2015

Resource Management in Large-scale Systems

Ashkan Paya
University of Central Florida

RESOURCE MANAGEMENT IN LARGE-SCALE SYSTEMS

by

ASHKAN PAYA

B.S. Sharif University of Technology, 2011
M.S. University of Central Florida, 2014

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2015

Major Professor: Dan C.Marinescu

ABSTRACT

The focus of this thesis is resource management in large-scale systems. Our primary concerns are energy management and practical principles for self-organization and self-management. The main contributions of our work are:

1. **Models.** We proposed several models for different aspects of resource management, e.g., energy-aware load balancing and application scaling for the cloud ecosystem, hierarchical architecture model for self-organizing and self-manageable systems and a new cloud delivery model based on auction-driven self-organization approach.

2. **Algorithms.** We also proposed several different algorithms for the models described above. Algorithms such as coalition formation, combinatorial auctions and clustering algorithm for scale-free organizations of scale-free networks.

3. **Evaluation.** Eventually we conducted different evaluations for the proposed models and algorithms in order to verify them. All the simulations reported in this thesis had been carried out on different instances and services of Amazon Web Services (AWS).

All of these modules will be discussed in detail in the following chapters respectively.

Dedicated to my brother Arash Paya.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dan C.Marinescu for the continuous support of my Ph.D. study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Moreover, I would like to thank the rest of my thesis committee: Prof. Mostafa Bassiouni, Prof. Eduardo Mucciolo, and Dr. Pawel Wocjan, for their encouragement, insightful comments, and hard questions.

My sincere thanks also goes to Dr. Lukas Marti, for offering me a Summer internship opportunity in their group and leading me working on diverse exciting projects at Apple Inc.

Last but not least, I would like to thank my brother Arash Paya and also my parents Jamshid Paya and Nazila Minbashi Zadeh, for supporting me spiritually throughout my life.

Ashkan Paya
April 2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION AND MOTIVATION

Today, utility computing, envisioned by John Mc. Carthy[1] and others is a social and technical reality, but cloud computing technology must evolve to avoid becoming the victim of its own success.

At this time, the average cloud server utilization is low, while the power consumption of clouds based on over-provisioning is excessive and has a negative ecological impact. We live in a world of limited resources and cloud over-provisioning is not sustainable either economically or environmentally. If successful, self-organization would allow cloud servers to operate more efficiently and thus, reduce costs for the Cloud Service Providers (CSPs), provide an even more attractive environment for cloud users, and support some form of interoperability. The pressure to provide new services, better manage cloud resources, and respond to a broader range of application requirements is increasing, as more US government agencies are encouraged to use cloud services[2].

We have known for years that distributed systems which maintain state information are neither scalable nor robust; this is the reason why most Internet services are delivered by stateless servers. We have also known for some time that collecting state information consumes a fair share of system resources and that system management decisions based on obsolete state information are far from optimal; this knowledge is critical for the communication and computing infrastructure built around the Internet but resource management in cloud computing is still based on hierarchical control models where state information is maintained at several levels. We have also known that

[1] In1961, in a speech given to celebrate MIT's centennial, he suggested that in the future computing power and applications could be sold through the utility business model.

[2] See for example the November 6 memorandum "The DoD Cloud Way Forward" which stresses the need for DoD to increase its use of cloud services.

assembling large collections of systems each with a small, but finite probability of failure, requires special design principles to guarantee availability.

For several decades we have designed and built heterogeneous computing systems with very large numbers of components interacting with each other and with the environment in intricate ways. The complexity of such systems is undeniable but their design was, and still is, based on traditional, mostly deterministic, system organization and management. This has to change, but the path to change is strenuous.

Hypothesis of our research is that self-organization could potentially solve the major challenges related to resource management in large-scale systems. *Self-organization* is the spontaneous emergence of global coherence out of local interactions. A self-organizing system responds to changes in the environment through adaptation, anticipation, and robustness. The system reacts to changes in the environment, predicts changes and reorganizes itself to respond to them, or is robust enough to sustain a certain level of perturbations. We argue that self-organization could be critical for the future of cloud computing and it is feasible and effective.

1.1
Self-organization Could Be Critical For The Future of Computer Clouds

In the early 2000s it was recognized that the traditional management of computer systems is impractical and IBM advanced the concept of *autonomic computing* [51, 75]. Progress in the implementation of autonomic computing has been slow. Autonomic computing is closely related to self-organization and self-management. The main aspects of autonomic computing as identified in [75] are: *Self-configuration* - configuration of components and systems follows high-level policies, the entire system adjusts automatically and seamlessly; *Self-optimization* - components continually seek opportunities to improve their own performance and efficiency; *Self-healing* - the system automatically detects, diagnoses, and repairs localized software and hardware problems; and *Self-protection* - automatically defend against malicious attacks and anticipate and prevent system-wide failures.

The cloud ecosystem is evolving, becoming more complex by the day. Some of the transformations expected in the future add to the complexity of cloud resource management and require different policies and mechanisms implementing these policies. Several factors affecting the complexity of cloud resource management are:

a. *The cloud infrastructure is increasingly more heterogeneous.* Servers with different configurations of multi-core processors, attached co-processors (GPUs, FPGAs), and data flow engines are already, or are expected to become elements of the cloud computing landscape. Amazon Web Services (AWS) already support G2-type instances with GPU co-processors.

b. *The spectrum of cloud services and cloud applications widens.* For example, in the last year AWS added some 10 new services, including Lambda, Glacier, Redshift, Elastic Cache, and DynamoDB. Several types of EC2 (Elastic Compute Cloud) profiles, M3 - balanced, C3 - compute optimized, R3 - memory optimized, I2 and HS1 - storage optimized were also identified in the last months. The spectrum of EC2 instance types is also broadening; each instance type provides different sets of computer resources measured by vCPUs (vCPU is a hyper-thread of an Intel Xeon core for M3, C3, R3, HS1, G2, and I2).

As the cloud user community grows, instead of a discrete menu of services and instance types we expect a continuum spectrum; policies and mechanisms allowing a cloud user to precisely specify the resources it needs and the conditions for running her application should be in place. At the same time, the cloud infrastructure should support an increasing number of data- and CPU-intensive *Big Data* applications.

Many big data applications in computational science and engineering do not perform well on cloud. A 2010 paper [73] presents the results of an HPCC (High Performance Computing Challenge) benchmark of *EC2* and three supercomputers at NERSC[3]. The results show that the floating performance of EC2 is about 2.5 times lower, 4.6 Gflops versus 10.2 Gflops. The memory bandwidth is also about 2.5 times lower, 1.7 versus 4.4 GB/sec. The network latency is significantly higher, 145 versus 2.1 μsec and the network bandwidth is orders of magnitude lower, 0.06 versus 3.4 GB/sec. One of the goals of ongoing research in cloud resource management is to show that the performance could be improved by a self-organization scheme which exploits locality and reduces

[3] National Energy Research Scientific Computing Center

the communication latency, as discussed in Section 1.5.

c. *Cloud over-provisioning demands high initial costs and leads to a low system utilization; this strategy is not economically sustainable* [41]. Cloud *elasticity* is now based on over-provisioning, assembling pools of resources far larger than required to satisfy the average needs. Elasticity allows cloud users to increase or decrease their resource consumption based on their needs. The average cloud server utilization is in the 18% to 30% range [25]. Low server utilization implies that the cloud power consumption is far larger than it should be. The power consumption of cloud servers is not proportional with the load, even when idle they use a significant fraction of the power consumed at the maximum load. Computers are not *energy proportional systems* [25] thus, power consumption of clouds based on over-provisioning is excessive and has a negative ecological impact [130]. A 2010 survey [30] reports that idle or under utilized servers contribute 11 million tones of unnecessary CO_2 emissions each year and that the total yearly cost for the idle servers is $19 billion.

d. *The cloud computing landscape is fragmented.* CSPs support different cloud delivery models: Amazon IaaS (Infrastructure as a Service), Microsoft PaaS (Platform as a Service), Google mostly SaaS (Software as a Service), and so on. An obvious problem with clear negative implication is the vendor lock-in; once becoming familiar and storing her data on one cloud it is very costly for the user to migrate to another CSP. An organization which can seamlessly support cloud interoperability and allow multiple cloud delivery models to coexist poses additional intellectual challenges.

In [108] we introduce a cloud self-organization architecture which addresses these challenges and shows that economic models support policies and mechanisms for implementing effective cloud resource management models.

1.2
Practical Implementation of Cloud Self-organization Is Challenging

Complexity and self-organization have preoccupied the minds of computing pioneers John von Neumann [120] and Alan Turing; the essence of self-organization is captured by Turing's observation [155] "Global order can arise from local interactions." The interest in complex systems and cloud self-organization is illustrated by a fair number of recent papers including [8, 16, 41, 85, 108, 109, 111, 124, 142, 159].

Our limited understanding of complex systems and the highly abstract concepts regarding self-organization developed in the context of natural science do not lend themselves to straightforward application to the design of modern computing and communication systems. Practical application of self-organization principles to computer clouds is extremely challenging due to the absence of a technically suitable definition of self-organization and to the other challenges discussed in Section 1.1. A powerful indication of the challenges posed by practical aspects of self-organization is that none of the existing large-scale computing systems can be accurately labeled as self-organizing.

Practical implementation of cloud self-organization is challenging for several reasons:

A. The absence of a technically suitable definition of self-organization, a definition that could hint to practical design principles for self-organizing systems and quantitative evaluation of the results. Marvin Minsky [102] and Murray Gell-Mann [50] have discussed the limitations of core concepts in complex system theory such as emergence and self-organization. The same applies to autonomic computing, there is no indication on how to implement any of the four principles and how to measure the effects of their implementation.

B. Computer clouds exhibit the essential aspects of complexity; it is inherently difficult to control complex systems. A complex system is one with a very large number components, each with distinct characteristics, and many interaction channels among individual components. Complex systems: (a) are nonlinear [4]; (b) operate far from equilibrium; (c) are intractable at the component level; (d) exhibit different patterns of behavior at different scales; (e) require a long history to draw conclusion about their properties; (f) exhibit complex forms of emergence[5]; (g) are affected by phase transitions - for example, a faulty error recovery mechanism in case of a power failure took down Amazon's East Coast Region operations; and (h) scale well. In contrast, simple systems are linear, operate close to equilibrium, are tractable at component level, exhibit similar patterns of behavior at different levels, relevant properties can be inferred based on a short history, exhibit

[4] The relation between cause and effect is often unpredictable: small causes could have large effects, and large causes could have small effects. This phenomena is caused by *feedback*, the results of an action or transformation are fed back and affect the system behavior.

[5] Emergence is generally understood as a property of a system that is not predictable from the properties of individual system components.

simple forms of emergence, are not affected by phase transitions, and do not scale well, see also Chapter 10 of [110]. There are also specific factors making even more challenging the application of self-organization principles to large-scale computing and communication systems:

1. Abstractions of the system useful for a particular aspect of the design may have unwanted consequences at another level.

2. Systems are entangled with their environment. The environment is man-made and the selection required by the evolution can either result in innovation, or generate unintended consequences, or both.

3. Systems are expected to function simultaneously as individual systems as well as groups of systems (systems of systems) [101].

4. Systems are both deployed and under development at the same time.

C. A quantitative characterization of complex systems and of self-organization is extremely difficult. We can only assess the goodness of a particular self-organization algorithm/protocol indirectly, based on some of the measures of system effectiveness, e.g., the savings in cost or energy consumption. We do not know how far from optimal a particular self-organization algorithm is.

Some of the challenges to the cloud delivery models discussed in Section 5.1 are amply documented in the literature. Security and privacy [10, 18, 35, 57, 60, 92, 131, 134], the virtualization overhead [38, 110, 114], and sustainability [41, 127, 160] are major concerns motivating research for new architectural solutions for cloud computing [37, 140].

