UNIVERSITY OF OSLO
Department of Informatics

Moving into the Cloud

Master thesis

Christian Mikalsen

Moving into the Cloud

Christian Mikalsen

Abstract

Cloud computing is the notion of abstracting and outsourcing hardware or software resources over the Internet, often to a third party on a pay-as-you-go basis. This emerging concept is sometimes claimed to represent a completely new paradigm, with disruptive effects on our means of viewing and accessing computational resources.

In this thesis, we investigate the state-of-the-art of cloud computing with the aim of providing a clear understanding of the opportunities present in cloud computing, along with knowledge about the limitations and challenges in designing systems for cloud environments. We argue that this knowledge is essential for potential adopters, yet is not readily available due to the confusion currently surrounding cloud computing.

Our findings reveal that challenges associated with hosting systems in cloud environments include increased latency due to longer network distances, limited bandwidth since packets must cross the Internet to reach the cloud, as well as reduced portability because cloud environments are currently lacking standardization. Additionally, systems must be designed in a loosely coupled and fault-tolerant way to fully exploit the dynamic features of cloud computing, meaning that existing applications might require significant modification before being able to fully utilize a cloud environment. These challenges also restrict some systems from being suitable for running in the cloud.

Furthermore, we have implemented a prototype in Amazon EC2 to investigate the feasibility of moving an enterprise search service to a cloud environment. We base the feasibility of our approach on measurements of response time, bandwidth and scalability. We conclude that our cloud-based search service is feasible due to its opportunities for implementing dynamic scaling and reducing local infrastructure in a novel fashion.

Acknowledgements

I would like to stop shortly to thank everyone who have given their time, assistance and patience so generously throughout the last year.

First, I would like to thank my supervisors Pål Halvorsen and Carsten Griwodz for their patience, guidance and valuable input during the process of completing this thesis. Similarly, I would like to express my appreciation to Rolf Michelsen and Torgeir Hovden in FAST for taking time out of your busy schedules to help me, providing interesting discussions and useful insight.

I also want to express my gratitude to everyone in ProsIT and at the lab for creating a social and fun atmosphere throughout my studies. Finally, I want to thank everyone at Simula Research Laboratory for providing a truly hospitable and inspiring environment.

This thesis would not be possible without your help. Thank you.

Christian Mikalsen
Oslo, Norway
May 2009

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Distributed computing has been widely used for many years to run software concurrently on multiple machines, with the conceptually simple client/server architecture being most prevalent. The advent of paradigms such as grid computing allows a large number of machines to be inter-connected to form an illusion of a powerful supercomputer, providing extensive computational resources allowing complex tasks to be executed. While grid computing has been widely deployed in academic environments for more than two decades, is has generally been in little use outside of these sectors.

Cloud computing, expanding on many of the principles of grid computing, is often claimed to represent a completely new paradigm, with disruptive effects on our means of viewing and accessing computational resources. In essence, cloud computing is the notion of abstracting and outsourcing hardware or software resources over the Internet, often to a third party on a pay-as-you-go basis. One of the primary motivations for cloud computing is the opportunity to deploy and use software systems without initial IT investments, instead outsourcing infrastructure to third parties and paying only variable costs associated with the actual resources consumed.

Along with initial adoption and early prototypes of cloud computing came a significant amount of market attention, resulting in something of a hype. Dedicated cloud computing products and services quickly emerged, and companies proved eager to attach cloud computing bumper stickers onto existing products to take part in the massive market attention. This has led cloud computing to become an ambiguous concept.

Cloud computing has potential to allow developers to cost-effectively implement dynamic scalability, scaling the hardware resources supporting a system with demand, as well as providing high availability through hosting in geographically dispersed data centers. A notion of computational resources as a utility, with on-demand and pay-as-you-go access, is provided through *Infrastructure as a Service* (IaaS) and *Platform as a Service* (PaaS) services.

1.2 Problem statement

In this thesis, we will investigate the state-of-the-art of cloud computing with the aim of providing a clear understanding of the opportunities present in cloud computing, along with knowledge about the limitations and challenges in designing systems for cloud environments. We argue that this knowledge is essential for potentials adopters, yet is not readily available due to the confusion currently surrounding cloud computing.

Like any computing paradigm, cloud computing is not suitable in all scenarios. To be able to successfully utilize emerging cloud computing technology, potential users need to know what classes of systems might benefit from being based in a cloud environment, as well as those that do not. To answer this question, we must consider key characteristics of enterprise systems relating to the characteristics of cloud environments. These characteristics include static scalability, tight coupling of components, latency requirements for interactive applications and large amounts of data for data-centric systems.

We will also implement an architectural model of the FAST ESP, an enterprise search platform widely deployed in organizations across the world, in Amazon EC2 as a case study to investigate if a cloud-based approach to an enterprise search service is feasible, given the challenges of increased latency, limited bandwidth and scalability. The feasibility of our approach will be evaluated based on experiments and measurements.

1.3 Main contributions

We have taken an in-depth look at state-of-the-art cloud computing, and placed it into context by relating it to the established paradigm of grid computing. Using an ontology, we have decomposed high level concepts into five specific layers of cloud computing, and discussed the functionality provided in each layer.

We have found that cloud environments provide an opportunity to design dynamically scalable services utilizing temporal load variance, as well as enabling cost-effective availability through geographically dispersed data centers. Still, this flexibility comes at some cost, especially in terms of non-functional challenges. Challenges include increased latency due to longer network distances, limited bandwidth since packets must cross the Internet to reach the cloud, as well as reduced portability because cloud environments are currently lacking standardization. Additionally, systems must be designed in a loosely coupled and fault-tolerant way to fully exploit the dynamic features of cloud computing, meaning that existing applications might require significant modification before being able to fully utilize a cloud environment.

Our investigation has identified important characteristics of enterprise systems related to cloud computing, including data centricity, tight coupling and static scalability. These characteristics affect how systems are moved to cloud environments, and restrict the classes of systems suitable for the cloud.

Systems with absolute characteristics that correspond to cloud challenges, such as strict latency requirements or large data amounts, are least suited for cloud environments. Examples of such applications include video streaming software requiring predictable CPU scheduling and high bandwidth, health management software dealing with sensitive information and business-critical software over which an organization requires full control. Still, we argue that there is no single metric for cloud suitability, as the importance of individual opportunities and challenges differ between systems, and must be balanced in each case.

Finally, we have implemented a cloud search service in Amazon EC2, based on an architectural model of FAST ESP [1]. We have used this as case study to investigate if a cloud-based version this service is feasible considering response time, bandwidth and scalability. We conclude that our cloud-based approach is feasible, though performance in terms of response time and bandwidth is inferior compared to a direct disk-based implementation. Still, we argue that the cloud-based implementation represents a novel approach due to the opportunities for implementing dynamic scalability along with a reduced need for local infrastructure operations.

1.4 Outline

This thesis is organized as follows:

In Chapter 2, we take a look at virtualization, serving as an enabling technology for cloud computing. We also give an overview of what the cloud computing paradigm represents, and how the concept relates to grid computing.

Chapter 3 gives an overview of key characteristics of cloud computing environments, along with a survey of some state-of-the-art IaaS and PaaS services.

In Chapter 4, we provide an overview of different approaches to moving systems to the cloud, along with an overview of some of the challenges in doing so. We also discuss characteristic of enterprise systems, including a case study of FAST ESP.

Chapter 5 provides a case study of a cloud-based search service based on an architectural model of FAST ESP, including an investigation of its feasibility based on measurements of response time, bandwidth and scalability.

Finally, Chapter 6 concludes the thesis with a summary and conclusion.

Chapter 2

Background

2.1 Introduction

This chapter introduces the concept of virtualization, serving as an enabling technology for cloud computing. We further describe grid computing, discussing motivation, concepts, architecture and existing implementations. Finally, we introduce cloud computing, with the goal of placing it into the context of virtualization and grid computing.

2.2 Distributed computing

With the advent of Information Technology (IT) in academic and corporate environments in the 1960s, service bureaus and timesharing facilities provided and controlled access to computational resources [2]. Users prepared punch cards in advance and sent them to a central facility for processing, or accessed central computers using light terminals over telephone lines. Computation was provided through mainframe computers, typically operating standalone without communication with other machines.

With the arrival of the personal computer (PC) in the 1980s, users were able to install software locally on individual computers. However, the limited computational capacity of the PC meant that not all computations could be feasibly executed locally. This lead to the prevalence of client-server architecture, in which clients request a server to perform some work and return the result. A major example of a client-server protocols include Hypertext Transfer Protocol [3] (HTTP), in which browsers running locally on clients request documents and information from remote servers.

When the number of interconnected machines increase, the client-server paradigm becomes an impractical paradigm to describe distributed systems (even though the client-server architecture is generally used internally to enable other paradigms). Complex tasks, like scientific experiments or weather forecasting, require large amounts of computational resources. Instead of relying on large mainframes or monolithic supercomputers to provide the required resources, researchers discovered that clusters of commodity PCs could be interconnected to form virtual super-computers. This paradigm is known as *cluster computing*, and is often realized with a large number of Commercial off-the-shelf (COTS) hardware interconnected using a dedicated network. One adopter of cluster computing is Google, proving that highly scalable systems can be cost-effectively created using clusters of commodity hardware.

Grid and cloud computing represent an evolution of this paradigm, based on the same principles of connecting a large number of commodity machines to provide cost-effective and scalable computing. In the following sections, we discuss grid and cloud computing closer, describing how they are realized. To better understand key principles behind cloud computing, we open with a discussion of virtualization.

2.3 Virtualization

Virtualization has received much attention during recent years, but has been subject to research since the mid-1960's. Initial research on virtualization started with IBM and MIT collaborating on the M44/44X experimental system, simulating virtual machines with software and hardware [4].

The term virtualization is today used to describe a wide range of technologies and approaches. A common property of the different aspects of virtualization is to abstract one or more resources away from a physical implementation, but both motivation and approach vary widely.

2.3.1 Motivation

Virtualization allows several advantages in modern software systems, such as abstraction, isolation, consolidation and cost efficiency. We will discuss these advantages closer in this section, outlining the motivation for using virtualization technology.

Abstraction

Modern computer systems are very complex, and abstraction has proved necessary for continued evolution of systems. Without different levels of abstraction, software developers and hardware makers would have to relate to vast amounts of implementation details and dependencies, making development time-consuming and prone to errors. Thanks to abstraction, we are able to develop software and hardware in modular fashion, communicating with the other components of the system using well-defined interfaces.

In addition to easing development, modular design with well-defined interfaces make it possible to run software and hardware in across different environments, for example on hardware made by different vendors. Modern computer systems have three logical interfaces separating distinct layers of the system, that are used to provide abstraction [5]. Figure 2.1 illustrates how virtualization can fill gaps between layers of software and hardware.

The lowest level is the *Instruction Set Architecture* (ISA), which represents the boundary between hardware and software, and defines the set of instructions a processor can execute. A widely-used ISA is the Intel IA-32 (x86) instruction set [6], implemented by Intel and AMD for their current processors.

The *Application Binary Interface* (ABI) allows software to access hardware and software available by user-mode instructions or through a system call interface. Instead of executing privileged instructions directly, applications invoke system calls which enables the operating system to mediate privileged operations, performing access control and security checks. The Intel Itanium Binary Compatibility Standard [7] is an ABI that allows software compiled on one operating system to run unmodified on different operating systems, as long as the underlying hardware is compatible.

On the source code level, software developers use *Application Programming Interfaces* (APIs) to interact with other software through high level language library calls. To simplify development, systems calls are typically also accessed through such high level language libraries. Using APIs, applications can be recompiled to run on other systems that support the same API libraries by recompilation. An example of a standardized API for system level operations is the widely used POSIX standard, designed to ease design of portable software.

Despite these advantages, this abstraction also has some limitations. Subsystems designed to work with one interface not necessarily work with similar (but not equivalent) interfaces, and software compiled for one ISA is tied to hardware supporting that particular ISA. Virtualization helps solve these problems by filling the gaps between incompatible interfaces, providing a level of abstraction allowing us to effectively combine and compose systems that are not by themselves compatible.

Isolation

Creating and supporting services and applications for a number of users can be challenging, especially when users require heterogeneous environments and access to a wide range of different services.

Performance isolation is a property of systems in which resource consumption in sub-systems is isolated, and is guaranteed not to affect the resources promised to or available to other sub-systems. This property is especially important in systems in which certain critical services are required to operate at a specific capacity, and for which a share of resources must be guaranteed to be available at all times. An example of such a scenario is an order processing system which must be able to process all incoming orders in a timely manner, independent of other services running on the same hardware.

The traditional approach to isolation is using conventional multi-user operating systems, such as Windows or Linux, where users are given individual accounts with appropriate permissions. However, system administration in such systems can quickly become a time-consuming task, especially when different users require applications with vastly different requirements. Also important is the fact that such systems don't adequately support performance isolation, since scheduling priority, memory usage, network traffic and disk accesses of one user can affect others [8]. This might be acceptable for systems that are over-provisioned or limited to a closed group (e.g., within a department), but not for critical tasks or in larger environments.

Attempts have been made to improve the performance isolation in conventional operating systems [9], but the issue remains challenging because of the complexity of interacting resources and components in modern systems. For example, cache miss and page replacement overhead may be hard to measure and trace back to individual user accounts or processes, but can affect performance isolation negatively.

Virtualization can help resolve this issue, since resource usage can be monitored and controlled at an abstraction layer above the operating system. Virtualization also provides us with isolation of failures, with which failures in one virtual machine can be kept contained in the same virtual instance, enabling other instances to continue unaffected. This property is valuable in creating fault tolerant systems, since it also offers protection against kernel crashes, which otherwise can render complete systems unusable.

Consolidation and cost efficiency

As computer systems have evolved in recent years, many organizations have invested in large data centers with a significant number of physical machines. Traditionally, physical hardware has been dedicated to different software, for example resulting in separate servers for database, file storage and e-mail. The separation was primarily done due to different hardware requirement and utilization properties for the systems, causing execution of one type of system to negatively affect other systems; e.g. a database server using large amounts of memory when the database is heavily accessed, resulting in possible degradation for an e-mail service on the same machine.

Additionally, since all systems were required to scale under heavy load, over provisioning is required for all services. This results in organizations investing in large quantities of over-provisioned hardware, which for most of the time is under-utilized.

Enterprise organizations have looked to virtualization for a solution for these reasons for several years, since it enables them to consolidate heterogeneous services on far fewer physical machines, and improve over-all utilization. While this reduces hardware cost, one of the major advantages is the savings in energy consumption. Fewer machines running at higher utilization rates consume less energy than a larger number of lower-utilized machines, both in terms of direct consumption by the hardware and in reduced need for energy-intensive air cooling in the data centers.

Virtualization also helps improve system utilization with systems that have different peak times. For example, large batch processing jobs can utilize the hardware resources outside of normal office hours, while still allowing an email service with a peak during daytime to co-exist on the same physical infrastructure.

2.3.2 Concepts

Virtualization provides a virtualized environment between a platform and an operating system or process, and can be applied to complete systems or selected components. Virtualizating a system or component maps its physical interfaces and resources onto an underlying, possibly different, real system [5].

While virtualization serves as an abstraction layer, the goal is typically not to simplify or hide implementation details, but instead fill the gap between different logical or physical interfaces, enabling systems to circumvent compatibility problems or avoid hardware constraints.

Figure 2.1 shows the architecture of virtualization. Virtualized systems or components are said to run inside a *virtual machine* (VM), which is an isolated environment provided by some virtualization infrastructure. A wide array of virtual machine implementation exist, all of which make different tradeoffs depending on the properties the virtual machine should provide.

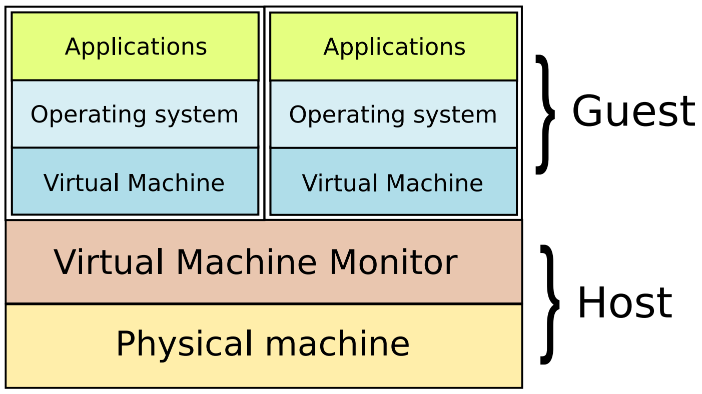


Figure 2.1: Architecture of virtualization

At the core of a virtualization infrastructure is a component called the *Virtual Machine Monitor* (VMM), commonly referred to as a *hypervisor*. The VMM sits at the layer above a virtual machine, and is responsible for providing the environment that make up the virtual machine.

A VMM is typically able host multiple virtual machines, and can be implemented in either hardware or software, or a combination of the two. The VMM runs on a *host*, and allow virtual machines to execute as *guests*. As with processes on a regular system, we make a distinction between a virtual machine image, made up by executable code, and a running *virtual machine instance*, made up by executable code with context and state.

2.3.3 Taxonomy

Various approaches to virtualization have been researched through the years, some gaining more popularity and widespread use than others. In [5], the authors attempt to classify the architecture of different virtualization approaches, and to present a virtual machine taxonomy, shown in Figure 2.2. The approaches taken by the different VMs will be described in the following sections.

The primary distinction between VMs is based on the target for the virtualization. Process VMs serve as virtual machines for individual application processes, and are de-

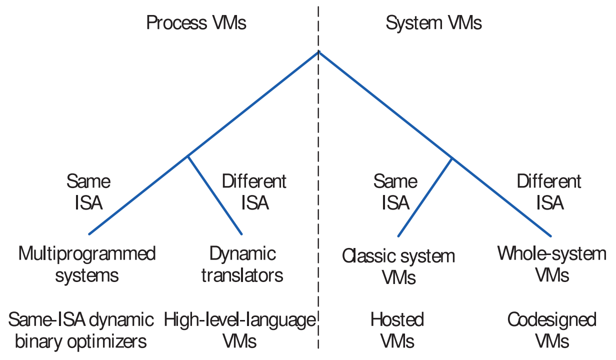


Figure 2.2: Taxonomy of Virtual Machines

scribed in Section 2.3.4. System VMs provide virtualized environments for complete operating systems to run, and are described in Section 2.3.5.

Figure 2.3(a) illustrates how a process VM provides an environment for executing processes encompassing hardware, the OS and the virtualization software. Similarly, Figure 2.3(b) shows how a system VM provides an OS as well as application processes with a virtualized hardware environment. We further distinguish VMs by their support for executing software compiled for different instruction sets, or if the VM is designed to only software compiled for the same ISA.

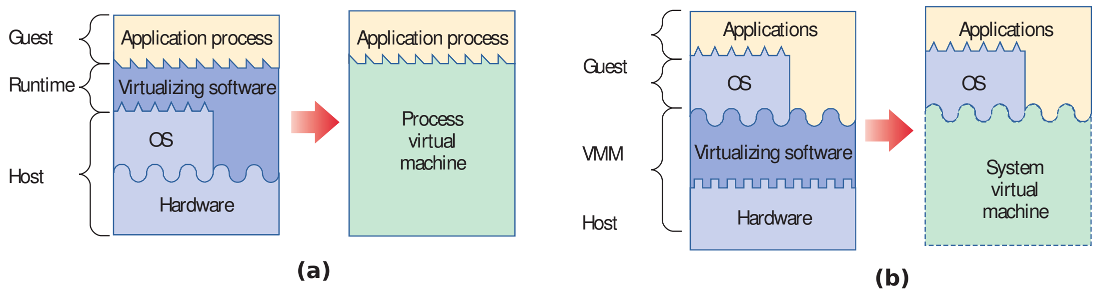


Figure 2.3: (a) Process VM, which virtualizes OS and hardware to processes. (b) System VM, which virtualizes hardware to both the operating system and processes.

2.3.4 Process virtualization

Process VMs provide a virtual machine environment for user applications, and is typically itself hosted inside an existing operating system. Most current operating systems support multiple concurrent processes through multiprogramming, which provides each process with an illusion of having a complete machine to itself. The operating system provides each process with its own address space and registers, and transparently time-shares hardware resources to realize this illusion. In effect, this implies that current operating systems already provide a kind of process VM for each of the concurrently executing applications [5].

Emulation and interpretation

If we want to execute binaries compiled for an ISA differing from a physical host, we must provide some transformation of the binary program code. An intuitive way to perform this transformation by interpretation and emulation, where interpreter software reads, processes and emulates individual guest instructions. This interpretive approach is intuitive and straightforward to implement, but results in slow execution since single instructions from the guest instruction stream may take tens or even hundreds of host instructions for interpretation and emulation [5]. This approach is similar to dynamic translation in system virtualization, discussed in Section 2.3.5.

The performance of interpreted approaches can be improved with dynamic binary translation, in which the host dynamically converts guest instructions to the host instruction set in blocks rather than individual instructions. Commonly used instruction transformations can be cached and reused later, thus reducing the overhead of interpretation when instructions are repeated. This approach is commonly referred to as *Just-In-Time compilation*.

High-level-language VMs

A major motivation for using process VMs is platform portability, namely the ability to run software unmodified on a wide array of different systems. A disadvantage to achieving this goal through emulation is that it requires considerable programming effort, since the emulation must be implemented on a case by case basis for each platform combination.

A better approach is to implement process VMs as part of an environment where software is created with a high-level language (HLL). The high-level language source code is converted by a front-end compiler into intermediate code, often called byte code. This intermediate code does not correspond to any physical platform, but is designed to be flexible, easy to port and to match the features of one or more HLLs.

When executed, the intermediate code is processed by another code generator that outputs binary code matching the instruction set of the host system on which the software is about to be run. Code generation is either done entirely for the application upon startup, or on an on-demand basis.

The primary advantage of HLL VMs is that software is easily ported, once the initial VM and libraries are ported to different host platforms. Examples of widely used HLL VMs are Sun Microsystem's Java [10], and Microsoft Common Intermediate Language [11] used in the Microsoft .NET framework [12].

2.3.5 System virtualization

System VMs provide an environment in which complete multi-user operating systems and their processes can run. By using system VMs, it is possible to run multiple, possibly different, operating systems on a single physical machine.

System VMs represent the original approach to virtualization, and emerged during the early 1970s [13]. Mainframes were common at the time, and were massive and expensive. The mainframes were typically shared between a large amount of users, and virtualization was used to allow different groups of users to run different operating systems. In the years that followed, hardware became less expensive and usage migrated to desktop computers of individual users. Accordingly, the interest in system VMs faded and research activity minimized. During recent years, system virtualization has enjoyed renewed popularity, as the large mainframe systems of the 1970s have been replaced with clusters of servers shared between many users of the systems.

One of the primary advantages of system VMs is that they may provide strong isolation between multiple systems running concurrently on the same hardware. This isolation means that the virtual machines run in a sandboxed environment, in which communication between different instances is not possible. This implies that compromised security in one of the guest instances does not affect any other instances.

With system virtualization, the primary task of the VMM is platform replication, and dividing the physical resources among the different guest instances. The virtual guest instances have no direct access to the physical hardware, and the guest instances themselves are (transparently) controlled by the VMM.

By nature of virtualization, the privileged state of a guest differs from the privileged state of the host. Since the VMM's task is to provide an execution environment that fits the expectation of the guest, the guest's state needs to be maintained by the VMM. To achieve this goal, system VMM's typically hold shadow structures of privileged data, such as page tables. For on-CPU structures (such as the page table pointer register) this can be trivially handled by keeping an image of the registers and performing operations against the image when the VMM intercepts traps from the guest.

However, other system structures may not be as easily kept synchronized. Off-CPU data, such as page tables, reside in guest memory, and access to it may not coincide with trapping instructions. For example, guest page table entries are privileged state due to their encoding of mapping and permissions, but this privileged state can be modified by any instruction in the guest instruction stream without causing a trap.

To maintain consistency of the shadow structures, VMMs typically use hardware page protection mechanisms to trap accesses to these in-memory structures. For example, guest page table entries may be write-protected, a concept known as *tracing*. System VMMs then intercept the trap, decode the instruction and emulate its effect against the primary structure while propagating the changes to the shadow structure.

Classical virtualization

In 1974, Popek and Goldberg defined three formal properties for a system to be considered a proper VMM [14]:

1. *The equivalence property* (fidelity). Execution of software on the VMM should be identical to its execution on hardware, except for timing effects.
2. *The efficiency property* (performance). Most of the guest instructions should be executed directly by the hardware without intervention from the VMM.
3. *The resource control property* (safety). The VMM should manage all hardware resources.

At the time of Popek and Goldberg's paper, a trap-and-emulate architecture of virtualization was so prevalent that it was considered the only practical style of virtualization [15]. With this approach to virtualization, which we will refer to as *classic virtualization*, the guest OS runs at a reduced privilege level, and all unprivileged instructions execute directly on the CPU without VMM intervention. Privileged operations are made to trap when executed in the guest's unprivileged context, and are caused by the guest attempting to execute a privileged instruction or attempting to access structures protected by the VMM.

When a trap is intercepted by the VMM, the VMM decodes and inspects the guest's instruction and emulates the instruction's behavior against the guest's virtual state, before control is returned to the guest. The trap-and-emulate behavior has been extensively discussed in the literature, and can be easily demonstrated to fulfill the Popek and Golberg criteria [15].

Binary translation

The classical trap-and-emulate approach to virtualization has a disadvantage in that it requires hardware that is classically virtualizable, which is not the case for all current architectures. For example, the common IA-32/x86 ISA is not classically virtualizable because guests can observe their de-privileged status by inspecting certain registers, and not all privileged operations generate traps when executed at a de-privileged level [15].

An intuitive solution to providing virtualization for systems that are not classically virtualizable is to use an interpreter, emulating all of the guest's instructions on a virtual state. This interpretive approach satisfies Popek and Goldberg's resource control and equivalence properties, but violates the efficiency property since performance would be significantly slower compared to direct execution on hardware.

To overcome the challenge of virtualizing systems that are not classically virtualizable while avoiding the performance penalties of interpretation, the concept of *binary translation* (BT) is used by some VMMs, such as VMware Workstation [15]. Binary translation combines the semantic precision provided by interpretation with high performance, making it possible to virtualize architectures such as the x86 while satisfying the formal properties of Popek and Goldberg.

BT works on principles similar to the Just-In-Time (JIT) compilers used in dynamic programming environments like Java. The host reads the memory at the location indicated by the guest's Program Counter (PC), and classifies the data in subsequent bytes as prefixes, opcodes or operands. This classification results in a set of Intermediate Representation (IR) objects, each corresponding to one guest instruction. The IR objects are then grouped into a Translation Unit (TU), which represents a block of instructions ready for translation. Translation units contain the instructions in the code flow until a terminating instruction is encountered or until a maximum number of instructions is reached.

The TU is then translated into a subset of the full instruction set, containing only non-privileged operations. Privileged instructions in the original instruction stream are converted to non-privileged instructions operating on virtual state or interacting with the VMM. Since privileged instructions are relatively rare, even in OS kernels, the performance of binary translation is mostly determined by the translation of regular instructions [15]. Due to the fact that most instructions are non-privileged, most code can be translated identically to the source instructions, or with only minor changes.

