CPU, memory, and network interfaces. Therefore, it is common that most servers in data centers are underutilized. A large amount of hardware, space, power, and management cost of these servers is wasted. Server consolidation is an approach to improve the low utility ratio of hardware resources by reducing the number of physical servers. Among several server consolidation techniques such as centralized and physical consolidation, virtualization-based server consolidation is the most powerful. Data centers need to optimize their resource management. Yet these techniques are performed with the granularity of a full server machine, which makes resource management far from well optimized. Server virtualization enables smaller resource allocation than a physical machine.

In general, the use of VMs increases resource management complexity. This causes a challenge in terms of how to improve resource utilization as well as guarantee QoS in data centers. In detail, server virtualization has the following side effects:

• Consolidation enhances hardware utilization. Many underutilized servers are consolidated into fewer servers to enhance resource utilization. Consolidation also facilitates backup services and disaster recovery.

• This approach enables more agile provisioning and deployment of resources. In a virtual environment, the images of the guest OSes and their applications are readily cloned and reused.

• The total cost of ownership is reduced. In this sense, server virtualization causes deferred purchases of new servers, a smaller data-center footprint, lower maintenance costs, and lower power, cooling, and cabling requirements.

• This approach improves availability and business continuity. The crash of a guest OS has no effect on the host OS or any other guest OS. It becomes easier to transfer a VM from one server to another, because virtual servers are unaware of the underlying hardware.

To automate data-center operations, one must consider resource scheduling, architectural support, power management, automatic or autonomic resource management, performance of analytical models, and so on. In virtualized data centers, an efficient, on-demand, fine-grained scheduler is one of the key factors to improve resource utilization. Scheduling and reallocations can be done in a wide range of levels in a set of data centers. The levels match at least at the VM level, server level, and data-center level.

Ideally, scheduling and resource reallocations should be done at all levels. However, due to the complexity of this, current techniques only focus on a single level or, at most, two levels.

Dynamic CPU allocation is based on VM utilization and application-level QoS metrics. One

method considers both CPU and memory flowing as well as automatically adjusting resource overhead based on varying workloads in hosted services. Another scheme uses a two-level resource management system to handle the complexity involved. A local controller at the VM level and a global controller at the server level are designed. They implement autonomic resource allocation via the interaction of the local and global controllers. Multicore and virtualization are two cutting techniques that can enhance each other. However, the use of CMP is far from well optimized. The memory system of CMP is a typical example. One can design a virtual hierarchy on a CMP in data centers. One can consider protocols that minimize the memory access time, inter-VM interferences, facilitating VM reassignment, and supporting inter-VM sharing. One can also consider a VM-aware power budgeting scheme using multiple managers integrated to achieve better power management. The power budgeting policies cannot ignore the heterogeneity problems. Consequently, one must address the trade-off of power saving and data-center performance.

Virtual Storage Management

The term ―storage virtualization‖ was widely used before the renaissance of system virtualization. Yet the term has a different meaning in a system virtualization environment. Previously, storage virtualization was largely used to describe the aggregation and repartitioning of disks at very coarse time scales for use by physical machines. In system virtualization, virtual storage includes the storage managed by VMMs and guest

OSes. Generally, the data stored in this environment can be classified into two categories: VM images and application data. The VM images are special to the virtual environment, while application data includes all other data which is the same as the data in traditional OS environments. The most important aspects of system virtualization are encapsulation and isolation.

Traditional operating systems and applications running on them can be encapsulated in VMs. Only one operating system runs in a virtualization while many applications run in the operating system. System virtualization allows multiple VMs to run on a physical machine and the VMs are completely isolated. To achieve encapsulation and isolation, both the system software and the hardware platform, such as CPUs and chipsets, are rapidly updated. However, storage is lagging. The storage systems become the main bottleneck of VM deployment.

In virtualization environments, a virtualization layer is inserted between the hardware and traditional operating systems or a traditional operating system is modified to support virtualization. This procedure complicates storage operations. On the one hand, storage management of the guest OS performs as though it is operating in a real hard disk while the guest OSes cannot access the hard disk directly. On the other hand, many guest OSes contest the hard disk when many VMs are running on a single physical machine. Therefore, storage management of the underlying VMM is much more complex than that of guest OSes (traditional OSes).

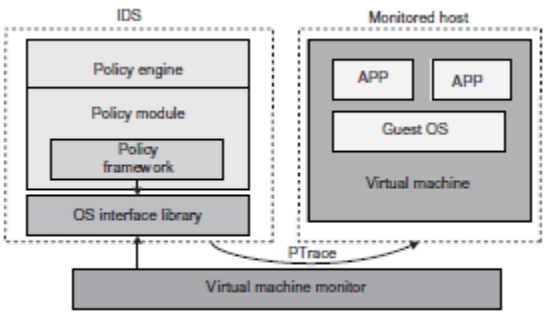


FIGURE 3.29
The architecture of livewire for intrusion detection using a dedicated VM.

In addition, the storage primitives used by VMs are not nimble. Hence, operations such as remapping volumes across hosts and checkpointing disks are frequently clumsy and esoteric, and sometimes simply unavailable. In data centers, there are often thousands of VMs, which cause the VM images to become flooded. Many researchers tried to solve these problems in virtual storage management. The main purposes of their research are to make management easy while enhancing performance and reducing the amount of storage occupied by the VM images. Parallax is a distributed storage system customized for virtualization environments. Content Addressable Storage (CAS) is a solution to reduce the total size of VM images, and therefore supports a large set of VM-based systems in data centers.

Since traditional storage management techniques do not consider the features of storage in virtualization environments, Parallax designs a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switchers are relocated into a federation of storage VMs. These storage VMs share the same physical hosts as the VMs that they serve. It provides an overview of the Parallax system architecture. It supports all popular system virtualization techniques, such as para virtualization and full virtualization. For each physical machine, Parallax customizes a special storage appliance VM. The storage appliance VM acts as a block virtualization layer between individual VMs and the physical storage device. It provides a virtual disk for each VM on the same physical machine.

UNIT IV

OPEN SOURCE GRID MIDDLEWARE PACKAGES

As reviewed in Berman, Fox, and Hey , many software, middleware, and programming environments have been developed for grid computing over past 15 years. Below we assess their relative strength and limitations based on recently reported applications. We first introduce some grid standards and popular APIs. Then we present the desired software support and middleware developed for grid computing includesfour grid middleware packages.

Grid Software Support and Middleware Packages

BOINC Berkeley Open Infrastructure for Network Computing.

UNICORE Middleware developed by the German grid computing community.

Globus (GT4) A middleware library jointly developed by Argonne National Lab., Univ. of Chicago, and USC Information Science Institute, funded by DARPA, NSF, and NIH. CGSP in

ChinaGrid

The CGSP (ChinaGrid Support Platform) is a middleware library developed by 20 top universities in China as part of the ChinaGrid Project .

Condor-G Originally developed at the Univ. of Wisconsin for general distributed computing, and later extended to Condor-G for grid job management. .

Sun Grid Engine (SGE)

Developed by Sun Microsystems for business grid applications. Applied to private grids and local clusters within enterprises or campuses.

Grid Standards and APIs

Grid standards have been developed over the years. The Open Grid Forum (formally Global Grid Forum) and Object Management Group are two well-formed organizations behind those standards. We have already introduced the OGSA (Open Grid Services Architecture) in standards including the GLUE for resource representation, SAGA (Simple API for Grid Applications), GSI (Grid Security Infrastructure), OGSI (Open Grid Service Infrastructure), and WSRE (Web Service Resource Framework).

The grid standards have guided the development of several middleware libraries and API tools for grid computing. They are applied in both research grids and production grids today. Research grids tested include the EGEE, France Grilles, D-Grid (German), CNGrid (China), TeraGrid (USA), etc. Production grids built with the standards include the EGEE, INFN grid (Italian), NorduGrid, Sun Grid, Techila, and Xgrid . We review next the software environments and middleware implementations based on these standards.

Software Support and Middleware

Grid middleware is specifically designed a layer between hardware and the software. The middleware products enable the sharing of heterogeneous resources and managing virtual organizations created around the grid. Middleware glues the allocated resources with specific user applications. Popular grid middleware tools include the Globus Toolkits (USA), gLight, UNICORE (German), BOINC (Berkeley), CGSP (China), Condor-G, and Sun Grid Engine, etc. summarizes the grid software support and middleware packages developed for grid systems since 1995. In subsequent sections, we will describe the features in Condor-G, SGE, GT4, and CGSP.