A review of some of the challenges faced by cloud computing hints that computer clouds are complex systems [4, 142]. Four groups of actors are involved in cloud computing: (i) the CSP infrastructure consisting of possibly millions of compute and storage servers and an interconnection network and the stack of software running on each of these systems; (ii) a very large population of individual and corporate users with different level of expertise, expectations, applications, and constraints; (iii) the regulators, the government agencies that enforce the rules governing the business; (iv) and, last but not least, the physical environment, including the networks supporting user access and the power grid supplying the energy for powering the systems, the heating and the cooling. These components interact with one another often in unexpected ways; a faulty error recovery procedure triggered by the power failure of a few systems could cause a chain reaction and shut down an entire data center, thus affecting a very large user population.

Today's clouds are designed and engineered using technics suitable for small-scale deterministic systems rather than complex systems with non-deterministic behavior. The disruptive technology we advocate for a cloud infrastructure is based on *self-organization* and *self-management*.

Informally, self-organization means synergetic activities of elements when no single element acts as a coordinator and the global patterns of behavior are distributed [50, 141]. More recent concepts such as *autonomic computing* introduced by IBM and *organic computing* have some intersection with self-organization and imply autonomy of individual system components and intelligent behavior.

Self-organization is prevalent in nature; for example, this process is responsible for molecular self-assembly, for self-assembly of monolayers, for the formation of liquid and colloidal crystals, and in many other instances. Spontaneous folding of proteins and other biomacromolecules, the formation of lipid bilayer membranes, the flocking behavior of different species, the creation of structures by social animals, are all manifestation of self-organization of biological systems. Inspired by biological systems, self-organization was proposed for the organization of different types of computing and communication systems [65, 107], including sensor networks, for space exploration [67], and even for economical systems [79]. A number of studies of self-organization in physical systems and the mechanisms to control such systems have been published recently [87, 117].

Though the virtues of self-management have long been recognized [7, 8, 61], there is, to our knowledge, no cloud computing infrastructure, or large-scale computing or communication system based on self-organizing principles. Some of the mechanisms used in our model have been discussed in the literature e.g., [11, 26, 27, 47, 58, 91], others have been incorporated in different cloud architectures [9, 143].

1.3
The Physical Model of The Cloud Infrastructure

The model is based on a description of Google's cloud infrastructure in [25]. Warehouse-scale computers (WSCs) are the building blocks of a cloud infrastructure. A hierarchy of networks connect $50,000$ to $100,000$ servers in a WSC.

The servers are housed in racks; typically, the 48 servers in a rack are connected by a 48-port Gigabit Ethernet switch. A switch has two to eight up-links which go to the higher level switches in the network hierarchy [25, 68]. A number of racks are connected into a *cell* and a WSC consists of tens of cells. As the latency increases when messages cross multiple layers of the hierarchy and the bandwidth decreases, a very careful application mapping is necessary.

1.4
Practical Principles For The Design of Self-organizing Clouds

Tensions between local and global objectives in a self-organizing cloud system exist. These tensions manifest themselves in questions such as: How to balance the individual cost of autonomous servers with global goals e.g., maximizing the CSP payoff? How to adapt the price for services to the actual demand? How to find an equilibrium between system reconfiguration and continuous system availability?

Several principles provide answers to some of these questions and guide our suggestions for cloud self-organization:

1. Base the design on the principle of rational choice; assume that an agent, in our case an autonomous server, will always choose the option that maximizes its utility. *Utility* is the measure of the value or the benefit of an action.

2. Support some form of coordination among groups of autonomous servers. The servers have to cooperate when responding to service requests, guaranteeing QoS by distributing and

balancing the workload, replicating services to increase reliability, and implementing other global system policies. Cooperation means that individual systems have to partially surrender their autonomy.

3. Take advantage of the properties of market-based strategies and auctions [14, 15, 106] to ensure system scalability and to guarantee that the system will eventually reach equilibrium.
4. Devise algorithms and mechanisms for coalition formation e.g., [99], to allow autonomous servers to act in concert and compare their effectiveness and weaknesses.
5. Devise a mechanism to support an effective reservation system. Self-organization cannot occur instantaneously therefore, give the autonomous servers interconnected by a hierarchy of networks the time to form coalitions in response to services requests. Thus, self-management requires an effective reservation system. Reservations are ubiquitous for systems offering services to a large customer population, e.g., airline ticketing, chains of hotels, and so on. Existing clouds, e.g., the Amazon Web Services, offer both reservations and "spot" instances, with spot access rates lower than those for reservations.
6. Devise effective mechanisms to support spatial and temporal locality.
7. Use consensus algorithms for reaching decisions.

1.5
A Reservation Systems Based on Coalition Formation and Auctions

We believe that coalition formation and auctions could support self-organization protocols that can exploit the hierarchical architecture of today's cloud infrastructure. *Coalition formation* supports aggregation of resources and of services. Resource aggregation is necessary because a single server may not be able to supply the resources demanded by a cloud client. Service aggregation is necessary to reduce the number of agents involved in an auction thus, the time and the space requirements of the auctioning algorithms. Coalition members will be located in "proximity" to one another and thus benefit from lower latency and higher bandwidth communication. This will guarantee lower communication costs for the cloud services provided by the coalition.

Auctions have been successfully used for resource management in the past. One of the advantages of auction-based resource management is that auctions do not require a model of the system, while traditional cloud resource management strategies do. We shall investigate reservation systems when auctions are organized periodically and the reservations are expressed in *allocation intervals.* An auction-based reservation protocol strikes a balance between low-cost services for cloud clients and a decent profit for the service providers, is scalable, and, though the computational algorithms involved are often fairly complex, the computations can be done efficiently. Cloud service packages will be auctioned; a *package* consist of combinations of services in one or more allocation intervals. The items sold are services advertised by sub-coalitions of autonomous servers and the bidders are the cloud users. A service is characterized by a type describing the resources offered, the conditions for service, and the allocation intervals when the service is available.

Spatial and temporal locality are critical requirements for the effectiveness of a self-organized and self-managed cloud. In the protocols we envision *spatial locality* means that an auction should favor packages which involve servers in the same sub-coalition; spatial locality reduces the communication costs. *Temporal locality* means that an auction should favor reservations consisting of a run of consecutive allocation intervals. This reduces the overhead involved in saving partial results of an application and then reloading them at a later time. The auctions provide incentives for packages that use services offered by one sub-coalition over a run of consecutive allocation intervals.

One of the challenges is to adapt the well-researched algorithms for combinatorial auctions [14] to a very different environment. Traditional auctions assume a continuous time, whereas we assume that a service can be offered for one or more allocation intervals. Instead of a single seller and many buyers, we have large sets of both sellers and buyers. The sellers are the autonomous service providers which have to form sub-coalitions subject to locality constraints when assembling the resources demanded by the service requests of the buyers. In the combinatorial auctions discussed in the literature, the buyers form coalitions to take advantage of price discounts when buying in large quantities. In the original clock-proxy auction there is one seller and multiple buyers who bid for packages of goods. For example, the airways spectrum in the US is auctioned by the FCC

(Federal Communications Commission) and communication companies bid for licenses. In this case a *package* consist of multiple licenses; the quantities in these auctions are the bandwidth allocated times the population covered by the license. Individual bidders choose to bid for packages during the proxy phase and pay the prices they committed to during the clock phase.

A *sub-coalition* consists of servers willing to provide a set of services. A *consensus algorithm* can be used by the members of a sub-coalition to elect a leader who can negotiate on their behalf. A consensus algorithm could also be used to reach agreement regarding the price for the services and resources the members of a sub-coalition are willing to offer during the auction. Paxos-type consensus algorithms [94] could be used, but the participants are selfish and the algorithms do not prevent manipulation by strategic users. Such algorithms are implemented by existing cloud services such as Zookeeper (*https://zookeeper.apache.org/*).

1.6
Problem Statement

In this thesis we discuss several practical principles and optimizations for cloud resource management. One of the major concerns of cloud resource utilization is the energy consumption of underlying components of the system. Energy consumption of a system does not scale linearly with the performance. Hence, we need to find a mechanism to efficiently distribute the load among a group of servers and allocate resources appropriately to them and then switch the rest of the idle servers to one of the sleep states. But selecting right candidates while dealing with heterogeneous systems is another challenge we face. There might be lot of criteria or parameters involved in selecting the suitable candidate for the individual request. Type matching, request duration, server available capacity are just few of them.

Sometimes a request might not fit into any of the servers in our system and we need to split it among a group of them. Hence, we need to introduce a new cloud delivery model, which has the potential to promote portability and diversity of choice. Therefore, we need homogenous subgroup formation in order to compete or place bids to win the access to the request. Even though it looks like a good solution to our problem, we still need a concrete design and architecture in order to be able to propose this as a solution. We discuss how self-organization will address each of the challenges described above. The approach is bid-centric. The system of heterogeneous cloud resources is dynamically, and autonomically, configured to bid to meet the needs identified in a high-level task or service specification. When the task is completed, or the service is retired, the resources are released for subsequent reuse.

1.7
Roadmap

We discuss all of our concerns and solutions addressed above in the rest of this thesis which has been organized as follows: Operating efficiency of a system and server consolidation are discussed in Chapter 2, Sections 2.2 and 2.3, respectively. The model described in Section 2.4 introduces the operating regimes of operation for processors and the conditions when to switch a server to one of the sleep states described in Section 2.2.3. Load balancing and scaling algorithms suitable for a clustered cloud organization based on the model are presented in Section 2.5; these algorithms aim to optimize the energy efficiency and to balance the load. Simulations experiments and conclusions are covered in Sections 2.6 and 2.7.

In Chapter 3, we compare hierarchical organization based on system monitoring and control with a simple bidding scheme. In this chapter we first present the hierarchical organization and control of existing clouds in Section 3.2 and then, in Section 3.3 we analyze the need for alternative strategies for cloud resource management and discuss market based strategies as an intermediate step towards cloud self-organization and self-management. To compare hierarchical control prevalent in existing clouds with a straightforward bidding scheme we conduct a series of simulation experiments and report the results in Sections 3.4 and 3.5, respectively. Finally, in Section 3.6 we present our conclusions and discuss future work.

In Chapter 4, we propose a two-stage protocol for resource management in a hierarchically organized cloud. The first stage exploits spatial locality for the formation of coalitions of supply agents; the second stage, a combinatorial auction, is based on a modified proxy-based clock algorithm and has two phases, a clock phase and a proxy phase. The clock phase supports price discovery; in the second phase a proxy conducts multiple rounds of a combinatorial auction for the package of services requested by each client. The protocol strikes a balance between low-cost services for cloud clients and a decent profit for the service providers. We also report the results of an empirical investigation of the combinatorial auction stage of the protocol.

In Chapter 5, we discuss the architecture of the system. The architecture we propose has its own limitations, it cannot eliminate all the inefficiencies inherent to virtualization, requires the development of new families of algorithms for resource management and the development of new software. On balance this approach has compelling advantages as we shall see in Section 5.2.

Scalability is a critical property of a large-scale systems. In Chapter 6, we propose a simple scheme for the organization of scale-free systems.

CHAPTER 2
ENERGY-AWARE LOAD BALANCING AND APPLICATION SCALING FOR THE CLOUD ECOSYSTEM

The work reported in this chapter is based on [130]. Computers are not energy proportional systems thus, power consumption of clouds based on over-provisioning is excessive and has a negative ecological impact. Here we introduce an energy-aware operation model used for load balancing and application scaling on a cloud. The basic philosophy of our approach is defining an energy-optimal operation regime and attempting to maximize the number of servers operating in this regime. Idle and lightly-loaded servers are switched to one of the sleep states to save energy. The load balancing and scaling algorithms also exploit some of the most desirable features of server consolidation mechanisms discussed in the literature.

2.1
Motivation And Related Work

In the last few years packaging computing cycles and storage and offering them as a metered service became a reality. Large farms of computing and storage platforms have been assembled and a fair number of Cloud Service Providers (CSPs) offer computing services based on three cloud delivery models SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service).