Hardware-assisted virtualization

The IBM System/370 [16] introduced a concept of *interpretive execution*, a hardware execution mode designed for running guest operating systems, in which the VMM encoded the state of the guest into a hardware-specific format followed by execution of a special instruction to start interpretive execution. In this execution mode, many of the guest operations which would normally trap in a deprivileged environment were able to directly access hardware shadow structures, reducing the frequency of relatively expensive trap operations.

Recently, hardware-assisted virtualization has gotten some renewed attention, and both AMD and Intel have performed architectural changes to allow classic virtualization of the x86 architecture. The hardware provides a number of new primitives to enable hardware-assisted virtualization. The main primitive is a new in-memory data structure, called the *Virtual Machine Control Block* (VMCB), which combines the host state with a subset of a guest's virtual CPU. A new CPU mode allows direct execution of guest code, including privileged instructions, and is activated when the CPU is set into guest mode by the VMM executing a *vmrun* instruction.

When entering guest mode, the CPU loads guest state from the VMCB, and resumes execution in guest mode. The CPU continues execution in guest mode until a condition defined in the VMCB is reached. An exit operation is then performed, which saves the guest state to the VMCB, loads a VMM-supplied state into the CPU and resumes execution in host mode.

Some privileged operations (such as page faults) result in exit operations, and hand control over to the VMM which emulates the operations and maintain shadow structures. The performance of this hardware-assisted virtualization therefore depends on the frequency of exists, since each exit operation triggers a switch between guest and host mode, in which state much be changed.

Early experiments show that while hardware-assisted virtualization achieves better performance than pure software virtualization in some cases, software virtualization with binary translation still has similar or superior performance in most cases [15]. Software virtualization also has the advantage of being quite dynamic, whereas hardware-assisted virtualization is more static in nature.

Paravirtualization

During the first virtualization boom, both hardware and guest operating systems were typically produced by a single vendor, such as IBM [15]. This allowed flexibility in the composition to be exploited in order to improve performance and efficiency of the virtualization. One approach exploited the relationship between the VMM and guest OS by modifying the OS to provide higher-level information to the VMM, and to provide additional features such as VM-to-VM communication. This modification relaxes the Popek and Goldberg equivalence property in order to improve the performance of the system. This idea of taking advantage of the fact that operating systems can be made aware that they are running in an a virtualized state has recently gained momentum under the name *paravirtualization* [17].

Using paravirtualization, the Xen VMM [8] aims to provide a solution for running hundreds of virtual machine instances simultaneous on modern server hardware. The designers recognize that full virtualization has the benefit of being able to run unmodified operating systems making them easy to virtualize, but argue that it also has a number of drawbacks. In particular, it might be desirable to combine access to physical and virtual hardware devices, such as Network Interface Cards, to better support time-sensitive tasks such as handling Transport Control Protocol [18] (TCP) timeouts.

Xen intends to provide an x86 abstraction with support for running full multi-user operating systems with unmodified application binaries. The authors argue that paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as the x86, and that completely hiding the effects of resource virtualization from guest operating systems risk both correctness and performance [8].

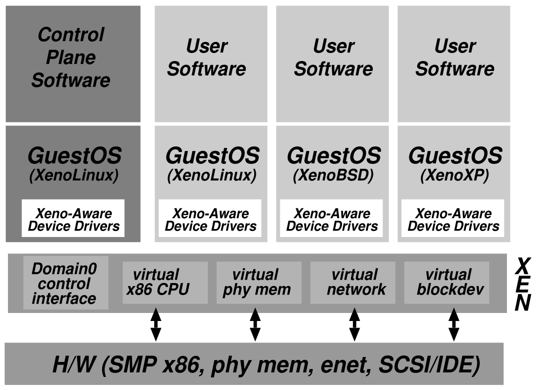


Figure 2.4: Structure of a machine running a Xen hypervisor.

Figure 2.4 shows the structure of a machine running the Xen hypervisor with a number of different operating systems. Operating systems are modified to run on top of Xen, and the kernel of guest operating systems are modified to communicate with Xen's virtual machine interface instead of physical hardware. Xen is designed to be operating system agnostic, and a number of operating systems have been ported to run on top of Xen. Examples of operating systems supporting Xen are XenoLinux and XenoXP as shown in Figure 2.4, which are custom versions of Linux and Windows XP with kernel changes to allow paravirtualization with Xen.

Instead of directly doing memory management with the physical memory, Xen restricts updates in a number of ways to allow it to keep its shadow structures updated. Xen also ensures that operating systems only access the memory they have permission to access, and prevents them from touching reserved memory.

Xen virtualizes the CPU, and traps and inspects privileged instructions before they are allowed to execute. Operating systems that normally run in privilege mode 0 (ring 0) on the x86 architecture are instead run in ring 1, with only the Xen hypervisor running at ring 0.

To improve performance, Xen optimizes the two commonly most frequent exceptions; system calls and page faults. System calls are implemented by modifying the guest operating system to register a fast system call handler, which avoids the need to indirect execution via Xen and triggering an expensive transition. Page faults can not be implemented this way, because only code running in ring 0 can read the faulting address from the control register. Page faults are therefore always processed by Xen.

Device I/O is handled by a set of device abstractions exposed by Xen. I/O data is transferred using shared-memory, asynchronous buffer descriptor rings, which provide high-performance buffer transfer vertically through the system while still allowing Xen to efficiently perform validation checks.

Xen also provides a less costly alternative to hardware interrupts by providing a lightweight event-delivery mechanism that is used for sending asynchronous notifications to guest instances. Guest operating systems register an event handler, used by Xen to notify the guest of new data. This event handler can also be "held off" using a mechanism similar to disabling interrupts on hardware.

Guest operating systems use *hypercalls* to transfer execution into the hypervisor, which is a mechanisms similar to system calls. An example use for hypercalls is to request a set of page-table updates, which Xen validates and applies before returning control to the guest instance.

Codesigned virtualization

While traditional VMs are implemented to achieve functionality and portability on hardware with a standard ISA, codesigned VMs are in contrast developed to take advantage of new, proprietary ISAs to improve performance or power efficiency [5].

A codesigned VM appears to be part of the hardware, and its purpose is to emulate a source ISA, which serves as the guest's ISA. The VM is typically loaded from ROM during boot, and resides in a concealed region of memory inaccessible from conventional software. Internally, the codesigned VM contains a binary translator that converts guest instructions (in a source ISA) into a target ISA for the hardware, and caches them in concealed memory. In effect, codesigned VMs decouple the source ISA from the underlying hardware, meaning that new CPU designs require only changes to the binary translator.

Hosted virtualization

The classical approach was to place the VMM directly on top of the hardware, executing in the most privileged mode. An alternative approach is to place the VMM inside an existing operating system (i.e., hosted), as is done in products like the VMWare Server [19]. Hosted VMs are installed just like normal software, and rely on the host operating system to provide device drivers and other services rather than providing them directly in the VMM.

An advantage of hosted virtualization is that users can install VMs like regular application software. Furthermore, the virtualization software can take advantage of existing driver software and services already present in the host OS. A disadvantage is the extra overhead and lack of control induced by running the virtualization software as a regular user process.

Whole-system virtualization

In conventional system VMs, all guest operating systems and applications use the same underlying instruction set. However, it may be advantageous to achieve compatibility by running operating systems made for different instruction sets.

In such cases, the VM must emulate both application and operating system code, and perform translation between the instruction sets. An example of such whole-system virtualization software is Virtual PC [20], which allows the Windows operating system to run on top of the previously PowerPC-based Macintosh platform.

2.4 Grid computing

Grid computing aims to create an illusion of a simple, yet large and powerful computer out of a large number of connected heterogeneous systems, sharing various resources [21]. Grid computing has gained most adoption in environments with large and complex tasks, such as performing complex mathematical simulations in research projects or analyzing large amounts of data to find trends in a company's sales figures. Organizing a set of machines in a grid can provide a dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [22].

Figure 2.5 shows the architecture of a computational grid. Software running on the machines in an organization communicate with the grid, submitting jobs to one or more coordinators. Jobs are scheduled and distributed internally for execution on one or more nodes in the grid, and the results are sent back to the original machine.

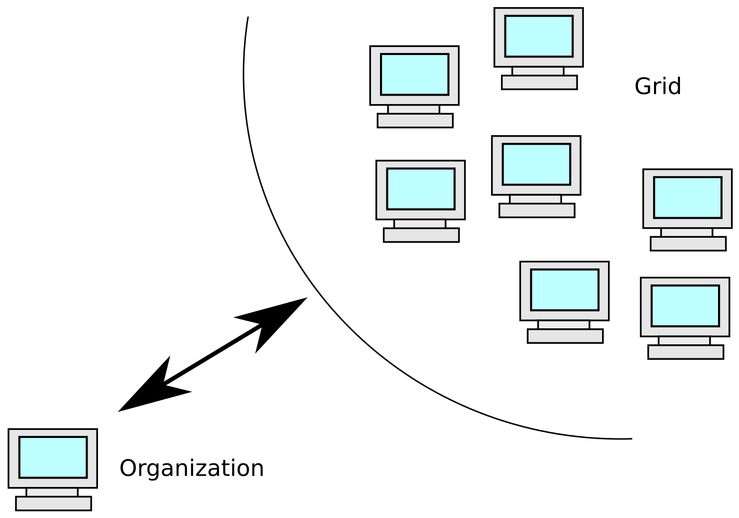


Figure 2.5: The logical architecture of a grid

Many of these characteristics also apply to *cluster computing*, which is similar in concept. With cluster computing, jobs are distributed among a set of highly interconnected computers, typically located at a single site. The grids of grid computing operate in a similar fashion, but tend to be looser in coupling and more geographically distributed. The individual nodes in grids are also more heterogeneous in nature, whereas clusters are sets of similar hardware.

2.4.1 Motivation

Exploiting under-utilized resources

If a machine on which an application is run is heavily loaded, a grid-enabled application can take advantage of an idle computer elsewhere in the grid to run computation-intensive tasks and report the results back to the source machine. For this to be possible, the application must be remotely executable without undue overhead, and the software must have required resources (such as special hardware and software) available on the machines the job is delegated to. Time-consuming batch jobs that spend time processing a set of input data to produce output are primary candidates to this type of distribution.

Most organizations have a large amount of under-utilized computing resources, and many desktop machines are fully utilized less than 5% of the time [21]. Due to over-provisioning, even server machines are heavily under-utilized most of the time. If available resources, such as CPU and storage capacity, on these idle machines are leveraged to perform other jobs in a grid fashion, organizations are able to better utilize their infrastructure investments. This can be done by dedicating a certain share of the resources to grid jobs, or by scheduling jobs when the machine is idle, a practice known as *cycle scavenging*.

Another property of grids is the ability to balance resource utilization among nodes. If one or more machines gets overloaded due to large amounts of tasks, grids can allow jobs to be distributed to other nodes in the grid with free capacity. Grids can provide a consistent way to balance loads on a large set of machines, applying to CPU, storage and other resources [21]. If supported by the grid infrastructure, jobs can also be *migrated* between nodes, transparently moving executing applications. If a priority system is in place, the grid can automatically also pause or reduce the resources dedicated to lower priority jobs to improve the capacity for higher priority jobs.

Parallel CPU capacity

An attractive feature of grids is the opportunity to provide massive parallel CPU capacity, especially useful in fields where computational power is driving innovation, such as in bioinformatics, financial modeling, oil exploration and motion picture animation.

To take advantage of this parallel CPU capacity, applications must be designed with highly parallelizable algorithms or the ability to be partitioned into many different sub-tasks that can execute independently. Ideally, perfectly parallel applications will be able to complete ten times faster with access to ten times the resources; a property known as *linear scalability*. Unfortunately, most typical systems are not this parallelizable because of dependencies between sub-tasks, or because of contention to shared resources such as files and databases. This implies that systems must be specifically designed to execute in grids, commonly using some sort of grid middleware.

Access to special resources

In addition to sharing access to CPU and storage resources normally present on all nodes in the grid, grids can also be used to provide access to other resources such as special software and devices.

For example, the license for a particular expensive piece of software may only allow the software to be installed on one physical machine. A grid could be set up to provide the functionality of the special software to other machines in the grid by accepting jobs, executing them with the software, and returning the result to the source machine. Another machine may have special hardware installed, such as a high-performance printer or an electron microscope. Using a grid, other users could access this resource using a reservation system together with remote operation of the device from other machines.

Reliability

The conventional approach to high availability is to use expensive redundant components, such as redundant and hot-swappable CPUs and power supplies for providing hardware failover. While this approach works for many failures, it is expensive since organizations must invest in redundant hardware. With a grid system, reliability can be achieved by migrating jobs from systems experiences failures to other unaffected machines in the grid. Additionally, since grids can be geographically distributed, larger-scale failures such as power outages or fire in individual sites can be limited to a subset of the total grid resources, provided that the physical infrastructure is designed with this in mind.

Grid management and monitoring software can automatically resubmit jobs to other machines in the grid if failures are detected. Real-time and critical jobs can be run in parallel simultaneously in several machines in the grid, and be checked for consistency to detect failures, data corruption or tampering.

Cost efficiency

Large-scale grid computing can be realized by combining a larger number of cost-effective commodity hardware to form a grid, instead of investing in costly and proprietary mainframe computers, which was the only available solution a few years ago.

Management

Virtualizing the resources in a grid can allow ease management of the infrastructure [21]. Capacity and utilization can be monitored and visualized by monitoring software, aiding in better planning of infrastructure investments. An intelligent grid system can also reroute jobs while maintenance is performed, and allow additional capacity to be automatically utilized as it is added to the grid.

2.4.2 Concepts

Management software

The management software keeps track of available resources in the grid, and which machines are part of the grid. This information is primarily used to device where jobs should be assigned. Secondly, the management software performs measurements of the capacities at the grid nodes and and their utilization rates. This measurement information is used to place jobs in the grid, and determine the health of the grid, alerting service personell to outages and congestion. This information can also be used to account and bill for the usage of the grid resources. Advanced grid management software can also automatically perform recovery actions autonomously, referred to as autonomic computing.

Donor software

Each of the nodes in the grid must install some software component that manages the grid's use of its' resources. The donor software may also perform authentication of the donor machine's identity and the grid itself, if required.

The donor software accepts jobs from the grid, and prepare it for execution on the machine. When the job is completed, the result is sent back to the grid. The donor software communicates with the management software, and more advanced implementations can also dynamically adjust job priority (in the case where multiple jobs execute on a machine), as well as periodically checkpoint execution to cope with failures or migrations of the job to other machines in the grid.

Schedulers

Most grid systems include job scheduling software, which decides what machine(s) on which a job should be executed. The simplest approach to scheduling is simple round-robin scheduling between machines, but more advanced systems also implement job priority systems in which jobs are queued for execution with differing priorities. The priorities of the jobs are decided by policies regulating different types of jobs, users and resources. In some cases, it may be desirable for users to be able to reserve capacity in advance in a calendar based system, which must also be managed by the scheduler. Advanced schedulers monitor jobs as they are executed in the grid, and manage the overall workflow. Jobs lost due to network outages or machine failures are automatically resubmitted on other nodes.

Submission software

End users submit jobs to the grid by using submission software, either located on dedicated submission nodes or clients, or installed on end users workstations.

2.4.3 Implementations

Condor

Condor High-Throughput Computing System [23] is a software framework for parallelizing computational tasks, and is a widely used framework to implement grid computing in both commercial and academic environments. The framework can either be used to manage a set of dedicated grid machines, or to perform jobs on idle machines (i.e. cycle scavenging).

Condor is able to run both sequential and parallel jobs, and provides different execution environments (called *universes*) in which jobs are run. One mode, called *vanilla universe*, provides an environment to run most batch-ready software without modifications. In a second execution mode, *standard universe*, software is linked to Condor-specific libraries that provide support for more advanced operations like job I/O and checkpointing.

Enabling Grids for E-science

Enabling Grids for E-science (EGEE) is Europe's largest grid computing project, involving 91 organizations from 32 countries, with a combined capacity of over 40 000 CPUs and petabytes of storage capacity [24]. The service has participating machines at different institutions spread over the world, mostly in academic environments, and aims to provide infrastructure to support collaborative scientific research.

EGEE is based on the gLite grid middleware [25], designed as a set of services. Virtual Organizations (VOs) represent individuals or organizations with access to the system, which are able to create and execute jobs on the grid. Computing resources are provided in the form of Computing Elements (CEs), offering a common interface for submitting and managing jobs on individual resources. A Workload Management System (WMS) schedules jobs on the available CEs according to user preferences and policies. A Job Provenance service tracks information about jobs executed on the infrastructure, and a Package Manager allows dynamic deployment of grid jobs.

Hadoop

The open source project Hadoop [26] attempts to provide a general-purpose framework for distributing tasks in a distributed system. By using Hadoop, developers can share jobs and large amounts of data across thousands of computing nodes without having to create custom software. Jobs are developed using the MapReduce [27] programming model, commonly used for parallelizing large jobs.

With MapReduce, users specify map functions that processes single key/value pairs to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. The MapReduce programming model, originally published by Google, has proved useful to make development of parallel tasks, but has been criticized for not being directly suitable for all types of tasks [28].

2.5 Cloud computing

Despite the significant attention cloud computing has received in the recent few years, the cloud computing community has been unable to arrive at a common definition. Making matters worse, all the hype surrounding cloud computing further clutters the message [29], resulting in confusion for consumers and potential adopters.

Although several definitions have been proposed, they generally lack a common denominator and focus on different aspects of the technology. In [30], the authors attempt to identify a set of characteristics common to a subset of the suggested definitions, but conclude that finding a fully common baseline is currently not possible. Still, they are able to isolate three distinct features that most closely resemble common denominators; scalability, pay-per-use utility model and virtualization.

Furthermore, the authors claim that cloud computing is primarily used in three scenarios; Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). We will take a further look at IaaS and PaaS later in this chapter, but argue that the two concepts are strongly related and even sometimes hard to distinguish, due to many common characteristics. Similarly, the authors of [31] argue that cloud computing has four aspects; centralized data centers, distributed computing, an utility grid approach to computing and SaaS; we argue that the three first aspects are strongly interrelated. Consequently, we argue that cloud computing has two primary aspects; a utility-approach to computational resources (IaaS and PaaS), and the notion of Software as a Service (SaaS). We will discuss these aspects in the following sections.

2.5.1 Computational resources as a utility

As a form of distributed computing, cloud computing expands on many of the fundamental principles of grid computing, but represents an evolution in terms of business models, applications, programming models, and use of virtualization.

The cloud serves as a metaphor for the Internet, representing a computing infrastructure that is logically abstracted away and in which major components reside on unseen computers with unknown whereabouts, possibly scattered across continents [2]. With cloud computing, organizations outsource software hosting from in-house data centers to centralized computing services, in which they have little or no control over the physical infrastructure.

Figure 2.6 shows a high-level overview of cloud computing. Users, or local software systems, in an organization access software hosted in a cloud environment, requesting tasks to be performed and accessing data and resources. The underlying architecture of the cloud is abstracted away, and remains internal to third-party providers. The physical infrastructure of the cloud can be spread across multiple cloud environments, potentially spanning countries and continents in data centers owned by different cloud computing providers.

The notion of computational resources as a utility is not new. When mainframe computers were common, computer usage was measured and sold in processing time.

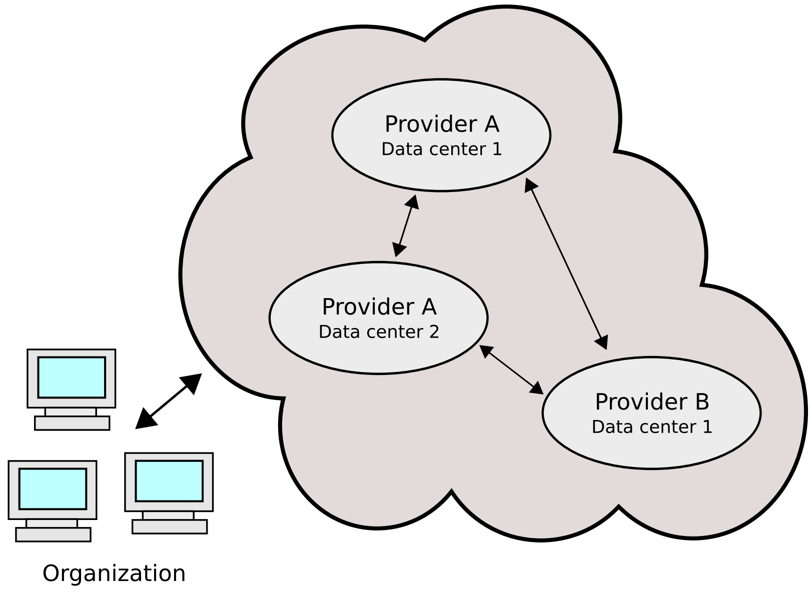


Figure 2.6: High-level overview of cloud computing.

Users submitted jobs to a mainframe computer, and were billed for the cycles or time the job consumed. A goal of cloud computing is to provide pervasive access to computational resources in a fashion similar to accessing power through electrical grids [22]. An analogy is the advent of electrical grids in the early 1900s, and the revolutionary development that followed due to the reliable, low-cost access it provided to industry and homes.

With cloud computing, the notion of utility computing once again gains traction. When computing resources are used as a utility, organizations avoid the need for major investments in hardware and facilities (such as air cooled data centers) upfront, instead paying for access to computational capabilities as a utility, meaning that it is dependable, consistent, pervasive and inexpensive [22].

In recent years, many large corporations have invested in extensive data centers designed to serve their own IT needs, with computational infrastructure based on expensive and high-powered servers [31]. With the advent of cloud computing, many of these organizations have turned the data center architecture upside down, instead basing their infrastructure on a larger number of cheap commodity hardware deployed in large numbers, as opposed to having fewer expensive servers.

With significant investments in data centers, several corporations (e.g., Google and Amazon) have seen an opportunity to generate revenue by selling the surplus capacity in their data centers; lately expanding this service to dedicated infrastructure as the market for these services has been proven lucrative. Virtualization has been a key factor in making this approach feasible, as it enables the infrastructure to be utilized in a dynamic fashion with acceptable management overhead, providing dynamic provisioning, resource sharing, isolation and security. We will discuss the use of virtualization in cloud computing in Section 2.6.3.

2.5.2 Software as a Service

Traditionally, software has been sold packaged by retailers, and users have paid for a license to use a specific version of software product. With the advent of the Internet, we have seen a number of applications instead provided to the user through Web browsers, starting with products such as Web-based e-mail and calendaring. In recent years, we have also started to see the advent of a new class of Web-based applications, like the Google Docs productivity suite and the Adobe Photoshop image editing software, offering Web-based versions of software previously only available as locally run software.

With the advent of Information Technology (IT) in academic and corporate environments in the 1960s, service bureaus and timesharing facilities provided and controlled access to computational resources [2]. Users prepared punch cards in advance and sent them to a central facility for processing, or accessed central computers using light terminals over telephone lines. With the arrival of the personal computer in the 1980s, users' increased control of their individual software and data was touted as a selling point, and users were to be liberated from the dependence on central computing facilities, and free to control their own environments, software and data. However, as the number of individual computers used as workstations or servers increase, IT departments are struggling to keep them all running smoothly. Due to the distributed nature, keeping machines fully operational and optimized is time-consuming, since properly enforcing common policies, regimes and administering updates and upgrades is difficult.

For end users, SaaS means that software can be used without local installation and configuration, and without needing to periodically purchase and perform upgrades to get the newest functionality. Also, users avoid having to invest in one-time licenses for software, instead paying smaller recurring fees on a pay-as-you-go basis, reducing entry barriers for new software while using software that is continously updated and improved.

Organizations using SaaS at a large scale generally achieve the same benefits as end users. However, since installation, configuration and maintenance generally increase with the amount of users, the opportunity for benefit in costs savings and reduced adminstrative overhead is increased. Additionally, web-based software is generally accessible from any location, as long as the user has Internet access and a web browser available, providing convenience and opportunity for increased productivity for employees.

The business model of selling software as a service is attractive for software companies because it allows for recurring revenue from customers, instead of a single upfront sale. It is likely that this trend will continue to affect the way software is licensed, deployed and used, enabled by the pervasiveness of Internet access.

However, SaaS also lead to some challenges, such as security, privacy, data ownership, reduced control and dependence on third party providers. To enable services to be accessed from everywhere, data is increasingly stored on central servers, rather than locally at users machines. This requires security at different levels; from transport over the Internet to storage in centralized data centers. Additionally, data ownership and rights have been discussed, and is a problem currently lacking standard solutions. Also, since software is hosted by a third party, users and organizations rely on third party vendors for potentially critical services. Locally run software can in principle be used without time constraints, whereas with SaaS a vendor might decide to change or cease operations, possibly rendering a large number of users without access to critical software, or even worse, their data. Further complicating is the fact that pay-as-you-go service may imply reduced commitments from software providers, since users only pay for consumed resources, as opposed to any future guarantees.

While SaaS is significantly affecting the way software is developed, deployed and accessed, we choose to focus on the IaaS and PaaS aspects of cloud computing in this thesis. Our primary reason for this choice is the fact that many of the opportunities and challenges of SaaS is rooted in economical, business model and legal areas, rather than being technically oriented. In the rest of this thesis, we will use the term cloud computing mainly to refer to the aspect of computational resources as a utility through IaaS and PaaS.

2.5.3 Motivation

For many organizations, a primary motivation for cloud computing lies in the reduced need for initial IT investments, instead paying only variable costs for resources consumed. The economy of cloud computing is illustrated in Figure 2.7, showing a qualitative relationship between fixed and variable costs.

Cloud computing provides an opportunity to host software services with reduced in-

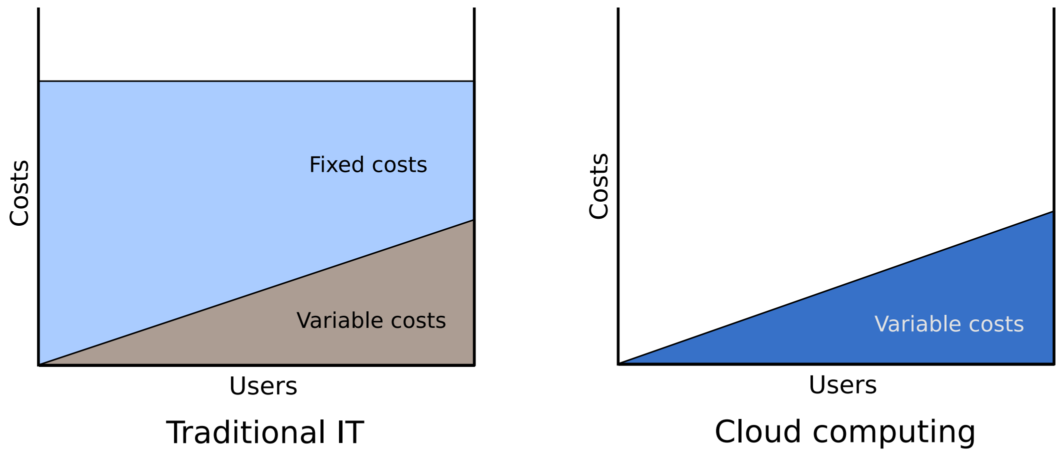


Figure 2.7: Qualitative illustration of the economy of cloud computing.[1]

vestments and running costs due to administrative and technical overhead. To reliably support large IT systems, many organizations have invested in large data centers, requiring extensive air conditioning, surveillance and perimeter security, Uninterruptable Power Supplies (UPS) to handle power outages and other expensive equipment. Organizations must also provision, purchase, install and configure hardware in the data center, in addition to keeping personell to handle data center monitoring, and hardware upgrades and repairs. By outsourcing this software hosting to a cloud computing provider, these investments are eliminated and replaced with variable pay-as-you-go costs.