THE GLOBUS TOOLKIT ARCHITECTURE (GT4)

The Globus Toolkit is an open middleware library for the grid computing communities. These open source software libraries support many operational grids and their applications on an international basis. The toolkit addresses common problems and issues related to grid resource discovery, management, communication, security, fault detection, and portability. The software itself provides a variety of components and capabilities. The library includes a rich set of service implementations. The implemented software supports grid infrastructure management, provides tools for building new web services in Java, C, and Python, builds a powerful standard-based.

Security infrastructure and client APIs (in different languages), and offers comprehensive command-line programs for accessing various grid services. The Globus Toolkit was initially motivated by a desire to remove obstacles that prevent seamless collaboration, and thus sharing of resources and services, in scientific and engineering applications. The shared resources can be computers, storage, data, services, networks, science instruments (e.g., sensors), and so on.

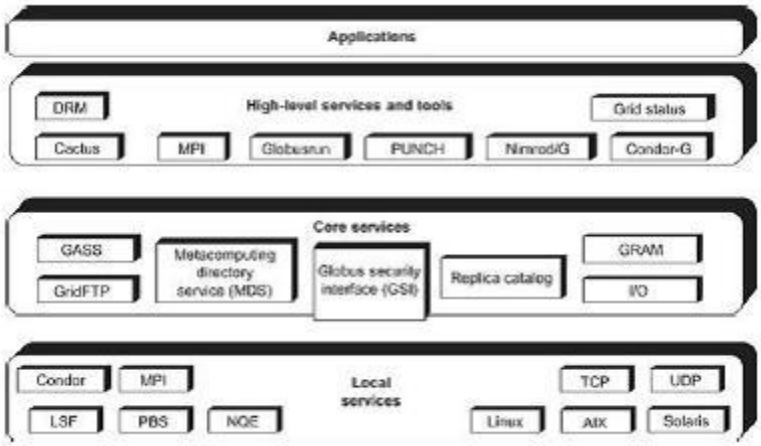


Figure: Globus Toolkit GT4 supports distributed and cluster computing services

The GT4 Library

The GT4 Library GT4 offers the middle-level core services in grid applications. The high-level services and tools, such as MPI, Condor-G, and Nirod/G, are developed by third

parties for general purpose distributed computing applications. The local services, such as LSF, TCP, Linux, and Condor, are at the bottom level and are fundamental tools supplied by other developers.

Globus Job Workflow

A typical job execution sequence proceeds as follows: The user delegates his credentials to a delegation service. The user submits a job request to GRAM with the delegation identifier as a parameter. GRAM parses the request, retrieves the user proxy certificate from the delegation service, and then acts on behalf of the user. GRAM sends a transfer request to the RFT, which applies GridFTP to bring in the necessary files.

GRAM invokes a local scheduler via a GRAM adapter and the SEG initiates a set of user jobs. The local scheduler reports the job state to the SEG. Once the job is complete, GRAM uses RFT and GridFTP to stage out the resultant files.

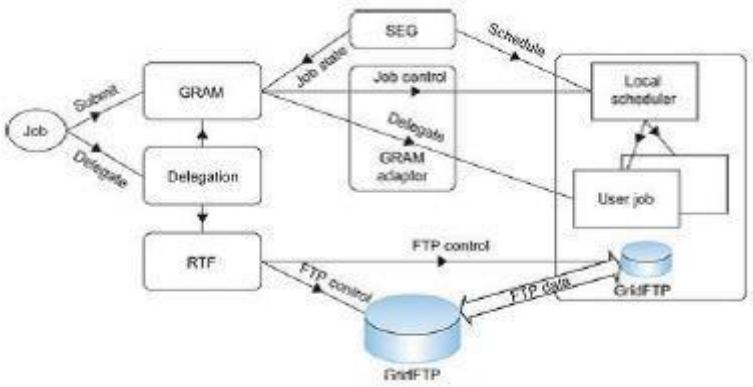


Figure: Globus job workflow among interactive functional modules.

Client-Globus Interactions

There are strong interactions between provider programs and user code. GT4 makes heavy use of industry-standard web service protocols and mechanisms in service

Description, discovery, access, authentication, authorization, and the like. GT4 makes extensive use of Java, C, and Python to write user code. Web service mechanisms define

specific interfaces for grid computing. Web services provide flexible, extensible, and widely adopted XML-based interfaces.

These demand computational, communication, data, and storage resources. We must enable a range of end-user tools that provide the higher-level capabilities needed in specific user applications. Developers can use these services and libraries to build simple and complex systems quickly.

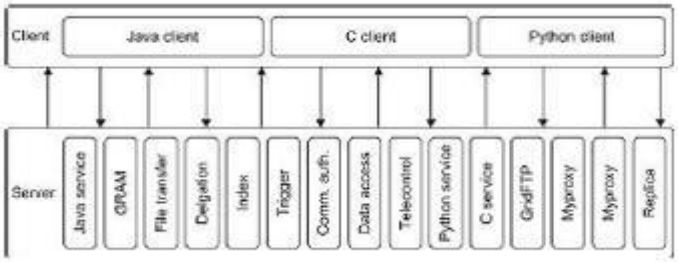
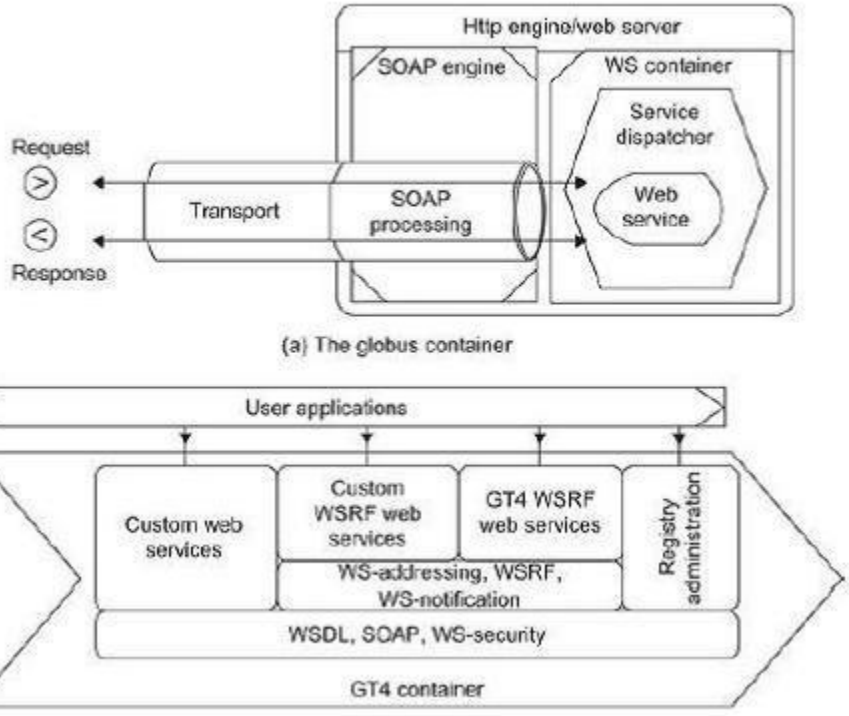


Figure: Client and GT4 server interactions; vertical boxes correspond to service programs and horizontal boxes represent the user codes.

The horizontal boxes in the client domain denote custom applications and/or third-party tools that access GT4 services. The toolkit programs provide a set of useful infrastructure services.

Three containers are used to host user-developed services written in Java, Python, and C, respectively. These containers provide implementations of security, management, discovery, state management, and other mechanisms frequently required when building services.



2. Explain MapReduce Model in detail

The model is based on two distinct steps for an application:

• ***Map***: An initial ingestion and transformation step, in which individual input records can be processed in parallel.

• ***Reduce***: An aggregation or summarization step, in which all associated records must be processed together by a single entity.

The core concept of MapReduce in Hadoop is that input may be split into logical chunks, and each chunk may be initially processed independently, by a map task. The results of these individual processing chunks can be physically partitioned into distinct sets, which are then sorted. Each sorted chunk is passed to a reduce task.

A map task may run on any compute node in the cluster, and multiple map tasks may be running in parallel across the cluster. The map task is responsible for transforming the input records into key/value pairs. The output of all of the maps will be partitioned, and each partition will be sorted. There will be one partition for each reduce task. Each partition's sorted keys and the values associated with the keys are then processed by the reduce task. There may be multiple reduce tasks running in parallel on the cluster.

The application developer needs to provide only four items to the Hadoop framework: the class that will read the input records and transform them into one key/value pair per record, a map method, a reduce method, and a class that will transform the key/value pairs that the reduce method outputs into output records.

My first MapReduce application was a specialized web crawler. This crawler received as input large sets of media URLs that were to have their content fetched and processed. The media items were large, and fetching them had a significant cost in time and resources.