Expansion of Cloud Services. Cloud elasticity, the ability to use as many resources as needed at any given time, and low cost, a user is charged only for the resources it consumes, represents solid incentives for many organizations to migrate their computational activities to a public cloud.

The number of CSPs, the spectrum of services offered by the CSPs, and the number of cloud users have increased dramatically during the last few years. For example, in 2007 the EC2 was the first service provided by AWS; five years later, in 2012, AWS was used by businesses in 200 countries. Amazon's S3 (Simple Storage Service) has surpassed two trillion objects and routinely runs more than 1.1 million peak requests per second. Elastic MapReduce has launched 5.5 million clusters since May 2010 when the service started [166].

Impact On Energy Consumption. The rapid expansion of the cloud computing has a significant impact on the energy consumption in US and the world. The costs for energy and for cooling large-scale data centers are significant and are expected to increase in the future. In 2006, the 6 000 data centers in the U.S. reportedly consumed 61×10^9 kWh of energy, 1.5% of all electricity consumption in the country, at a cost of $4.5 billion [162]. The energy consumption of data centers and of the network infrastructure is predicted to reach $10,300$ TWh/year (1 TWh = 10^9 kWh) in 2030, based on 2010 efficiency levels [135]. These increases are expected in spite of the extraordinary reduction in energy requirements for computing activities.

Idle and under-utilized servers contribute significantly to wasted energy, see Section 2.2. A 2010 survey [30] reports that idle servers contribute 11 million tons of unnecessary CO_2 emissions each year and that the total yearly costs for idle servers is $19 billion. Recently, Gartner Research [148] reported that the average server utilization in large data-centers is 18%, while the utilization of *x86* servers is even lower, 12%. These results confirm earlier estimations that the average server utilization is in the $10 - 30\%$ range [29].

Load Balancing And Scaling. The concept of "load balancing" dates back to the time when the first distributed computing systems were implemented. It means exactly what the name implies, *to evenly distribute the workload to a set of servers* to maximize the throughput, minimize the response time, and increase the system resilience to faults by avoiding overloading the systems.

An important strategy for energy reduction is concentrating the load on a subset of servers and, whenever possible, switching the rest of them to a state with a low energy consumption. This observation implies that the traditional concept of load balancing in a large-scale system could be reformulated as follows: *distribute evenly the workload to the smallest set of servers operating at optimal or near-optimal energy levels, while observing the Service Level Agreement (SLA) between* *the CSP and a cloud user. An optimal energy level is one when the performance per Watt of power is maximized.*

Scaling is the process of allocating additional resources to a cloud application in response to a request consistent with the SLA. We distinguish two scaling modes, horizontal and vertical scaling. *Horizontal scaling* is the most common mode of scaling on a cloud; it is done by increasing the number of Virtual Machines (VMs) when the load of applications increases and reducing this number when the load decreases. Load balancing is critical for this mode of operation. *Vertical scaling* keeps the number of VMs of an application constant, but increases the amount of resources allocated to each one of them. This can be done either by migrating the VMs to more powerful servers or by keeping the VMs on the same servers, but increasing their share of the server capacity. The first alternative involves additional overhead; the VM is stopped, a snapshot is taken, the file is migrated to a more powerful server, and the VM is restarted at the new site.

Related Work. The alternative to the wasteful resource management policy when the servers are *always on*, regardless of their load, is to develop *energy-aware load balancing and scaling* policies. Such policies combine *dynamic power management* with load balancing and attempt to identify servers operating outside their optimal energy regime and decide if and when they should be switched to a sleep state or what other actions should be taken to optimize the energy consumption. The vast literature on energy-aware resource management concepts and ideas discussed in this chapter includes [8, 29, 31, 32, 33, 55, 56, 93, 103, 135, 160, 161, 162].

Questions Posed by Energy-aware Load Balancing And Application Scaling. Some of the questions posed by energy-aware load balancing and application scaling are: (a) Under what conditions should a server be switched to a sleep state? (b) What sleep state should the server be switched to? (c) How much energy is necessary to switch a server to a sleep state and then switch it back to an active state? (d) How much time it takes to switch a server to a running state from a sleep state? (e) How much energy is necessary for migrating a VM running on a server to another one? (f) How much energy is necessary for starting the VM on the target server? (g) How to choose the target where the VM should migrate to? (h) How much time does it takes to migrate a VM?

The answers to some of these questions depend on the server's hardware and software, including the virtual machine monitor and the operating systems, and change as the technology evolves and energy awareness becomes increasingly more important. In this chapter we are concerned with high-level policies which, to some extent are independent of the specific attributes of the server's hardware and, due to space limitation, we only discuss (a), (b), and (g). We assume that the workload is predictable, has no spikes, and that the demand of an application for additional computing power during an evaluation cycle is limited. We also assume a *clustered organization*, typical for existing cloud infrastructure [25, 68].

Contributions of This Work. There are three primary novelties described in this chapter: (1) a new model of cloud servers that is based on different operating regimes with various degrees of "energy efficiency" (processing power versus energy consumption); (2) a novel algorithm that performs load balancing and application scaling to maximize the number of servers operating in the energy-optimal regime; and (3) analysis and comparison of techniques for load balancing and application scaling using three differently-sized clusters and two different average load profiles.

Models for energy-aware resource management and application placement policies and the mechanisms to enforce these policies such as the ones introduced in this chapter can be evaluated theoretically [8], experimentally [54, 55, 56, 93], through simulation [31, 33, 127], based on published data [17, 30, 81, 82], or through a combination of these techniques. Analytical models can be used to derive high-level insight on the behavior of the system in a very short time but the biggest challenge is in determining the values of the parameters; while the results from an analytical model can give a good approximation of the relative trends to expect, there may be significant errors in the absolute predictions. Experimental data is collected on small-scale systems; such experiments provide useful performance data for individual system components but no insights on the interaction between the system and applications and the scalability of the policies. Trace-based workload analysis such as the ones in [55] and [161] are very useful though they provide information for a particular experimental set-up, hardware configuration, and applications. Typically trace-based simulation need more time to produce results. Traces can also be very large and it is hard to generate representative traces from one class of machines that will be valid for all the classes of simulated machines. To evaluate the energy aware load balancing and application scaling policies and mechanisms introduced in this chapter we chose simulation using data published in the literature [25].

Operating efficiency of a system and server consolidation are discussed in Sections 2.2 and 2.3, respectively. The model described in Section 2.4 introduces the operating regimes of a processors and the conditions when to switch a server to a sleep state. Load balancing and scaling algorithms suitable for a clustered cloud organization based on the model are presented in Section 2.5; these algorithms aim to optimize the energy efficiency and to balance the load. Simulations, experiments and conclusions are covered in Sections 2.6 and 2.7.

2.2
Energy Efficiency of A System

The *energy efficiency* of a system is captured by the ratio "performance per Watt of power". During the last two decades the performance of computing systems has increased much faster than their energy efficiency [29].

2.2.1
Energy Proportional Systems

In an ideal world, the energy consumed by an idle system should be near zero and grow linearly with the system load. In real life, even systems whose energy requirements scale linearly, when idle, use more than half the energy they use at full load. Data collected over a long period of time shows that the typical operating regime for data center servers is far from an optimal energy consumption regime.

An *energy-proportional* system consumes no energy when idle, very little energy under a light load, and gradually, more energy as the load increases. An ideal energy-proportional system is always operating at 100% efficiency [29].

2.2.2 Energy Efficiency of A Data Center; The Dynamic Range of Subsystems

The energy efficiency of a data center is measured by the *power usage effectiveness* (PUE), the ratio of total energy used to power a data center to the energy used to power computational servers, storage servers, and other IT equipment. The PUE has improved from around 1.93 in 2003 to 1.63 in 2005; recently, Google reported a PUE ratio as low as 1.15 [17]. The improvement in PUE forces us to concentrate on energy efficiency of computational resources [33].

The *dynamic range* is the difference between the upper and the lower limits of the energy consumption of a system as a function of the load placed on the system. A large dynamic range means that a system is able to operate at a lower fraction of its peak energy when its load is low.

Different subsystems of a computing system behave differently in terms of energy efficiency; while many processors have reasonably good energy-proportional profiles, significant improvements in memory and disk subsystems are necessary. The largest consumer of energy in a server is the processor, followed by memory, and storage systems. Estimated distribution of the peak power of different hardware systems in one of the Google's datacenters is: CPU 33%, DRAM 30%, Disks 10%, Network 5%, and others 22% [25] .

The power consumption can vary from 45W to 200W per multi-core CPU. The power consumption of servers has increased over time; during the period 2001 - 2005 the estimated average power use has increased from 193 to 225 W for volume servers, from 457 to 675 for mid range servers, and from 5,832 to 8,163 W for high end ones [81]. Volume servers have a price less than $25 K, mid-range servers have a price between $25 K and $499 K, and high-end servers have a price tag larger than $500 K. Newer processors include power saving technologies.

The processors used in servers consume less than one-third of their peak power at very-low load and have a dynamic range of more than 70% of peak power; the processors used in mobile and/or embedded applications are better in this respect. According to [29], the dynamic power range of other components of a system is much narrower: less than 50% for DRAM, 25% for disk drives, and 15% for networking switches. Large servers often use $32 - 64$ dual in-line memory modules (DIMMs); the power consumption of one DIMM is in the 5 to 21 W range.

A server with $2 - 4$ hard disk drives (HDDs) consumes $24 - 48$ W. A strategy to reduce energy consumption by disk drives is concentrating the workload on a small number of disks and allowing the others to operate in a low-power mode. One of the techniques to accomplish this is based on replication [162]. Another technique is based on data migration [63].

A number of proposals have emerged for *energy proportional* networks; the energy consumed by such networks is proportional with the communication load. An example of an energy proportional network is the *InfiniBand*. A network-wide power manager, which dynamically adjusts the network links and switches to satisfy changing datacenter traffic loads, called Elastic Tree is described in [64].

2.2.3
Sleep States

The effectiveness of sleep states in optimizing energy consumption is analyzed in [56]. A comprehensive document [69] describes the advanced configuration and power interface (ACPI) specifications which allow an operating system (OS) to effectively manage the power consumption of the hardware.

Several types of *sleep sates*, are defined: C-states ($C1 - C6$) for the CPU, D-states ($D0 - D3$) for modems, hard-drives, and CD-ROM, and S-states ($S1 - S4$) for the basic input-output system (BIOS). The C-states allow a computer to save energy when the CPU is idle. In a sleep state, the idle units of a CPU have their clock signal and the power cut. The higher the state number, the deeper the CPU sleep mode, the larger the energy saved, and the longer the time for the CPU to return to the state $C0$ which corresponds to the CPU fully operational. The clock signal and the power of different CPU units are cut in states $C1$ to $C3$, while in state $C4$ to $C6$ the CPU voltage is reduced. In state $C1$ the main internal CPU clock is stopped by the software, but the bus interface and the advanced programmable interrupt controller (APIC) are running, while in state $C3$ all internal clocks are stopped, and in state $C4$ the CPU voltage is reduced.

2.2.4
Resource Management Policies For Large-scale Data Centers

These policies can be loosely grouped into five classes: (a) Admission control; (b) Capacity allocation; (c) Load balancing; (d) Energy optimization; and (e) Quality of service (QoS) guarantees. The explicit goal of an admission control policy is to prevent the system from accepting workload in violation of high-level system policies; a system should not accept additional workload preventing it from completing work already in progress or contracted. Limiting the workload requires some knowledge of the global state of the system; in a dynamic system such knowledge, when available, is at best obsolete. Capacity allocation means allocating resources for individual instances; an instance is an activation of a service. Some of the mechanisms for capacity allocation are based on either static or dynamic thresholds [110].

Economy of scale affects the energy efficiency of data processing. For example, Google reports that the annual energy consumption for an Email service varies significantly depending on the business size and can be 15 times larger for a small business than for a large one [54]. Cloud computing can be more energy efficient than on-premise computing for many organizations [17, 118].