A key feature of cloud computing is the notion of *dynamic scalability*, meaning that the resources dedicated to services or applications can be scaled up and down based on demand. For instance, a customer invoicing system can be scaled down from five to two servers outside of business hours, avoiding under-utilization due to over-provisioning, commonly found in typical enterprise environments in which systems must be scaled for the heaviest possible load. Similarly, the service can be scaled from five to ten servers in the last week of the month when invoices are generated and sent to customers, avoiding over-utilization leading to increased response time and elevated error rates. To support dynamic scalability, a cloud environment must support load monitoring based on some criteria (e.g., processing queue length or average CPU utilization) as well as support dynamic provisioning (starting and stopping instances dynamically).

Another opportunity presented by cloud computing is achieving high availability through

[1]Based on original figure by Sam Johnston (released with a Creative Commons license).

geographically redundant and dispersed data centers in a cost-effective manner. This means that software can be hosted in multiple data centers, with fully independent and redundant hardware, energy supply and connectivity, enabling systems to remain available even in the case of extensive failures or catastrophes like fires and floods, at any site. Providing this level of redundancy directly in an organization would be very expensive, effectively limited to only the largest corporations.

2.5.4 Ontology

In [32], the authors present an ontology of cloud computing, dissecting it into five distinct layers, shown in Figure 2.8. The authors claim that the interrelations between the different parts of cloud computing is currently ambiguous, and that a unified ontology can help enable increase interoperability, as well as help us better understand the potential and limitations of cloud computing technology. They also argue that a unified ontology is essential to facilitate maturation and further innovation in the area.

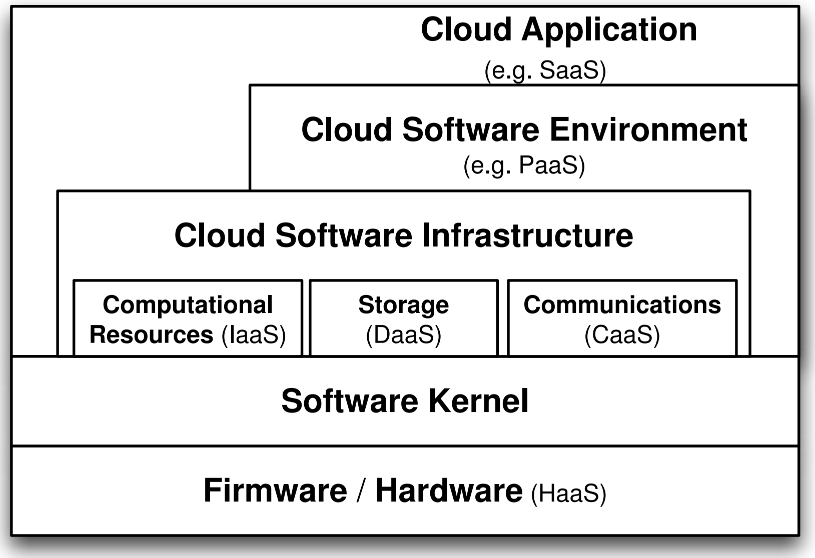

Cloud Application
(e.g. SaaS)
Cloud Software Environment
(e.g. PaaS)
Cloud Software Infrastructure
Computational Resources (IaaS)
Storage (DaaS)
Communications (CaaS)
Software Kernel
Firmware / Hardware (HaaS)

Figure 2.8: Ontology of cloud computing

The ontology is based on composability, and the principle that every layer can be built by consuming and extending the services provided by the layers below. According to this ontology, all cloud computing components and services fall into one of five layers: applications, software environments, software infrastructure, software kernel, or hardware. In the following sections, we will examine the characteristics and limitations of each layer, and describe their functionality and interfaces.

Cloud application layer

The cloud application layer is the topmost layer in the ontology, and the layer most visible to end-users. The services in this layer are typically accessed through Web portals, either free or for a recurring fee.

This model has proven popular with users because it alleviates the need to provision and support hardware to run applications and services, as well as eliminate the need for local installation and configuration. Users instead offload the computational work from their terminals to the data centers in which the cloud is located.

The advantages of providing applications and services in this manner for developers lie in the fact that it also eases work related to upgrading, patching and testing the code since there is only one system to keep updated, as well as protecting the intellectual property since users are unable to access the source code of the systems. Developers are able to roll out new features and updates without disturbing the users, as long as the system is backwards compatible with existing data.

Services provided with this model are normally referred to as SaaS, and has the advantage of ensuring recurring revenue for the service provides, as well as reducing the need for initial investments for users. Two examples of SaaS are the Salesforce Customer Relationship Management (CRM) [33] and Google Apps [34].

In the ontology, these cloud applications are developed on cloud software environments or on top of cloud infrastructure components, depending on the service. In addition, the application can be provided by composing services from other cloud systems. As an example, a cloud-based e-commerce service may consume a payment system based in another cloud.

Despite the advantages provided by this model, issues with security and availability currently might hinder adoption for many systems. Vendors must be able to fully address end user's concerns with security and storing confidential data in the cloud, as well as provide reliable uptime guarantees. Availability is an especially big concern when cloud services are composed by services from different clouds.

Cloud Software Environment Layer

The second layer in the ontology is the cloud software environment layer, which provides services to cloud application developers using the services to implement their own applications and services. The cloud software environment provides the developers with a programming language-level environment with a set of APIs to aid the interaction between cloud components and applications, provide support for scalability, and ease deployment and management. The services provided in this layer are normally referred to as *Platform as a Service* (PaaS), referring to the fact that one in effect rents access to a platform on which applications and services can be built.

An example of a service in this category is Google's AppEngine [35], which provides Python and Java runtime environments and APIs for interacting with the cloud runtime environment. Another example is Joyent Accelerator [36] which provides a platform to run Web applications written in languages like Ruby to run on top of Joyent's cloud infrastructure, also providing mechanisms to help the applications scale.

The cloud software environment may also be provided by using an existing framework like Hadoop directly on top of the infrastructure layer. The major benefit for developers implementing their services in this layer is that the environment provides useful features easing development and reducing development time, including automatic scaling and load balancing, as well as integration with other services like authentication, e-mail or communication. In other words, much of the overhead of developing cloud applications is handled at the environment level. Developers are able to dedicate more of their focus on implementing specific business logic, while outsourcing base functionality to platform services. The downside of this approach is that the resulting software will potentially be tightly coupled to the specific platform, meaning that transitioning to other providers is non-trivial and require rewriting parts of the system.

Cloud Infrastructure Layer

The cloud infrastructure layer, as the name suggest, provides the fundamental resources needed to provide upper level platforms and services. The services provided in this layer can be split into three categories; computational resources, data storage and communication.

Many properties and design features are shared between the three categories, such as availability and security. Also, they tend to share the same interfaces, typically based on SOAP [37] or Representational State Transfer [38] (REST) communication over standard HTTP [3], a principle borrowed from Service Oriented Architecture.

Developers are free to design their systems directly on top of this layer, skipping the platform layer. This results in increased freedom and flexibility, since developers can opt to use an existing platform that matches the individual system, or even implement their own platform for specific cases. This approach can also be used to transition existing enterprise systems to a cloud to reduce infrastructure investments, since existing middleware and software can ported directly by treating IaaS as a virtual data center.

However, this approach typically involves a lot of design decisions and will require more development effort than basing development on a PaaS approach.

Computational resources In cloud systems, computational resources are normally provided in the form of virtual machines, since it gives the users significant flexibility due to the fact that they have super-user access to their infrastructure, while protecting the data center by ensuring isolation between different systems. This approach is called *Infrastructure as a Service* (IaaS), and has been made economically feasible due to the recent adoption of paravirtualization and hardware-assisted virtualization.

However, these virtualization techniques still have some issues providing adequate computational service. The lack of strict performance isolation between VMs sharing physical nodes result in difficulty for vendors to provide strong performance guarantees, which in turn result in providers offering weaker SLAs to ensure cost-effective services.

Examples of IaaS products are Amazon's Elastic Compute Cloud [39] (EC2), currently the most popular commercial IaaS implementation, and the open-source Eucalyptus project [40]. We take a closer look at EC2 and Eucalyptus in Chapter 3.

Data storage Data storage is the second infrastructure resource, providing cloud systems ability to store data at remote disks, with access from several locations. This storage service is sometimes referred to as *Data-storage as a Service* (DaaS), and is essential to facilitate scaling applications beyond individual servers.

Cloud storage services must meet several requirements, such as high availability, adequate performance, replication and consistency. However, many of these requirements are conflicting in nature, so different systems make different tradeoffs with regard to what properties they wish to focus on. In general, cloud storage services are designed to be highly scalable, and focus primarily on availability and performance at the cost of consistency, often using the notion of *eventual consistency* [41].

DaaS services are based existing research into distributed storage, building on distributed storage systems like the Google File System [42] (GFS), replicated relational databases (RDBMS) like Bayou [43], and distributed key-value stores like Dynamo [44].

The most popular commercial implementation of DaaS is currently Amazon Simple Storage Service [45] (S3), which is based on the concept of storing pieces of data, called *buckets*, which can be manipulated using a REST-based HTTP interface. The S3 service is used as internal storage in many popular desktop and Web applications, such as the backup and file synchronization software Dropbox [46]. It is also increasingly used as an alternative to host widely-accessed static content, such as images, on many popular

Web sites like Twitter [47]. S3 is also widely used as storage for EC2 instances, as in our cloud search prototype discussed in Chapter 5.

Communication For large and mission-critical cloud applications, proper Quality of Service (QoS) for network communication is required at the infrastructure level. Consequently, cloud infrastructure must be designed to provide reliable, configurable, schedulable and predictable network traffic. These requirements can be fulfilled by a *Communication as a Service* (CaaS) component, providing network security, overlay provisioning, latency guarantees and encryption. This service is currently in little use commercially, but is expected to emerge with demands [48].

Software kernel

Software kernels provide basic software management of the physical servers that compose the cloud, and can be implemented in OS kernels, hypervisors or in clustering middleware. In traditional grid computing, this layer is used to deploy and run applications on a set of inter-connected nodes. However, in the absence of virtualization, grid applications and jobs are closely tied to the physical hardware, making migration, checkpointing and load balancing a cumbersome task.

This area has been subject to extensive research within the grid computing community, and several successful implementations, such as Condor, exist. Much of the work in this area can be used to provide similar functionality in the area of cloud computing, where virtualization technology is used to a greater extent.

Hardware and firmware

The bottom layer in the cloud stack is the physical hardware and switches that form the physical infrastructure of the cloud. The users of this layer are typically large enterprises with extensive IT requirements and an interest in leasing access to physical hardware.

Physical infrastructure can be provided as Hardware as a Service (HaaS), offered by vendors which operate, manage and upgrade hardware on behalf of a set of customers within some lease time. These vendors have technical expertise required to setup and host complex setups, as well as access to cost-effective infrastructure to host the systems. The economic advantage of HaaS arise from the economy of scale, since massive floor space, huge amounts of power and cooling, as well as significant operations expertise is needed.

2.6 Discussion

In this section, we attempt to compare and put cloud and grid computing into a clear context. We also discuss how virtualization is used to enable cloud computing, relate cloud computing to Service-oriented Architecture (SOA), and outline some research issues.

2.6.1 Comparing grid and cloud computing

A clear distinction between grid and cloud computing is hard to make, and the terms are often used interchangeably. The two concepts have a lot in common, and yet differ in some ways. A comparison of some key properties of grid and cloud computing can be found in Table 2.1, and will be elaborated in the following.

Area	**Grid computing**	**Cloud computing**
Motivation	Increased resource capacity	Flexibility and scalability
Infrastructure	Owned by participants	Provided by third party
Business model	Share costs	Pay as you go
Typical applications	Research, heavy batch jobs	On-demand infrastructure, scalable services, Web apps.
Virtualization	Infrequent	Prevalent
VM requirements	Scalability, isolation	Scalability, isolation, runtime migration, security
Advantages	Mature frameworks	Low entrance cost, flexibility
Disadvantages	Initial investment, less flexibility	Open issues, immature technology

Table 2.1: Characteristics of grid and cloud computing

Grid computing has been in development and use for over a decade, and several established frameworks and programming models exist. The original motivation for grid computing was to be able to tap into large amounts of computational resources, especially useful for computationally intensive tasks like weather forecasts, research simulation and High Performance Computing (HPC) applications. The motivation for cloud computing tend to differ, and is typically more related to achieving scalability and flexibility in systems design.

Business model

Setting up a computational grid requires significant hardware infrastructure, which in turn means large investments for organizations interested in taking advantage of grid computing. This lead to the adoption of shared grids, where organizations give others access to some resources in return for access to a collectively larger pool of resources. This way, the cost of acquiring and running the grid is shared between a set of users. Users of such shared grids each receive some share of resources, such as a fixed amount of CPU hours, that they can utilize without cost.

Cloud computing differs from this approach in that it is typically offered on a pay-as-you-go basis with little or no initial investment. Instead, users lease access to computational resources from a third-party provider. While distributed computing is a also key feature of grid computing, a distinguishing characteristic of cloud computing is that the distributed computing is also available to end-users or organizations without their own infrastructure.

Application model

Although grids support many types of applications, grid applications are typically job-oriented and batch scheduled [49]. Grid applications are typically not interactive, and are queued for parallel execution on a given number of nodes when resources are available.

As cloud computing is still in an early stage, we have not yet clearly seen what kind of applications that will run in the clouds. However, early applications differ from grid applications in that they execute continuously, opposed to being scheduling in batches, as well as often being interactive. However, many grid applications should be able to take advantage of cloud computing, possibly with the exception of applications that require low-latency interconnects for efficient scaling to many processors [49].

These different application models imply that cloud computing differs from grids in the way resources are shared. In grid computing, access to a pool of resources are governed by a scheduler, and multiplexed sequentially. Whereas in cloud computing, the resources in the cloud are being shared by all users simultaneously.

Programming model

Grid programming does not differ fundamentally from traditional distributed programming, but is complicated by issues like grids crossing administrative domains, heterogeneous resources and performance, as well as stability and performance [49]. In addition, since grids often cater to users with computationally intensive tasks, software must be highly efficient as well as scale well to achieve results within allotted limits.

To achieve these goals, grid software is often designed using paradigms and models common with parallel computing. One of the most popular programming models used in parallel computing is Message Passing Interface [50] (MPI), in which processes use their local memory to perform execution, and communicate by sending and receiving messages. MPICH-G2 [51] is a grid-enabled implementation of MPI, which can be used to realize parallel grid applications. Another popular programming model is MapReduce [27].

In cloud computing, the focus is more on composability, with a lot of principles borrowed from SOA. Services are designed with standard interfaces (APIs), that can be accessed over HTTP using SOAP or REST. For example, parallel execution can be achieved by having several worker nodes sharing a queue which is accessed with a SOAP API.

Virtualization

Cloud computing has been made possible due to efficient virtualization techniques, such as hardware-assisted virtualization and paravirtualization, on which it relies heavily. Virtualization is used to provide isolation and resource sharing, as well as implementing support for rapidly starting, pausing, stopping and migrating computational nodes as they are needed.

Grid computing does not utilize virtualizaton to the same extent, possibly due to organizations wanting to retain control of their resources [49]. Although some grids use virtualization to provide homogeneous execution environments, many grid systems do not use it at all. A contributing fact to this is likely the fact that virtualization in the past lead to significant performance losses for some applications [49]. Another explanation is that grids are typically shared between users cooperating (to some extent), focusing on fair resource sharing through account resource limitations. In cloud computing, the users of the shared infrastructure are not known to each other, and may even be direct competitors. This means that stronger isolation is needed to ensure security and fairness under all conditions.

2.6.2 Cloud computing and SOA

With SOA, systems are designed by combining and integrating functionality into distinct, interoperable services. Services communicate asynchronously over a network, passing messages using an Enterprise Service Bus (ESB). Due to its prospects of delivering modular and scalable systems, SOA has become popular in enterprise organizations during recent years.

The high level objectives of SOA closely resemble the principles of cloud computing, namely enabling resources and systems to be delivered as loosely coupled services, that can be combined to realize other services. Practically, cloud computing is an option for hosting SOA services, providing cost-effective infrastructure and platform as a service.

Some characteristics of SOA services make them highly suitable for cloud environments. SOA principles advocate that services should be designed in a loosely coupled fashion, communicating asynchronously using message passing. This might enable SOA services to be transparently composed of both in-house and cloud based services, with the ESB transparently routing messages between the two environments. Also, SOA services can be composed of third-party services, much in the spirit of SaaS, enabling a service to utilize a third-party cloud-based service.

Still, non-functional system requirements such as security and latency, need to be taken into account when considering cloud-enabling SOA systems, like any other enterprise system. Issues like handling transactional integrity and enforcing security and privacy across networks, as well as practical issues like firewall management, must be considered. We return to the challenges of porting enterprise systems to cloud environments in Chapter 5.

2.6.3 The role of virtualization

Virtualization serves as a major technology aiding in implementation of cloud computing, and will likely continue to do so in the coming years. We use this section to describe the properties of virtualization that make it useful in realizing cloud computing infrastructure.

Isolation

One of the most important characteristics of virtualization is the isolation it provides between different instances. Virtual machines running different classes of services can execute arbitrary code without affecting each other, provided that the VMM schedules resources fairly, with proper Quality of Service enforcement ensuring the each instance gets the resources it requires. Isolation also helps ensuring correctness of services, and that failures in one or more instances does not adversely effect other instances. In addition, virtualization helps security by limiting namespaces and performing access control to prevent code from accessing resources and data without proper permissions.

Heterogeneity

Over the years, computer architecture has evolved dramatically, leaving a lot of legacy systems around for critical tasks, especially in some sectors such as finance, insurance and public service. These systems may be hard to maintain because of lack of replacement parts for legacy hardware, and lack of competent developers for running these services. Luckily, the advent for virtualization has made it possible to run these systems on top of virtualized hardware, which allows systems to continue running on reliable and state of the art hardware.

Since virtualization provides isolation between systems, it also allows services with different behaviors to co-exist on the same physical hardware. Where typical companies had distinct hardware for database and Web servers due to different resource consumption and requirements, such heterogeneous services can now co-exist on the same physical hardware provided that proper Quality for Service is provided by the VMM.

Flexibility

Virtualization provides abstraction between services and the underlying hardware, and therefore allows a greater flexibility in infrastructure. When demand for a service increases, the service can be scaled up by awarding it increased resources from the VMM, or migrating it to a more powerful environment. Provided support is available in the middleware, this migration can happen with little or no delay, completely transparently to users.

Since virtual instances can be started and stopped very rapidly, virtualization also provides flexibility in resource consumption. If a test server is needed for quality assurance in a software development company, a virtual instance can easily be setup and be ready for use in hours or minutes. When the tests are complete, the server can be safely discarded and the hardware repurposed for other useful tasks. Furthermore, virtualization provides IaaS providers with an opportunity to evolve the underlying infrastructure without affecting users and customers, since the virtualization handles the interaction with the hardware.

2.6.4 Research issues

In [52], the authors outline some of the areas in which cloud computing still has open research issues.

One of the major issues is construction of services and images for realizing cloud applications. To provide cost-efficiency, mature tool and middleware support is needed to allow developers to create services that properly scale and migrate within the cloud or among clouds. Development time must be reduced to make distributed computing cost effective for more projects than the ones that are already distributed today.

With many vendors appearing within the areas of cloud computing, portability is needed to allow VM images to be run in cloud environments from different vendors, and avoid infrastructure lock-ins and reduced competition and flexibility. An interesting project in this regard is the *Open Virtualization Format* (OVF) [53], which aims to standardize the packaging of virtual machine images so they can easily and efficiently be deployed independently of platforms and vendors. Projects such as Citrix' *Project Kensho* [54] aim to provide tools for working with virtual machines packaged according to the OVF standard.

Although many organizations have already started utilizing virtualization to improve utilization of their existing infrastructure, the issue of return on investment is complicated, since direct comparison to existing systems is difficult [52].

Cloud computing also raises significant questions about security and privacy. In [2], the authors raise a few of the questions organizations must consider when transitioning to cloud computing. One challenge is how organizations can take their data with them when they move to another provider, and another is what happens with the data if a company fails to pay the bill or wants to completely expunge some information. Also, if the government issues a subpoena or search warrant against a customer's servers, a vendor might be less likely to go to court to prevent access to customer data than the customer would if it retained full control of the data. A related challenges is the need for organizations to document regulatory compliance, requiring guarantees for data storage, transport and access.

2.7 Summary

Virtualization is an established technology with a long history, and has recently acquired renewed interest due to the prospect of consolidating and improving hardware utilization. Paravirtualized VMs like Xen offer relatively resource-efficient virtualization, and serves as enabling technology for cloud computing.

Grid computing is an established form of distributed computing, in which a large number of machines are connected to form a single grid, typically with the goal of exploiting available resources or performing computationally intensive tasks. Grid computing is often found in academic environments, and several popular implementations exist.

Cloud computing expands on many of the fundamental principles of grid computing, but represents an evolution in terms of business models, applications, programming models, use of virtualization. We argue that cloud computing has two primary aspects; a utility approach to computing, and Software as a Service (SaaS). In this thesis, we focus on the utility approach to computing, through Infrastructure as a Service (IaaS) and Platform as a Service (PaaS).

SaaS represent a return to the original idea of centralized computing services, where the software itself is hosted on centralized machines at service providers, with users accessing the services through thin clients, such as Web browsers.

In the next chapter, we will take a closer look at some state-of-the-art cloud environments, and discuss some inherent properties and characteristics. These characteristics will later serve as a basis for an investigation into the approaches and challenges in designing systems for cloud environments.

Chapter 3

Characteristics of cloud computing infrastructure and platforms

3.1 Introduction

To successfully design applications and services for the cloud, developers must have knowledge about the opportunities and limitations of cloud infrastructure and platforms, and know the characteristics of cloud environments compared to classical enterprise environments.

This chapter discusses key characteristics of cloud computing infrastructure and platforms offered by state-of-the-art IaaS and PaaS vendors, in turn forming basic requirements for cloud software design.

3.2 Characteristics

3.2.1 Horizontal scalability

The traditional way to scale applications is to gradually add more capacity to the machines running a service as demands increase; a model that historically has proven successful for many applications. This approach has led to the popularity of large and expensive mainframe computers, such as high-end servers from IBM and Sun. In 1965, Gordon Moore successfully predicted that the number of transistors used in microprosessors double approximately every two years, an observation known as Moore's Law, which has generally materialized through CPU frequencies doubling at the same rate.

However, recently this approach has shown some limitations. One of the major limitations is the extra heat generated by adding more and more transistors to scale up performance, increasing the need for cooling to avoid over-heating the hardware. This results in systems that require vast amounts of electricity, making them less economically attractive.

A trend popularized by Google, is to use large numbers of commodity machines instead of large mainframe systems. This approach also forms the basis for cloud computing, and affects how cloud software should be designed to scale. Instead of scaling vertically by adding more resources to individual nodes, we scale horizontally by adding more nodes to the pool of nodes collectively running an application. The difference between horizontal (scale-out) and vertical (scale-up) scaling is shown in Figure 3.1.

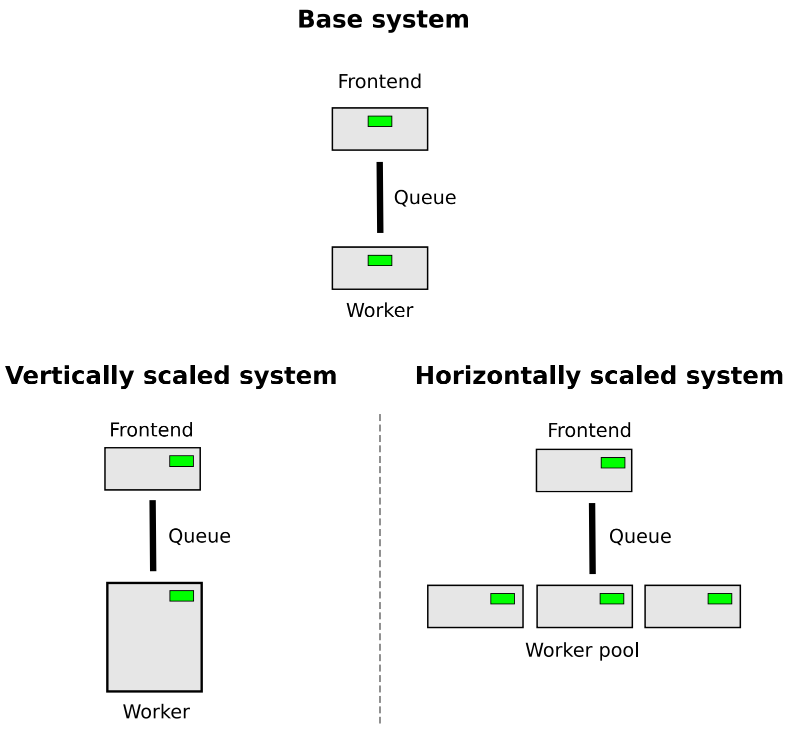


Figure 3.1: The difference between vertical and horizontal scaling

While this approach is intuitive in principle, it does pose some requirements to the way software is designed, since many traditional software design patterns do not enable systems to directly utilize horiztonal scaling. A trend is to use best practices from SOA to allow the systems to scale horizontally, using asynchronous communication such distributed message queuing or communication over a shared bus. This helps systems scale since worker nodes can be added to collectively process messages received from a shared message queue or bus.

Some PaaS vendors (e.g., Scalr [55]) provide tools that aid in system scaling, performing continuous load and health monitoring, comparing the observations with predefined scaling criteria and automatically starting and stopping nodes as required. The scaling criteria can be based on measured parameters like CPU load or memory consumption in individual nodes, or more complex parameters such as a work queue length or service latency. For cloud applications with varying load, automatic scaling can be a key advantage helping to achieve cost-effective operation, taking full advantage of the pay-as-you-go nature of cloud computing.

3.2.2 Unknown physical topology

When designing software for cloud environments, developers have little or no information about the physical infrastructure on which the software will execute. As long as platform vendors are able to fulfill their Service Level Agreements, generally limited to guaranteeing a minimum level of availability, they are free to configure the underlying infrastructure as they wish.

In addition to the challenges involved in running software on unknown hardware, comes the effects from using virtualization to share physical machines between multiple customers. This has several implications that must be taken into account when designing software for the cloud.

In corporate data centers, network latency is relatively constant and predictable, at least in the absence of network congestion. The servers hosting different components and sub-systems have fixed locations, and can be placed near each other to optimize network traffic. For example, a database server can be placed on the same part of the network as a server making intensive queries, avoiding affecting the rest of network while achieving low latency for requests.

Developers designing systems for cloud platforms are unaware of, and unable to control, the underlying network infrastructure, and consequently they are not able to perform these optimalizations. Sub-systems might be located on nodes in different parts of the network, with little control of the cross traffic in the network. In extreme cases, nodes may even be located in geographically distant data centers, although some cloud vendors (e.g., Amazon EC2) provide a coarse-grained control of infrastructure (like enabling customers to choose between geographical zones when starting nodes).