The job had several steps:

1. Ingest the URLs and their associated metadata.
2. Normalize the URLs.
3. Eliminate duplicate URLs.
4. Filter the URLs against a set of exclusion and inclusion filters.
5. Filter the URLs against a do not fetch list.
6. Filter the URLs against a recently seen set.
7. Fetch the URLs.
8. Fingerprint the content items.
9. Update the recently seen set.
10. Prepare the work list for the next application.

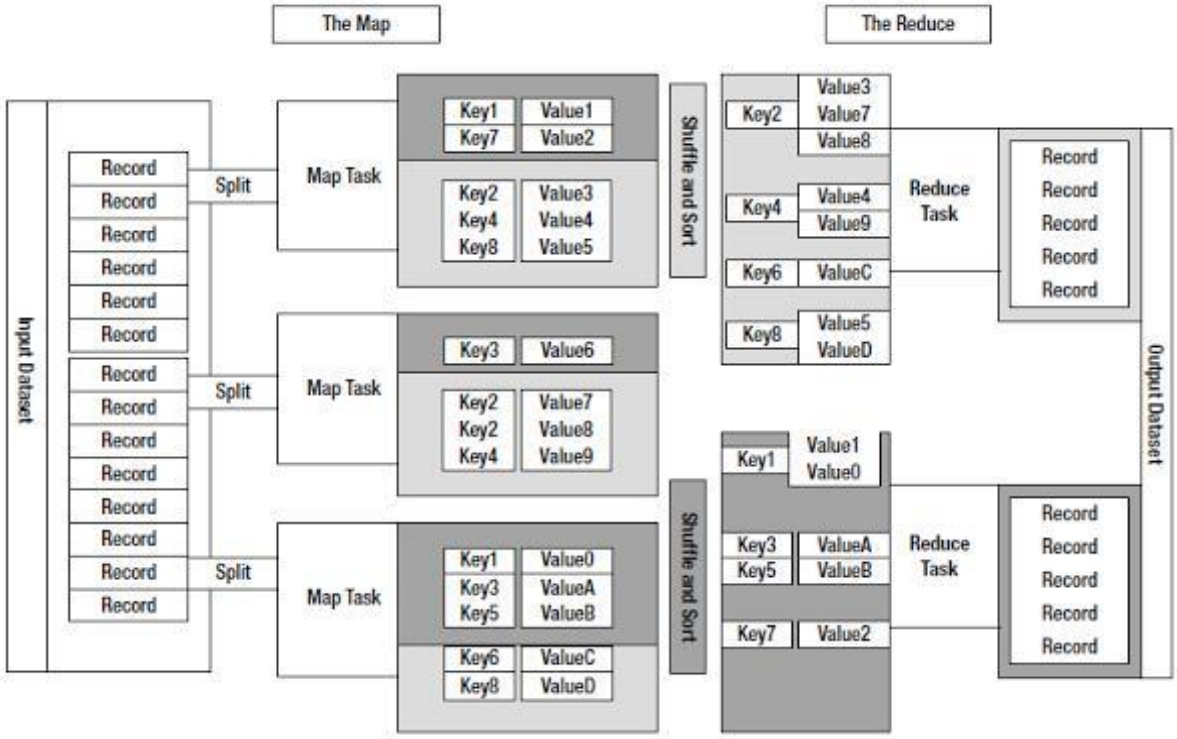


Figure: *The MapReduce model*

Introducing Hadoop

Hadoop is the Apache Software Foundation top-level project that holds the various Hadoop subprojects that graduated from the Apache Incubator. The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications. The Hadoop framework handles the processing details, leaving developers free to focus on application logic.

The introduction on the Hadoop project web page states:
The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing, including:

Hadoop Core, our flagship sub-project, provides a distributed filesystem (HDFS) and support for the MapReduce distributed computing metaphor.
HBase builds on Hadoop Core to provide a scalable, distributed database.

Pig is a high-level data-flow language and execution framework for parallel computation.It is built on top of Hadoop Core.

ZooKeeper is a highly available and reliable coordination system. Distributed applications use ZooKeeper to store and mediate updates for critical shared state.

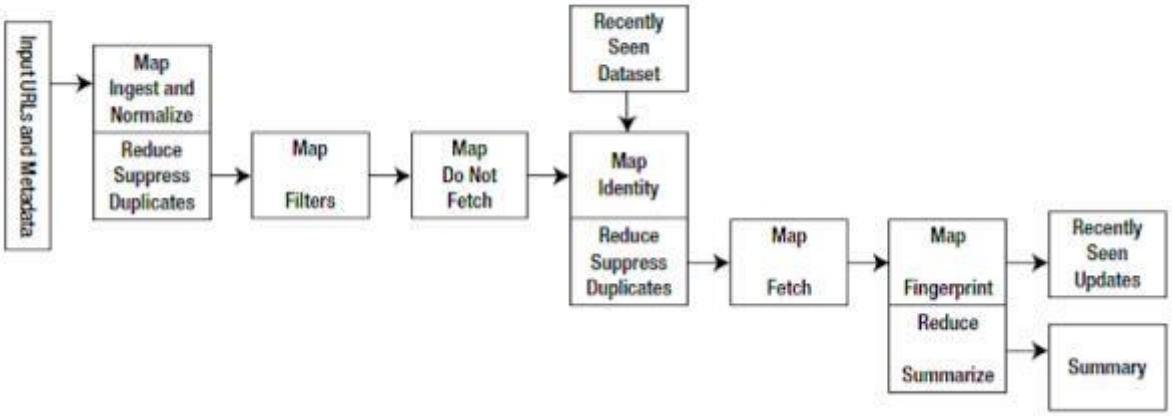
Hive is a data warehouse infrastructure built on Hadoop Core that provides data summarization, adhoc querying and analysis of datasets.

The Hadoop Core project provides the basic services for building a cloud computing environment with commodity hardware, and the APIs for developing software that will run on that cloud.

The two fundamental pieces of Hadoop Core are the MapReduce framework, the cloud computing environment, and he Hadoop Distributed File System (HDFS).

The Hadoop Core MapReduce framework requires a shared file system. This shared file system does not need to be a system-level file system, as long as there is a distributed file system plug-in available to the framework.

The Hadoop Core framework comes with plug-ins for HDFS, CloudStore, and S3. Users are also free to use any distributed file system that is visible as a system-mounted file system, such as Network File System (NFS), Global File System (GFS), or Lustre.



The Hadoop Distributed File System (HDFS)MapReduce environment provides the user with a sophisticated framework to manage the execution of map and reduce tasks across a cluster of machines.

The user is required to tell the framework the following:

• The location(s) in the distributed file system of the job input
• The location(s) in the distributed file system for the job output
• The input format
• The output format
• The class containing the map function
• Optionally. the class containing the reduce function

• The JAR file(s) containing the map and reduce functions and any support classes

The final output will be moved to the output directory, and the job status will be reported to the user.MapReduce is oriented around key/value pairs. The framework will convert each record of input into a key/value pair, and each pair will be input to the map function once. The map output is a set of key/value pairs—nominally one pair that is the transformed input pair. The map output pairs are grouped and sorted by key. The reduce function is called one time for each key, in sort sequence, with the key and the set of values that share that key. The reduce method may output an arbitrary number of key/value pairs, which are written to the output files in the job output directory. If the reduce output keys are unchanged from the reduce input keys, the final output will be sorted. The framework provides two processes that handle the management of MapReduce jobs:

• TaskTracker manages the execution of individual map and reduce tasks on a compute node in the cluster.

• JobTracker accepts job submissions, provides job monitoring and control, and manages the distribution of tasks to the TaskTracker nodes.

The JobTracker is a single point of failure, and the JobTracker will work around the failure of individual TaskTracker processes.

The Hadoop Distributed File System

HDFS is a file system that is designed for use for MapReduce jobs that read input in large chunks of input, process it, and write potentially large chunks of output. HDFS does not handle random access particularly well. For reliability, file data is simply mirrored to multiple storage nodes. This is referred to as *replication* in the Hadoop community. As long as at least one replica of a data chunk is available, the consumer of that data will not know of storage server failures.

HDFS services are provided by two processes:

• NameNode handles management of the file system metadata, and provides management and control services.

• DataNode provides block storage and retrieval services.

There will be one NameNode process in an HDFS file system, and this is a single point of failure. Hadoop Core provides recovery and automatic backup of the NameNode, but no hot failover services. There will be multiple DataNode processes within the cluster, with typically one DataNode process per storage node in a cluster.

3. Explain Map & Reduce function?

A Simple Map Function: IdentityMapper

The Hadoop framework provides a very simple map function, called IdentityMapper. It is used in jobs that only need to reduce the input, and not transform the raw input. All map functions must implement the Mapper interface, which guarantees that the map function will always be called with a key. The key is an instance of a WritableComparable object, a value that is an instance of a Writable object, an output object, and a reporter.