2.3
Server Consolidation

The term *server consolidation* is used to describe: (1) switching idle and lightly loaded systems [161] to a sleep state; (2) workload migration to prevent overloading of systems [33]; or (3) any optimization of cloud performance and energy efficiency by redistributing the workload [103].

2.3.1
Server Consolidation Policies

Several policies have been proposed to decide when to switch a server to a sleep state. The *reactive* policy [158] responds to the current load; it switches the servers to a sleep state when the load decreases and switches them to the running state when the load increases. Generally, this policy leads to SLA violations and could work only for slow-varying, predictable loads. To reduce SLA violations one can envision a *reactive with extra capacity* policy when one attempts to have a safety margin and keep running a fraction of the total number of servers, e.g., 20% above those needed for the current load. The *AutoScale* policy [55] is a very conservative *reactive* policy in switching servers to sleep state to avoid the power consumption and the delay in switching them back to running state. This can be advantageous for unpredictable spiky loads.

A very different approach is taken by two versions of *predictive* policies [161]. The *moving window policy* estimates the workload by measuring the average request rate in a window of size Δ seconds uses this average to predict the load during the next second (second $(\Delta + 1)$) and then slides the window one second to predict the load for second $(\Delta + 2)$, and so on. The *predictive linear regression* policy uses a linear regression to predict the future load.

2.3.2
Optimal Policy

In this chapter we define an *optimal* policy as one which guarantees that running servers operate within their optimal energy regime or for limited time in a suboptimal regime, and, at the same time, the policy does not produce any SLA violations. At this time SLAs are very general and do not support strict QoS guarantees, e.g., hard real-time deadlines. Different policies can be ranked by comparing them with an *optimal* policy. The mechanisms to implement energy-aware application scaling and load balancing policies should satisfy several conditions: (i) Scalability - works well for large farms of servers; (ii) Effectiveness - leads to substantial energy and cost savings; (iii) Practicality - uses efficient algorithms and requires as input only data that can be measured with low overhead and, at the same time, accurately reflects the state of the system; and last, but not least, (iv) Consistency with the global objectives and contractual obligations specified by Service Level Agreements.

The workload can be slow- or fast-varying, have spikes or be smooth, can be predicted or is totally unpredictable; the admission control can restrict the acceptance of additional load when the available capacity of the servers is low. What we can measure in practice is the *average energy* used and the average server *setup time*. The setup time varies depending on the hardware and the operating system and can be as large as 260 seconds; the energy consumption during the setup phase is close to the maximal one for the server [55].

The time to switch the servers to a running state is critical when the load is fast varying, the load variations are very steep, and the spikes are unpredictable. The decisions when to switch servers to a sleep state and back to a running state are less critical when a strict admission control policy is in place; then new service requests for large amounts of resources can be delayed until the system is able to turn on a number of sleeping servers to satisfy the additional demand.

2.4
The System Model

The model introduced in this section assumes a clustered organization of the cloud infrastructure and targets primarily the IaaS cloud delivery model represented by AWS. AWS supports a limited number of instance families, including M3 (general purpose), C3 (compute optimized), R3 (memory optimized), I2 (storage optimized), G2 (GPU) and so on. An *instance* is a package of system resources; for example, the `c3.8xlarge` instance provides 32 vCPU, 60 GiB of memory, and 2×320 GB of SSD storage. AWS used to measure the server performance in ECUs (Elastic Compute Units) but has switched recently to, yet to be specified, vCPU units; one ECU is the equivalent CPU capacity of a $1.0 - 1.2$ GHz 2007 Opteron or 2007 Xeon processor.

Applications for one instance family have similar profiles, e.g., are CPU-, memory-, or I/O-intensive and run on clusters optimized for that profile; thus, the application interference with one another is minimized. The normalized system performance and the normalized power consumption are different from server to server; yet, warehouse scale computers supporting an instance family use the same processor or family of processors [25] and this reduces the effort to determine the parameters required by our model.

In our model the migration decisions are based solely on the vCPU units demanded by an application and the available capacity of the host and of the other servers in the cluster. The model could be extended to take into account not only the processing power, but also the dominant resource for a particular instance family, e.g., memory for R3, storage for I2, GPU for G2 when deciding to migrate a VM. This extension would complicate the model and add additional overhead for monitoring the application behavior.

The model defines an energy-optimal regime for server operation and the conditions when a server should be switched to a sleep state. Also the model gives some hints regarding the most appropriate sleep state the server should be switched to and supports a decision making framework for VM migration in horizontal scaling.

2.4.1 Clustered Organization

A cluster $\mathcal{C}$ has a *leader*, a system which maintains relatively accurate information about the free capacity of individual servers in the cluster and communicates with the leaders of the other clusters for the implementation of global resource management policies. The leader could consist of a multi-system configuration to guarantee a fast response time and to support fault-tolerance.

An advantage of a clustered organization is that a large percentage of scheduling decisions are based on local, therefore more accurate, information. Server $\mathcal{S}_k$ makes scheduling decisions every τ_k units of time. The servers in the cluster report to the leader the current load and other relevant state information every τ^i units of time, or earlier if the need to migrate an application is anticipated.

We consider three levels of resource allocation decision making: (a) the local system which has accurate information about its state; (b) the cluster leader which has less accurate information about the servers in the cluster; and (c) global decisions involving multiple clusters. In this chapter we are only concerned with *in-cluster scheduling* coordinated by the leader of the cluster. *Inter-cluster scheduling* is based on less accurate information as the leader of cluster $\mathcal{C}$ exchanges information with other leaders less frequently.

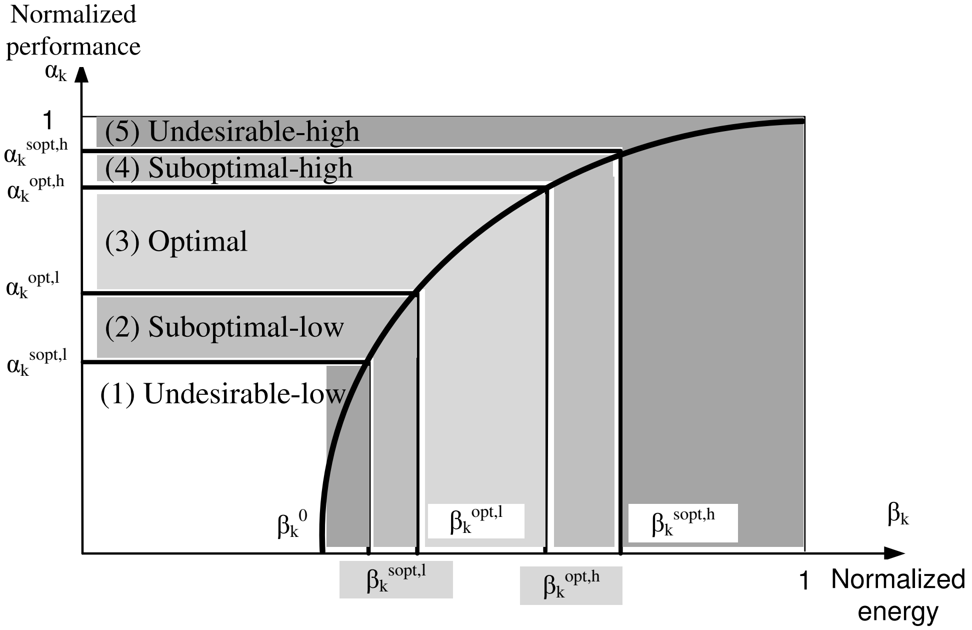


Figure 2.1: Normalized performance versus normalized power; boundaries of the five operating regimes are shown.

2.4.2 System And Application Level Resource Management

The model is based on a two-level decision making process, one at the system and one at the application level. The scheduler of the *Virtual Machine Monitor* (VMM) of a server interacts with the *Server Application Manager (SAM)* component of the VMM to ensure that the QoS requirements of the application are satisfied. SAM gathers information from individual application managers of the VMs running on the server.

2.4.3 Model Parameters

The cluster leader maintains static and dynamic information about all servers in cluster $\mathcal{C}$. *Static information* includes:

$\mathcal{S}_k$ - the serverId;

γ_k - a constant quantifying the processing power of server $\mathcal{S}_k$; the processing power is expressed in vCPUs.

The model illustrated in Figure 2.1 uses several parameters:

$\alpha_k^{sopt,l}$ $\alpha_k^{opt,l}$, $\alpha_k^{opt,h}$, and $\alpha_k^{sopt,h}$, the normalized performance boundaries of different operating regimes.

τ_k - the reallocation interval. Ever τ_k units of time the system determines if and how to reallocate resources.

The application record of application $\mathcal{A}_{i,k}$ includes the *applicationId* and several other parameters:

(1) $a_{i,k}(t)$ - current demand of application $\mathcal{A}_i$ for processing power on server $\mathcal{S}_k$ at time t in vCPU units, e.g., 0.4.

(2) $\lambda_{i,k}$ - highest rate of increase in demand for processing power of application $\mathcal{A}_i$ on server $\mathcal{S}_k$.

(3) $p_{i,k}(t)$ - migration cost.

(4) $q_{i,k}(t)$ - horizontal scaling cost.

The increase in demand for processing power of application $\mathcal{A}_{i,k}$ during a reallocation interval is limited

$$a_{i,k}(t + \tau_k) \leq a_{i,k}(t) + \lambda_{i,k}\tau_k. \tag{2.1}$$

$a_k(t)$, the fraction of processing power of server $\mathcal{S}_k$ used by all active applications at time t is an example of *dynamic information.* This information is reported with period τ^i, or whenever the server determines that it needs to migrate an application or to create additional VMs for an application. To minimize communication costs, the reporting period τ^i is much larger than the rescheduling period of individual clusters. A server $\mathcal{S}_k$ has a computational constant γ_k which quantifies the highest level of performance it can deliver.

2.4.4

Server Operating Regimes

The normalized performance of server $\mathcal{S}_k$ depends on the power level, $\alpha_k(t) = f_k[\beta_k(t)]$. We distinguish five operating regimes of a server, an optimal one, two suboptimal, and two undesirable, Figure 2.1:

$\mathcal{R}_1$ - undesirable-low regime

$$\beta_k^0 \leq \beta_k(t) \leq \beta_k^{sopt,l} \quad \text{and} \quad 0 \leq a_k(t) \leq \alpha_k^{sopt,l}. \tag{2.2}$$

$\mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$, - suboptimal-low, optimal, and suboptimal high regimes

$$\beta^{r1,llim} \leq \beta_k \leq \beta^{r2,hlim} \;\; \text{and} \;\; \alpha^{r1,llim} \leq \alpha_k \leq \alpha^{r2,hlim} \tag{2.3}$$

with:

$\mathcal{R}_2$: r1=sopt, r2=opt, llim=l, hlim=1

$\mathcal{R}_3$: r1=r2=opt, llim=l, hlim=h

$\mathcal{R}_4$: r1=opt, r2=sopt, llim=h, hlim=h

$\mathcal{R}_5$ - undesirable-high regime

$$\beta_k^{sopt,h} \leq \beta_k(t) \leq 1 \quad \text{and} \quad \alpha_k^{sopt,h} \leq a_k(t) \leq 1. \tag{2.4}$$

Our choice of five operating regimes is motivated by the desire to distinguish three types of system behavior in terms of power utilization: optimal, suboptimal, and undesirable. The optimal regime is power efficient and gives a degree of confidence that the sever will not be forced to request VM migrations during the next few scheduling cycles while the sub-optimal regimes are acceptable only for limited periods of time. The five-regime model coupled with delayed actions, e.g., allowing a system to operate within a suboptimal or undesirable-high regimes, could reduce the system overhead and even save power by reducing VM migration costs. One can consider a more refined system behavior, e.g., a seven-regime model, but this adds to the model complexity without clear benefits.