Additionally, since systems execute in a virtualized environment, they might also be prone to differences in scheduling. If the VMs running on a physical machine are not properly isolated, some instances might experience unfair scheduling, in turn affecting latency. For example, if I/O operations are not properly isolated between instances, an I/O intensive application can adversely affect the performance of another customer's system. Developers have no control over load inflicted by other customers on shared physical machines, and this load can change at any time as other instances are started and stopped on the machines.

3.2.3 Increased latency

Network latency is limited by the speed of light, and further increased by queueing delays and congestion in the network. Since cloud data centers are currently centralized in a few locations, packets headed to or from cloud environments must generally travel significant geographical distances to or from client networks. Compared to a traditional in-house network, the latency required to reach a cloud might be an order of magnitude higher.

When designing cloud applications that need to transport data between different data centers and networks, such as between a corporate data center and a cloud, latency can become an issue. Similarly, latency might also be a challenge when applications in one cloud need to communicate with systems running in another cloud, for example to consume a third-party service. If latency is a critical issue, such as for highly interactive tasks where users are continually working with the system and waiting for the results of operations, such inter-network communication may not be desirable.

3.2.4 Loose coupling

Since the underlying infrastructure is unknown, nodes in cloud systems will be loosely coupled by nature. When new instances are instantiated, they receive network addresses dynamically. Since cloud environments should support quickly scaling applications up and down by instantiating and releasing instances on demand, the traditional approach of using the Domain Name System (DNS) to address other nodes might be unsuitable due to DNS propagation delay.

In enterprise systems, the different sub-systems typically use DNS to discover and address each other, communicating directly by sending packets directly to the recipient over the network. For this model to work in a cloud environment, some other mechanism must be used to enable nodes to discover and address one another. This mechanism can be custom to the application, or be provided as a service within the cloud. To overcome the limitations of slow DNS propagation upon node instantiation, Amazon EC2 offers special IP addresses called *elastic IP addresses*, which are allocated in advance and related to a user account. These addresses can be mapped to different instances in a relatively rapid manner, making them useful for fail over and load balancing.

The recommended communication pattern is indirectly using shared queues and messaging, as in SOA. This avoids both the need for direct discovery and addressing, while ensuring that the architecture is scalable since sub-systems can be horizontally scaled.

3.2.5 Designing for failures

In massively distributed systems, failures will occur both in nodes and in the network. This is also the case in cloud computing, where the infrastructure is based on large volumes of commodity hardware, each of which can (and will) periodically fail. Cloud environments are still able to provide high availability in the face of hardware failures because there is a large amount of redundant hardware available on which execution can resume. If a node fails, it can be transparently restarted or migrated to another node and resume execution. Still, this failure model has several implications for developers designing systems for cloud environments.

Software running in the cloud must be able to sufficiently handle failure without crashing, meaning that failures must be treated like a part of normal application workflow rather than catastrophic events. When a node fails, other parts of the system should be left unaffected, and the failing node should restart or be replaced by a new instance, either directly by the system, or transparently by the cloud environment.

Handling these events imposes some requirements on the different components in a cloud application. The principle requirement is that every node in the system must be replaceable at any time, and that the system contains no *golden nodes* that are irreplaceable [56]. Due to the nature of clouds, nodes can start and stop at any time due to failures, or simply due to scaling, a scenario every component must be designed for.

A challenge in overcoming this problem is how to manage *application state*. Application state consist of configuration and the data of the application. While configuration can typically be re-fetched at any time from some fixed location, application data requires the system to actively perform replication and updates in a reliable manner.

One approach to address this issue is to view each node as a cache [56], and keep state replicated in a reliable fashion. Whenever a node performs an operation on some data, the updated data is pushed and replicated across different nodes, with local data only used to increase performance. When a node or component crashes, the node should be able to fully reinitialize itself upon restart, fetching both configuration and data from other nodes in the network. No assumptions must be made that any state is available locally on the machine.

In addition to handling state in the face of failures, applications should be able to resume execution after failures or scaling without risking data consistency. This issue can be handled by designing cloud systems using stateless and idempotent operations, meaning that operations may be safely executed several times without resulting in errors or incorrect data.

3.2.6 Sandboxed environment

To facilitate isolation between customers, as well as limiting potentially malicious activity, software executing in a cloud environment run in a sandboxed context. Depending on the specific cloud environment, the sandbox prohibits certain actions or access to specific resources. These limitations exist on top of the isolation provided by the virtualization software.

On some platforms, such as AppEngine, applications are not able to access any files on the physical disk directly. This limitation is due to the fact that most Web applications it is designed to support do not require disk access to perform their intended tasks. Other environments may restrict sending e-mail (by blocking specific network ports), in an attempt to prevent the use of their platform to send unsolicited e-mail. AppEngine tackles this issue by imposing a maximum number of e-mail messages that can transmitted within a time frame, along with a standard API for sending e-mail (while blocking other means of transmission).

To protect the network and instances against malicious traffic, firewalls are employed to filter both incoming and outgoing traffic. In the case of both EC2 and Azure Services Platform, application developers must explicitly enable traffic to and from specific port/protocol pairs, to ensure that a minimal potential attack interface is exposed.

3.2.7 Security and privacy

With traditional in-house hosting of servers and data, companies need only to secure their own services to protect their data. With cloud computing, the data is moved out of the company network, and onto the servers of third parties. Since security is only as strong as the weakest link, adding more links to the security chain risk making it more vulnerable.

Companies that move their data to the clouds need to fully trust the cloud infrastructure providers with access to their systems. Data can potentially be intercepted on the way to the cloud, leading to a new set of challenges for secure transportation. Some of these challenges are met with security protocols like Virtual Private Networks (VPNs), safely tunnelling encrypted data between sites. However, data must still be decrypted on the cloud to allow processing.

Cloud computing also raises some legal questions, such as who is accountable if sensitive data leaks to competitors or the public. These challenges must be taken into consideration before moving systems with sensitive data onto a cloud infrastructure.

3.2.8 Continuous service

In many enterprise scenarios, systems are periodically taken offline for short periods of time to perform maintenance tasks and upgrades. In contrast, some cloud platforms are designed to allow continuous service, with tools in place to support live upgrades. For example, Azure Services Platform provides support for defining *upgrade zones*, which partitions your system into distinct sets of nodes, each of which can be individually taken offline for upgrades while ensuring that service remains available. System upgrades are managed by setting up a new version of the system, and then activating a *rolling upgrade* that happens transparently in the individual zones. The upgrade zones are also used transparently to keep the underlying operating system up to date, by rolling out periodic patches incrementally in the background.

A version switch mechanism is also provided, in which one configures the new version of the system in a staging environment separate from the production environment. This mechanism is illustrated in Figure 3.2. When the upgrade is activated, the two systems change places by redirecting load balancers, enabling the system prepared in the staging environment to go into production. If there is an error with the new version, the same mechanism can be used to roll back to the previous version of the system. For this mechanism to work optimally, applications must be able to handle schema changes in the underlying data, either by performing conversion as needed or working with versioned data.

3.2.9 Development and platform constraints

Some cloud platforms impose constraints on the development process or runtime environment that restrict developers' choices. For example, some platforms (e.g., the Azure Services Platform), provide a single software platform consisting of an operating sys-

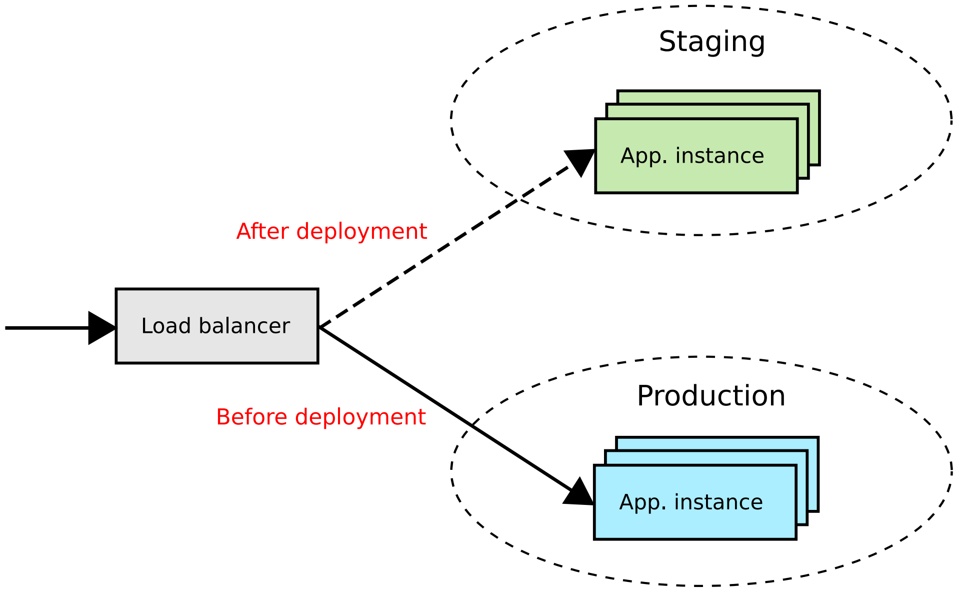


Figure 3.2: Continous service through version switching. New versions are prepared in a staging environment, after which the load balancer instantly redirects incoming requests.

tem with preconfigured features and APIs, which developers extend with custom software. In Azure Services Platform, software must be written in a *managed language* like C#, Microsoft's equivalent to Java, enabling it to be easily sandboxed and supported in the cloud environment.

Google AppEngine is another example of a platform that imposes constraints on development. AppEngine only supports applications written in the Java or Python languages, also restricting some parts of the standard libraries (such as direct file access). To deploy applications, they must be packaged in a specific format and upload with a custom tool. Consequently, AppEngine is not platform or language agnostic.

In addition to specific development tools and runtime environment, vendors might require users to use a provided control interface for starting and stopping nodes, performing upgrades or accessing status and health information. However, many cloud providers also provide access to the same functionality via standardized web services API, enabling integration with external tools and services.

3.2.10 Development complexity

The difficulty of transitioning development from sequential software into concurrent multi-threaded applications has been a subject to significant research for many years. When programming for the cloud, developers must not only design software to run on single machines, but potentially scale to tens or hundreds of machines. This requires strict attention to the effects of interaction between the individual machines, and coordination between different subtasks to avoid inconsistency, deadlocks and race conditions.

In addition, debugging cloud applications is currently more challenging than debugging local applications, since tool support is not yet mature. Intermittent synchronization problems in application scaling to hundreds of machines also make problems significantly harder recreate and isolate. To alleviate this problem, frameworks like GridBatch [57] and MapReduce [27] aim to provide middleware infrastructure which abstracts away common tasks and everyday challenges for developers designing software for the cloud. However, these frameworks still have a significant learning curves, and it may be hard to find developers experienced with these emerging technologies.

3.2.11 Provided tools and features

Although the specific details differ, cloud platforms provide tools and abstractions to make it easier for developers to design scalable software for cloud environments. For instance, since the database or storage systems can become bottlenecks when designing massively scalable services, cloud platforms generally offer interfaces to work with distributed databases or a replicated storage service. These databases and storage services are typically designed to be heavily scalable, but are not generally not feature complete compared with full relational databases or file systems.

Communication between the nodes in the cloud is commonly handled by using a distributed queue or messaging service, which is exposed with a simple interface for cloud applications. The queuing service can optionally fulfill some additional guarantees, such as guaranteed delivery of messages.

Because logging and debugging applications distributed over tens, hundreds or even thousands of nodes quickly become challenging, most platforms provide some mechanism for distributed logging, and tools for debugging. Windows Azure also provides a local version of the cloud environment (called *fabric*), which enables developers to run cloud applications locally with full debugging support.

3.2.12 Limited vendor portability

Currently, most cloud platform vendors use custom packaging and meta data formats proprietary to the specific cloud environment. In addition, code using specific APIs or tools provided by a cloud platform will be tied to that particular cloud environment. This means that it may be non-trivial to move cloud applications to another provider without rewriting and repackaging substantial parts of the application. However, standardization work, such as Open Virtualization Format [53], is currently in progress in this area, and it is likely that this situation will improve as cloud technology matures.

3.3 IaaS services

In this section, we will take a look at the functionality provided in state-of-the-art IaaS offerings, with the goal of understanding what kind of functionality and features to expect, as well as discuss the ways these features can be utilized.

3.3.1 Amazon EC2

Amazon EC2 was the first commercial IaaS to receive widespread attention and gain market traction. It was released in beta form in August 2005, and made generally available in October 2008.

Infrastructure

EC2 is based on Xen virtualization technology, and is hosted in Amazon's data centers in USA and Europe. Customers reserve and instantiate virtual machines, called *instances*, which are equivalent to virtual private servers (VPSs).

Instances are classified and priced based on the *EC2 Compute Unit*, which is the basis for the instance types available to users. Because the EC2 infrastructure is based on commodity hardware, this is an abstract unit used to describe the different instance types in a consistent and predictable way, while allowing Amazon to evolve the underlying physical hardware. An EC2 Compute Unit is equivalent to a 2007 1.0-1.2Ghz Opteron or Xeon processor. In addition to CPU specification, each instance type has a I/O performance indicator which indicates the share of I/O resources an instance is allowed to use. Customers can currently choose between three standard instance types (small, large and extra large) as well as two high-CPU classes (medium and extra large) intended for especially CPU-intensive tasks.

EC2 provides support for running a wide range of operating systems, including several flavors of Linux, OpenSolaris and Windows. Support for these operating systems are provided in the form of preconfigured images that can be instantiated directly. In addition, users can deploy operating systems other than the preconfigured ones by packaging and uploading custom images, provided that they support the Xen hypervisor.

Environment, development and deployment

Being an IaaS, EC2 is a flexible environment, giving developers a wide range of options with regards to OS, development platform, database architecture, Web and application servers as well as support for custom middleware. The service is development platform and language agnostic, and developers are free to use scompiled languages like C, dynamic languages like Java or scripting languages such as Python to realize their systems.

The penalty for this flexibility is that EC2 provides no direct language-integrated API for services, and does not directly offer any integrated debug support. Developers must manually setup and configure libraries and debugging support themselves. However, this functionality is often supported by middleware and frameworks such as Hadoop [26] or Condor [23], which can run on top the raw EC2 infrastructure, available as preconfigured base images. EC2 also supports several different common database products, such as Oracle, Microsoft SQL Server and MySQL, which can be deployed and distributed across instances, enabling the traditional enterprise database model to be used instead of or in addition to Amazon's proprietary cloud database solutions.

Systems are set up by instantiating preconfigured base images, or by deploying complete images with OS, software and configuration settings. To facilitate creation of these images, called *Amazon Machine Images* (AMIs), developers have access to tools for packaging and uploading virtual machines in a standardized format with Amazon-specific meta-data. Packaged AMIs can be kept private to developers, or be publically shared with other developers free of charge or for a cost.

Integrated services

While the focus of EC2 is to provide a flexible IaaS, Amazon have provided integration with its other cloud services. When combining the infrastructure provided by EC2 with the services, developers effectively end up with a light-weight PaaS environment.

The Simple Queue Service [58] (SQS) offers a reliable hosted queue service for storing messages, and is designed to be highly scalable. Developers can use SQS to pass messages between different components in the system, residing on different instances potentially in different data centers. All components can run independently, and are temporally coupled.

Developers can create an unlimited number of queues, each dedicated for a specific task. Each queue can hold an unlimited number of ordered messages, with each message up to 8KB in size. The service provides reliability by locking messages while they are being processed, expiring the lock if the processing task does not complete within a fixed time interval. The service is based on polling, and worker instances must periodically poll the queue for new messages. The SQS service also supports monitoring, and the control interface can be used to monitor attributes such as the number of messages stored in the queue at any time. This allows developers to monitor health and load based on the work queues, scaling the application as required.

Amazon Simple Storage Service [45] (S3) provides a scalable SaaS, and can be used to store an unlimited amount of files in a reliable fashion. Files are stored as objects, containing data and some meta data such as modification time. Objects can be up to 5 gigabytes in size, and are stored in containers called *buckets*, identified by a unique key. Buckets are used to separate between different namespaces, and the combination of a bucket and an object key uniquely identifies one object. File download is over HTTP by default, but the service also provides a BitTorrent interface which can lower distribution costs by utilizing peer to peer distributions.

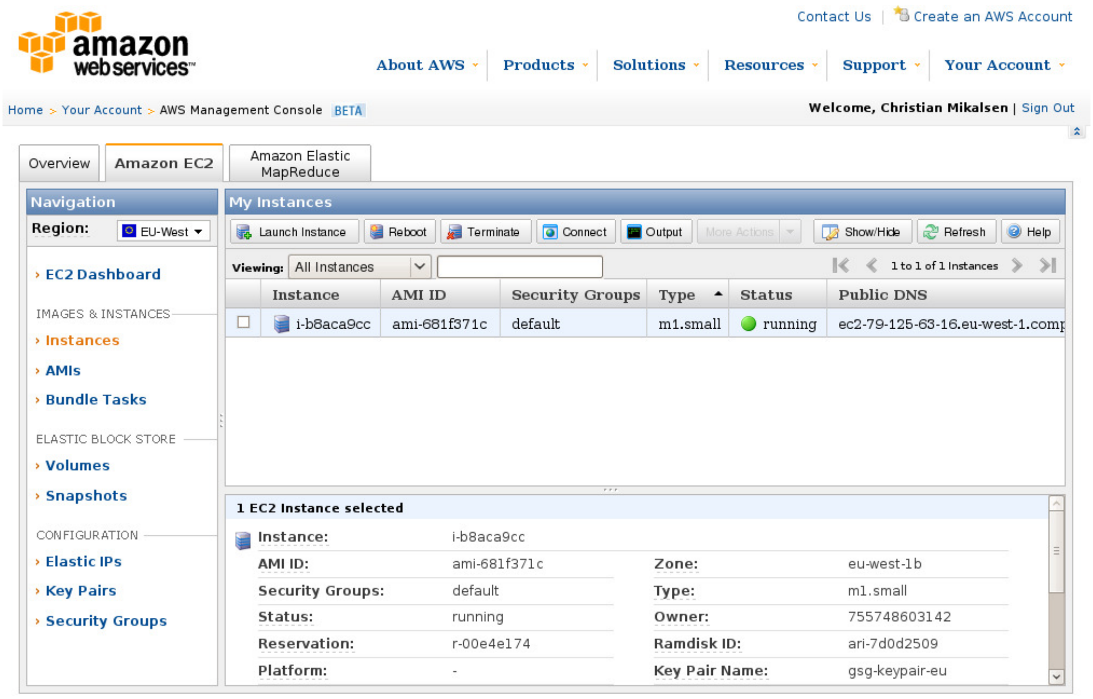
Developers have some control of the zone in which individual buckets are stored, but objects are otherwise transparently stored and replicated automatically. Updates are atomic, and consistency is based on timestamps of writes. However, data consistency is not guaranteed across all replicas, and outdated or deleted data may be returned from read requests until all changes have successfully propagated.

SimpleDB provides simple database functionality, such as storing and querying cloud data sets. The service is designed to be highly scalable at the cost of functionality, and does not provide advanced features found in modern relational database management systems (RDBMS). The system is based on *structured data*, organized into domains. Queries can be run across all data stored in a domain, which are comprised of items described by name, value pairs. A major difference from traditional RDBMS is the lack of support for defining schemas with attributes and constraints. Data is added and manipulated using a REST [38] based protocol, while queries are supported by a API that supports a subset of the query functionality found in Structured Query Language (SQL) [59].

Control interface

Amazon provides APIs for controlling EC2 and related services over HTTP, using SOAP [37] or REST protocols. The interface is described in a Web Services Description Language [60] (WSDL) document, which facilitates service integration with most programming languages and environments.

Available API methods include operations for listing running instances, starting and stopping instances, configuring networking, reading console output, and attaching/detaching volumes. Requests are encrypted and secured with HTTPS and X.509 [61] public key infrastructure, to prevent eavesdropping and unauthorized access. EC2 comes with a set of downloadable command line tools that can be used when designing and deploying cloud systems for EC2, serveing as thin wrappers around the API. A web-based console, shown in Figure 3.3, is also available to graphically manage and configure instances.



Instance	AMI ID	Security Groups	Type	Status	Public DNS
i-b8aca9cc	ami-681f371c	default	m1.small	running	ec2-79-125-63-16.eu-west-1.comp

1 EC2 Instance selected

Instance:	i-b8aca9cc		
AMI ID:	ami-681f371c	Zone:	eu-west-1b
Security Groups:	default	Type:	m1.small
Status:	running	Owner:	755748603142
Reservation:	r-00e4e174	Ramdisk ID:	ari-7d0d2509
Platform:	-	Key Pair Name:	gsg-keypair-eu

Figure 3.3: Amazon EC2 web-based administration console.

Scaling support

While EC2 is designed to support highly scalable systems, it does not directly provide support for load balancing. Instead, developers are required to setup their own load balancing systems. Neither does it support automatic scaling, instead relying on developers to configure such functionality themselves.

However, using services like SQS, developers can use EC2 to create systems that scale well with demand, as long as they are able to configure the software directly. EC2s main contribution to realizing scalable systems is the fact that instances can be started and stopped in a matter of seconds or minutes, in addition to providing simple access to a set of highly scalable services.

Price model and Service Level Agreement

Amazon charges customers for resources consumed using a variable pricing model based on usage, without any flat fees. EC2 compute instances are billed per instance-hour, with the price varying based on instance type and configuration. Customers requiring specific guarantees that instances of a desired type are available when needed can optionally utilize *Reserved Instances*, which are instances for which an advance fee has been paid in return for being privately reserved and available at an reduced hourly rate.

In addition, customers are charged for traffic passing in and out of the EC2 network. Users also pay fees for using services such as SQS and S3; for instance is a fee charged per SQS operation and per GB of data stored in S3. However, Amazon claim that the pricing of these services is intended to be highly competetive compared to comparable solutons due to economy-of-scale.

EC2 and related services are bound by a Service Level Agreement, guaranteeing a minimum amount of service uptime. In the case of EC2, Amazon guarantees that uptime should be no lower than 99.95% during a period of one year. If uptime falls under this requirement, customers receive service credits in compensation.

3.3.2 GoGrid

GoGrid is a commercial IaaS provider, claiming to be the first cloud computing platform that allows users to manage their cloud infrastructure in a Web browser [62].

Infrastructure

Similar to EC2, GoGrid is based on Xen virtualization, running on a cluster of machines in their data center. Users can choose between a set of machine configurations when provisioning new machines, and is guaranteed a certain amount of memory and a specified share of an Intel P4 2.0 equivalent CPU. The machine is also tied to a specific amount of *cloud storage*.

GoGrid infrastructure consists of three predefined roles; database servers, application servers and load balancers. Users provision and setup servers from a pool of preconfigured templates, selecting memory and CPU configurations.

Environment, development and deployment

The service supports a range of Windows and Linux operating systems, available through *templates* with different combinations of operating systems and software. In addition, users adjust settings such as public IP address and hardware configuration using the control interface. Users are currently not able to create custom templates, but GoGrid has announced that this feature will be available at a later stage.

Integrated services

GoGrid provides few integrated services, currently limited to a cloud storage service. The cloud storage is a DaaS service, mounted as a disk volume on servers. The storage is only available inside the cloud network, and not accessible outside. Users access the cloud storage using protocols like File Transfer Protocol (FTP) or Samba [63] file sharing. Customers are given an initial free quota of storage, and are billed per gigabyte exceeding this quota. Storage quotas are updated daily in batches of 100GB, which is increased when 80% of the current quota is used.

Control interface

GoGrid is designed to be configured and administered graphically using a Web-based control interface. The interface allows users to setup a virtual infrastructure consisting of machines and load balancers. After servers are provisioned, they are available within a couple of minutes. They can then be started, paused or stopped using the control interface.

GoGrid also provides an API for controlling the infrastructure, available using REST, like EC2. The API provides full operations to setup and configure infrastructure, start and stop servers, as well as retrieving billing information.

Scaling support

GoGrid contribution to helping developers create scalable applications is by automating common challenges related to scaling, and easing or eliminating manual configuration when scaling up or down.

Users configure hardware load balancers to help balance traffic between different servers, that can be configured to use different modes of operation, such as round robin where requests are sequentially balanced between a set of servers, and hash-based balancing based on sessions or source IP addresses.

Price model and Service Level Agreement

GoGrid provide both pay-as-you-go and pre-paid pricing plans, with the former charging users per *RAM hour* of use. One RAM hour is equivalent to 1 gigabyte of memory provisioned per hour, meaning that a machine with 512MB memory running for an hour is 0.5 RAM hours. This reflects the fact that GoGrid uses RAM amount as the basis for the available instance types, with a corresponding CPU capacity. Pre-paid plans are packages consisting of a certain amount of RAM hours within a period of one month, which must be pre-paid by customers. The pre-paid plans are metered the same way as the pay-as-you-go plan, but offer lower prices per RAM hour.

The GoGrid Service Level Agreement guarantees 100% availability, compensating customers experiencing outages with service credits 100 times the outage period. In addition, the SLA guarantees that GoGrid will make every *commercially practical effort* to ensure data saved onto cloud storage is persistent. GoGrid also provides network traffic guarantees, including figures for latency, acceptable packet loss, jitter and outage, both for internal and external traffic.

3.3.3 Eucalyptus

Eucalyptus [40] [64] is an open-source IaaS implementation, developed at University of California, Santa Barbara. Eucalyptus aims to provide standards-based software for deploying and maintaining computational clouds, as opposed to the proprietary and opaque systems supporting EC2 and GoGrid.

The goal of the Eucalyptus project is currently to provide a framework for further research related to cloud computing rather than providing an efficient infrastructure for production environments, although this is possible. Eucalyptus is designed to be installed locally on a cluster of machines, effectively resulting in a private cloud. It is based on standard Linux software, requiring few or no modifications to host systems, and is based on Xen virtualization.

One of the main features of Eucalyptus is that fact that it is interface-compatible with EC2, meaning that management tools and systems for EC2 can be used unchanged with Eucalyptus. The management architecture is designed to be flexible, potentially expanding to support other interfaces. Additionally, Eucalyptus is bundled with an open-source interface-compatible implementation of the S3 cloud storage service, dubbed Walrus.

3.4 PaaS services

3.4.1 Google AppEngine

Google AppEngine is a PaaS offering from Google, released as a beta version in April 2008. The initial release of AppEngine focused on providing a platform for creating consumer Web applications, supporting a modified version of the Python [65] runtime and the Django [66] web framework. Recently, support for the Java [10] programming language has been added, making the service interesting for more enterprise users.

Developers designing systems for AppEngine download a Software Development Kit (SDK) containing libraries and software enabling applications to be developed and run locally, emulating the services available in the AppEngine cloud environment. To deploy a system on the AppEngine cloud, developers use a custom upload tool provided by Google along with a Web-based control interface.

One of AppEngine's biggest selling points is the tight integration with BigTable, Google's massively scalable distributed storage system [67]. Developers are able to easily store and retrieve objects using a provided Object Relational Mapper (ORM) tool integrated in the runtime environment. With the data store being the most difficult component to reliably scale with load, AppEngine aims to provide an attractive solution for many Web application developers.

In addition to the BigTable service, AppEngine provides developers with API access to a range of other useful services, such as caching, e-mail sending, image manipulation and account management. For example, distributed caching is available through a custom version of *memcached* [68] offered through an integrated API.