IdentityMapper.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Implements the identity function, mapping inputs directly to outputs.
*/ public class IdentityMapper<K, V>
extends MapReduceBase implements Mapper<K, V, K, V> {
/** The identify function. Input key/value pair is written directly to
* output.*/
public void map(K key, V val,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
output.collect(key, val);
}
}
```

A Simple Reduce Function: IdentityReducer

The Hadoop framework calls the reduce function one time for each unique key. The framework provides the key and the set of values that share that key.

IdentityReducer.java

```
package org.apache.hadoop.mapred.lib;
import java.io.IOException;
import java.util.Iterator;
```

```
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;
/** Performs no reduction, writing all input values directly to the output.
*/ public class IdentityReducer<K, V>
extends MapReduceBase implements Reducer<K, V, K, V> {
```
Chapter 2 ■ THE BASICS OF A MAPREDUCE JOB **35**

```
/** Writes all keys and values directly to output. */
public void reduce(K key, Iterator<V> values,
OutputCollector<K, V> output, Reporter reporter)
throws IOException {
while (values.hasNext()) {
output.collect(key, values.next());
}
}
```

If you require the output of your job to be sorted, the reducer function must pass the key objects to the output.collect() method unchanged. The reduce phase is, however, free to output any number of records, including zero records, with the same key and different values.

4. Explain HDFS Concepts in detail? Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes. HDFShas the concept of a block, but it is a much larger unit—64 MB by default.File0sin HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Simplicity is something to strive for all in all systems, but is especially important for a distributed system in which the failure modes are so varied. The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns

Namenodes and Datanodes

An HDFS cluster has two types of node operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree.

The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. Datanodes are the workhorses of the filesystem. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount.

It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

HDFS Federation

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.

HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second

Namenode might handle files under */share*.Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Block pool storage is *not* partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

HDFS High-Availability

The combination of replicating namenode metadata on multiple filesystems, and using the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a *single point of failure* (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly-available shared storage to share the edit log.

When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.

- Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

If the active namenode fails, then the standby can take over very quickly since it has the latest state available in memory: both the latest edit log entries, and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), since the system needs to be conservative in deciding that the active namenode has failed.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*. Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active. Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, in the case of routine maintenance, for example.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*. The system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory, and disabling

its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique rather graphically known as *STONITH*, or "shoot the other node in the head", which uses a specialized power distribution unit to forcibly power down the host machine. Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname which is mapped to a pair of namenode addresses, and the client library tries each namenode address until the operation succeeds.

5. Explain Anatomy of a File Read?

The client opens the file it wishes to read by calling open () on the FileSystem object, which for HDFS is an instance of DistributedFileSystem. DistributedFileSystem calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file. The namenode returns the addresses of the datanodes that have a copy of that block.

If the client is itself a datanode ,then it will read from the local datanode, if it hosts a copy of the block .The DistributedFileSystem returns an FSDataInputStream to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

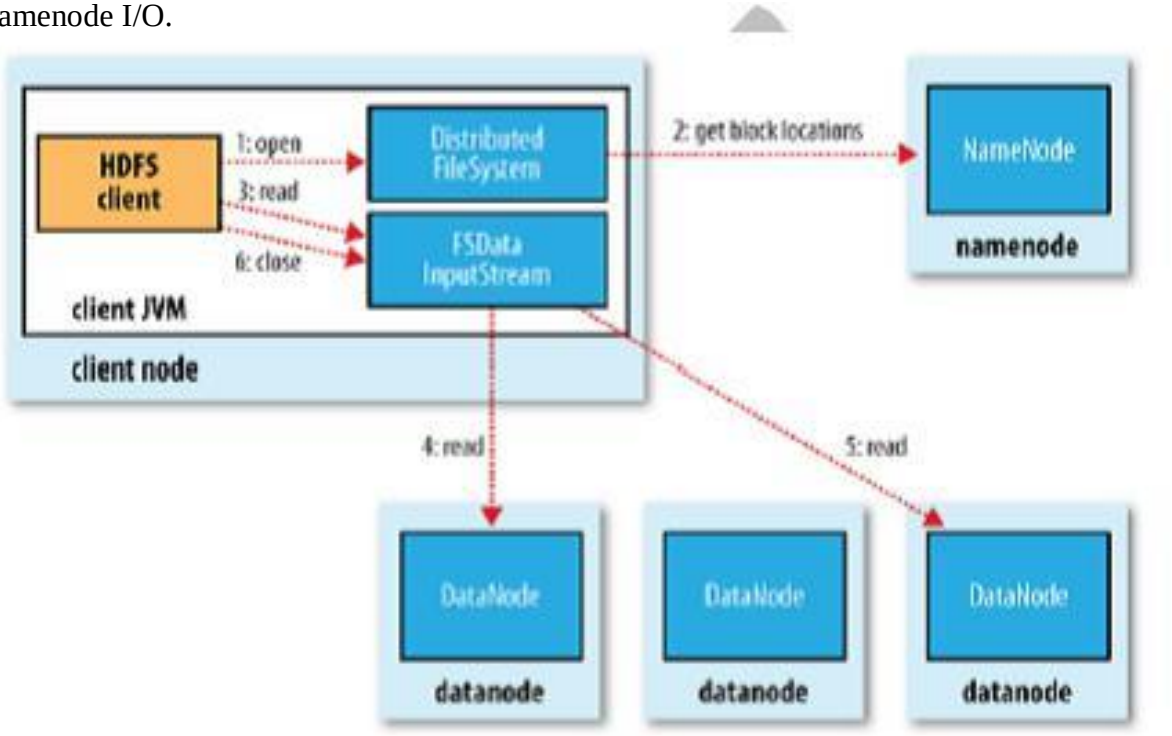


Figure: *A client reading data from HDFS*

The client then calls read () on the stream. DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read () repeatedly on the stream. When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block. This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close () on the FSDataInputStream .

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInputStream also verifies checksums for the data transferred to it from the datanode.
If a corrupted block is found, it is reported to the namenode before the DFSInput Stream attempts to read a replica of the block from another datanode.One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients, since the data traffic is spread across all the datanodes in the cluster.

6. Explain Anatomy of a File write?

The client creates the file by calling create () on DistributedFileSystem. DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist, and that the client has the right permissions to create the file. If these checks pass, the namenode makes a record of the new file.

A client writing data to HDFS

The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutput Stream, which handles communication with the datanodes and namenode.As the client writes data (step 3), DFSOutputStream splits it into packets, which it writes to an internal queue, called the *data queue*. The data queue is consumed by the Data Streamer, whose responsibility it is to ask the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline—we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).If a datanode fails while data is being written to it, then the following actions are taken, which are transparent
to the client writing the data.

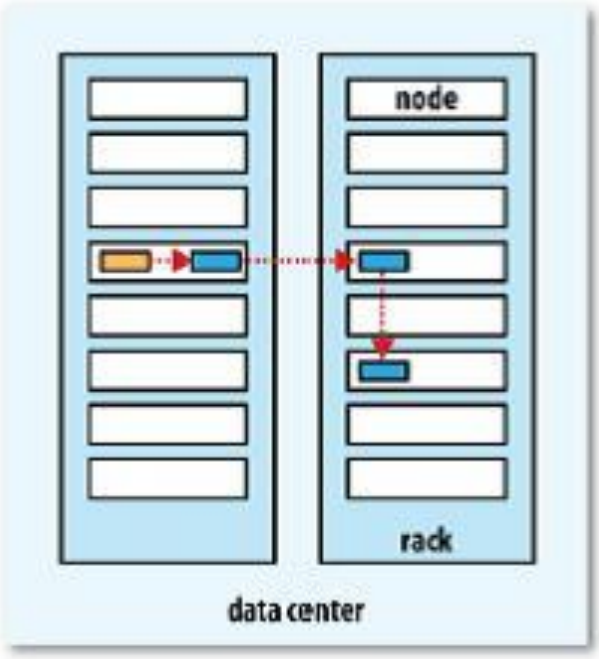


Figure: *A typical replica pipeline*

First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block 0on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed. Datanode recovers later on. The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal. It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas (default one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached.

When the client has finished writing data, it calls close () on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up.

The Globus Toolkit, started in 1995 with funding from DARPA, is an open middleware library for the grid computing communities. These open source software libraries support many operational grids and their applications on an international basis. The toolkit addresses common problems and issues related to grid resource discovery, management, communication, security, fault detection, and portability. The software itself provides a variety of components and capabilities. The library includes a rich set of service implementations. The implemented software supports grid infrastructure management, provides tools for building new web services in Java, C, and Python, builds a powerful standard-based security infrastructure and client APIs (in different languages), and offers comprehensive command-line programs for accessing various grid services. The Globus Toolkit was initially motivated by a desire to remove obstacles that prevent seamless collaboration, and thus sharing of resources and services, in scientific and engineering applications. The shared resources can be computers, storage, data, services, networks, science instruments (e.g., sensors), and so on. The Globus library version GT4. Globus Tookit GT4 supports distributed and cluster computing services.
Courtesy of I. Foster

The GT4 Library

GT4 offers the middle-level core services in grid applications. The high-level services and tools, such as MPI, Condor-G, and Nirod/G, are developed by third parties for general-purpose distributed computing applications. The local services, such as LSF, TCP, Linux, and Condor, are at the bottom level and are fundamental tools supplied by other developers summarizes GT4's core grid services by module name. Essentially, these functional

moduleshelp users to discover available resources, move data between sites, manage user credentials, and so on. As a de facto standard in grid middleware, GT4 is based on industry-standard web service technologies.

Functional Modules in Globus GT4 Library

Service Functionality Module Name Functional Description

Global Resource Allocation Manager GRAM Grid Resource Access and Management (HTTP-based)

Communication Nexus Unicast and multicast communication

Grid Security Infrastructure GSI Authentication and related security services

Monitory and Discovery Service MDS Distributed access to structure and state information

Health and Status HBM Heartbeat monitoring of system components

Global Access of Secondary Storage GASS Grid access of data in remote secondary storage