This classification captures the current system load and allows us to distinguish the actions to be taken to return to the optimal regime. A system operating in the suboptimal-low regime is lightly loaded; the server is a candidate for switching to a sleep state. The undesirable-high regime should be avoided because a scaling request would immediately trigger VM migration and, depending on the system load, would require activating one of the servers in a sleep state. This classification also captures the urgency of the actions taken; suboptimal regimes do not require an immediate attention, while the undesirable-low does. The time spent operating in each suboptimal regime is also important.

2.4.5

Measuring Server Energy Efficiency

A recent benchmark [151] compares the energy efficiency of typical business applications running on a Java platform. For example, Table 2.1 based on data reported in Figure 5.3 of [25] shows the results for the SPECpower_ssj2008 benchmark for a server with a single chip 2.83 GHz quad core Intel Xeon processor, 4GB of DRAM, and one 7.2 k RPM 3.5" SATA disk drive.

Table 2.1: The average workload as a percentage of the maximal workload, the power used (P) in Watts, the number of transactions (T), and the computational efficiency (T/P), the ratio of transactions to the average power consumption, from [25]

Load (%)	0	10	20	30	35	40	50	60	70	80	90	100
P	165	180	185	190	195	200	210	220	225	235	240	250
T	0	175	335	484	552	620	738	854	951	1,049	1,135	1,214
T/P	0	0.97	1.81	2.55	2.83	3.10	3.51	3.88	4.23	4.46	4.73	4.84

From Table 2.1 we see that the energy efficiency is nearly-linear. Consider the case when the workload of n servers operating in the $\mathcal{R}_1$ regime migrates to n^{opt} servers already in the $\mathcal{R}_3$ regime and the n servers are forced to a sleep state. The energy saved over an interval of time T, $E^s(T)$, while delivering the same level of performance satisfies the condition

$$E^s(T) \geq n \times \beta_0 \times T - n \times (\bar{p} + \bar{s}) - n^{opt} \times (\beta^{opt,h} - \beta^{opt,l}), \tag{2.5}$$

assuming that all servers have the same: (1) energy consumption when idle, β_0; (2) average migration costs $\bar{p}$; and (3) average setup cost $\bar{s}$. The first term of Equation 2.5 shows that the longer the system operates in this mode, the larger the energy savings. The second term measures the energy used for VM migration and for server setup when they need to be brought back from the sleep state. The last term of Equation 2.5 accounts for the increase in energy consumption due to additional load of the n^{opt} servers operating in the optimal regime. For the example in Table 2.1, $\beta_0 = 165$ W. In [55] the setup time is 260 seconds and during that time the power is consumed at the peak rate of $\bar{s} = 200$ W. There are no reliable estimations of the migration costs. Processors with low-power halt states, such as the C1E state of `x86`, have a lower setup cost [25].

The results in Table 2.1 suggest the following boundaries for the five operating regions:

$$\alpha^{sub,l} = 35\%, \alpha^{opt,l} = 50\%, \alpha^{sub,l} = 80\%, \alpha^{und,h} = 90\% \tag{2.6}$$

These boundaries will be used for the simulation discussed in Section 2.6. The parameters for the five operating regimes $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$ and $\mathcal{R}_5$ corresponding to the boundaries in Equation 2.6 are summarized in Table 2.2.

Table 2.2: The thresholds α, the average power consumption per processor, $\bar{P}$, the average performance measured as the number of transactions, $\bar{T}$, and the ratio $\bar{T}/\bar{P}$ for the five regimes: $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4$ and $\mathcal{R}_5$.

	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$	$\mathcal{R}_4$	$\mathcal{R}_5$
α	0-34.9	35-49.9	50-79.9	80-89.9	90-100
$\bar{P}$	180	202.5	222.5	237.5	245
$\bar{T}$	276	645	894	1,092	1,175
$\bar{T}/\bar{P}$	1.410	3.165	3.985	4.595	4.785

2.4.6

Application Scaling

We assume that the SAM of server $\mathcal{S}_k$ updates every τ_k units of time the $a_{i,k}(t)$ of all applications $\mathcal{A}_{i,k}$ and predicts the consumption at the beginning of the next reallocation interval. The SAM maintains a control data structure including all currently running applications, ordered by $a_{i,k}(t)$. At each reallocation instance, server $\mathcal{S}_k$ determines its available capacity

$$d_k(t) = \alpha_k^{opt,h} - \frac{1}{\gamma_k} \sum_i a_{i,k}(t), \tag{2.7}$$

as well as the largest possible demand for processing capacity at the end of that reallocation interval

$$g_k(t + \tau_k) = \sum_i \left(a_{i,k}(t) + \lambda_{i,k}\tau_k \right). \tag{2.8}$$

2.5
Energy-aware Scaling Algorithms

The objective of the algorithms is to ensure that the largest possible number of active servers operate within the boundaries of their respective optimal operating regime. The actions implementing this policy are: (a) migrate VMs from a server operating in the undesirable-low regime and then switch the server to a sleep state; (b) switch an idle server to a sleep state and reactivate servers in a sleep state when the cluster load increases; (c) migrate the VMs from an overloaded server, a server operating in the undesirable-high regime with applications predicted to increase their demands for computing in the next reallocation cycles.

The clustered organization allows us to accommodate some of the desirable features of the strategies for server consolidation discussed in Section 2.3. For example, when deciding to migrate some of the VMs running on a server or to switch a server to a sleep state, we can adopt a conservative policy similar to the one advocated by autoscaling [55] to save energy. Predictive policies, such as the ones discussed in [161] will be used to allow a server to operate in a suboptimal regime when historical data regarding its workload indicates that it is likely to return to the optimal regime in the near future.

The cluster leader has relatively accurate information about the cluster load and its trends. The leader could use predictive algorithms to initiate a gradual wake-up process for servers in a deeper sleep state, $C4 - C6$, when the workload is above a “high water mark” and the workload is continually increasing. We set up the high water mark at 80% of the capacity of active servers; a threshold of 85% is used for deciding that a server is overloaded in [59], based on an analysis of workload traces. The leader could also choose to keep a number of servers in $C1$ or $C2$ states because it takes less energy and time to return to the $C0$ state from these states. The energy management component of the hypervisor can use only local information to determine the regime of a server.

2.5.1
Scaling Decisions

The Server Application Manager SAM$_k$ is a component of the Virtual Machine Monitor (VMM) of a server $\mathcal{S}_k$. One of its functions is to classify the applications based on their processing power needs over a window of w reallocation intervals in several categories: rapidly increasing resource demands (RI), moderately increasing (MI), stationary (S), moderately decreasing (MD), and rapidly decreasing (RD). This information is passed to the cloud leader whenever there is the need to migrate the VM running the application.

SAM$_k$ interacts with the cluster leader and with the application managers of servers accepting the migration of an application currently running on server $\mathcal{S}_k$. A report sent to the cluster leader includes the list of applications currently running on $\mathcal{S}_k$, their additional demands of over the last reallocation cycle and over a window of w reallocation intervals, and their classification as RI/MI/S/MD/RD over the same window. The scaling decisions are listed in the order of their energy consumption, overhead, and complexity:

(1) Local decision - whenever possible, carry out vertical application scaling using local resources only.

(2) In-cluster, horizontal or vertical scaling - migrate some of the VMs to the other servers identified by the leader; wake-up some of the servers in a sleep state or switch them to one of the sleep states depending on the cluster workload.

(3) Inter-cluster scaling - when the leader determines that the cluster operates at 80% of its capacity with all servers running, the admission control mechanism stops accepting new applications. When the existing applications scale up above 90% of the capacity with all servers running then the cluster leader interacts with the leaders of other clusters to satisfy the requests. This case is not addressed in this chapter.

All decisions take into account the current demand for processor capacity, as well as the anticipated load.

I. Local, vertical scaling. The anticipated load at the end of the current and the following scheduling cycle allow the server to continue operating in the optimal regime

$$g_k(t+\tau_k) \leq \gamma_k a_k^{opt,h} \ \& \ \gamma_k a_k^{opt,l} \leq g_k(t+2\tau_k) \leq \gamma_k a_k^{opt,h}. \tag{2.9}$$

II. In-cluster scaling. The anticipated load could force the server to transition to the suboptimal-low, suboptimal-high, or undesirable-high regimes, respectively:

$$\begin{gathered} g_k(t+\tau_k) \leq \gamma_k \alpha_k^{opt,h} \ \& \ \gamma_k \alpha_k^{sopt,l} \leq g_k(t+2\tau_k) \leq \gamma_k \alpha_k^{opt,l} \\ g_k(t+\tau_k) \leq \gamma_k \alpha_k^{opt,h} \ \& \ \gamma_k \alpha_k^{opt,h} \leq g_k(t+2\tau_k) \leq \gamma_k \alpha_k^{sopt,h} \\ g_k(t+\tau_k) \leq \gamma_k \alpha_k^{opt,h} \ \& \ g_k(t+2\tau_k) > \gamma_k \alpha_k^{sopt,h}. \end{gathered} \tag{2.10}$$

The leader includes the server in the `WatchList` when Equations 2.10 are satisfied.

III. Inter-cluster scaling. The anticipated load could force the server to transition to the undesirable-low regime.

$$g_k(t+\tau_k) \leq \gamma_k \alpha_k^{opt,h} \quad \& \quad g_k(t+2\tau_k) \leq \gamma_k \alpha_k^{sopt,l}. \tag{2.11}$$

The leader includes the server in the `MigrationList` in case of Equation 2.11.

In addition to the lazy approach discussed in (II) when a server operates within the boundaries of the suboptimal-high regime until its capacity is exceeded, one could use an *anticipatory* strategy. To prevent potential SLA violations in the immediate future, in this case we force the migration from the suboptimal-high region as soon as feasible.

A. *Synchronous operation.* The algorithm executed by SAM$-k$ every τ_k units of time is:

```
1. Order applications based on the demand. Compute the actual rate of increase or decrease in demand over a window of w reallocation cycles according to Equation 6. Compute servers available capacity.
2. If Equations 9 or 11 are  satisfied send an imperative request for application migration.
3. If first or third equations in 10 are satisfied send a warning  including application history.
4. Else, reallocate CPU cycles to allow each application its largest  rate of increase. If time elapsed from the last status report is larger than in-cluster reporting period send an update to the leader.
```

A server operating in the undesirable-high regime has a high energy efficiency, but gives us a warning that we should be prepared to activate servers in the sleep states and that VM migration is very likely.

B. *Asynchronous operation.* When a server receives a message related to VM migration, its SAM reacts immediately:

B.1. When $\mathcal{S}_k$ receives a request from the cluster leader to accept the migration or vertical scaling on an application, it first checks that by accepting the request it will still be operating on an optimal regime. If so, it sends an accept message to the leader and to $\mathcal{S}_v$, the server requesting the migration or vertical scaling of application. In the former case it starts one or more VMs for the application; in the later case it waits to receive from $\mathcal{S}_v$ the snapshot of the VM image and then starts the new VMs.

B.2. When, in response to a report of operation in a suboptimal regime, server $\mathcal{S}_k$ receives an accept message for vertical scaling of application $\mathcal{A}_{i,k}$ from another server, $\mathcal{S}_v$, it stops the application, constructs the image of the VM running the application, and then sends it to $\mathcal{S}_v$. For horizontal scaling, it sends $\mathcal{S}_v$ the location of the image.

2.5.2

Cluster Management

The leader of a cluster maintains several control structures:

`OptimalList` - includes servers operating in an optimal regime or those in suboptimal regime projected to migrate back to the optimal regime; the list is ordered in the increasing order of computing power. Within a group of servers with similar γ_k, the servers are ordered in the increasing order of available capacity.

`WatchList` - includes servers running in two suboptimal regime and the undesirable-high regimes whose applications are candidates for VM migration.

`MigrationList` - includes servers operating in undesirable regimes. The leader selects candidates for migration and identifies possible targets for migration.