To aid in the task of exchanging information and data between the AppEngine cloud environment and in-house LANs, Google provide a component called the *Secure Data Connector* (SDC). The SDC runs inside an organizational LAN, setting up an encrypted tunnel to servers within the AppEngine cloud environment. Developers configure resources rules, defining the resources that can be accessed in different locations, including support for authorization.

The primary advantage of AppEngine, like most PaaS offerings, is that it is relatively straightforward to use, providing developers with an opportunity to rapidly get started developing applications. The flip side is that systems are constricted within the provided platform, which may not be flexible enough for some scenarios. Additionally, developers must specifically design their applications for the AppEngine platform, resulting in non-portable applications.

Google AppEngine is free for users up to a certain level of consumed resources, after which resources like CPU hours, storage and bandwidth is billed.

3.4.2 Azure Services Platform

The goal of the Azure Services Platform is to provide a *cloud operating system*, allowing developers to host and run applications in Microsoft data centers. The Azure Services Platform uses a specialized operating system, Windows Azure, serving as a runtime for systems running on the platform. The Windows Azure runtime environment runs on top of a *fabric controller*, that manages replication, load balancing as well as providing cloud services like storage.

Developers design systems for the Azure Services Platform as managed code for the Microsoft .NET framework, currently using the programming languages C# or VB.Net [12]. Integration with cloud services is provided through client libraries integrated in the .NET framework, enabling developers to use the Visual Studio [69] Integrated Development Environment (IDE), including debugging support.

The Azure Services Platform provides a a range of services to ease development of scalable applications. The provided services include a distributed message bus and workflow service, designed to aid communication between applications and services, as well as a cloud-based relational database supporting structured, unstructured or semistructured data.

Developers design their application as a set of *web roles* and *worker roles*, illustrated in Figure 3.4. Web roles accept incoming HTTP requests passing through a load balancer, and communicate with the worker roles through message queues provided by the fabric controller. Web roles are intended to perform short tasks and handle presentation, with worker roles performing more intensive tasks, like resizing an image. When the application is deployed, the system topology is described in a configuration file, allowing the fabric controller to understand how the individual components of the application are organized. This knowledge allows the application to be dynamically scaled, for example by adding and removing worker roles as required. Additionally, it enables support for features like *rolling upgrades*, in which individual components can be upgraded without taking the application off-line.

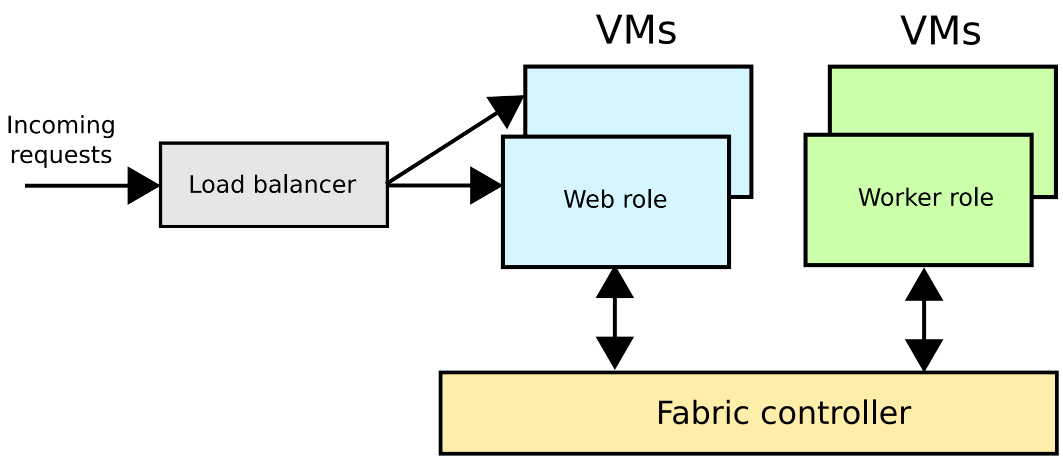


Figure 3.4: Web and worker roles in Azure Services Platform.

The Azure Services Platform is currently only available as a technical preview, and no pricing model has been released.

3.5 Summary

In this chapter, we have discussed key characteristics inherent in many cloud environments, some of which result in challenges for developers designing systems for the clouds, as well as place constraints on how such systems are designed and operated. The main challenges include increased latency, reduced bandwidth, loose coupling and unknown physical topology.

We have also provided a survey of some state-of-the-art IaaS and PaaS systems, describing infrastructure, development and runtime environments, control interface and provided services. In the following, we will focus primarily on EC2, currently being the most actively developed service.

In the following chapter, we attempt to characterize a range of enterprise IT systems, focusing on key properties relating to cloud computing, in an attempt to classify their cloud suitability. We will also take a look at some approaches to designing and porting systems to cloud environments, including a case study in the FAST ESP system.

Chapter 4

Moving systems to the cloud

4.1 Introduction

In this chapter, we will discuss the approaches to and challenges in moving existing systems to the cloud, focusing on common characteristics of enterprise systems. We will also investigate some approaches to moving existing systems to the cloud; from cloud-enabling systems with minor or no modifications, to completely re-architecting applications to fully utilize all services and paradigms provided by the cloud.

To further illustrate how such transitions can be realized, we will end the chapter by looking at a concrete case study, investigating how the different architectural components of an existing real-world system can map onto a cloud environment.

4.2 What is an enterprise system?

In this context, we define an *enterprise system* as a system designed to support key operations for an organization. These operations can range from major business processes, like sales and order processing, to administrative processes like customer service support, business reporting or accounting. Because of their nature, enterprise systems are often business critical, with the company or organization relying on the systems for daily operations.

We do not distinguish systems based on target users, so our definition includes systems with both external (i.e., customers) and internal (i.e., employees) users. Enterprise systems can be developed specifically for an organization, either in-house or with contractors, or licensed from a third party. Enterprise systems are typically distributed over a network, with users accessing the services from different locations. The systems are often designed with a data-centric approach, often developed around a central data repository such as a database.

Examples of enterprise systems include internal systems like enterprise resource planning (ERP), customer relationship management (CRM), human resource management systems, as well as external services like online shopping carts, search services and customer portals. Systems that do not fall into the enterprise system category include consumer applications installed locally on end users machines, and other systems in which the data relates only to a single user (such as online web mail services).

4.3 Enterprise system challenges and characteristics

Despite the wide range of enterprise systems, many share a set of characteristics. In the following sections, we will discuss these characteristics, limited to the ones that are relevant in the context of porting systems to the cloud.

4.3.1 Data centricity

A common characteristic of many enterprise systems is that they are data-centric, meaning that the primary functionality is based upon processing or manipulating a large data set shared between the users of the system. Data sets include general cases like customer databases, employee and order history and product catalogs, as well as other cases like search indices and photo/video collections. These data sets can grow large over time, and may itself be distributed to ensure sufficient storage space and replication.

These data sets are normally stored in databases, oftentimes using a relational database management system (RDBMS). A widespread pattern is accessing the database using Structured Query Language (SQL), either directly or indirectly using an Object Relational Mapper (ORM) tool.

This data centricity has some consequences. Firstly, it is often impractical to perform remote operations on the data. Secondly, it can be time consuming to transport large amounts of data across the Internet. This implies that the data should be stored within the cloud to allow the data to be accessed efficiently.

A possible approach to storing data in the cloud is moving an existing RDBMS setup into the cloud. This should be trivial to do, but provides few advantages compared to the existing setup in terms of performance and scalability. Another possibility is using a cloud storage service or distributed database provided in the cloud environment. This should provide scalability, at the cost of some control and functionality. However, since cloud storage and databases have been designed with scalability as the most important goal, they are often thin on functionality and features compared to a proper RDBMS. This means that developers must move some of the logic and functionality previously placed in the data layer (i.e., using SQL) to the application layer.

4.3.2 Static scalability

Many enterprise systems of today are used by hundreds or thousands of concurrent users, and are designed to handle heavy loads. However, most of them share a single trait - they must be dimensioned and scaled for handling the worst-case load, based on estimated or empirical traffic patterns. The pattern of scaling for the worst-case load means that both software and hardware is often tied in a specific configuration. The application software runs on a fixed pool of hardware, and the only factor varying with load is the degree of utilization. When systems need to be scaled, the traditional approach has been upgrading components in or replacing the entire server. For example, more RAM and a more powerful CPU can be installed, often enabling the application to process a higher amount of transactions.

Static scalability has some issues pertaining to cloud computing, and is an obstacle to fully utilizing the dynamic scalability in many cloud environments, since applications are designed for static environments. While it is still possible to move the current setups into a cloud infrastructure, effectively using the cloud environment as a virtual data center, does not provide the advantages of scalability inherent in the cloud computing environment.

In order to fully utilize fully dynamic scalability, many current systems must be modified to handle a dynamic hardware and resource pool, in which servers and resources appear and disappear as required. In many cases, especially for older systems, this implies rewriting parts of the software, whereas newer systems might be able to address this at a middleware layer.

4.3.3 Tight coupling

Another challenge, closely related to static scalability, is the notion of tight coupling between components in an enterprise system. Tight coupling is possible at different levels of the system, ranging from hardware resources to dependencies on external systems.

At the hardware level, static relationships between servers such as application and database servers, are examples of tight coupling potentially reducing scalability. DNS or similar naming schemes are commonly used to allow servers to discover and address each other in a network. While load balancing is possible with these schemes (e.g., using mechanisms such as round-robin DNS), the more common case is locating individual servers to perform certain tasks, like processing a payment or downloading a file.

Another challenge is systems directly dependant on other systems. Consider an internal customer support application where operators communicate with customers, performing tasks like issuing refunds. If the support systems is closely tied to the invoicing and payment processing system for processing refunds, moving the support system to a cloud environment might be complicated. Inter-dependencies like this are common in large systems, and lead to additional challenges, such as how systems moved to the cloud can access internal systems in a secure and reliable manner. A possible solution is setting up network links between the cloud environment and internal network using technology such a VPN, but this might have security and reliability implications in addition to the complexity of distributing inter-dependent systems.

One of the core philosophies of SOA is that individual components should be loosely coupled, communicating by passing messages through an ESB. In the case of SOA systems, it should be possible to extend ESBs to make them able to transparently route messages between an organization-internal network and a cloud environment. This would allow SOA applications to take advantage of scalability of cloud computing, as well as transparent partial migration of individual SOA components.

Applications designed to be loosely coupled should be able to easily move into a cloud environment, in contrast to tightly coupled applications. To resolve tight coupling, some systems may need to be partially redesigned to handle dynamic resources and avoid network dependencies. Figure 4.1 illustrates the difference between tight coupling and loose coupling using a shared message bus or queue, with boxes representing servers. By indirecting communication through some mechanisms, such as a queue or message bus, systems can scale by adding or removing nodes with demand. For SOA applications, cloud transition may be facilitated by the flexibility designed into the notion of communication using an Enterprise Service Bus (ESB).

4.3.4 Latency requirements

Highly interactive systems, in which users are constantly providing input and waiting for results, form the basis for many tasks in different organization. These systems include customer support services, enterprise search and business intelligence systems,

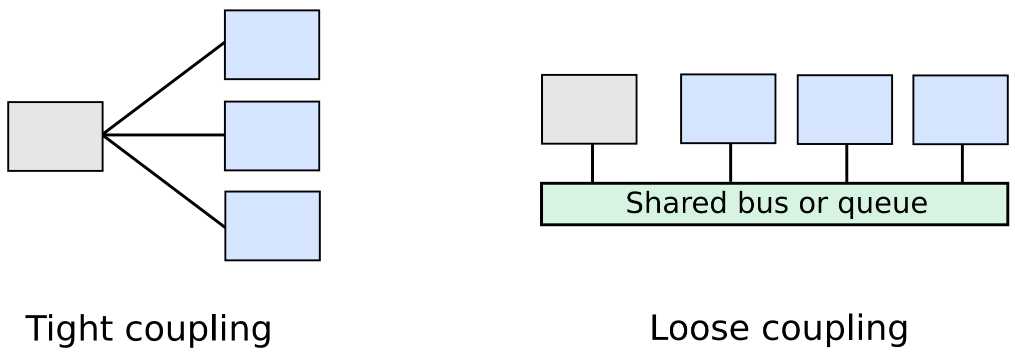


Figure 4.1: The difference of tight and loose coupling.

for which keeping response times low is critical to enable employees to work efficiently.

Many factors combine to form the total response time for services, which will be further discussed in Chapter 5. One source of increased latency is unavoidable when moving to a cloud environment, namely the increased round-trip time of network traffic. Compared to organization-internal networks, packets will need to travel farther to reach the cloud environment and back, with latency depending on the geographical distance to the cloud provider.

Within organizational LANs, RTTs of under 10 milliseconds are normal. When the LAN is replaced with a WAN required to reach the cloud provider, however, RTTs might increase an order of magnitude. For some systems, especially when multiple requests and responses and required, this added RTT lead to a slower-responding system. The RTT is even more important if the cloud-based system is depending on systems in other clouds or networks, in turn leading to even more round trips.

In other cases, the added response time may not be significant due to geographical closeness or other factors, or may actually be reduced on average if the cloud environment means that the system is able to scale better under load. Still, the factor of Internet latency must be taken into account when considering running systems in the cloud.

4.3.5 Sensitive information

Many computer systems manage sensitive information, including medical information, employee information or business secrets like operations details and product development. Even systems that are generally not sensitive, such as enterprise search engines or intranets, might contain some information sensitive to the organization.

When such systems are considered to be moved to a cloud, effort must be made to ensure the information is kept safe, in transport, processing and storage. Additionally, national laws and regulations may impose specific requirements on how the data is to be processed and stored. When outsourcing IT infrastructure to a third party cloud provider, it is more difficult to document that laws and regulations are followed if an audit should occur, or even which laws should apply. Legal issues and possible solutions fall outside the scope of this thesis, but are mentioned for completeness and to underline their importance.

4.3.6 SLA requirements

Most organizations rely on at least one computer system for their business operation. These systems are vital for operations, and must be operating properly to allow revenue to be generated. These systems have strict uptime requirements, often documented in Service Level Agreements internally with an IT department or with an external provider.

When critical systems are moved to a cloud environment, care must be taken to ensure that the SLA guaranteed by the cloud provider is sufficient. Currently, most SLAs provided by cloud computing providers are limited, generally restricted to service credits as compensation if the service is not delivered at the required level.

Still, a cloud provider is more likely to have equipment and expertise to provide a more reliable service environment than most organizations would be cost-effectively able to do themselves. Still, the lack of control of the service environment means that some organization are skeptical to outsourcing their systems.

4.3.7 Legacy technology

Legacy technology refers to software and technology that has been replaced by newer generations, but continues to be used. Legacy technology is often unsupported by vendors and out of development. Many complex enterprise systems, called legacy information systems [70], are based on technology developed years or decades ago, due to the complexity and cost involved in replacing it. This is especially common in industries that were early adopters of IT systems, such as financial or airline industries. Organizations in these sectors still rely in systems written in languages such as Fortran and COBOL, running on outdated OSes like old versions of UNIX or Windows NT.

Moving these systems to a cloud environment would likely be expensive due to the large number of required changes. Also, the fact that these systems still run on legacy software and technology might indicate that the organizations prefer reliability and stability at the cost of cost-efficiency and dynamic scalability.

4.4 Cloud suitability

When considering what kinds of applications are suitable for cloud computing, there are two primary factors that must be taken into account. First, organizations must consider the potential opportunities a cloud computing platform can deliver. Secondly, they must consider whether any of the challenges outweigh the advantages, making the transition too extensive, expensive or altogether impossible. This trade-off between opportunities and challenges associated with hosting new or existing systems in a cloud environment is illustrated in Figure 4.2.

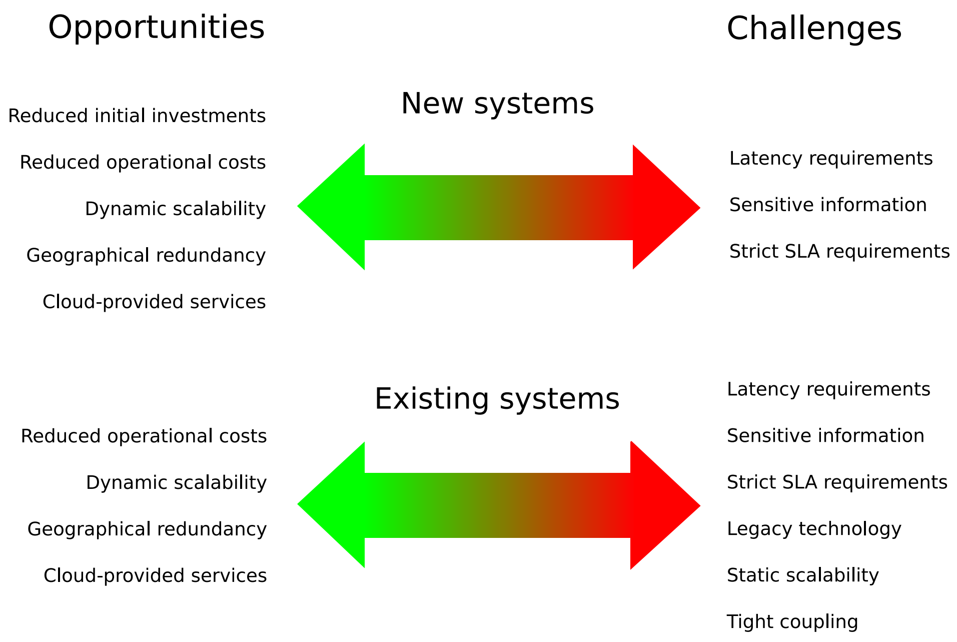


Figure 4.2: Trade-offs between opportunities and challenges in cloud-basing systems.

For most systems, the primary advantages of cloud computing is the potential for dynamic scalability, in addition to reduced costs by eliminating initial hardware investments, and lowering operating costs when outsourcing infrastructure. When developing new systems, both of these factors might weigh equally as they both potentially reduce the cost of running the system. For existing systems, the hardware investment is typically already done, so the largest motivation is the potential for increased scalability and reduced operational costs.

Cloud computing providers also enable systems to take advantage of geographical redundancy through data centers located in different locations, without organizations having to invest and acquire this infrastructure specifically. An additional advantage is being able to utilize the scalable services provided in the cloud environments to ease development of scalable software.

The need for, and benefits of, dynamic scalability might differ between systems, in turn affecting the motivation for moving it to a cloud environment. If the system has a relatively static load with only minor fluctuations, and this is not expected to change, the motivation for scalability is small. Similarly, if an organization already has the infrastructure in place, such as a virtualized data center in which hardware is sufficiently utilized, the motivation for outsourcing operations is limited.

Applications with absolute characteristics that correspond to challenges in Figure 4.2 are likely currently not suitable for cloud environments. This class of software might for instance include video streaming services requiring predictable CPU scheduling and high bandwidth, medical management software dealing with sensitive medical information, and business-critical software over which an organization needs full control.

Some systems have some potential challenges, but the opportunities might outweigh the negative aspects. Systems falling into this category can include distributed backup systems for which the geographical redundancy and cost-effective cloud services for storage outweigh the challenge of transporting large amounts of data, since transfer time is not critical. We argue that FAST ESP falls into this category, since the opportunity for dynamic scalability might outweigh the challenge of transporting data for indexing.

In general, developers will need consider and balance the individual opportunities and challenges against each other. We argue that there is no single metric for cloud computing suitability, as the importance of individual factors will differ between organizations and systems.

4.5 Approaches to cloud-enabling systems

When discussing the motivation and challenges in cloud-basing systems, it is also relevant to take a quick glance at the different approaches to cloud-enabling systems.

4.5.1 Moving systems unchanged

At one end of the scale is moving a system to a cloud environment without any changes to the code. This approach requires little work beyond installation and setup, and merely uses the cloud environment as a virtual data center. IaaS providers, like GoGrid and EC2, are the natural choice for this setup, and servers and hardware are provisioned once and left running after initial configuration.

With little effort, the virtual data center approach has the potential of delivering some benefits in the form of outsourcing data center operation to a cloud environment and avoiding initial hardware investments. This approach can be useful in some cases, such as for software testing, in which an organization can setup a complete test setup in a cloud environment and test how a system behaves in a distributed fashion, without investing in dedicated test hardware. Similarly, some users may want to eliminate the need for local server hardware, or have a cloud-based backup system in place if something happens to a primary setup. While this approach potentially provides some benefits with little effort, it does not allow users to fully take advantage of the dynamic nature of cloud computing.

4.5.2 Moving systems with some changes

Depending on system specifics, some changes in how components and subsystems interact may allow it to benefit from dynamic scaling. By loosening the coupling between components and implementing support for adding and removing servers dynamically, the systems can potentially benefit from optimizing hardware utilization with load. Scaling can be implemented either manually, with system operators selecting the number of server instances for different application roles, or fully automatic with some software component monitoring the load and traffic of different components, adjusting resources as needed.

This hybrid solution can potentially deliver several benefits at the cost of some development effort, depending on the specific implementation. In some cases, the flexibility may be implemented at a framework or middleware level, whereas other cases may require application logic to be rewritten. Potential benefits include dynamic scaling, improved fault tolerance and reduced operational costs. In this approach, both IaaS and PaaS providers may be appropriate, depending on the specific system and its requirements for flexibility and control.

Another example of a hybrid solution is the emerging concept of *cloud bursting*, in which cloud environments are used in addition to regular data center resources when a service experiences heavy load. A shopping site experiencing high load after a marketing campaign might have a setup in place that is started when load reaches a critical point, after which a load balancer redirects a share of incoming requests to the cloud service instead of the regular service, in order to handle the extra traffic. When normal traffic levels resume, the cloud service is reset into standby mode. This setup might also be attractive as a backup service to keep systems running should local hardware fail.

4.5.3 Fully enabling systems for a cloud environment

When designing systems from the ground up, it is possible to fully utilize the characteristics and advantages of cloud computing by developing the system specifically for a cloud environment. When the system is designed in a loosely coupled and distributed manner, scaling can easily be implemented dynamically. Also, developing software specifically for cloud environments allows developers to take advantage of the services provided in the cloud, such as scalable cloud storage and distributed message queues. Using these mechanisms can speed up development time and ease the development of scalable applications. Developing specifically for a cloud environment also makes it possible to use PaaS environments, designed specifically to make development easier.

While the development effort is reduced by utilizing the provided cloud services and tailoring applications to specific environments helps to reduce development overhead, the price paid is generally in terms of vendor lock-in. Since cloud environments currently are unstandardized, significant changes might be required if the application is later moved to another cloud provider (or an in-house data center).

4.6 Case study: FAST ESP

FAST ESP [1] is an enterprise search platform deployed in over 4000 organizations across the world. As an enterprise search service, FAST ESP aims to provide an organization's internal and external users with access to a unified interface for locating and navigating in a large amounts of structured and unstructured data. The goals of the service include helping customers discover interesting products in an e-commerce solution, helping employees find relevant information in the organization intranet, or allowing users to lookup the right people in a directory service.

4.6.1 Architectural overview

A high-level architectural model of FAST ESP is illustrated in Figure 4.3, and will be discussed in the following sections. For brevity, we constrain the discussion to a high-level overview of relevant components, with significant simplifications compared to a realistic implementation. Further information about the implementation and optimization of these components can be found in [71], along with a thorough description of maintenance and usage of the search index.

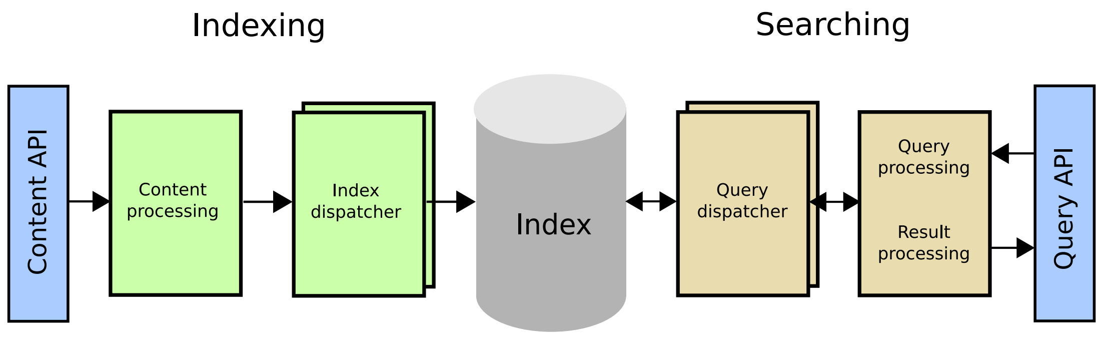


Figure 4.3: FAST ESP architecture

Index

At the core of FAST ESP, like any search service, is the index. The index contains a list of all terms present in the indexed documents, mapping each term to its occurrences within a set of indexed documents. To keep the index as small as possible, an additional data structure is used to map each term to a unique term ID, allowing the index to deal with smaller-sized term IDs instead of the more verbose terms directly.

The processing required to maintain the index and perform queries is done in a number of machines containing the index, called *search nodes*. To distribute work, search nodes are organized in a matrix of rows and columns, illustrated in Figure 4.4. When indexes are expected to become large, the index can be partitioned across multiple columns, with all servers in a column all storing a part of the complete index. To handle a high query rate or provide fault-tolerance, searches can be balanced across multiple rows.

Collectively, all servers in a row contain a complete replica of the index, enabling it to operate independently from other rows. The query dispatcher is designed to balance queries by directing them to one or more rows of search nodes using a round-robin (or similar) scheduling algorithm.

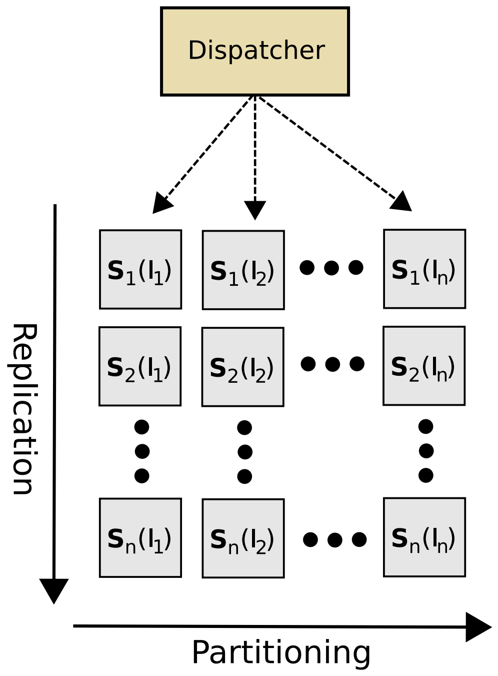


Figure 4.4: FAST ESP scalability model. Columns of search nodes partition the index, and rows provide replication for fault tolerance and scalability.

Content refinement (indexing)

Before content can be included in search results, it must be indexed. In FAST ESP, this process is called *content refinement* or *indexing*, and is accessed through a *Content API* interface. Content can be pushed or pulled into the system, supporting a range of audio, video, image and document formats.

In the *Document processor*, incoming content is fed through a document processing subsystem, performing a variety of tasks depending on the content format. Document processing includes document conversion, classification, language detection and summary generation. Output from the document processor consist of a representation of the document and some additional meta-data, and is subsequently fed into an *index* *node* via an *index dispatcher*. The documents is converted to a format ready for inclusion in the search index, consisting of term mappings along with relevant meta data like modification dates and access permissions.