Grid File Transfer GridFTP Inter-node fast file transfer

Nexus is used for collective communications and HBM for heartbeat monitoring of resource nodes. GridFTP is for speeding up internode file transfers. The module GASS is used for global access of secondary storage. More details of the functional modules of Globus GT4 and their applications are available at www.globus.org/toolkit/.

Globus Job Workflow

The typical job workflow when using the Globus tools. A typical job execution sequence proceeds as follows: The user delegates his credentials to a delegation service. The user submits a job request to GRAM with the delegation identifier as a parameter. GRAM parses the request, retrieves the user proxy certificate from the delegation service, and then acts on behalf of the user. GRAM sends a transfer request to the RFT (Reliable File Transfer), which applies GridFTP to bring in the necessary files. GRAM invokes a local scheduler via a GRAM adapter and the SEG (Scheduler Event Generator) initiates a set of user jobs. The local scheduler reports the job state to the SEG. Once the job is complete, GRAM uses RFT and GridFTP to stage out the resultant files. The grid monitors the progress of these operations and sends the user a notification when they succeed, fail, or are delayed.

Globus job workflow among interactive functional modules.

Client-Globus Interactions

GT4 service programs are designed to support user applications .There are strong interactions between provider programs and user code. GT4 makes heavy use of industry-standard web

service protocols and mechanisms in service description, discovery, access, authentication, authorization, and the like. GT4 makes extensive use of Java, C, and Python to write user code. Web service mechanisms define specific interfaces for grid computing. Web services provide flexible, extensible, and widely adopted XML-based interfaces.

Client and GT4 server interactions; vertical boxes correspond to service programs and horizontal boxes represent the user codes. Courtesy of Foster and Kesselman GT4 components do not, in general, address end-user needs directly. Instead, GT4 provides a set of infrastructure services for accessing, monitoring, managing, and controlling access to infrastructure elements. The server code in the vertical boxes in corresponds to 15 grid services that are in heavy use in the GT4 library. These demand computational, communication, data, and storage resources. We must enable a range of end-user tools that provide the higher-level capabilities needed in specific user applications. Wherever possible, GT4 implements standards to facilitate construction of operable and reusable user code. Developers can use these services and libraries to build simple and complex systems quickly.

A high-security subsystem addresses message protection, authentication, delegation, and authorization. Comprising both a set of service implementations (server programs at the bottom of Figure 7.21) and associated client libraries at the top, GT4 provides both web services and non-WS applications. The horizontal boxes in the client domain denote custom applications and/or third-party tools that access GT4 services. The toolkit programs provide a set of useful infrastructure services. Globus container serving as a runtime environment for implementing web services in a grid platform. Courtesy of Foster and Kesselman Three containers are used to host user-developed services written in Java, Python, and C, respectively. These containers provide implementations of security, management, discovery, state management, and other mechanisms frequently required when building services. They extend open source service hosting environments with support for a range of useful web service specifications, including WSRF, WS-Notification, and WS-Security. A set of client libraries allow client programs in Java, C, and Python to invoke operations on both GT4 and user-developed services. In many cases, multiple interfaces provide different levels of control: For example, in the case of GridFTP, there is not only a simple command-line client (globus-url-copy) but also control and data channel libraries for use in programs—and the XIO library allowing for the integration of alternative transports. The use of uniform abstractions and mechanisms means clients can interact with different services in similar ways, which facilitates construction of complex, interoperable systems and encourages code reuse Parallel Computing and Programming Paradigms

Consider a distributed computing system consisting of a set of networked nodes or workers. The system issues for running a typical parallel program in either a parallel or a distributed manner would include the following :

• Partitioning This is applicable to both computation and data as follows:

• Computation partitioning This splits a given job or a program into smaller tasks. Partitioning

greatly depends on correctly identifying portions of the job or program that can be performed concurrently. In other words, upon identifying parallelism in the structure of the program, it can be divided into parts to be run on different workers. Different parts may process different data or a copy of the same data.

• Data partitioning This splits the input or intermediate data into smaller pieces.Similarly, upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers. Data pieces may be processed by different parts of a program or a copyof the same program.

• Mapping This assigns the either smaller parts of a program or the smaller pieces of data to underlying resources. This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.

• Synchronization Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed.

Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.

• Communication Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.

• Scheduling For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers. It is worth noting that the resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy. For multiple jobs or programs, a scheduler selects a sequence of jobs or programs to be run on the distributed computing system. In this case, scheduling is also necessary when system resources are not sufficient to simultaneously run multiple jobs or programs.

Motivation for Programming Paradigms

Because handling the whole data flow of parallel and distributed programming is very time-consuming and requires specialized knowledge of programming, dealing with these issues may affect the productivity of the programmer and may even result in affecting the program's time to market. Furthermore, it may detract the programmer from concentrating on the logic of

the program itself. Therefore, parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

In other words, these models aim to provide users with an abstraction layer to hide implementation details of the data flow which users formerly ought to write codes for. Therefore, simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms. Other motivations behind parallel and distributed programming models are (1) to improve productivity of programmers, (2) to decrease programs' time to market, (3) to leverage underlying resources more efficiently, (4) to increase system throughput, and (5) to support higher levels of abstraction .

MapReduce, Hadoop, and Dryad are three of the most recently proposed parallel and distributed programming models. They were developed for information retrieval applications but have been shown to be applicable for a variety of important applications . Further, the loose coupling of components in these paradigms makes them suitable for VM implementation and leads to much better fault tolerance and scalability for some applications than traditional parallel computing models such as MPI .

MapReduce, Twister, and Iterative MapReduce

MapReduce, is a software framework which supports parallel and distributed computing on large data sets . This software framework abstracts the data flow of running a parallel program on a distributed computing system by providing users with two interfaces in the form of two functions:

Map and Reduce. Users can override these two functions to interact with and manipulate the data flow of running their programs illustrates the logical data flow from the Map to the Reduce function in MapReduce frameworks. In this framework, the –value‖ part of the data, (key, value), is the actual data, and the –key‖ part is only used by the MapReduce controller to control the data flow .

Formal Definition of MapReduce

The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling. Here, although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: Map and Reduce . These two main functions can be overridden by the user to achieve specific objectives the MapReduce framework with data flow and control flow.Therefore, the user overrides the Map and Reduce functions first and then invokes the provided MapReduce (Spec, & Results) function from the library to start the flow of data. The MapReduce function, MapReduce (Spec, & Results), takes an important parameter which is a specification object, the Spec. This object is first initialized inside the user's program, and then

the user writes code to fill it with the names of input and output files, as well as other optional tuning parameters. This object is also filled with the name of the Map and Reduce functions to identify these user-defined functions to the MapReduce library.

The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