`SleepList` - includes servers in one of the sleep states; the list is ordered on the type of sleep state and then in the increasing order of computing power reflected by the constant γ_k.

The leader performs several functions: admission control for new applications, server consolidation and reactivation, and VM migration.

1. Admission control. When the cluster leader receives a request to accept a new application it computes the available capacity and admits it if the system is not overloaded

$$\frac{d_{\mathcal{C}}(t)}{\sum_{k=1}^{n_{\mathcal{C}}^{a}} \gamma_k} \leq 0.8 \quad \text{with} \quad d_{\mathcal{C}}(t) = \sum_{k=1}^{n_{\mathcal{C}}^{a}} d_k(t) \tag{2.12}$$

with $d_{\mathcal{C}}(t)$ is the available capacity and $n_{\mathcal{C}}^{a}$ represents the number of servers operating in the optimal and suboptimal regimes, those included in the `OptimalList` and the `WatchList`. If the `SleepList` is not empty, the leader has the choice to place the application on a standby queue and then wake up one or more servers in this list and assign the application to these servers.

2. Server consolidation and re-activation. The leader examines each server in the `MigrationList` and attempts to pair those operating in the lower undesirable regime with the ones in the upper undesirable regime and to migrate applications from the later to the former ones taking into account the application profile. Servers left to operate in the lower undesirable region are then ordered to switch to one of the sleep states, depending on the current system load and the predicted future load. These servers are added to the `SleepList`.

When the server load reaches the *high water mark*, then the leader initiates gradual server transitions from deeper sleep states to the lighter sleep states and reactivation of some servers in state C1.

3. VM migration. A VM can only be migrated at the time when checkpointing is possible and a consistent cut [110] can be defined. The leader acts as a broker for the selection of a target for horizontal or vertical scaling of application $\mathcal{A}$. Once it receives a request for in-cluster scaling it first identifies a potential target; then the two servers, the one sending the request and the one accepting to be the target for horizontal or vertical scaling, negotiate the VM migration. Once an agreement has been reached, the two servers carry out the operation without the intervention of the leader. The target selection is guided by two objectives:

(i) Ensure that the selected target server will be able to accommodate application scaling for an extended period of time, while operating in its optimal regime; this will help reduce the migration costs and the power consumption.

(ii) Keep the user costs low by selecting the least costly server, the one with the lowest γ_k that satisfies condition (i).

The strategy is to consider a window of future intervals, ϕ, and determine the largest possible increase in resource demand of application $\mathcal{A}_i$.

$$a_i(t + \phi \times \lambda_i) = a_i(t) + \phi\lambda_i \tag{2.13}$$

Then search the *WatchList* for a server operating in the lower suboptimal regime with suitable available capacity

$$d_v(t) > a_i(t + \phi \times \lambda_i). \tag{2.14}$$

If such a server does not exist, then server $\mathcal{S}_u$ from *SleepList* with the lowest γ_u is selected if it satisfies the conditions

$$a_i(t) \geq \gamma_u \alpha_u^{sopt,l} \quad \text{and} \quad a_i(t + \phi \times \lambda_i) \leq \alpha_u^{sopt,l}. \tag{2.15}$$

The algorithms assume that the thresholds for normalized performance and power consumption of server $\mathcal{S}_k$ are constant. When the processor supports dynamic voltage and frequency scaling [160] the thresholds, $\alpha_k^{opt,l}(t)$, $\alpha_k^{opt,high}(t)$, $\alpha_k^{sopt,l}(t)$, $\alpha_k^{sopt,high}(t)$, $\beta_k^{opt,l}(t)$, $\beta_k^{opt,high}(t)$, $\beta_k^{sopt,l}(t)$, $\beta_k^{sopt,high}(t)$ will vary in time. The basic philosophy will be the same, we shall attempt to keep every server in an optimal operating regime. An additional complication of the algorithms is that we have to determine if it is beneficial to increase/decrease the power used thus, push up/down the thresholds of the operating regimes of the server. We still want to make most scaling decisions locally.

2.6 Simulation Experiments

The effectiveness of an energy-aware load balancing and scaling algorithm is characterized by its computational efficiency, the number of operations per Watt of power, and by its negative impact reflected by the potential SLA violations. In our study we shall use the data in Table 2.2 based on the measurements reported in Table 2.1 from [25]. These measurements are for a transaction processing benchmark, *SPECpower_ssj2008*, thus, the computational efficiency will be the number of transactions per Watt. We define the ideal computational efficiency as the number of transactions when all servers operate at the upper limit of the optimal region, at 80% load, and no SLA violations occur. In this case

$$\left(\frac{T}{P}\right)_{ideal} = \frac{T_{80\%}}{P_{80\%}} = \frac{1049}{235} = 4.46 \; transactions/Watt. \tag{2.16}$$

We conduct a simulation study to evaluate the effectiveness of the algorithms discussed in Section 2.5.

Table 2.3: The effects of application scaling and load balancing algorithm on a system with the parameters described in Table 2.2. We experimented with two average system loads, 30% and 70% of the server capacity and three different cluster sizes, $10^2, 10^3$, and 10^4. The data before/after the application of the algorithm are: (a) the number of servers in each one of the five operating regimes, $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4, \mathcal{R}_5$ and in the sleep state (slp) (columns 3-8); (b) $\mathbb{P}$ - the average power consumption per processor in Watts (column 9); (c) $\mathbb{T}$ - the performance measured as the average number of transactions per processor (column 10); and (d) the average ratio $\mathbb{T}/\mathbb{P}$ (column 11).

Load	Size	# in $\mathcal{R}_1$	# in $\mathcal{R}_2$	# in $\mathcal{R}_3$	# in $\mathcal{R}_4$	# in $\mathcal{R}_5$	# in slp
30%	10^2	65/0	35/27	0/64	0/2	0/3	0/4
	10^3	550/0	450/300	0/598	0/20	0/64	0/18
	10^4	5500/0	4500/3050	0/5950	0/50	0/319	0/631
70%	10^2	0/0	0/58	20/35	80/2	0/5	0/0
	10^3	0/0	0/430	190/490	810/20	0/56	0/4
	10^4	0/0	0/4000	2000/4500	8000/250	0/233	0/17

Load	Size	$\mathbb{P}$	$\mathbb{T}$	$\mathbb{T}/\mathbb{P}$
30%	10^2	188/197	264/746	1.4/3.8
	10^3	190/214	442/825	2.3/3.8
	10^4	190/203	442/771	2.3/3.8
70%	10^2	234/204	1,052/742	4.5/3.6
	10^3	234/215	1,054/823	4.5 /3.7
	10^4	235/193	1,052/715	4.5/3.7

The simulation experiments reported in this chapter were conducted on the Amazon cloud; an *c3.8xlarge* EC2 instance with 60 G memory and 32 cores was used.

The study will give us some indications about the operation of the algorithm in clusters of different sizes and subject to a range of workloads. The metrics for assessing the effectiveness and the overhead of the algorithms are:

(i) The evolution of the number of servers in each of the five operating regimes as a result of the load migration mandated by the algorithm.

(ii) The computational efficiency before and after the application of the algorithm.

(iii) The ratio of local versus in-cluster scaling decisions during simulation. This reflects the overhead of the algorithm.

For simplicity we chose only two sleep states C_3 and C_6 in the simulation. If the load of the cluster is more than 60% of the cluster capacity we do not choose the C_6 state because the probability that the system will require additional resources in the next future is high. Switching from the C_6 state to C_0 requires more energy and takes more time. On the other hand, when the cluster load is less than 60% of its capacity we choose the C_6 state because it is unlikely that the server will be reactivated in the next future.

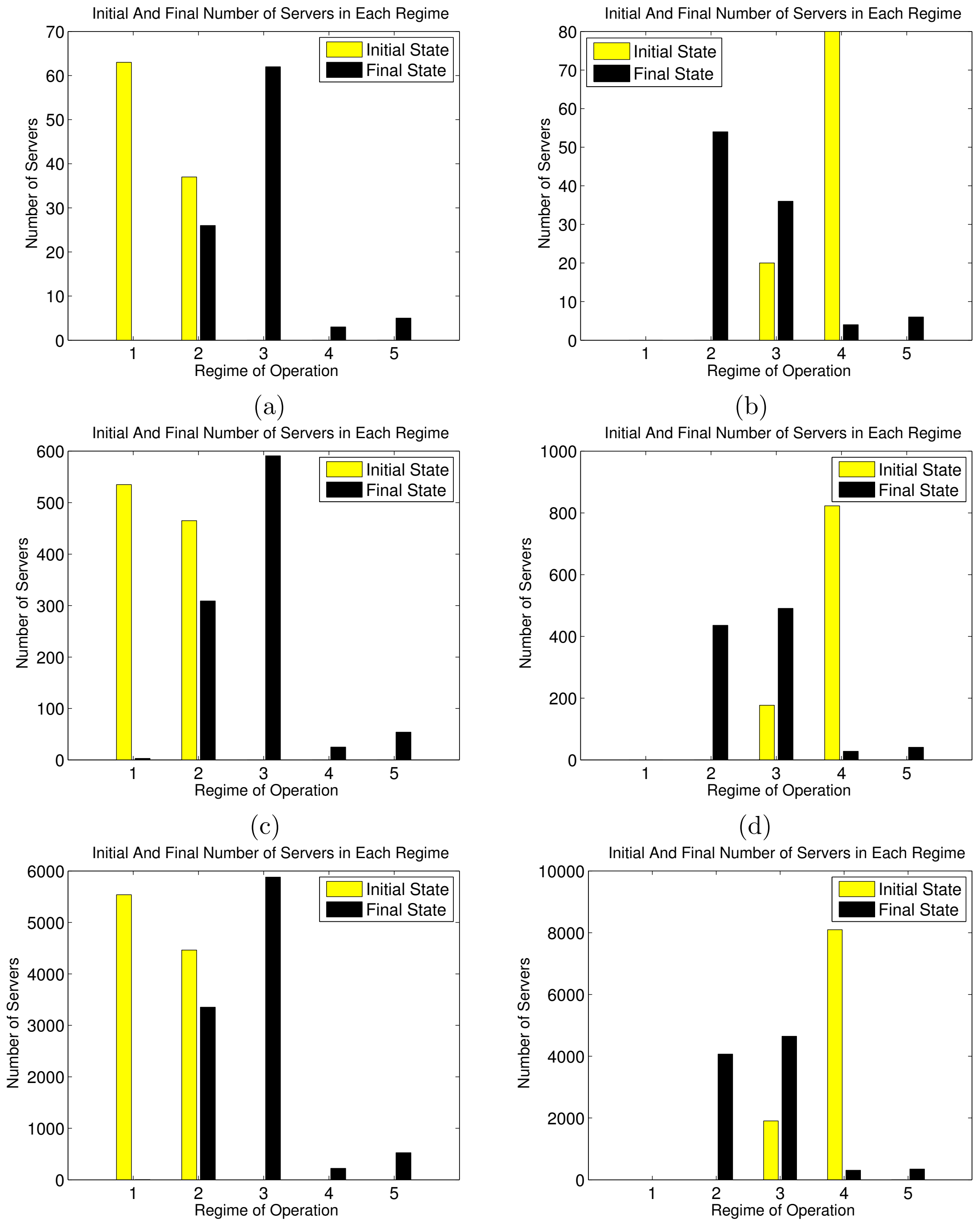
The simulation uses Amazon Web Services (AWS). AWS offers several classes of services; the servers in each class are characterized by the architecture, CPU execution rate, main memory, disk space, and I/O bandwidth. The more powerful the server, the higher the cost per hour for the class of service.

Table 2.3 summarizes the effects of application scaling and load balancing algorithm on a cluster when the parameters of the servers are given in Table 2.2 and the application is a transaction processing system. In a transaction processing system there is no workload migration, a front-end distributes the transactions to the servers thus, we did not include migration costs. To assess the power consumption and the performance measured by the number of transactions we use average values for each regime. Recall that in this case the boundaries of the five operating regimes are given in Equation 2.6.