Content and result analysis (searching)

Search queries are submitted through a *Query API*, arriving from users at desktop PCs or mobile terminals, or programmatically from other systems. Incoming queries are first sent into *query processing*, in which operations like natural language query mining, spell-checking and language detection is performed. The resulting query is subsequently converted into a set of term IDs using the mapping dictionary maintained by the indexing process, which in turn allows an efficient lookup to be performed on the index by a *query dispatcher*. The query dispatcher directs the query to one or more search nodes that perform the actual query.

With results returned from the index, the content is passed to *result processing*, performing operations like formatting summaries, enforcing access controls and enhancing and transforming the result set, before it is returned to the user.

4.6.2 Deployment

FAST ESP is deployed in a wide range of organizations, covering both small companies with a few employees and international corporations with thousands of users. This means that the platform is designed to be scalable, handling both small and large amounts of traffic.

Currently, the FAST ESP platform is deployed on servers located in the data centers of customer organizations. This means that customers require local or leased expertise for installation and configuration, as well as personell able to solve technical issues, handle hardware upgrades and replace failed components.

The amount of hardware required to support the FAST ESP platform depends on several factors, such as amount of documents, index size, query rate, freshness requirements and processing complexity. For small customers, a single server running all components of the platform may suffice. For larger customers, the system is designed to scale gracefully across multiple servers with demand, with the largest customers having up to hundreds of servers dedicated to the FAST ESP platform.

In a typical setup for medium to large customers, one or a small number of servers are dedicated content and search gateways. The content gateway server(s) provide the Content API, along with document processing and indexing functionality. Similarly, the search gateway server(s) host the Query API, performing query/result processing and query dispatching.

4.6.3 Motivations for FAST ESP in a cloud environment

For small customers, hosting FAST ESP in a cloud environment might allow customers to take advantage of the service while eliminating the need for any local servers for query processing, dispatching and index lookup. A reduction in initial investment costs and continuous operating costs, may lead more customers to consider FAST ESP, as the entry barrier is reduced.

A reduced need for local servers is also likely to be attractive for larger customers, which currently must acquire and support up to hundreds of servers to fulfill their search needs. Outsourcing the infrastructure services by accessing the service in a cloud environment can lead to cost savings both in terms of required data center investments and running costs, as well as in a reduced need for technical personell to ensure proper operation of the service.

Another motivation for cloud computing lies in the inherent temporal traffic variability found in such services. Since FAST ESP is currently statically provisioned with a fixed hardware pool available, the system must be configured for the highest load it is required to handle under normal circumstances. This implies two challenges; improving hardware utilization (i.e., under-utilization), and handling extra-ordinary load (i.e., over-utilization).

When FAST ESP is used internally for employees in an organization, the system must be ready to handle most of its daily amount of requests within office hours. While users expect the system to be available at all times, the system load is likely marginal outside business hours compared to within the same time period. However, with a static hardware pool, the amount of hardware resources utilized remains constant, independent of the load. For many other uses of FAST ESP, we expect this pattern of temporal load variance to hold true, although the specific traffic patterns will differ. For example, when used as a search service in a consumer shopping Web site, most traffic might occur after business hours and in the weekends, with low-utilization periods within office hours and at night.

By adapting the service to run in a cloud environment, hardware resources can be dynamically adjusted based factors like predicted traffic based on time of day, continuous traffic measurements or observed queue lengths. Dynamic scaling might also help to handle extra-ordinary amounts of traffic, useful for applications like a shopping application experiencing the effect of *Black Friday*[1]. In this case, not being able to scale as needed due to a static and insufficiently provisioned hardware pool is likely to negatively affect sales, resulting in lost revenue.

4.6.4 Cloud architecture

Moving a system like FAST ESP to a cloud environment can be done in many ways, and developers implementing such a cloud-based systems face a range of alternative approaches. In the following section, we will describe one such possible implementation, for which the primary motivation is to keep the architecture simple and similar to the existing system.

A possible architectural overview of this simple cloud-based version of the FAST ESP is shown in Figure 4.5. The lower part of the figure illustrate users in an organization, residing in one or more physical locations. Where FAST ESP would typically be deployed within an organizational LAN, the primary components of the system have been moved to an external cloud. Users perform searches using the Search API interface located at an externally available endpoint in the cloud environment. Within the cloud, the architecture is largely kept unchanged compared with the non-cloud-based system, except for the internal operation of some components, which will be discussed in the following sections.

Indexing agent

In our suggested approach, the only component present in the organizational LAN is a component we have called a *local indexing agent*. The local indexing agent is responsible for indexing available content, locating and transporting data for indexing in the cloud, preferably using an efficient and secure transport channel. To ensure only new or updated content is indexed, the local indexing agent keeps a local history of meta data for indexed content. The local indexing agent uses the Content API, available at an externally reachable endpoint in the cloud, to push content into the indexing component of the system.

Index storage and lookup

In the current system, most the work of indexing and searching is performed in the search nodes, each holding a full or partial copy of the index. When moving FAST ESP

[1]One of the busyiest retail shopping days of the year in the USA [72]

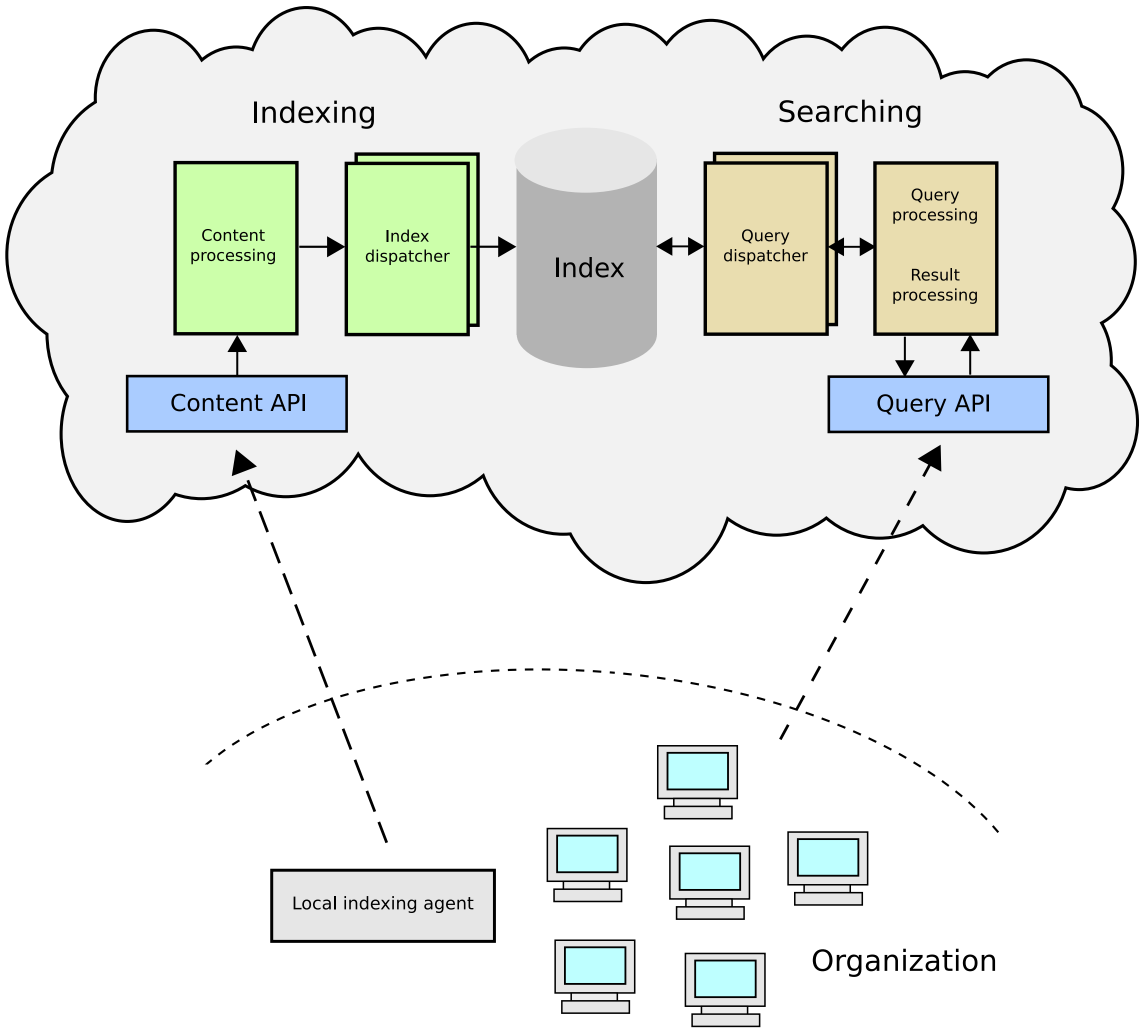


Figure 4.5: FAST ESP cloud architecture

to the cloud, it is certainly possible to retain this architecture, but investigating a more cloud-like approach might be interesting.

Specifically, we are curious as to the feasibility of using cloud-provided services like distributed storage or database services to hold the index. Using a cloud storage service, we would be able to query the index for specific index blocks, that in turn would be processed by a search component to return query results. With a cloud database service, the query might be expressed directly in a query language supported by the database, offloading much of the work to the service. The advantages of both these approaches is that they offload complexity to external services, taking advantage of built-in replication and scalability.

The challenge is that these services are likely not designed or optimized for this use. While the storage service might provide more than adequate storage and scalability, its performance might be insufficient due to high access latencies, probably at least an order of magnitude slower than reading directly from a hard disk. The performance argument also applies to a cloud-based database service, since it is often based on top of the same architecture as the distributed storage service. In addition, the distributed database service may not provide a query language flexible enough to handle all queries efficiently.

A natural trade off may include storing the primary index in a cloud storage service, while keeping complete or partial index caches in search nodes. This way, search nodes can easily be added and removed as necessary, fetching their copy of the index from cloud storage upon startup. A mechanism must also be employed to ensure that search node caches are kept up to date.

Scalability

Our primary target for scalability is the search nodes, which must be able to scale up and down as needed. By storing the index in cloud storage, we should easily be able to implement dynamic scaling of the search nodes. The query dispatcher is able to easily track the query rate, issuing commands to the cloud service to start and stop search nodes on demand. Nodes starting up proceed to fetch the complete index, or a partition as instructed by the query dispatcher, from the storage and into a local cache (in memory, disk or a combination).

In many distributed systems, scalability is complex because state needs to be kept in processing nodes. In the case of our search nodes, only read operations are performed on data, so no state needs to be kept. If a node crashes or disappears, the query dispatcher can simply resubmit the query to another search node. Note that this does not apply to index nodes, in which indexing state must be managed.

When experiencing high load, query/result and content processors can be dynamically scaled by instantiating the required number of instances to process the load, configuring load balancing between them. This also applies to the indexing and query dispatcher nodes.

4.6.5 Challenges

While the implementation above appears relatively simple, several potential challenges must to be considered before the approach can be deemed feasible. In this section, we will describe some of them, leading up to a more detailed investigation in Chapter 5.

Large data amounts

The nature of search engines implies that they deal with large datasets, often with indexes in the gigabyte, terabyte and potentially petabyte ranges. Datasets of this magnitude are impractical to transport over the Internet on demand due to bandwidth limitations.

The challenge is primarily relevant with regard to indexing, in which all content must be read and processed to build a search index. The classical approach to indexing involves feeding the document contents into an indexer component, which extracts terms and performs processing before outputting a set of updates to insert into the index. If FAST ESP document processing and indexing were to be fully placed in a cloud environment, for which content must be transported across the Internet to reach, indexing would be significantly slower than the current process, in which content is only transported across a LAN.

To alleviate some of the challenges with reduced bandwidth, we could optimize the local indexing agent to optimize and reduce the amount of required network traffic. For example, to ensure only new and updated content is indexed, the indexing agent can keep some local history of meta-data describing indexed content. This approach, similar to the approaches taken by cloud-based backup software like Dropbox [46] and Jungledisk [73], should lower bandwidth requirements significantly. Additionally, the indexing agent should be designed to transfer the content compressed, possibly with an adaptive compression scheme ensuring optimal compression for different kinds of content.

High latency

Compared with a search service within a LAN environment, a cloud environment entails higher network latency due to the larger network distance packets must travel. With the advent of Internet search engines, users have come to expect instant results for even complex queries over vast amounts of data.

Increased latency is relevant in two ways. First, if affects the time required to sent queries to the service, and for results to return. Secondly, using cloud services for core mechanisms in the architecture means that I/O is performed over a network environment instead against a local disks. This is likely to adversely affect the total response time, since the service will use more time to wait for data to arrive.

4.7 Summary

Enterprise systems are systems designed to support key operations in an organization, and are found in most medium and large organizations. These systems often share a set of key characteristics, such as being data-centric, statically scalable and relatively tightly coupled. When considering to transition such systems to a cloud environment, these characteristics might dictate certain changes to be made, and might even restrict cloud suitability.

Different approaches can be taken when designing or porting systems to the cloud, ranging from moving applications unchanged to fully designing the system for a cloud environment. These approaches require different amounts of effort, but in turn also offer a varying amount of dynamic scalability as well as utilization of other attractive cloud features.

To further investigate the motivation, challenges and characteristics in moving an existing enterprise system to a cloud environment, we have performed a case study of the widely deployed FAST ESP system, outlining how the current system might map onto a cloud context.

In the next chapter, we will implement a cloud search service based on an architectural model of FAST ESP, allowing us to perform experiments and discuss the feasibility of porting such a system to a cloud environment, focusing on response time, bandwidth and scalability.

Chapter 5

Cloud search performance

5.1 Introduction

In this chapter, we investigate three key challenges of running a cloud search service, based on an architectural model of FAST ESP, in a cloud environment. These challenges are increased latency, limited bandwidth and scalability. The goal of the investigation is to determine if a cloud-based search service is feasible, based on benchmarks and experiments related to these challenges.

We have selected these particular challenges because we consider them important in determining the overall feasibility of our approach, and because they are relevant also for many other systems. Other potential challenges, such as security, privacy and portability, fall outside the scope of this thesis, and remain as future work.

5.2 Cloud search service

Like any search service, the core task of our cloud search service is to accept queries from users (e.g., a phrase a user is interested in), and return a set of matching results. The queries are based on an index file, keeping track of the searchable content. To keep focused and avoid taking on too much at once, we have chosen to focus on the core search functionality, excluding all functionality related to indexing.

A primary motivation for implementing search services in a cloud environment is the temporal traffic variances inherent in such services. In enterprise environments in which FAST ESP is typically used, the volume of search requests can be orders of magnitude higher within office hours compared to outside hours. The opportunity for dynamic scalability in the cloud environment provides us with a way of scaling the resources consumed by the search service with demand, ensuring optimal utilization of the hardware. The cloud environment also allows the service to scale up in the event of especially high load, for example when used to provide product search in a consumer shopping service after a new series of products has been released. This scenario also motives us to implement the search service in a highly scalable fashion.

5.2.1 Metrics

We base our investigation of the cloud search service feasibility on a set of key concepts, which we define and describe in the this section. *Service time* is the time taken to process a single search request, including factors such as time required to set up a thread to process a request, parse parameters, perform an index lookup and process the results. In short, the service time is the elapsed time from a query is received by the search service to the result is ready.

However, service time does not fully reflect the actual time a single request takes from an end user perspective. To better describe perceived service time for end users, we introduce *response time*, defined as:

$$\text{Response time} = \text{service time} + \text{queuing time} + \text{transfer time}$$

In short, response time describes the time interval from a query is submitted by a user until a response arrives. In our search service, response time can be decomposed into the following components, illustrated in Figure 5.1:

1. Latency in sending request from client to the cloud service.
2. Queuing and processing time in the cloud service.
3. Latency in sending index lookup request to cloud storage.
4. Queuing and processing time in cloud storage.
5. Latency in sending index lookup response back to the search service.
6. Latency in sending results back to client.

Some of these factors are mainly functions of network locality and traffic congestion. For example, latencies 1 and 6 correspond to the RTT between the client and the service endpoint, while latencies 3 and 5 are functions of the RTT between the serving node in the cloud and the cloud storage endpoint, presumably located in the same network. We measure RTT using the Internet Control Message Protocol (ICMP) *ping* mechanism.

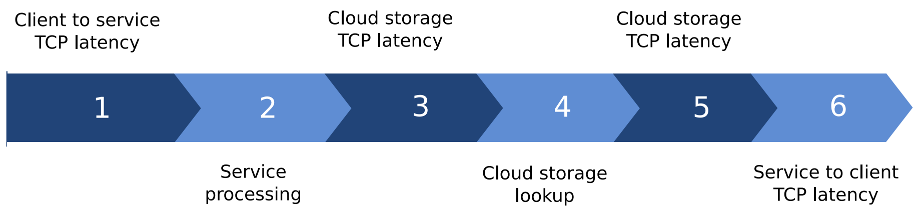

Client to service TCP latency
Cloud storage TCP latency
Cloud storage TCP latency
1
2
3
4
5
6
Service processing
Cloud storage lookup
Service to client TCP latency

Figure 5.1: Cloud search response time breakdown

5.2.2 Goals

To determine the feasibility of our cloud search service, we define four goals that the service should be able to fulfill:

1. Response time under 400 ms.
2. Acceptable search throughput per server
3. Linearly scalable index storage
4. Acceptable bandwidth between search service and cloud storage

The perceived response time dictates the user-friendliness of the service, and users have come to expect instant results to their search queries. This means that our cloud search service must be able to return results within a short time. Research indicates that response times up to 100 ms are unnoticeable for most users, with response times approaching one or two seconds being acceptable [74]. When the response time passes a couple of seconds, a number of users will find the service unacceptably slow to use. In this context, we have defined the highest acceptable response time as 400 ms, corresponding to the acceptance criteria in [71], discussing early benchmarks of FAST ESP performance. By requiring the response time to be under 400 ms, we allow sufficient time to enable browsers to process and render the results, including other elements such as graphics, within a time frame of one or two seconds.

To ensure cost-efficiency, another goal is that individual search servers should be able to handle an acceptable amount of concurrent queries, which we define as 100 queries per second. Also, the search service should support horizontal scalability, meaning that adding more search servers should linearly increase the aggregate capacity of the service. Since the index storage is a shared resource between search nodes, this implies that the index storage must be scalable. Finally, we require that the bandwidth between the search service and the index storage is sufficient to allow nodes to be instantiated in reasonable time, e.g., no more than 2-3 minutes.

5.2.3 Operation and simplifications

The high level architecture of our cloud search service is depicted in Figure 5.2. A number of clients submit queries to the service over HTTP, and results are returned to the client when the query has been processed. Clients are located in networks external to the cloud (e.g., in corporate networks).

The implementation of the cloud search service has been deliberately designed to be simple and incomplete compared with a fully developed search service, to ensure results are focused and easy to break down. In the following subsections, we elaborate on some of the simplications and assumptions we have made.

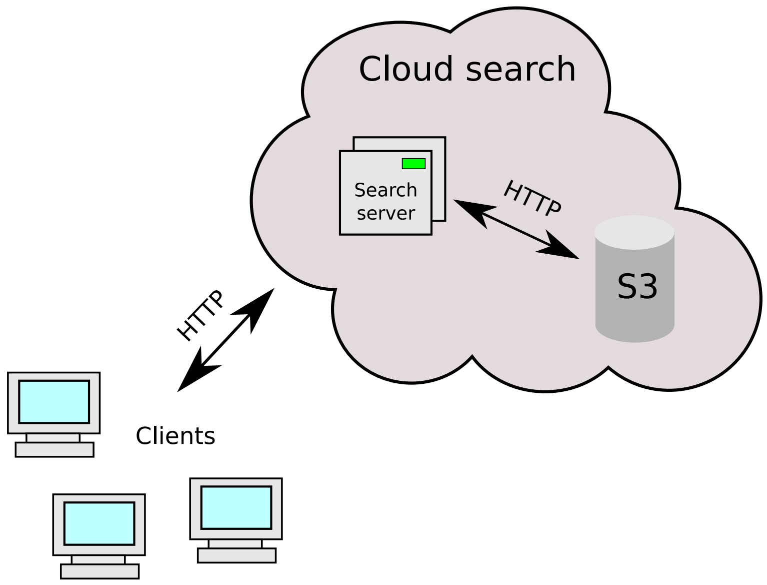


Figure 5.2: Cloud search architecture with index in cloud storage

Interface

The cloud search interface is designed to be minimal, making the service easy to use while incurring little overhead. In addition, the interface should be independent of the internal implementation of the service. We fulfill these criterias using standard HTTP GET requests, made to a Uniform Resource Identifier (URI) at an endpoint in the cloud. Search results are returned as a regular HTTP response.

Incoming requests are handled by one or more search servers, which are web servers able to parse and execute queries. Clients can be human users submitting queries from web browsers or other software services.

Index

With scalability as one of our goals, we must design the service so that the index is stored in a way that does not negatively impact scalability. In an effort to utilize most of the inherent scalability offered by the cloud environment, we choose to investigate how the cloud storage service is able to fulfill the role as storage for our index. Specifically, we will use the S3 cloud storage service as our index store.

Since our experiments concern only the core search functionality, we have left out functionality related to indexing. Instead, searches are performed against a 2GB dummy index file generated organized in a format designed to be similar to a realistic index. In practice, the index contains 2GB of randomly generated data, split into approximately 260 000 individual pieces of 8KB blocks. The 2GB total index size was chosen to ensure that the amount of data is too large to be practical to keep in cache during the experiments.

Query processing

When the cloud search service receives a search query, it first extracts the search criterion, embedded from the request. In our architectural model, the criterion is simply a pseudo-random number generated by the client, corresponding to an index block to look up. After the criterion has been extracted, the cloud search service proceeds to fetch the desired block from the index. When the desired index block has been read from the index, the cloud search service returns the block contents to the client, simulating a moderately sized query response.

For simplicity, other tasks present in normal search systems, such as extensive query parsing, optimization and results processing, are not present in our implementation, because the functionality is not relevant to our experiments.

I/O operations

In our example, we perform a single I/O read from the cloud storage for each query. This represents a simplification compared to a realistic implementation of a search service, where the number of I/O requests would be higher.

In a realistic scenario, the number of I/O requests per search would vary based on a number of factors, such as the number of terms in the query. In addition to performing a number of index lookups for each term, the search engine would typically perform I/O operations to return excerpts of the matching documents.

FAST uses the following formula to estimate the number of I/O operations required for a single search in their ESP platform [71]:

$$IO = Q\,[2TN + H]$$

In this formula, Q is the number of queries, T is the number of terms in the query, and N the number of parallel search nodes (columns). H represents the number of hits, and must be included if the search service should return context information (i.e., excerpts) for matching documents. The constant factor two indicates that each term in the query must be looked up twice in each search column; first to map the term to a term ID, and subsequently to find its occurrence in the index.

However given the response time and capacity of a search service with single-read queries, it should be trivial to estimate the response time and capacity of a search node with multiple I/O operations, given that the service scales predictably.

5.2.4 Runtime environment

In our experiments, all server machines have been run on Amazon EC2 in Europe (EC2 region *eu-west-1*). This region was selected because it is located in data centers in Ireland, which is the geographically closest of Amazon's current data centers at the time of writing. This region generally has the lowest network latency from Europe compared to the EC2 data centers located in North America.

All server instances are of type *m1.large*, which is the medium instance type offered by Amazon. This instance type corresponds to a 64-bit platform with 7.5GB memory, 850GB hard disk capacity and 4 EC2 Compute Units, equivalent to two virtual cores each with the CPU capacity of two 1.0-1.2 Ghz 2007 Opteron or 2007 Xeon processor. Although it is possible that the *m1.small* instance type, with 1.7 GB memory, 120 GB hard disk capacity and one EC2 Compute Unit worth of CPU capacity, would be enough for our experiments, we opted to instead use the higher class because it is given a greater share of shared resources like network I/O, resulting in higher capacity and less variance under load.

Client processes are executed in a non-virtualized environment on 64-bit versions of Linux, attached to a wired Ethernet network with direct Internet access. Both client and server components have been implemented in Java 1.5, using the *jets3t* library [75] on servers to access the S3 cloud storage service.

5.3 Response time and throughput

To investigate the first three of our goals, concering response time, throughput and scalability, we have performed an experiment with a number simulated clients accessing the search service concurrently. The test setup is shown in Figure 5.2, with search servers accessing S3 within the cloud, and clients running outside the cloud (in Akershus, Norway). The experiment has been run for a period of five minutes, and has been performed multiple times to confirm that the results are stable.

By performing the experiment with a single search server processing all incoming requests, we are able to investigate the response time we can achieve, as well as the number of concurrent requests a single server can handle. Since we expect that the cloud storage lookups will be the dominating operation, we will also attempt to isolate the performance of the cloud storage service.

To investigate scalability, we also perform the same experiment with dual search servers, balancing incoming request between them. Since the cloud storage service is a shared resource between the two servers, this will indicate if the cloud storage service is horizontally scalable.

When dual servers are used, we perform a logical load balancing by directing all clients on a single client machine to one of the servers. This is done to simplify the setup, as well as restrict the potential error sources. In a realistic scenario, the setup would likely instead involve at least one frontend servers (running software like the *HAProxy* TCP/HTTP load balancer [76], commonly used in EC2 setups) balancing incoming requests across search servers.

In addition to running the experiment with lookups performed on the index, we have performed an equivalent experiment with a static (but equally sized) response to form a baseline result. Comparing the normal results with the baseline allows us to isolate the response time resulting from the index cloud storage lookup.

Clients record the local system time when sending a search request, along with a corresponding timestamp as results are received. The difference between these timestamps correspond to the response time for the client, and includes all latency factors.

Concurrent clients are simulated with multi-threaded processes running on a set of machines. Each thread executes a single search requests, and blocks until a response is received. The number of threads in each process is varied to simulate different traffic loads. Initial experiments indicate that a single client machine can reliably simulate 80-100 parallel clients. Consequently, we have used six machines throughout our experiments to achieve the required load.

5.3.1 Observations

Response and service time

Figure 5.3 shows observed response and service time for both single server and dual server setups with an increasing number of concurrent clients. The response time starts out at approximately 200 ms for both single and dual server setups. By subtracting the round trip time (RTT) needed to transmit requests and corresponding results, we can calculate the service time, also included in Figure 5.3. We have used an RTT of 55 ms in our calculations, based on average ICMP ping results from the client network. The service time is initially approximately 80 ms, which we want to further decompose.