```
Map Function (… . )
{
 … …
}
Reduce Function (… . )
{
 … …
}
Main Function (… . )
{
 Initialize Spec object
 … …
 MapReduce (Spec, & Results)
}
```

MapReduce Logical Data Flow

The input data to both the Map and the Reduce functions has a particular structure. This also pertains for the output data. The input data to the Map function is in the form of a (key, value) pair. For example, the key is the line offset within the input file and the value is the content of the line. The output data from the Map function is structured as (key, value) pairs called intermediate (key, value) pairs. In other words, the user-defined Map function processes each input (key, value) pair and produces a number of (zero, one, or more) intermediate (key, value) pairs. Here, the goal is to process all input (key, value) pairs to the Map function in parallel .

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs. In turn, the Reduce function receives the intermediate (key, value) pairs in the form of a group of intermediate values associated with one intermediate key, (key, [set of values]). In fact, the MapReduce framework forms these groups by first sorting the intermediate (key, value) pairs and then grouping values with the same key. It should be noted that the data is sorted to simplify the grouping process. The Reduce function processes each (key, [set of values]) group and produces a set of (key, value) pairs as output. To clarify the data flow in a sample MapReduce application, one of the well-known

MapReduce problems, namely word count, to count the number of occurrences of each word in a collection of documents is presented here demonstrates the data flow of the word-count problem for a simple input file containing only two lines as follows: (1) –most people ignore most poetry| and (2) –most poetry ignores most people.| In this case, the Map function simultaneously produces a number of intermediate (key, value) pairs for each line of content so that each word is the intermediate key with 1 as its intermediate value; for example, (ignore, 1). Then the MapReduce library collects all the generated intermediate (key, value) pairs and sorts them to group the 1's for identical words; for example, (people, [1,1]). Groups are then sent to the Reduce function in parallel so that it can sum up the 1 values for each word and generate the actual number of occurrence for each word in the file; for example, (people, 2).

The data flow of a word-count problem using the MapReduce functions (Map, Sort, Group and Reduce) in a cascade operations.

Formal Notation of MapReduce Data Flow

The Map function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs as follows:

Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the –key| part. It then groups the values of all occurrences of the same key. Finally, the Reduce function is applied in parallel to each group producing the collection of values as output as illustrated here:

Strategy to Solve MapReduce Problems

As mentioned earlier, after grouping all the intermediate data, the values of all occurrences of the same key are sorted and grouped together. As a result, after grouping, each key becomes unique in all intermediate data. Therefore, finding unique keys is the starting point to solving a typical MapReduce problem. Then the intermediate (key, value) pairs as the output of the Map function will be automatically found. The following three examples explain how to define keys and values in such problems:

Problem 1: Counting the number of occurrences of each word in a collection ofdocuments

Solution: unique –key‖: each word, intermediate –value‖: number of occurrences

Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents

Solution: unique –key‖: each word, intermediate –value‖: size of the word

Problem 3: Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words –listen‖ and –silent‖).

Solution: unique –key‖: alphabetically sorted sequence of letters for each word (e.g., eilnst‖), intermediate –value‖: number of occurrences

MapReduce Actual Data and Control Flow

The main responsibility of the MapReduce framework is to efficiently run a user's program on a distributed computing system. Therefore, the MapReduce framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows . We summarize this in the following distinct steps:

1. Data partitioning The MapReduce library splits the input data (files), already stored in GFS, into M pieces that also correspond to the number of map tasks.

2. Computation partitioning This is implicitly handled (in the MapReduce framework) byobliging users to write their programs in the form of the Map and Reduce functions. therefore, the MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the Map and the Reduce functions, distributes them, and starts them up on a number of available computation engines.

3. Determining the master and workers The MapReduce architecture is based on a master-worker model. Therefore, one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them. A map/reduce worker is typically a computation engine such as a cluster node to run map/reduce tasks by executing Map/Reduce functions. Steps 4–7 describe the map workers.

4. Reading the input data (data distribution) Each map worker reads its corresponding portion of the input data, namely the input data split, and sends it to its Map function. Although a map worker may run more than one Map function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.

5. Map function Each Map function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.

6. Combiner function This is an optional local function within the map worker which applies to intermediate (key, value) pairs. The user can invoke the Combiner function inside the user program. The Combiner function runs the same code written by users for the Reduce function as its functionality is identical to it. The Combiner function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs. As mentioned in our discussion of logical data flow, the MapReduce framework sorts and groups the data before it is processed by the Reduce function. Similarly, the MapReduce framework will also sort and group the local data on each map worker if the user invokes the Combiner function.

7. Partitioning function As mentioned in our discussion of the MapReduce data flow, the intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one Reduce function to generate the final result. However, in real implementations, since there are M map and R reduce tasks, intermediate (key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one Reduce function only.

Therefore, the intermediate (key, value) pairs produced by each map worker are partitioned into R regions, equal to the number of reduce tasks, by the Partitioning function to guarantee that all (key, value) pairs with identical keys are stored in the same region. As a result, since reduce worker i reads the data of region i of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker I accordingly . To implement this technique, a Partitioning function could simply be a hash function (e.g., Hash(key) mod R) that forwards the data into particular regions. It is also worth noting that the locations of the buffered data in these Rpartitions are sent to the master for later forwarding of data to the reduce workers shows the data flow implementation of all data flow steps. The following are two networking steps:

8. Synchronization MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts when all map tasks finish.

9. Communication Reduce worker i, already notified of the location of region i of all mapworkers, uses a remote procedure call to read the data from the respective region of all map workers. Since all reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network. This issue is one of the major bottlenecks in increasing the performance of such systems . A data transfer module was proposed to schedule data transfers independently .Steps 10 and 11 correspond to the reduce worker domain:

10. Sorting and Grouping When the process of reading the input data is finalized by a reduce worker, the data is initially buffered in the local disk of the reduce worker. Then the reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys. Note that the buffered data is sorted and grouped

because the number of unique keys produced by a map worker may be more than R regions in which more than one key exists in each region of a map worker .

1. Reduce function The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the Reduce function.

Then this function processes its input data and stores the output results in predetermined files in the user's program.

Use of MapReduce partitioning function to link the Map and Reduce workers.

To better clarify the interrelated data control and control flow in the MapReduce framework, shows the exact order of processing control in such a system contrasting with dataflow. Data flow implementation of many functions in the Map workers and in the Reduce workers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.

Control flow implementation of the MapReduce functionalities in Map workers and Reduce workers (running user programs) from input files to the output files under the control of the master user program. Courtesy of Yahoo! Pig Tutorial

Compute-Data Affinity

The MapReduce software framework was first proposed and implemented by Google. The first implementation was coded in C. The implementation takes advantage of GFS as the underlying layer. MapReduce could perfectly adapt itself to GFS. GFS is a distributed file system where files are divided into fixed-size blocks (chunks) and blocks are distributed and stored on cluster nodes. As stated earlier, the MapReduce library splits the input data (files) into fixed-size blocks, and ideally performs the Map function in parallel on each block. In this case, as GFS has already stored files as a set of blocks, the MapReduce framework just needs to send a copy of the user's program containing the Map function to the nodes' already stored data blocks. This is the notion of sending computation toward data rather than sending data toward computation. Note that the default GFS block size is 64 MB which is identical to that of the MapReduce framework.

Twister and Iterative MapReduce

It is important to understand the performance of different runtimes and, in particular, to compare MPI and MapReduce . The two major sources of parallel overhead are load imbalance and communication (which is equivalent to synchronization overhead as communication synchronizes parallel units [threads or processes] in Categories 2 and 6). The communication overhead in MapReduce can be quite high, for two reasons:

• MapReduce reads and writes via files, whereas MPI transfers information directly between nodes over the network.

• MPI does not transfer all data from node to node, but just the amount needed to update information. We can call the MPI flow δ flow and the MapReduce flow full data flow.

The same phenomenon is seen in all –classic parallel‖ loosely synchronous applications which typically exhibit an iteration structure over compute phases followed by communication phases. We can address the performance issues with two important changes:

1. Stream information between steps without writing intermediate steps to disk.

2. Use long-running threads or processors to communicate the δ (between iterations) flow.

These changes will lead to major performance increases at the cost of poorer fault tolerance and ease to support dynamic changes such as the number of available nodes.

This concept has been investigated in several projects while the direct idea of using MPI for MapReduce applications is investigated in . The Twister programming paradigm and its implementation architecture at run time are illustrated whose performance results for K means are shown in Figure 6.8 [55,56], where Twister is much faster than traditional MapReduce. Twister distinguishes the static data which is never reloaded from the dynamic δ flow that is communicated.

Twister: An iterative MapReduce programming paradigm for repeated MapReduce execution

HADOOP LIBRARY FROM APACHE

Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache. The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS) as its underlying layer rather than GFS. The Hadoop core is divided into two fundamental layers: the MapReduce engine and HDFS. The MapReduce engine is the computation engine running on top of HDFS as its data storage manager. The following two sections cover the details of these two fundamental layers.

HDFS: HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.