2.6.1
The Effect of The System Load

In [127] we report on simulation experiments designed to evaluate the effectiveness of the algorithms for load balancing and energy optimization during application scaling. One of the questions we addressed was whether the system load has an effect on the resource management strategy to force the servers in a cluster to operate within the boundaries of the optimal regime. The experiments we report in this section are for clusters with $10^2, 10^3$, and 10^4 servers consisting of multiple racks of a WSC. For each cluster size we considered two load distributions:

(i) Low average load - an initial load uniformly distributed in the interval $20 - 40\%$ of the server capacity. Figure 2.2 (e) shows the distribution of the number of servers in the five operating regimes for clusters with 10^4 servers, before and after load balancing.

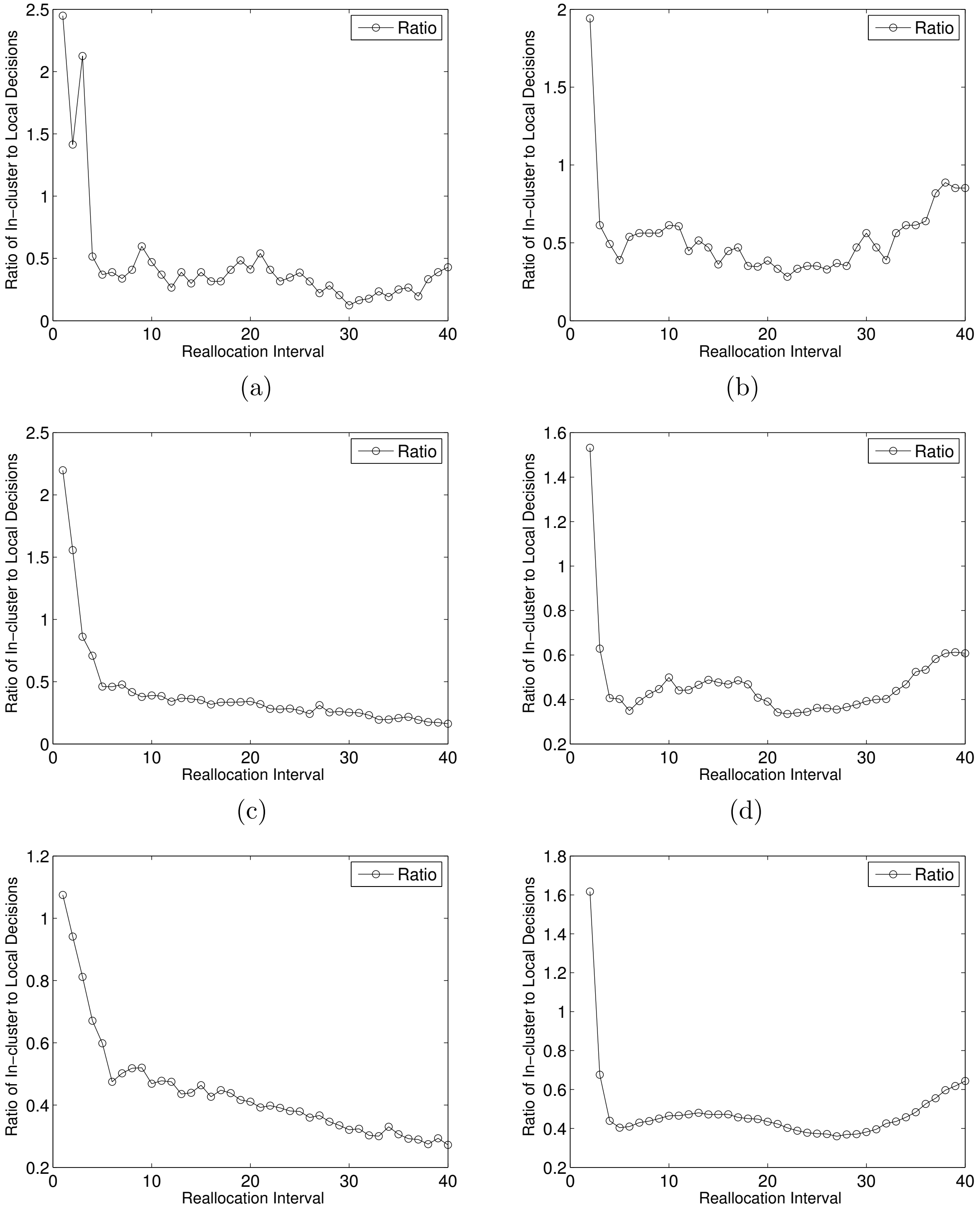


(a) (b)

(c) (d)

(e) (f)

Figure 2.2: The effect of average server load on the distribution of the servers in the five operating regimes, $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$ and $\mathcal{R}_5$, before and after energy optimization and load balancing. Average load: (a),(c),(e) 30% and (b),(d),(e) 70%. Cluster size: (a),(b) 10^2, (c),(d) 10^3 and (e),(f) 10^4.



(a) (b)

(c) (d)

(e) (f)

Figure 2.3: Time series of in-cluster to local decisions ratios for a transient period of 40 reallocation intervals. Average load: 30% of the server capacity in (a), (c), and (e); 70% in (b), (d), and (f). The cluster size: 10^2 in (a) and (b); 10^3 in (c) and (d); 10^4 in (e) and (f). After 40 reallocation intervals almost double rations when the load is 70% versus 30%.

The distributions for 10^2 and 10^3 servers in a cluster are similar. When the average server load is 30% of their capacity, the algorithm is very effective; it substantially improves the energy efficiency to 3.8 from as low as 1.4. After load balancing, the number of servers in the optimal regime increases from 0 to about 60% and a fair number of servers are switched to the sleep state.

(ii) High average load - initial server load uniformly distributed in the $60 - 80\%$ of the server capacity. Figure 2.2 (f) shows that when the average server load is 70% of their capacity, the average computational efficiency decreases from about 4.5 to 3.6 as many servers, about 80% of them, are forced from $\mathcal{R}_4$ to the $\mathcal{R}_2$ and $\mathcal{R}_3$ regimes to reduce the possibility of SLA violations. Initially, no servers operated in the $\mathcal{R}_5$ regime. In this experiment we did not use the anticipatory strategy and allowed servers to operate in the $\mathcal{R}_5$ regime after load balancing; as a result, a small fraction of servers ended up in the $\mathcal{R}_5$ regime. There is a balance between computational efficiency and SLA violations; the algorithm can be tuned to maximize computational efficiency or to minimize SLA violations according to the type of workload and the system management policies.

These results are consistent with the ones reported in [127] for smaller cluster sizes, $20, 40, 60$, and 80 servers. This shows that the algorithms operate effectively for a wide range of cluster sizes and for lightly, as well as, heavily loaded systems.

Table 2.4: Average and standard deviation of in-cluster to local decisions ratio for 30% and 70% average server load for three cluster sizes, 10^2, 10^3, and 10^4.

Cluster size	Average load	Average ratio	Standard deviation	Average load	Average ratio	Standard deviation
10^2	30%	0.69	0.57	70%	0.51	0.89
10^3	30%	0.33	0.21	70%	0.58	0.92
10^4	30%	0.49	0.27	70%	0.53	0.98

2.6.2 High-cost Versus Low-cost Application Scaling

Cloud elasticity allows an application to seamlessly scale up and down. In the next set of simulation experiments we investigate horizontal and vertical application scaling.

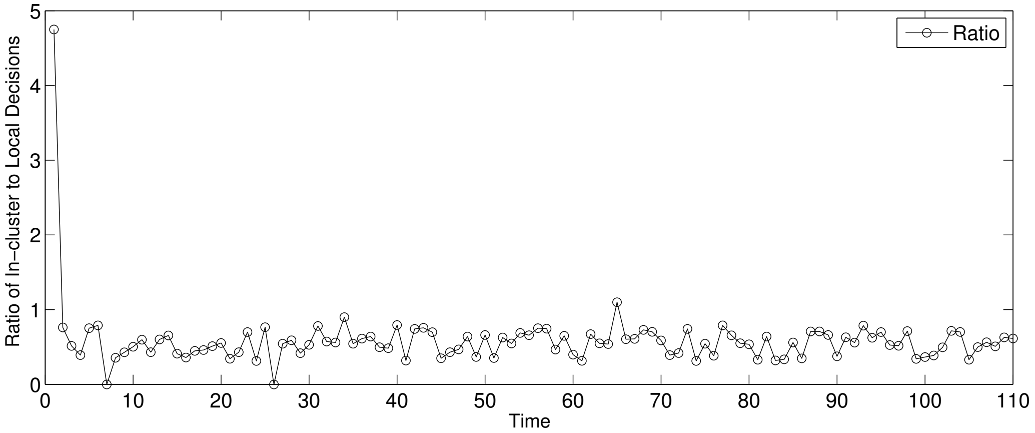


Figure 2.4: The ratio of in-cluster to local decisions in response to scaling requests versus time for a cluster with 40 servers when the average workload is 50% of the server capacity.

Horizontal scaling requires the creation of additional VMs to run the application on lightly loaded servers. In-cluster, horizontal scaling incurs higher costs than vertical scaling for load balancing. The higher costs in terms of energy consumption and time are due to the communication with the leader to identify the potential targets and then to transport the VM image to one or more of them. Local, *vertical scaling*, allows the VM running an application to acquire additional resources from the local server; local vertical scaling has lower costs, but it is only feasible if the server has sufficient free capacity.

The average ratio of high-cost in-cluster horizontal scaling to low-cost local vertical scaling for a transient period of 40 reallocation intervals are summarized in Table 2.4 and in Figures 2.3 (a), (b), (c), (d), and (e). When the average workload is 30% the algorithm works better; the larger the cluster, the more effectively the algorithm handles the application scaling as most decisions are done locally. After 40 reallocation intervals the ratios for the two average workload are 0.3 and 0.7, respectively, for cluster size 10^4; this shows that at higher workload VM migrations are more intense. The trends are different for the two cases, the ratio decreases in time when the average load is 30% and increases when the average load is 70%; we expect these trends to continue in the steady-state regime.

We carried out measurements of computational efficiency on a system with a 3GHz Intel Core i7 with 8 cores and a solid state disk. The system was running under OS X Yosemite. The application used for our measurements runs under `Xcode 6`, an integrated development environment containing a suite of software tools for OS X and iOS. The application is I/O intensive, it reads a record from the secondary storage carries out some trivial computations and writes back the record. `Xcode` allowed us to measure the normalized performance and the corresponding normalized power consumption and thus, to calculate the computational efficiency at the boundaries of the five operating regimes described by Equation 2.6. `Xcode` reports only the power used by the cores running the application, so the average computational efficiency we were able to measure refers only to the processor, rather than the entire system.

We then considered a cloud infrastructure consisting of $10,000$ servers identical with the system we have measured. We used the distribution of the servers in each of the five regimes in Table 2.3 to compute the computational efficiency before and after the application of the algorithm.

Table 2.5: The effects of application scaling and load balancing algorithm on a system with 10^4 servers 3GHz Intel Core i7. Shown are the number of servers in each one of the five operating regimes before and after the application of the algorithm according to Table 2.3 and the average computational efficiency in each regime $\bar{C}_{ef}^{\mathcal{R}_i}, \quad 1 \le i \le 5$ determined by our measurements. $\bar{C}_{ef}$ shows the average computational efficiency before and after the application of the algorithm.