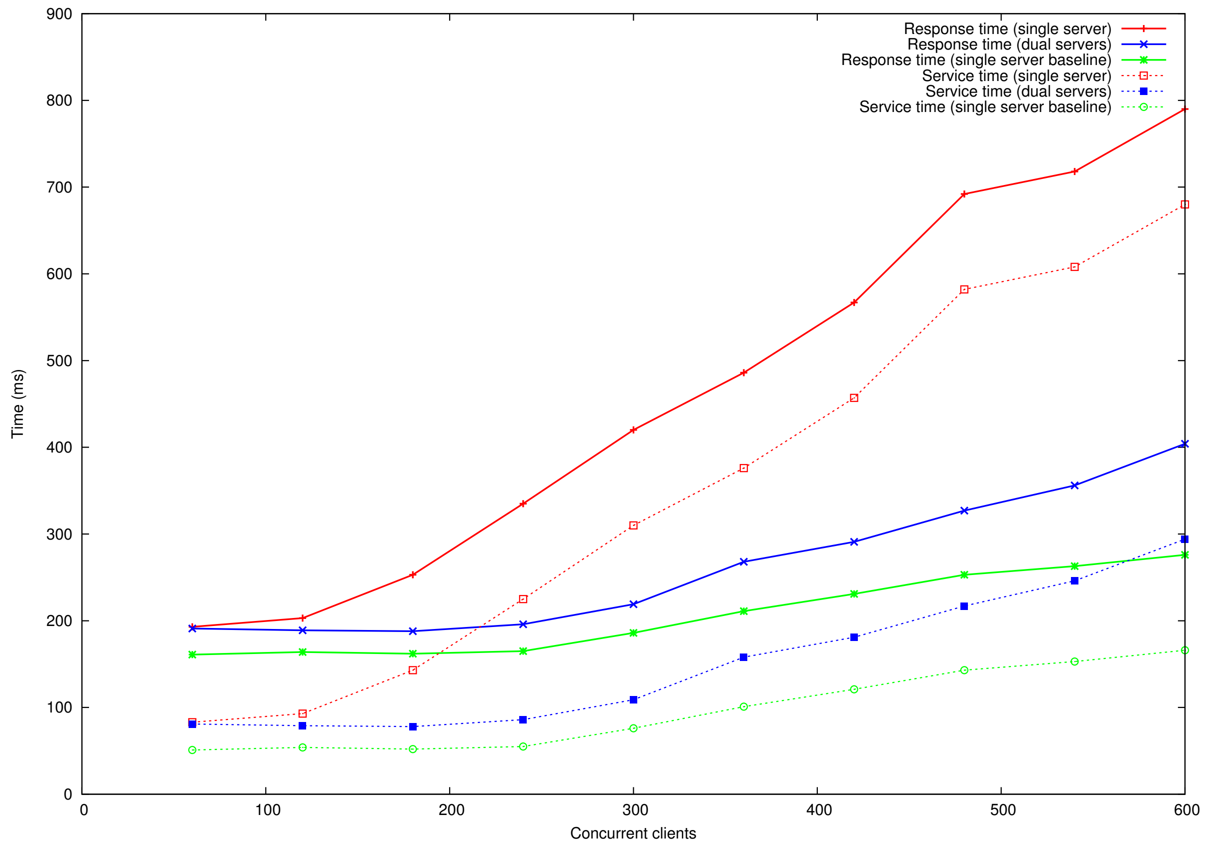


Figure 5.3: Cloud search response time

When a request arrives at the cloud search server, a thread is created to handle the request, after which the request is parsed and processed, including an index lookup. This means that we can decompose the service time into two distinct components; thread creation/request processing, and index lookup. Since our baseline experiment has the same creation and processing overhead, excluding the index lookup, we are able to isolate the overhead of thread creation/request processing from the index lookup. In Figure 5.3, this overhead is visible as the single server baseline service time, starting out at about 50 ms. This tells us that the time taken to perform a single index lookup is initially approximately 30 ms, confirmed by the initial storage response time in Figure 5.4, which shows the relationship between the total service time and the response time for performing lookups against cloud storage.

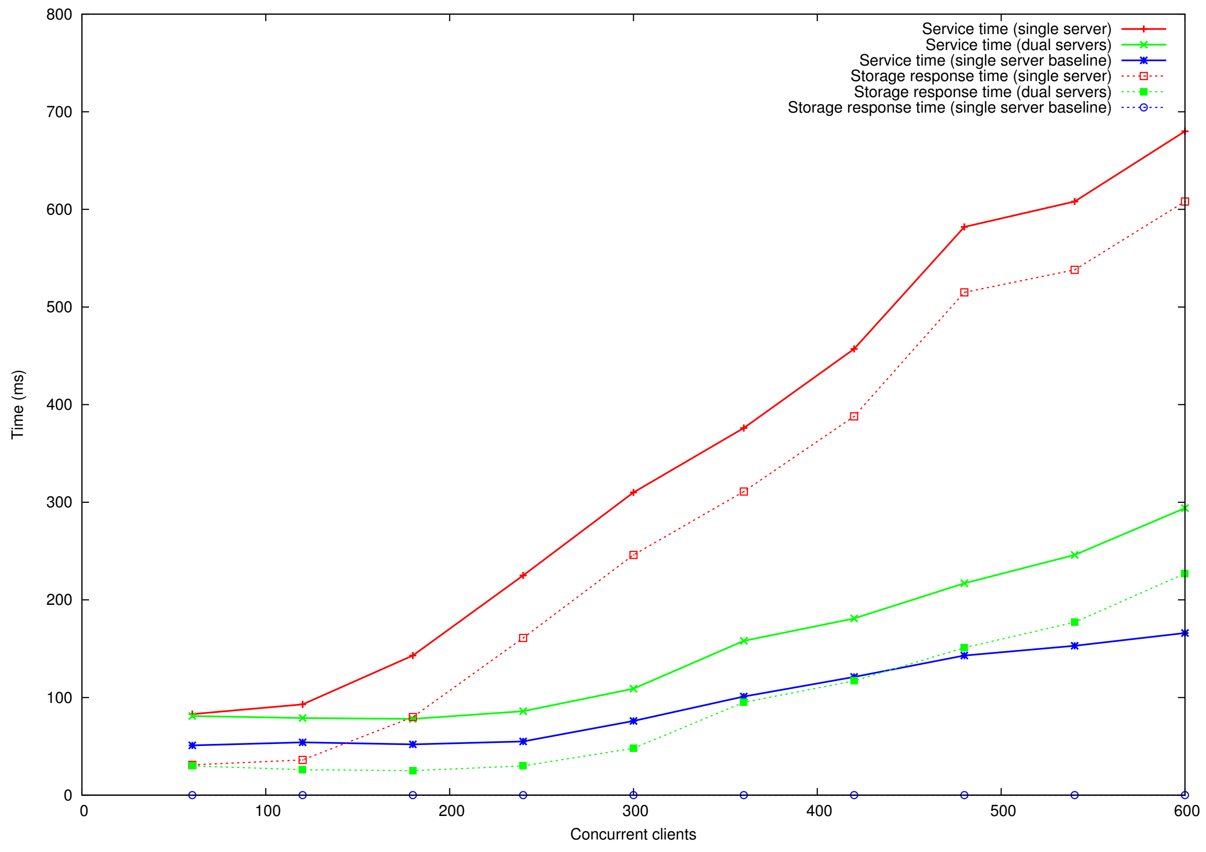


Figure 5.4: Cloud storage response time

With a single server, the response time is stable until about 125 concurrent clients, after which the response time starts increasing linearly with the number of concurrent clients. With the linear increase in response time, we reach our maximum acceptable response time using a single server at approximately 400 concurrent clients.

Throughput and capacity

In order to pinpoint the bottleneck limiting the number of concurrent clients within an acceptable response time, we will need to take a look at the throughput and capacity of our system.

Throughput is an expression of the rate at which a system performs work. In our case, we express the throughput as the number of search queries performed per second (QPS). For our search service, we calculate the average throughput T for $Q_{performed}$ queries performed over $T_{runtime}$ seconds using the following formula:

$$T = \frac{Q_{performed}}{T_{runtime}}$$

Closely related is the *capacity* of a system, which is an expression of the maximum throughput a system can handle without queuing requests. Workloads with rates less than the capacity can be handled directly without queueing. Workloads leading to rates higher than the capacity means that requests must be queued for delayed processing, since the system is unable to cope. When the system is combined from multiple operations, the bottleneck operation (i.e., the operation that takes the longest time to complete) defines the total capacity.

The observed throughput and capacity of our search system is shown in Figure 5.5. The solid lines indicate the average throughput with the workload varying with the number of concurrent clients. The dashed lines indicate the estimated capacity of the different setups based on the maximum throughput observed during all workloads.

When the throughput is equal to the capacity of a system, it is said to be *saturated*. When the system is saturated, the response time for individual requests increase as the requests are forced to queue longer before they are processed. This means that while the service time starts out as the dominating factor in the total response time, it is quickly surpassed by the queuing delay when the system is saturated.

Figure 5.3 clearly shows this pattern, with the initial increase in response time at approx. 125 concurrent clients corresponding to the point at which the throughput starts flattening in Figure 5.5. At about 175 concurrent clients, the single server throughput reaches its capacity, meaning that it is unable to perform more queries per second than the current workload. This is reflected in the response time, which starts to increase linearly with the increase in incoming requests.

When the rate of incoming requests is higher than the capacity, queues will in theory grow infinitely since the server is unable to cope with the incoming rate. This means that no matter how many clients we add, the rate of completion will not increase - only response times. In practice, however, incoming requests would soon start to drop since

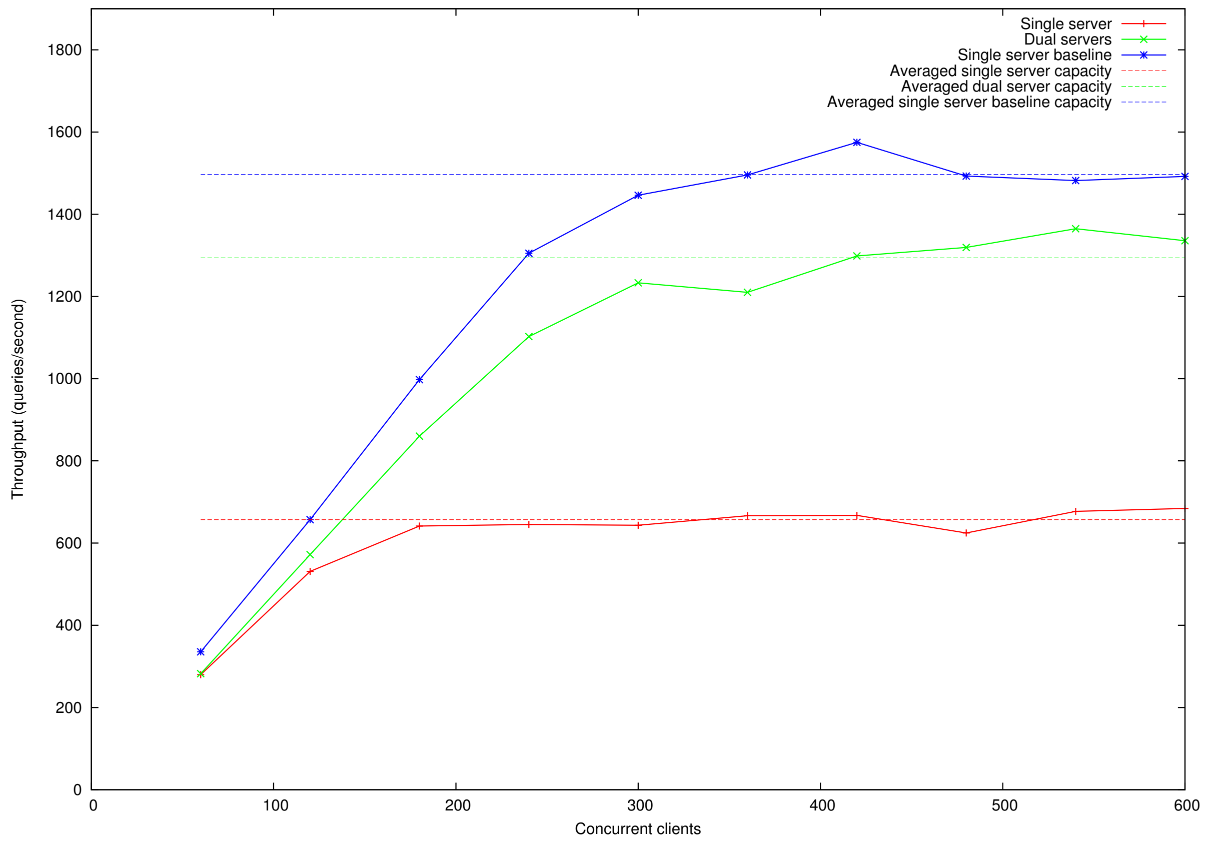


Figure 5.5: Cloud search throughput

the queues are not unbound in reality, and the resulting service would be unreliable for users.

Based on these observations, we conclude that a single server in our search service has a capacity of approximately 650 queries per second.

5.3.2 Potential bottlenecks

Because the slowest operation in a system, the bottleneck, limits its total capacity, we are eager to identify the bottleneck operation in our search service. In the following section, we will look at a some potential bottlenecks, discussing whether our observations indicate that they are in fact limiting system capacity.

Service limitations

Although the S3 documentation does not specifically detail any service limitations, it is highly likely that Amazon has implemented certain limitations on incoming requests to prevent abuse and attacks on the S3 service, such as Denial of Service (DoS) attacks in which large numbers of illegitimate requests are sent to a server with the intent of exhausting all available resources, rendering the server unable to provide legitimate service. For example, limitations are likely in place to restrict the maximum number of requests within a certain time interval for a single object or bucket, or received from an individual IP address or pair of account credentials.

However, we see no indications that the capacity of our search system is limited by any imposed service limitation. In fact, our figures show that our service was able to achieve a linearly increasing request rate when requesting objects from the same bucket using the same credentials at multiple servers. Since the S3 service is designed to be massively scalable, any imposed limitations are likely far higher than our current usage patterns, requiring a significantly larger amount of concurrent client resources to reach.

Insufficient replication

S3 is designed to transparently provide replication, and the replication degree is dynamically adjusted by the system based on the request patterns of specific buckets and objects. The degree of replication can currently not be influenced specifically by the user, and the replication mechanism is not documented. Consequently, the specific thresholds and delays for increasing replication are unknown. Replication is handled at the bucket level, meaning that all objects in the same bucket is equally replicated.

Despite the unknown internal replication mechanism of S3, object requests must clearly be distributed among some set of servers with access to a replica of the requested object. This load distribution is possible at the DNS level (changing the IP address corresponding to the object URI) as well as internally when fetching the object from disk. Currently it seems like a combination of the two is used.

Should a requested object be located in a bucket with too few replicas to process incoming requests at the current rate, requests must be queued for processing, resulting in higher service times. If this case of insufficient replication was a bottleneck in our benchmarks, the cloud search service would not have been able to scale almost linearly when adding more servers, since additional servers would be contending for the same replicas. Also, Figure 5.7 indicate that the service time of the cloud storage service remains relatively constant throughout our tests, indicating that queuing delay is not a factor in our tests. We conclude that there are no signs that indicate that the S3 service provide insufficient replication for our tests, although we have no way of establishing or adjusting the exact degree of replication.

VM I/O limitations

Running in a virtualized environment means that I/O resources are shared between a number of virtual machines sharing physical hardware. Since the cloud search service does not use hard disks directly, we are primarily concerned with the network bandwidth available to an instance. The EC2 documentation states that the service does not impose fixed limits on available bandwidth, except for guaranteeing a minimum share of the physical bandwidth to each VM sharing physical hardware. This means that an instance is often able to achieve a higher bandwidth than its strictly allotted share, at the cost of some variance.

To protect the internal network health, Amazon might additionally perform traffic shaping and filtering, for example to enforce a maximum rate of outgoing TCP packets from an instance. Such shaping might lead to situations in which a bottleneck is formed by limiting incoming or outgoing network traffic. Performing a large amount of cloud storage requests results in a large number of small HTTP requests, potentially affected by traffic shaping or network rate limiting.

To investigate if the amount of parallel HTTP connections could be a bottleneck, we ran our experiments replacing the cloud storage access with a HTTP request to a static document located on Amazon web servers. This service consistently operated with a capacity similar to the baseline service, confirming that traffic shaping/rate limiting is not the bottleneck in our service.

Server resource exhaustion

If the VMM is unable to provide sufficient resources like CPU and memory to the service, it will be unable to process more requests. We monitored CPU load and memory consumption during our experiments to ensure that we did not exhaust system resources, and we saw no signs that limited CPU and memory were affecting the capacity.

As long as clients have sufficient bandwidth and the server has resources to process the incoming rate of requests, the throughput of the baseline scenario shown in Figure 5.3 should scale linearly with load. We have executed the experiment with the client load spread over more machines to ensure that the limitations are not due to lack of bandwidth in the clients. This implies that the capacity we observe in the baseline scenario is limited by server resources, and represents the maximum throughput achievable on a single server, independently of cloud storage performance.

Little's law [77] is well-known in queuing theory, and describes the mean number of requests in a system at any time, L, as a function of the arrival rate of new requests λ and the expected time W spent to process a single request:

$$L = \lambda W$$

Little's law is often used in software benchmarking to ensure that observed results are not limited by bottlenecks in the test setup. In a multi-threaded scenario, it allows us to calculate the effective concurrency, i.e., the average number of threads actually running at any time.

We suspect that thread creation and management is the most resource intensive task in the otherwise minimal baseline scenario. We therefore want to investigate if thread creation and overhead in our server processes is in fact limiting the throughput. The results shown in Figure 5.5 indicate that a single server is able to handle a throughput of approximately 1500 queries per second in this baseline scenario (i.e., without accessing the cloud storage service). Using Little's law, we are able to calculate the average number of concurrent threads (units) active in the system based on the maximum throughput (arrival rate) and corresponding unit service time:

$$L = 1500 units/s \cdot 0.075 s/unit = 112.5 units$$

Using the same calculation with Little's law, we arrive at an equivalent mean number of concurrent threads per server of 105 for both the single and dual server scenarios, including cloud storage access. These numbers tell us that the actual achieved concurrency is almost equivalent in all scenarios, independently of the work performed in each case.

This leads us to conclude that thread creation and management overhead is in fact the current bottleneck in our cloud search service, and that the current system is unable to handle more than approximately 100-110 concurrent requests per server, limiting throughput to 650 QPS. To investigate this, we would have liked to implement a more efficient version of the client based on asynchronous I/O operations instead of threads, but time did not allow us to complete that in time for this thesis.

5.4 Index storage scalability

While the effects of individual cloud storage requests are interesting to discuss the feasibility of the implementation, we must also take into account that the cloud storage service is designed scalability in mind. To investigate how this affects our implementation, we have performed the experiment with dual servers.

With dual servers balancing incoming requests, the response and service time starts out equivalent to the single server setup, as is expected when the system is under light load. As the workload increase, we see in Figure 5.4 that the response time for the dual server setup increases at about half the rate of increase for the single server setup, indicating that the service scales linearly. While this might seem obvious at first glance, it has some implications in terms of the scalability of the cloud storage service, which in principle is a resource shared between the individual servers.

In addition to the reduced increase in response time with more concurrent clients, we see that the capacity of the dual server setup is approximately 1300 queries per second, roughly double the capacity of a single server. Based on our observations, the cloud storage service seems to be able to handle an increase in request rate in an almost linear fashion. This in turn implies that we can scale our cloud search service horizontally, i.e., by adding more servers, provided that the cloud storage service continues to scale linearly across servers. To ensure that this linear scalability persists with higher load, tests should be performed with a larger number of concurrent servers.

Storage service isolated

To isolate the performance of the storage service, we have performed an experiment with a setup similar to the running directly on EC2 servers. The test setup is illustrated in Figure 5.6. In this setup, a local process running on an instance within EC2 spawns a number of threads, each requesting random blocks from the index as rapidly as possible.

Our motivation for setup is two-fold. Frist, we want to independently confirm that the service time as reported above in Figure 5.4 is correct, by checking that it corresponds to the response time we are able to achieve in isolation. Additionally, we want to investigate achievable throughput when issuing several concurrent requests.

We have based the results on two different sources; response time as observed by the client process, and service time reported by cloud storage service log files. S3 offers request logging functionality, in which a log file of requests are written to a bucket of choice. These log files contain two relevant measurements, namely *total time* and *turn-*

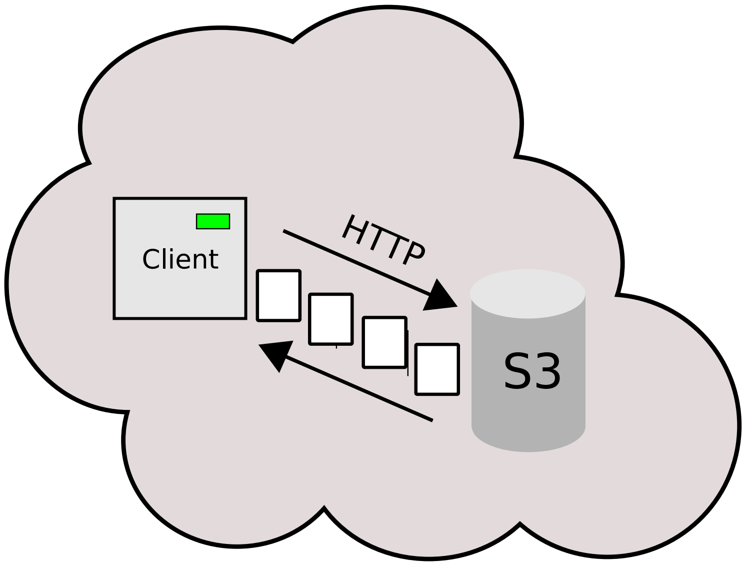


Figure 5.6: Test setup for storage service benchmark.

around time [78]. *Total time* indicates the number of milliseconds the packet is in-flight from a server perspective, and is measured from the request is received until the last byte of the response is sent by the service. *Turn-around time* represents the amount of time the service takes to process the request, including fetching the requested data, and is measured from the last byte of the request is received until the first byte of the response is sent.

Figure 5.7 shows the observed response time from the client perspective, together with the total time and turn-around time as reported by S3. Note that the S3 measurements overlap almost completely, likely caused by both requests and responses being relatively small in size (some hundred bytes for requests, and 8KB for responses). We attribute the increasing gap between client and server-reported time to latencies due to overhead when the number of threads increase.

The response time reported by the client starts out at approximately 25 ms, consequently increasing with the concurrency level to about 50 ms. Our initial assumption was that the increase in response time was caused by a similar increase in service time at the cloud storage level, but the measurements tell us otherwise. In fact, the service logs show the entirely opposite; the service time starts out at about 20 ms, and proceeds to stabilize at approximately 15 ms. While service time actually dropping with increased load might seem counter-intuitive, we attribute this to the fact that an increased request rate is likely to lead to a higher degree of cache hits within the service,

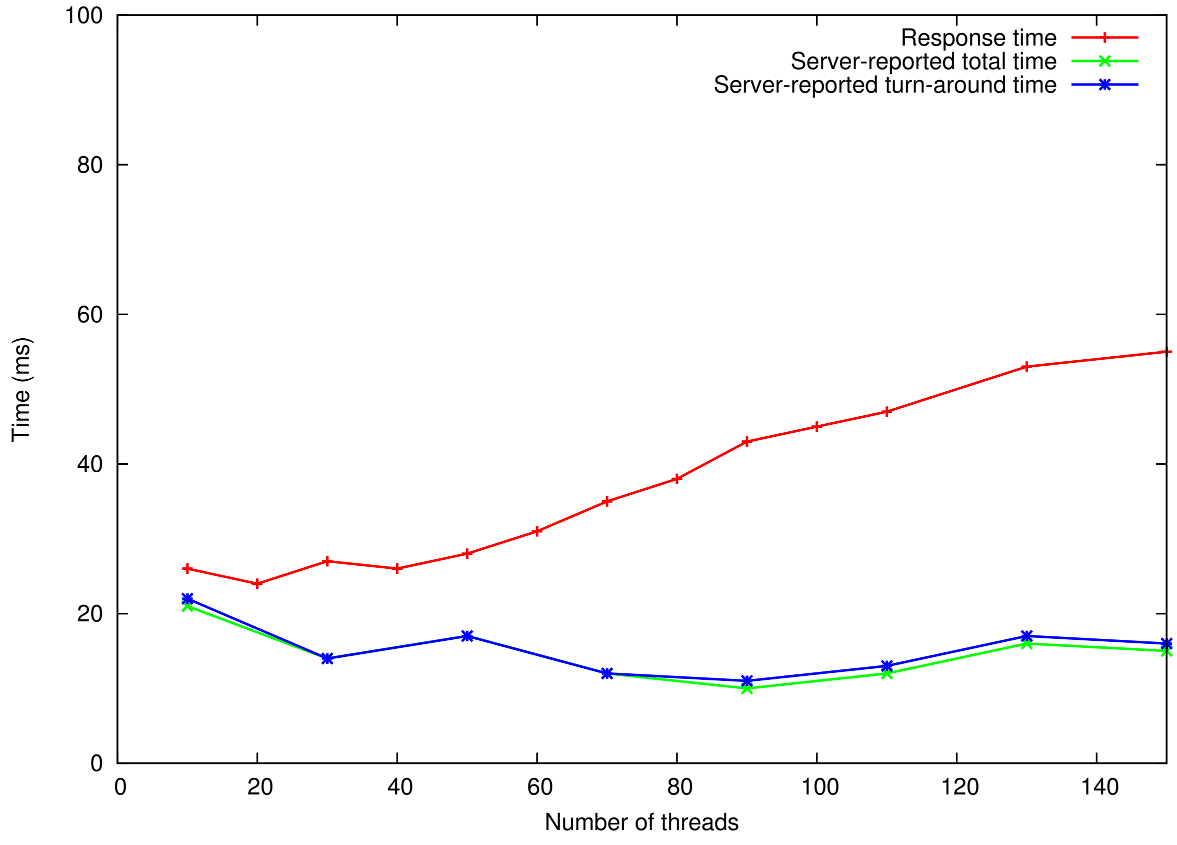


Figure 5.7: Storage service response time

because requests are made for blocks distributed randomly within the space of about 260 000 available index blocks.

Additionally, these observations confirm the observed performance of the cloud search service in Figure 5.4, in which the service time starts out at approximately 30 milliseconds.

Figure 5.8 shows the recorded throughput at different concurrency levels, in terms of transactions per second (TPS) and corresponding data rate. Plots a) and b) show the aggregate throughput of all running threads, while plots c) and d) show the throughput for a single thread (specifically the first running thread).