HDFS Architecture: HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves). To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes). The mapping of blocks to DataNodes is determined by the NameNode. The NameNode (master) also manages the file system's metadata and namespace. In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files. For example, NameNode in the metadata stores all information regarding the location of input splits/blocks in all DataNodes. Each DataNode,

usually one per node in a cluster, manages the storage attached to the node. Each DataNode is responsible for storing and retrieving its file blocks .

HDFS Features: Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements , to operate efficiently. However, because HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system. For example, security has never been supported for HDFS systems. The following discussion highlights two important characteristics of HDFS to distinguish it from other generic distributed file systems .

HDFS Fault Tolerance: One of the main aspects of HDFS is its fault tolerance characteristic. Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception. Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system :

• Block replication To reliably store data in HDFS, file blocks are replicated in this system. In other words, HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster. The replication factor is set by the user and is three by default.

• Replica placement The placement of replicas is another factor to fulfill the desired fault tolerance in HDFS. Although storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability, it is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack. Therefore, sometimes HDFS compromises its reliability to achieve lower communication costs. For example, for the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data .

• Heartbeat and Blockreport messages Heartbeats and Blockreports are periodic messages sent to the NameNode by each DataNode in a cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode. The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

HDFS High-Throughput Access to Large Data Sets (Files): Because HDFS is primarilydesigned for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency. Also, because applications run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64 MB) to allow HDFS to decrease the amount of metadata storage required per file. This provides two advantages: The list of blocks per file will shrink as the size of individual blocks increases, and by keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

HDFS Operation: The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations. In this section, the control flow of the main operations of HDFS on files is further described to manifest the interaction between the user, the NameNode, and the DataNodes in such systems .

• Reading a file To read a file in HDFS, a user sends an –open‖ request to the NameNode to get the location of file blocks. For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file. The number of addresses depends on the number of block replicas. Upon receiving such information, the user calls the read function to connect to the closest DataNode containing the first block of the file. After the first block is streamed from the respective DataNode to the user, the established connection is terminated and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.

• Writing to a file To write a file in HDFS, a user sends a –create‖ request to the NameNode to create a new file in the file system namespace. If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function. The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode. Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.

The steamer then stores the block in the first allocated DataNode. Afterward, the block is forwarded to the second DataNode by the first DataNode. The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.

Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

Architecture of MapReduce in Hadoop

The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems shows the MapReduce engine architecture cooperating with HDFS. Similar to HDFS, the MapReduce engine also has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers). The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers. The TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster. HDFS and MapReduce architecture in Hadoop where boxes with different shadings refer to different functional nodes applied to different blocks of data.

Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task. Slots are defined as the number of simultaneous threads supported by

CPUs of the TaskTracker node. For example, a TaskTracker node with N CPUs, each supporting M threads, has M * N simultaneous execution slots . It is worth noting that each data block is processed by one map task running on a single slot. Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

Running a Job in Hadoop

Three components contribute in running a job in this system: a user node, a JobTracker, and several TaskTrackers. The data flow starts by calling the runJob(conf) function inside a user program running on the user node, in which conf is an object containing some tuning parameters for the MapReduce framework and HDFS. The runJob(conf) function and conf are comparable to the MapReduce(Spec, &Results) function and Spec in the first implementation of MapReduce by Google, depicts the data flow of running a MapReduce job in Hadoop . Data flow in running a MapReduce job at various task trackers using the Hadoop library.

• Job Submission Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the followingprocedure:

• A user node asks for a new job ID from the JobTracker and computes input file splits.

• The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.

• The user node submits the job to the JobTracker by calling the submitJob() function.

• Task assignment The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers.

The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers. The JobTracker also creates reduce tasks and assigns them to the TaskTrackers. The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

• Task execution The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system. Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.

• Task running check A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers. Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.

UNIT V

TRUST MODELS FOR GRID SECURITY ENFORCEMENT

Many potential security issues may occur in a grid environment if qualified security mechanisms are not in place. These issues include network sniffers, out-of-control access, faulty operation, malicious operation, integration of local security mechanisms, delegation, dynamic resources and services, attack provenance, and so on. Computational grids are motivated by the desire to share processing resources among many organizations to solve large-scale problems.
Indeed, grid sites may exhibit unacceptable security conditions and system vulnerabilities.

On the one hand, a user job demands the resource site to provide security assurance by issuing a security demand (SD). On the other hand, the site needs to reveal its trustworthiness, called its trust index (TI). These two parameters must satisfy a security-assurance condition: TI ≥ SD during the job mapping process. When determining its security demand, users usually care about some typical attributes. These attributes and their values are dynamically changing and depend heavily on the trust model, security policy, accumulated reputation, self-defense capability, attack history, and site vulnerability. Three challenges are outlined below to establish the trust among grid sites .

The first challenge is integration with existing systems and technologies. The resources sites in a grid are usually heterogeneous and autonomous. It is unrealistic to expect that a single type of security can be compatible with and adopted by every hosting environment. At the same time, existing security infrastructure on the sites cannot be replaced overnight. Thus, to be successful, grid security architecture needs to step up to the challenge of integrating with existing security architecture and models across platforms and hosting environments.

The second challenge is interoperability with different –hosting environments.Services are often invoked across multiple domains, and need to be able to interact with one another. The interoperation is demanded at the protocol, policy, and identity levels.

For all these levels, interoperation must be protected securely. The third challenge is to construct trust relationships among interacting hosting environments. Grid service requests can be handled by combining resources on multiple security domains. Trust relationships are required by these domains during the end-to-end traversals. A service needs to be open to friendly and interested entities so that they can submit requests and access securely.

Resource sharing among entities is one of the major goals of grid computing. A trust relationship must be established before the entities in the grid interoperate with one another. The entities have to choose other entities that can meet the requirements of trust to coordinate with. The entities that submit requests should believe the resource providers will try to process their requests and return the results with a specified QoS. To create the proper trust relationship between grid entities, two kinds of trust models are often used. One is the PKI-based model, which mainly

exploits the PKI to authenticate and authorize entities; we will discuss this in the next section. The other is the reputation-based model.

The grid aims to construct a large-scale network computing system by integrating distributed, heterogeneous, and autonomous resources. The security challenges faced by the grid are much greater than other computing systems. Before any effective sharing and cooperation occurs, a trust relationship has to be established among participants.

Otherwise, not only will participants be reluctant to share their resources and services, but also the grid may cause a lot of damage .

A Generalized Trust Model

At the bottom, we identify three major factors which influence the trustworthiness of a resource site. An inference module is required to aggregate these factors. Followings are some existing inference or aggregation methods. An intra-site fuzzy inference procedure is called to assess defense capability and direct reputation. Defense capability is decided by the firewall, intrusion detection system (IDS), intrusion response capability, and anti-virus capacity of the individual resource site. Direct reputation is decided based on the job success rate, site utilization, job turnaround time, and job slowdown ratio measured. Recommended trust is also known as secondary trust and is obtained indirectly over the grid network.

A general trust model for grid computing. Courtesy of Song, Hwang, and Kwok, 2005

Reputation-Based Trust Model

In a reputation-based model, jobs are sent to a resource site only when the site is trustworthy to meet users' demands. The site trustworthiness is usually calculated from the following information: the defense capability, direct reputation, and recommendation trust. The defense capability refers to the site's ability to protect itself from danger. It is assessed accoding to such factors as intrusion detection, firewall, response capabilities, anti-virus capacity, and so on. Direct reputation is based on experiences of prior jobs previously submitted to the site. The reputation is measured by many factors such as prior job execution success rate, cumulative site utilization, job turnaround time, job slowdown ratio, and so on. A positive experience associated with a site will improve its reputation. On the contrary, a negative experience with a site will decrease its reputation.

A Fuzzy-Trust Model

In this model , the job security demand (SD) is supplied by the user programs. The trust index (TI) of a resource site is aggregated through the fuzzy-logic inference process over all related parameters. Specifically, one can use a two-level fuzzy logic to estimate the aggregation of numerous trust parameters and security attributes into scalar quantities that are easy to use in the job scheduling and resource mapping process.

The TI is normalized as a single real number with 0 representing the condition with the highest risk at a site and 1 representing the condition which is totally risk-free or fully trusted. The fuzzy inference is accomplished through four steps: fuzzification, inference, aggregation, and defuzzification. The second salient feature of the trust model is that if a site's trust index cannot match the job security demand (i.e., SD > TI), the trust model could deduce detailed security features to guide the site security upgrade as a result of tuning the fuzzy system.

Authentication and Authorization Methods

The major authentication methods in the grid include passwords, PKI, and Kerberos. The password is the simplest method to identify users, but the most vulnerable one to use.The PKI is the most popular method supported by GSI. To implement PKI, we use a trusted third party, called the certificate authority (CA). Each user applies a unique pair of public and private keys. The public keys are issued by the CA by issuing a certificate, after recognizing a legitimate user. The private key is exclusive for each user to use, and is unknown to any other users. A digital certificate in IEEE X.509 format consists of the user name, user public key, CA name, and a secrete signature of the user. The following example illustrates the use of a PKI service in a grid environment.

Authorization for Access Control

The authorization is a process to exercise access control of shared resources. Decisions can be made either at the access point of service or at a centralized place. Typically, the resource is a host that provides processors and storage for services deployed on it. Based on a set predefined policies or rules, the resource may enforce access for local services. The central authority is a special entity which is capable of issuing and revoking polices of access rights granted to remote accesses. The authority can be classified into three categories: attribute authorities, policy authorities, and identity authorities. Attribute authorities issue attribute assertions; policy authorities issue authorization policies; identity authorities issue certificates. The authorization server makes the final authorization decision.

Three Authorization Models

The subject is the user and the resource refers to the machine side. The subject-push model is shown at the top diagram. The user conducts handshake with the authority first and then with the resource site in a sequence. The resource-pulling model puts the resource in the middle. The user checks the resource first. Then the resource contacts its authority to verify the request, and the authority authorizes at step 3. Finally the resource accepts or rejects the request from the subject at step 4. The authorization agent model puts the authority in the middle. The subject check with the authority at step 1 and the authority makes decisions on the access of the requested resources. The authorization process is complete at steps 3 and 4 in the reverse direction.

GRID SECURITY INFRASTRUCTURE (GSI)

Although the grid is increasingly deployed as a common approach to constructing dynamic, interdomain, distributed computing and data collaborations, –lack of security/trust between different services‖ is still an important challenge of the grid. The grid requires a security infrastructure with the following properties: easy to use; conforms with the VO's security needs while working well with site policies of each resource provider site; and provides appropriate authentication and encryption of all interactions.

The GSI is an important step toward satisfying these requirements. As a well-known security solution in the grid environment, GSI is a portion of the Globus Toolkit and provides fundamental security services needed to support grids, including supporting for message protection, authentication and delegation, and authorization. GSI enables secure authentication and communication over an open network, and permits mutual authentication across and among distributed sites with single sign-on capability. No centrally managed security system is required, and the grid maintains the integrity of its members' local policies. GSI supports both message-level security, which supports the WS-Security standard and the WS-SecureConversation specification to provide message protection for SOAP messages, and transport-level security, which means authentication via TLS with support for X.509 proxy certificates.

GSI Functional Layers

GT4 provides distinct WS and pre-WS authentication and authorization capabilities. Both build on the same base, namely the X.509 standard and entity certificates and proxy certificates, which are used to identify persistent entities such as users and servers and to support the temporary delegation of privileges to other entities, respectively. As shown , GSI may be thought of as being composed of four distinct functions: message protection, authentication, delegation, and authorization.

TLS (transport-level security) or WS-Security and WS-Secure Conversation (message-level) are used as message protection mechanisms in combination with SOAP. X.509 End Entity Certificates or Username and Password are used as authentication credentials.

X.509 Proxy Certificates and WS-Trust are used for delegation. An Authorization Framework allows for a variety of authorization schemes, including a –grid-mapfile‖

ACL, an ACL defined by a service, a custom authorization handler, and access to an authorization service via the SAML protocol. In addition, associated security tools provide for the storage of X.509 credentials (MyProxy and Delegation services), the mapping between GSI and other authentication mechanisms (e.g., KX509 and PKINIT for Kerberos, MyProxy for one-time passwords), and maintenance of information used for authorization (VOMS, GUMS, PERMIS).

The remainder of this section reviews both the GT implementations of each of these functions and the standards that are used in these implementations. The web services portions of GT4 use SOAP as their message protocol for communication. Message protection can be provided either by transport-level security, which transports SOAP messages over TLS, or by message-level security, which is signing and/or encrypting portions of the SOAP message using the WS-Security standard. Here we describe these two methods.

Transport-Level Security

Transport-level security entails SOAP messages conveyed over a network connection protected by TLS. TLS provides for both integrity protection and privacy (via encryption). Transport-level security is normally used in conjunction with X.509 credentials for authentication, but can also be used without such credentials to provide message protection without authentication, often referred to as –anonymous transport-level security.| In this mode of operation, authentication may be done by username and password in a SOAP message.

Message-Level Security

GSI also provides message-level security for message protection for SOAP messages by implementing the WS-Security standard and the WS-Secure Conversation specification.

The WS-Security standard from OASIS defines a framework for applying security to individual SOAP messages; WS-Secure Conversation is a proposed standard from IBM and Microsoft that allows for an initial exchange of messages to establish a security context which can then be used to protect subsequent messages in a manner that requires less computational overhead (i.e., it allows the trade-off of initial overhead for setting up the session for lower overhead for messages).

GSI conforms to this standard. GSI uses these mechanisms to provide security on a per-message basis, that is, to an individual message without any preexisting context between the sender and receiver (outside of sharing some set of trust roots). GSI, as described further in the subsequent section on authentication, allows for both X.509 public key credentials and the combination of username and password for authentication; however, differences still exist. With username/password, only the WS-Security standard can be used to allow for authentication; that is, a receiver can verify the identity of the communication initiator.

GSI allows three additional protection mechanisms. The first is integrity protection, by which a receiver can verify that messages were not altered in transit from the sender. The second is encryption, by which messages can be protected to provide confidentiality. The third is replay prevention, by which a receiver can verify that it has not received the same message previously. These protections are provided between WS-Security and WS-Secure Conversation. The former applies the keys associated with the sender and receiver's X.509 credentials. The X.509 credentials are used to establish a session key that is used to provide the message protection.

Authentication and Delegation

GSI has traditionally supported authentication and delegation through the use of X.509 certificates and public keys. As a new feature in GT4, GSI also supports authentication through plain usernames and passwords as a deployment option. We discuss both methods in this section. GSI uses X.509 certificates to identify persistent users and services.

As a central concept in GSI authentication, a certificate includes four primary pieces of information: (1) a subject name, which identifies the person or object that the certificate represents; (2) the public key belonging to the subject; (3) the identity of a CA that has signed the certificate to certify that the public key and the identity both belong to the subject; and (4) the digital signature of the named CA. X.509 provides each entity with a unique identifier (i.e., a distinguished name) and a method to assert that identifier to another party through the use of an asymmetric key pair bound to the identifier by the certificate.

The X.509 certificate used by GSI are conformant to the relevant standards and conventions. Grid deployments around the world have established their own CAs based on third-party software to issue the X.509 certificate for use with GSI and the Globus Toolkit. GSI also supports delegation and single sign-on through the use of standard X.509 proxy certificates.
Proxy certificates allow bearers of X.509 to delegate their privileges temporarily to another entity. For the purposes of authentication and authorization, GSI treats certificates and proxy certificates equivalently. Authentication with X.509 credentials can be accomplished either via TLS, in the case of transport-level security, or via signature as specified by WS-Security, in the case of message-level security.

Trust Delegation

To reduce or even avoid the number of times the user must enter his passphrase when several grids are used or have agents (local or remote) requesting services on behalf of a user, GSI provides a delegation capability and a delegation service that provides an interface to allow clients to delegate (and renew) X.509 proxy certificates to a service. The interface to this service is based on the WS-Trust specification. A proxy consists of a new certificate and a private key. The key pair that is used for the proxy, that is, the public key embedded in the certificate and the private key, may either be regenerated for each proxy or be obtained by other means. The new certificate contains the owner's identity, modified slightly to indicate that it is a proxy.
The new certificate is signed by the owner, rather than a CA .

A sequence of trust delegations in which new certificates are signed by the owners rather by the CA. The certificate also includes a time notation after which the proxy should no longer be accepted by others. Proxies have limited lifetimes. Because the proxy isn't valid for very long, it doesn't have to stay quite as secure as the owner's private key, and thus it is possible to store the proxy's private key in a local storage system without being encrypted, as long as the permissions on the file prevent anyone else from looking at them easily. Once a proxy is created

and stored, the user can use the proxy certificate and private key for mutual authentication without entering a password. When proxies are used, the mutual authentication process differs slightly. The remote party receives not only the proxy's certificate (signed by the owner), but also the owner's certificate. During mutual authentication, the owner's public key (obtained from her certificate) is used to validate the signature on the proxy certificate. The CA's public key is then used to validate the signature on the owner's certificate. This establishes a chain of trust from the CA to the last proxy through the successive owners of resources. The GSI uses WS-Security with textual usernames and passwords. This mechanism supports more rudimentary web service applications. When using usernames and passwords as opposed to X.509 credentials, the GSI provides authentication, but no advanced security features such as delegation, confidentiality, integrity, and replay prevention. However, one can use usernames and passwords with anonymous transport-level security such as unauthenticated TLS to ensure privacy.