We see that a single thread is able to perform about 40 requests per second, yielding a data rate of a little under 300KB/s. In total the throughput is 400 requests per second at approximately 3MB/s. The throughput increases linearly up to 60 concurrent threads, after which it stabilizes at 2000 TPS and 15MB/s. While S3 documentation indicate

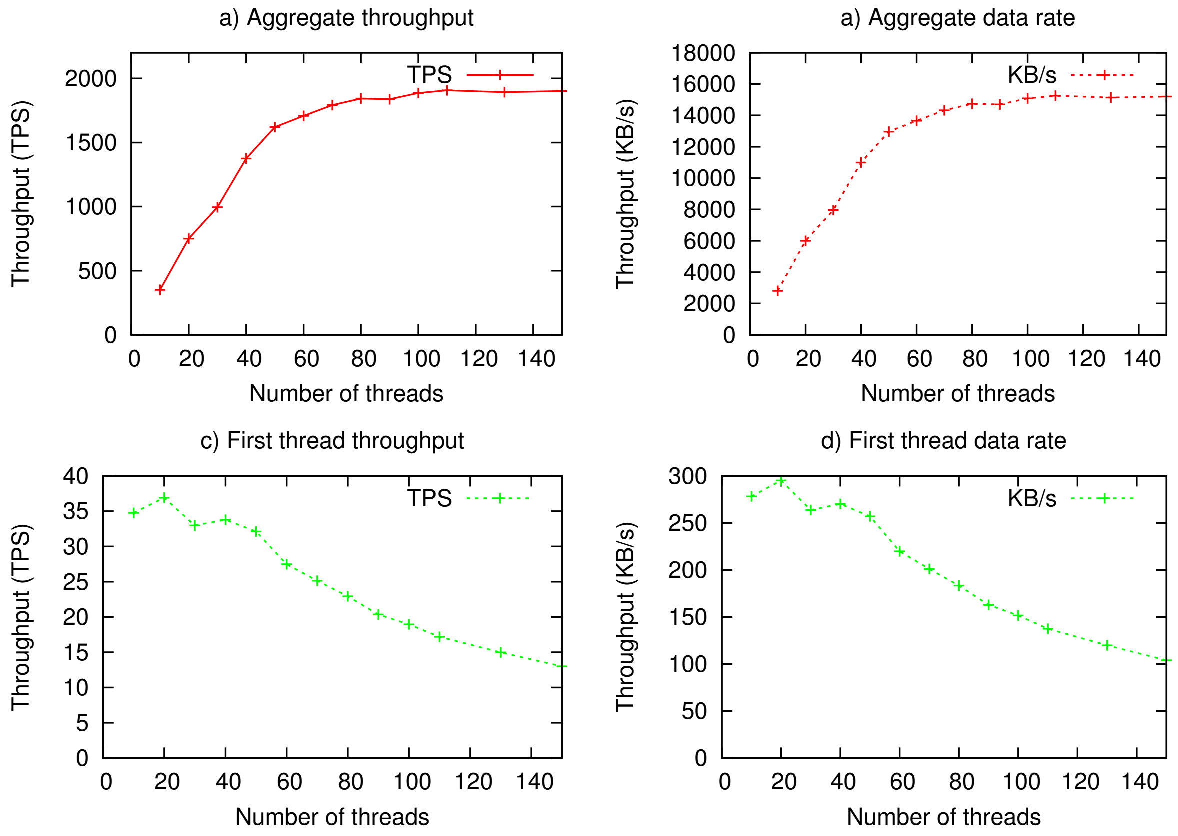


Figure 5.8: Storage service throughput in transactions per second (TPS) and corresponding data rate (KB/s)

that a higher bandwidth should be available between EC2 and S3, the throughput is likely limited by the large number of requests made for small objects, instead of fewer requests for larger objects; an assumption confirmed by other benchmarks [79].

5.5 Index storage bandwidth

While 15 MB/s should be representative for the throughput we can expect to see with only search operations, we also want to understand what kind of throughput we can achieve in other scenarios. Since the cloud search service is designed to scale dynamically, we are especially interested in available throughput for nodes initially fetching a copy of the index into their local caches upon startup. In this case, a larger amount of data is transferred in one piece, which should lead to higher throughput.

To measure the throughput of larger read operations, we repeatedly fetched 20MB and 100MB files from cloud storage to an EC2 instance. The results of our measurements are shown in Figure 5.9.

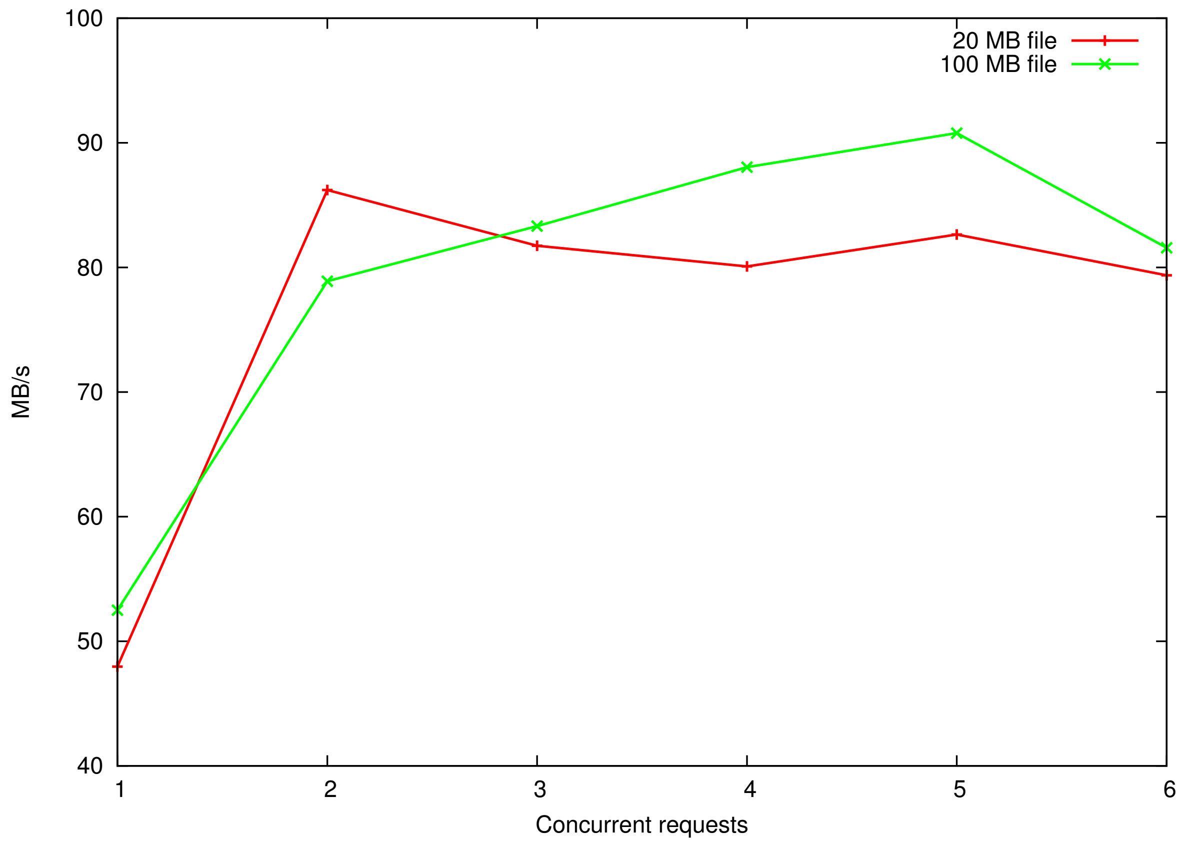


Figure 5.9: Storage service bandwidth

Initially, a single read operation is able to achieve a fairly acceptable throughput of 40-50MB/s. When performing two requests concurrently, the aggregate throughput almost doubles to 80-90 MB/s. Subsequently increasing the concurrency level to more than two parallel reads provides no increase in aggregate throughput, stabilizing at 80-90 MB/s shared between the number of concurrent read operations.

Note that we did not see significant changes in the throughput between initial and subsequent transfers, indicating that caching is not significantly affecting our results. We also note that we periodically experienced significant variance in the aggregate throughput, sometimes becoming as low as 30-40 MB/s, especially with more than two concurrent read operations. We attribute this to the fact that instances are able to use more than their strict share of I/O bandwidth when the shared connection is under-utilized.

With a bandwidth of 80 MB/s, transferring a complete index of 2GB data would take approximately 30 seconds. If we instead base our estimates on a slightly more pessimistic average throughput of 40 MB/s, the same index transfer will take about a minute. This means that instantiating and preparing new index nodes in a cloud-based version of FAST ESP as described in the previous chapter will require about a minute to fetch a full or partial index into its local cache, in addition to the time incurred to prepare and boot a VM instance, which is currently 30-60 seconds.

To enable dynamic scalability in the cloud search service, additional server instances can be added when the throughput reaches some threshold; for example 500 QPS per server. This way, additional capacity is made available in 1-2 minutes, depending on index transfer bandwidth and VM initialization time. Similarly, capacity can be scaled down when the throughput of a server is low, for example 200 QPS over some time.

5.6 Discussion

5.6.1 Conclusion

To conclude the discussion of the feasibility of our approach, we return to the original goals for the cloud search service.

Our first experiment show that a single search server is able to support approximately 650 queries per second. While this number is not directly comparable to a realistic search service (primarily because of the reduced number of I/O operations per query), we conclude that a single server is able to support an acceptable query rate even if we reduce the capacity to account for more realistic queries. At this query rate, the average response time is 275 ms, and within our goal.

Experiments with concurrent requests from the cloud storage service indicate that S3 provides linear scalability and sufficient replication. This means that we are able to double the capacity of our search service by adding a second search server. Still, we have only tested scalability with two servers, and more tests must be performed to investigate if the linear scalability persists with higher loads.

Finally, our experiments establish that we are able to achieve a bandwidth of 80-90 MB/s between EC2 and S3, although we see some variance which we attribute to periodic under-utilization of bandwidth on instances sharing the physical hardware. This bandwidth means we can transfer a complete copy of our 2GB index in approximately 30 seconds, or about a minute with a slightly pessimistic 40 MB/s bandwidth.

5.6.2 Non-cloud performance

To put the performance of our cloud search service into some perspective, we have taken a look at the performance of FAST ESP in a traditional non-cloud configuration, as reported in [71]. We note, however, that these numbers are somewhat dated (from 2003) and may not fully reflect the current state of hardware. Still, we argue that they should be relevant for providing some perspective.

To classify throughput, we turn to Figure 12 on p. 105 in [71], in which QPS is described as a function of the number of query terms. For single term queries, the throughput is reported to be 975 QPS, dropping exponentially as the number of search terms increase. With ten terms per query, the throughput drops to approximately 100 QPS.

The cloud search service performs the equivalent of a single term queries, and is able to provide a throughput of 650 QPS, yielding a difference of approximately 325 QPS. However, since our implementation is limited primarily by the number of concurrent threads the server is able to support, it is probable that a more efficient implementation of the cloud search service might be able to support a higher throughput. On the other hand, the cloud search service performs only a single I/O operation for a single term query, whereas FAST ESP performs two (in the absence of caching); one to lookup the termID, and another to fetch the list of occurrences.

With FAST ESP deployed within a LAN, network latency drops significantly due to geographical locality. In our experiments, we found that the average RTT to EC2 data centers in Europe was 55 ms. Within a LAN environment, the RTT would likely be under 10 ms, and often only a few milliseconds. This implies that FAST ESP is able to significantly outperform the cloud search service in highly interactive scenarios, even if service time were equal for the two services. However, we argue that the exact quantified response time is of less importance as long as it resides within the acceptable range, which we have defined as 400 ms.

5.6.3 Complicating factors

Larger indices

In our implementation, we have chosen to use a 2 GB index, chosen to be sufficiently large to represent a realistic index for a small organization, as well as being large enough to be impractical to keep cached in the cloud storage service. Still, large organizations with a lot of content might have significantly larger indices, possibly even in the order of terabytes or petabytes.

In our implementation, the index is stored as objects each containing an individual 8KB index block, with all objects placed in a shared S3 bucket. While the internal operation of S3 is not documented, the documentation claims that the service is designed to scale gracefully both in terms of traffic and data size [78]. This implies that the number of objects in a bucket should not affect the performance of object fetch operations, and there are no limitations imposed on the number of objects placed in a single bucket. Still, additional tests should be carried out with larger index sizes to ensure that the S3 performance is not negatively affected with an increase in aggregate data size.

Multiple I/O operations per query

In a realistic search service, queries generally require multiple I/O operations. In our cloud search service, this would mean performing multiple requests to the cloud storage service. Based on Figure 5.4, we see that each storage request takes about 30 ms. If these are performed sequentially, the service time for each request would quickly increase. For example, the service time would double from 100 ms for a single I/O query to a little over 200 ms for a query requiring four I/O requests, quickly becoming the dominating factor.

However, Figure 5.8 suggests that read operations are parallelizable. This means that requests for multiple objects can be sent to the storage service in virtually at the same time, reading them concurrently. However, increasing the average number of I/O operations required per query will still reduce the total capacity of the search service, because each incoming query uses a larger amount of the limited network I/O bandwidth. Also, I/O operations depending on previous operations (e.g., looking up the occurrences of a term ID still waiting to be looked up), must still be sequentially executed.

5.6.4 Alternative approaches

As previously mentioned, the goal of our implementation of the cloud search service has been to investigate the feasibility of using cloud services for index storage, rather than arriving at a complete or optimized implementation. With this in mind, we provide a brief overview of some alternative, perhaps more realistic, approaches. To ensure relevancy, we limit our scope to an architectural level, but further details on the functionality and optimizations present in FAST ESP can be found in [71].

Keep current FAST ESP architecture

A natural approach is keeping the current architecture of FAST ESP, only moving it from an in-house data center to a cloud IaaS environment, replacing each physical server with a VM instance. Presumably moderate modifications would be required in the existing code base to allow the search and index dispatchers to be able to dynamically start and stop instances along with the load, balancing between them as today. Instead of accessing the index from cloud storage, search nodes would instead rely on the local disk to store index data, like the current approach in in FAST ESP.

Intuitively, this should lead to performance comparable to current installations of FAST ESP, except for potentially increased network latency when the cloud data center is located geographically more distant. However, since FAST ESP is very disk I/O intensive, the performance of this configuration would strongly rely on the disk I/O performance provided by the VMM. To verify that the disk I/O performance and isolation provided by the VMM is adequate for this approach, tests must be performed.

Caching

A significant potential for optimization lies in the strategy chosen for caching. In the current cloud search service there is no support for caching, resulting in a cloud storage lookup being performed for every request. In [71], the authors claim that a sampling of the queries performed by users of the *AllTheWeb.com* search service show that only 7.28% of the queries were unique; all other queries were performed two or more times during the sampling period. This provides motivation for implementing caching support.

Several caching strategies are possible. One approach is configuring nodes to fetch a full or partial copy of the complete index from cloud storage upon startup, periodically refreshing data to ensure freshness. Queries are then performed on the local replica, residing on disk. Additionally, parts of the index can be kept in memory, according to some appropriate caching strategy. This should provide the same performance as the solution outlined above, but also relies on adequate disk I/O performance and isolation to be provided by the VMM. A disadvantage of this scheme is that the initial transfer of index data means that search nodes will require a longer startup time.

Another possibility is caching individual results of cloud storage lookups, invalidating them after some time according to some freshness criteria. This caching can be performed on disk and/or in memory, like the previous approach. While this approach ensures that search node startup time is kept low, search nodes will likely require some time to reach maximum performance, as the cache is filled.

Index location

An alternative to keeping the index data on local disks in the search nodes or in a cloud storage service, is to use the cloud database services provided by cloud environments like EC2 and Azure Services Platform, providing scalable support for storing, manipulating and fetching structured or unstructured data. Depending on the query requirements of the search service, these services can potentially be harnessed to perform lookups directly on the index. For example, Amazon claims that their their cloud database service, *SimpleDB*, is designed for real-time applications and provides near-LAN latencies [80]. The latency of this approach would in all likelyhood be comparable to the S3 approach. Also, limitations in the database query mechanisms might mean that not all queries can be performed by the service directly.

Another example is placing the index in a service like *Elastic Block Storage* [81], which is effectively a Storage Area Network (NAS) device that can be mounted as a disk drive in EC2 instances. However, both of these alternatives require performance testing to investigate if they are able provide sufficient performance to be useful in this context.

5.6.5 Difference in EU/US latency

EC2 users can select what region instances should be located in, currently choosing between two; *eu-west* and *us-east*. Amazon indicates that more regions might be added in the future. Table 5.1 shows the RTT for the two regions, measured from Fornebu, Norway. We see that hosting instances in the US data centers result in a doubling of the average RTT, which would lead to higher response times due to increased geographical distance.

Region	**Location**	**Minimum RTT**	**Average RTT**	**Maximum RTT**
eu-west	Ireland	54.58	56.54	60.525
us-east	US east coast	110.20	116.06	125.11

Table 5.1: Measured RTTs to EC2 data centers in EU and US.

5.7 Summary

We have successfully implemented an architectural model of a cloud search service, using S3 as index storage. Our primary goal was to investigate the feasibility of supporting a cloud search service using cloud services rather than an arriving at an efficient and realistic implementation.

Looking up individual index blocks stored in the S3 service from within Amazon EC2 yields a response time of approximately 30 milliseconds. We are able to serve about 650 requests/second using a single server, seemingly limited by the number of threads our service implementation is able to support.

When isolating the performance of the cloud storage service, we were able to perform approximately 1000 requests/second for randomly distributed 8KB index blocks, yielding a maximum throughput of 15 MB/s. The S3 service seems to properly scale under increased load, at least to the extent we were able to apply during testing. The response time increasing with concurrency seems to be caused by local request queuing due to contention, since service time measured at the S3 level is relatively stable.

While raw performance, especially with regards to response time, is significantly lower compared to local disk I/O, the possibility of easily implementing dynamic scalability ensures that the S3 cloud storage service remains potentially attractive as a novel approach to index and data storage for some applications. With an EC2 instance instantiation time of approximately 30-60 seconds, search nodes can be added and removed in an order of minutes, allowing the inherent temporal load variance in many services to be exploited.

However, both bandwidth and response time is subject to some variance. This implies that some of the observed results may be influenced by EC2 and S3 services currently being under-utilized, due to their relatively recent release. Both EC2 and S3 offers conceptually simple control interfaces, enabling applications to implement self-monitoring and dynamic scaling.

We conclude that basing FAST ESP in a cloud environment seem feasible, both in terms of achievable response times and bandwidth. Still, the many simplifications made in our architectural model mean that our results do not fully reflect the complexities inherent in a real-world implementation, and our results should be considered primarily as initial observations. The simulated workload in our prototype also represents a relatively small installation of the solution, and more extensive investigation must be performed to discover how the service performs and scales under higher load.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have taken an in-depth look at the current state of cloud computing, starting with virtualization as a key enabling technology and grid computing as a significant influence. We distill cloud computing into two primary aspects; computational resources provided as a pay-as-you-go utility, and the notion of Software as a Service. We have also used an ontology to describe five layers of cloud computing; firmware/hardware, software kernel, infrastructure, platform and application layers.

Furthermore, we have discussed key characteristics of IaaS and PaaS environments, including horizontal scalability, increased latency, loose coupling, bandwidth constraints and limited platform portability. We have also taken a closer look at several state-of-the-art cloud offerings, such as Amazon EC2 and Azure Services Platform.

To determine what classes of systems are suitable for cloud environments, as well as investigate possible approaches to moving systems to the clouds, we have looked at typical cloud-related characteristics of many enterprise systems. These include data centricity, static scalability, tight coupling and low-latency requirements. Together, they restrict what classes of systems are suitable for cloud environments, and what benefits cloud computing can provide.

6.2 Contributions

Through an in-depth look at the current state of cloud computing, we have found that cloud computing provides attractive opportunities to enable systems to exploit temporal load variance through dynamic scalability. It also allows organizations to cost-effectively support highly available applications deployed in geographically redundant data centers. Another attractive feature is the ability to deploy systems with little or no initial investments in hardware, by outsourcing infrastructure operations. These opportunities suggest that cloud computing will find a wide range of use in the future, ranging from hosting complete systems to providing temporary infrastructure for research projects or software testing.

Still, the scalability and flexibility of cloud computing comes at some costs. Challenges associated with hosting systems in cloud environments include increased latency due to longer network distances, limited bandwidth since packets must cross the Internet to reach the cloud, as well as reduced portability because cloud environments are currently lacking standardization. Additionally, systems must be designed in a loosely coupled and fault-tolerant way to fully exploit the dynamic features of cloud computing, meaning that existing applications might require significant modification before being able to fully utilize a cloud environment. Organizations designing systems for cloud environments also risk becoming locked to a specific vendor, since current offerings are highly opaque and non-standardized.

Furthermore, we have looked at common characteristics of enterprise systems, and found that they include data centricity, tight coupling and static scalability. These characteristics affect how systems are moved to cloud environments, and restrict the classes of systems suitable for cloud environments. Systems with absolute characteristics that correspond to the challenges of cloud environments, such as strict latency requirements or large data amounts, are least suited for cloud environments. Examples of such applications include video streaming software requiring predictable CPU scheduling and high bandwidth, health management software dealing with sensitive information and business-critical software over which an organization requires full control. Still, we argue that there is no single metric for cloud suitability, as the importance of individual opportunities and challenges differ between systems, and must be balanced in each case.

Finally, we have implemented a cloud search service based on an architectural model of FAST ESP used to investigate some of potential challenges; namely response time, bandwidth and scalability. We conclude that implementing a search service like FAST ESP in a cloud environment is feasible, despite providing inferior performance in terms of response time and bandwidth compared to a direct disk-based implementation. Still, we argue that our cloud-based prototype represents a novel approach to implementing dynamic scalability along with a reduced need for local infrastructure operations.

6.3 Future work

Due to time constraints, we were unable to investigate all challenges related to developing and running services in a cloud environment. Open issues include development issues like middleware and debugging support, as well as operational issues like instrumentation, dynamic scaling and load balancing.

Also, a more extensive range of systems should be tested for suitability in multiple cloud environments. Establishing a standardized test suite and evaluation methodology for benchmarks of cloud computing environments would help to enable a more systematic approach to cloud evaluation.

Appendix A

Source code

The attached CD-ROM contains the source code for the cloud search service and the experiments performed in Chapter 5.

The code is also made available at the following URL:
`http://heim.ifi.uio.no/chrismi/master/`

Bibliography

[1] FAST . FAST ESP . `http://fast.no/l3a.aspx?m=1031`, March 2009.

[2] Brian Hayes. Cloud Computing. *Commun. ACM*, 51(7):9–11, 2008.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Standard) , 1999.

[4] S.J. Vaughan-Nichols. New Approach to Virtualization Is a Lightweight. *Computer*, 39(11):12–14, Nov. 2006.

[5] J.E. Smith and Ravi Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, May 2005.

[6] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, November 2008.

[7] Intel. *Intel Itanium Processor-specific Application Binary Interface (ABI)*, November 2008.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[9] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. pages 45–58, 1999.

[10] Sun Microsystems. Java . `http://java.sun.com`, November 2008.

[11] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, June 2006.

[12] Microsoft Corporation. .NET Framework . `http://microsoft.com/NET/`, December 2008.

[13] R.J.Creasy. The Origin of the VM/370 Time-Sharing System. *IBM J. Research and Development*, pages 483–490, Sept, 1981.

[14] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[15] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[16] Peter H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983.

[17] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.

[18] J. Postel. Transmission Control Protocol. RFC 793 (Standard) , August 1981.

[19] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[20] Eric Traut. Building the virtual PC. *BYTE*, 22(11):51–52, 1997.

[21] Viktors Berstis. Fundamentals of Grid Computing. *IBM Redbooks*, 2002.

[22] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[23] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. Concurrency and Computation: Practice and Experience. 17:2–4, 2005.

[24] EGEE Project. Enabling Grids for E-science (EGEE) . `http://eu-egee.org/`, May 2009.

[25] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the Grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006.

[26] Apache Software Foundation . Apache Hadoop . `http://hadoop.apache.org`, November 2008.

[27] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[28] Steffen Viken Valvåg and Dag Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 113–122, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Cloud Computing Journal. Twenty-One Experts Define Cloud Computing . `http://cloudcomputing.sys-con.com/node/612375`, April 2009.

[30] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[31] Aaron Weiss. Computing in the Clouds. *netWorker*, 11(4):16–25, 2007.

[32] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a Unified Ontology of Cloud Computing. *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov. 2008.

[33] Salesforce.com, Inc. . Salesforce CRM . `http://www.salesforce.com`, November 2008.

[34] Google . Google Apps . `http://www.google.com/apps/`, November 2008.

[35] Google . Google AppEngine . `http://code.google.com/appengine/`, December 2008.

[36] Joyent . Joyent Accelerator . `http://www.joyent.com/accelerator`, December 2008.

[37] W3C . SOAP Specifications . `http://www.w3.org/TR/soap/`, February 2009.

[38] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, May 2002.

[39] Amazon.com . Amazon EC2 . `http://aws.amazon.com/ec2`, November 2008.

[40] University of California, Santa Barbara . Eucalyptus . `http://eucalyptus.cs.ucsb.edu/`, January 2009.

[41] Werner Vogels. Eventually consistent. *ACM Queue*, 2008.

[42] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[43] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *In Proceedings 7th SIGOPS European Workshop*, pages 275–280. ACM, 1996.

[44] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM Press.

[45] Amazon.com . Amazon S3 . `http://aws.amazon.com/s3`, November 2008.

[46] Dropbox . `http://getdropbox.com`, January 2009.

[47] Twitter . `http://twitter.com`, March 2009.

[48] William Johnston. Network Communication as a Service-Oriented Capability . In *High Performance Computing and Grids in Action* , volume 16. Lawrence Berkeley National Laboratory, 2008.

[49] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov. 2008.

[50] M.P.I Forum. MPI: A Message-Passing Interface Standard Message Passing Interface Forum. 1994.

[51] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *CoRR*, 2002.

[52] M.A Vouk. Cloud computing - Issues, research and implementations. In *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, pages 31–40, 2008.

[53] Simon Crosby et. al. Open Virtualization Format Specification. 2008.

[54] Citrix Community. Project Kensho . `http://community.citrix.com/display/xs/Kensho`, November 2008.

[55] Intridea. Scalr . `http://scalr.net`, December 2008.

[56] Yousef Khalidi. Windows Azure: Architecting and Managing Cloud Services. In *Professional Developers Conference*, 2008.

[57] Huan Liu and Dan Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 295–305, 2008.

[58] Amazon.com . Amazon SQS . `http://aws.amazon.com/sqs`, December 2008.

[59] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 249–264. ACM Press, 1974.

[60] W3C . Web Services Description Language (WSDL) 1.1 . `http://www.w3.org/TR/wsdl/`, January 2009.

[61] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure - Certificate and Certificate Revocation List (CRL) Profile. RFC 3280.* . 2002.

[62] GoGrid Inc. . GoGrid Cloud Hosting . `http://www.gogrid.com`, November 2008.

[63] Samba . `http://samba.org`, November 2008.

[64] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-source Cloud-computing System . In *Proceedings of Cloud Computing and Its Applications*, October 2008.

[65] Python . `http://python.org`, February 2009.

[66] Django Web Framework . `http://djangoproject.org`, March 2009.

[67] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[68] Memcached . `http://danga.com/memcached`, January 2009.

[69] Microsoft Corporation. Microsoft Visual Studio . `http://microsoft.com/visualstudio`, December 2008.

[70] J. Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16:103–111, 1999.

[71] Knut Magne Risvik, Yngve Aasheim, and Mathias Lidal. Multi-Tier Architecture for Web Search Engines. In *LA-WEB '03: Proceedings of the First Conference on Latin American Web Congress*, page 132, Washington, DC, USA, 2003. IEEE Computer Society.

[72] Wikipedia: Black Friday (shopping) . `http://en.wikipedia.org/wiki/Black_Friday_(shopping)`, January 2009.

[73] Jungledisk . `http://jungledisk.com`, January 2009.

[74] Charles J. Testa and Douglas B. Dearie. Human factors design criteria in man-computer interaction. In *ACM 74: Proceedings of the 1974 annual conference*, pages 61–65, New York, NY, USA, 1974. ACM.

[75] James Murty. Jets3t . `http://jets3t.s3.amazonaws.com/index.html`, January 2009.

[76] HAProxy . `http://haproxy.1wt.eu`, February 2009.

[77] J. D. C. Little. A Proof of the Queueing Formula $L = \lambda W$. *Operations Research* , 9:380–387, 1961.

[78] Amazon.com . Amazon S3 Documentation . `http://docs.amazonwebservices.com/AmazonS3/2006-03-01/`, December 2008.

[79] Rightscale. Network performance within Amazon EC2 and to Amazon S3 . `http://blog.rightscale.com/2007/10/28/network-performance-within-amazon-ec2-and-to-amazon-s3/`, February 2009.

[80] Amazon.com . Amazon SimpleDB . `http://aws.amazon.com/simpledb`, December 2008.

[81] Amazon.com . Amazon Elastic Block Storage . `http://aws.amazon.com/ebs`, December 2008.