Load	# in $\mathcal{R}_1$ $\bar{C}_{ef}^{\mathcal{R}_1}= 0.725$	# in $\mathcal{R}_2$ $\bar{C}_{ef}^{\mathcal{R}_2}= 1.420$	# in $\mathcal{R}_3$ $\bar{C}_{ef}^{\mathcal{R}_3}= 1.245$	# in $\mathcal{R}_4$ $\bar{C}_{ef}^{\mathcal{R}_4}= 1.055$	# in $\mathcal{R}_5$ $\bar{C}_{ef}^{\mathcal{R}_5}= 1.050$	$\bar{C}_{ef}$
30%	5500/0	4500/3050	0/5950	0/50	0/319	1.03/1.21
70%	0/0	0/4000	2000/4500	8000/250	0/233	1.09/1.18

The results are summarized in Table 2.5. We see that the average computational efficiency decreases from 1.42 in $\mathcal{R}_2$ to 1.245 in $\mathcal{R}_3$, and 1.055 in $\mathcal{R}_4$. As a result, the computational efficiency due to application of our algorithm increases only from 1.03 to 1.21 for the lightly loaded system and shows a modest increase from 1.09 to 1.18 for the heavy load case. As noted in Section 2.2, the processors consume less than one-third of their peak power at the very-low load thus, the actual improvement due to the application of our algorithm should be considerably higher.

Finally, we attempted to carry out measurements in a realistic cloud environment and we investigated the possibility of using EC2 instances. First, we realized that no Mac OS-based AMIs (Amazon Machine Images) are supported, so we could not use `Xcode`. We then discovered that the AWS virtual machine monitors prevent both `Linux` and `Ubuntu` AMIs to collect information related to power consumption. As a result, tools such as `dmidecode` or `lshw` used to report hardware information by monitoring kernel data structures return the fields related to power consumption as *unknown*. This lack of transparency makes the investigation of energy consumption in cloud environments a rather challenging task.

2.7
Conclusions And Future Work

The realization that power consumption of cloud computing centers is significant and is expected to increase substantially in the future motivates the interest of the research community in energy-aware resource management and application placement policies and the mechanisms to enforce these policies. Low average server utilization [148] and its impact on the environment [30] make it imperative to devise new energy-aware policies which identify optimal regimes for the cloud servers and, at the same time, prevent SLA violations.

A quantitative evaluation of an optimization algorithm or an architectural enhancement is a rather intricate and time-consuming process; several benchmarks and system configurations are used to gather the data necessary to guide future developments. For example, to evaluate the effects of architectural enhancements supporting Instruction-level or Data-level Parallelism on the processor performance and their power consumption several benchmarks are used [68]. The results show different numerical outcomes for the individual applications in each benchmark. Similarly, the effects of an energy-aware algorithm depend on the system configuration and on the application and cannot be expressed by a single numerical value.

Research on energy-aware resource management in large-scale systems often use simulation for a quasi-quantitative and, more often, a qualitative evaluation of optimization algorithms or procedures. As stated in [25] “First, they (WSCs) are a new class of large-scale machines driven by a new and rapidly evolving set of workloads. Their size alone makes them difficult to experiment with, or to simulate efficiently.” It is rather difficult to experiment with the systems discussed in this chapter and this is precisely the reason why we choose simulation.

The results of the measurements reported in the literature are difficult to relate to one another. For example, the wake-up time of servers in the sleep state and the number of servers in the sleep state are reported for the AutoScale system [55]; yet these figures would be different for another processor, system configuration, and application.

We choose computational efficiency, the ratio of the amount of normalized performance to normalized power consumption, as the performance measure of our algorithms. The amount of useful work in a transition processing benchmark can be measured by the number of transactions, but it is more difficult to assess for other types of applications. SLA violations in a transaction processing benchmark occur only when the workload exceeds the capacity of all servers used by the application, rather than the capacity of individual servers. Thus, in our experiment there are no SLA violation because there are servers operating in low-load regimes.

We need to balance computational efficiency and SLA violations; from Table 2.3 we see that the computational efficiency increases up to $3.6 - 3.8$, while the optimum is 4.46 transactions/Watt. Figure 2.2 (f) and Table 2.3 show that the computational efficiency decreases after the application of the algorithm to a system with the average load 70% because servers operating in the suboptimal-high regime are forced to reduce their workload. The *lazy* approach discussed in Section 2.5 would eliminate this effect.

Even the definition of an ideal case when a clairvoyant resource manager makes optimal decisions based not only on the past history, but also on the knowledge of the future can be controversial. For example, we choose as the ideal case the one when all servers operate at the upper boundary of the optimal regime; other choices for the ideal case and for the bounds of the five regimes could be considered in case of fast varying, or unpredictable workloads.

The five-regime model introduced in this chapter reflects the need for a balanced strategy allowing a server to operate in an optimal or near-optimal regime for the longest period of time feasible. A server operating in the optimal regime is unlikely to request a VM migration in the immediate future and to cause an SLA violation, one in a sub-optimal regime is more likely to request a VM migration, while one in the undesirable-high regime is very likely to require VM migration. Servers in the undesirable-low regime should be switched to a sleep state as soon as feasible.

The model is designed for clusters built with the same type of processors and similar configurations; the few parameters of the model are then the same for all the servers in the cluster. The clustered organization allows an effective management of servers in the sleep state as they should be switched proactively to a running state to avoid SLA violations. It also supports effective admission control, capacity allocation, and load balancing mechanisms as the cluster leader has relatively accurate information about the available capacity of individual servers in the cluster.

Typically, we see a transient period when most scaling decisions require VM migration, but in a steady-state, local decisions become dominant. Table 2.4 shows that the average system load affects the average ratio of in-cluster to local decisions thus, the migration costs, increases from 0.33 to 0.55 for a cluster with 10^3 servers when the average system load increases from 30% to 70%. As pointed out in [25] there is a real need for energy benchmarks such as the one in [151] which could be used to guide the design choices for new systems.

From the large number of questions posed by energy-aware load balancing policies we restrict our analysis to the conditions when a server should be switched to a sleep state and the choice of the sleep state, and how to chose the target server where to move a VM. The mechanisms for load balancing and application scaling policies are consistent with the requirements discussed in Section 2.3, scalability, effectiveness, practicality, and consistency with global system objectives.

Though designed specifically for the IaaS cloud delivery model, the algorithms discussed in Section 2.5 can be adapted for the other cloud delivery models. Typically, PaaS applications run for extended periods of time and the smallest set of servers operating at an optimal power level to guarantee the required turnaround time can be determined accurately.

Our future work will evaluate the overhead and the limitations of the algorithm proposed in this chapter; it will also include the implementation of a Server Application Manager and the evaluation of the overhead for the algorithm proposed in this chapter. The algorithm will be incorporated in the self-management policies introduced in [111].

CHAPTER 3
HIERARCHICAL CONTROL VS A MARKET MODEL

The work reported in this chapter is based on [112]. Assume we have a million servers, how do we organize them? How to minimize the cost and optimize the shared resources utilization while we are connecting them together? In this chapter we investigate a hierarchically organized cloud infrastructure and compare distributed hierarchical control based on resource monitoring with market policies for resource management. Mechanisms for implementing market policies do not require a model of the system, incur a low overhead, are robust, and satisfy several other desiderates of autonomic computing. We introduce several performance measures and report on extensive simulation studies which show that a straightforward bidding scheme supports an effective admission control mechanism, while reducing the communication complexity by several orders of magnitude and also increasing the acceptance rate than hierarchical control and monitoring mechanisms. Resource management based on market-based policies can be seen as an intermediate step towards cloud self-organization, an ideal alternative to current policies and mechanisms for cloud resource management.

3.1
Introduction

Cloud computing is a form of utility computing, a service delivery model in which a provider makes computing resources and infrastructure management available to the customer as needed and charges them for the actual resource consumption. Today, utility computing, envisioned by John Mc. Carthy and others is a social and technical reality.

Cloud computing emulates traditional utilities such as electricity, water, and gas, and attempts to reduce costs by assembling large pools of computing resources into data centers. These data centers take advantage of the latest computing and communication technologies, exploit economies of scale, reduce the overhead for delivering services, and provide an appealing computing environment for a large user population. Over the last decade cloud service providers (CSPs) such as Amazon, Google, and Microsoft, have built data centers at unprecedented scales. It is estimated that in 2013 these three CSPs were running roughly one million servers each.

The policies for cloud resource management can be loosely grouped into five classes: (1) admission control; (2) capacity allocation; (3) load balancing; (4) energy optimization; and (5) quality of service (QoS) guarantees.

The explicit goal of an *admission control* policy is to prevent the system from accepting workload in violation of high-level system policies [61]. Limiting the workload requires some knowledge of the global state of the system. *Capacity allocation* means to allocate resources for individual instances; an instance is an activation of a service. Locating resources subject to multiple global optimization constraints requires a search in a very large search space when the state of individual systems changes rapidly.

Load balancing and *energy optimization* are correlated and affect the cost of providing the services; they can be done locally, but global load balancing and energy optimization policies encounter the same difficulties as the the capacity allocation [80]. *Quality of service* is probably the most challenging aspect of resource management and, at the same time, possibly the most critical for the future of cloud computing.

The resource management policies must be based on a disciplined approach, rather than ad hoc methods. Basic mechanisms for the implementation of resource management policies are:

Control theory. Control theory uses the feedback to guarantee system stability and to predict transient behavior [80], but can be used only to predict local, rather than global behavior; applications of control theory to resource allocation are covered in [47]. Kalman filters have been used for unrealistically simplified models as reported in [77], and the placement of application controllers is the topic of [157].

Machine learning. Machine learning techniques do not need a performance model of the system [154]; this technique could be applied for coordination of several autonomic system managers [78].

Utility-based. Utility based approaches require a performance model and a mechanism to correlate user-level performance with cost [84].

Economic models. Auction models such as the one discussed in [144], cost-utility models [15], or macroeconomic models [106] are an intriguing alternative and have been the focus of research in recent years.

Resource management in large-scale computing and communication systems such as computer clouds poses significant challenges and generates multiple research opportunities. For example, at this time, the average cloud server utilization is low, while the power consumption of clouds based on over-provisioning is excessive and has a negative ecological impact [129, 130]. We live in a world of limited resources and cloud over-provisioning is not sustainable either economically or environmentally.

New strategies, policies and mechanisms to implement these policies are necessary to allow cloud servers to operate more efficiently and thus, reduce costs for the Cloud Service Providers (CSPs), provide an even more attractive environment for cloud users, and support some form of interoperability. The pressure to provide new services, better manage cloud resources, and respond to a broader range of application requirements is increasing, as more US government agencies are encouraged to use cloud services.

In this chapter we first present the hierarchical organization and control of existing clouds in Section 3.2 and then, in Section 3.3 we analyze the need for alternative strategies for cloud resource management and discuss market based strategies as an intermediate step towards cloud self-organization and self-management. To compare hierarchical control prevalent in existing clouds with a straightforward bidding scheme we conduct a series of simulation experiments and report the results in Sections 3.4 and 3.5, respectively. Finally, in Section 3.6 we present our conclusions and discuss future work.

3.2
Hierarchical Organization And Control of The Existing Clouds

The large-scale data center infrastructure is based on the so called *warehouse-scale computers* (WSCs) [25]. The key insight of the analysis of the cloud architecture in [25] is that a discontinuity in the cost of networking equipment leads to a hierarchical, rather than flat, network topology. Furthermore, a limited numbers of uplinks are typically allocated to cross-switch traffic, resulting in disparities between intra- and inter-rack communication bandwidth.

In a typical WSC, racks are populated with low-end servers with a `1U3` or blade enclosure format. The servers within a rack are interconnected using a local Ethernet switch with 1 or 10 Gbps links. Similar cross-switch connections are used to connect racks to cluster[1]-level networks spanning more than 10,000 individual servers. If the blade enclosure format is being used, then a lower-level local network is present within each blade chassis, where processing blades are connected to a smaller number of networking blades through an I/O bus such as PCIe.

1 Gbps Ethernet switches with up to 48 ports are commodity components. The cost to connect the servers within a single rack – including switch ports, cables, and server NICs – costs less than $30/Gbps per server. However, network switches with higher port counts are needed to connect racks into WSC clusters. These switches are up to 10 times more expensive, per port, than commodity 48 port switches. This cost discontinuity results in WSC networks being organized into a two-level hierarchy, where a limited portion of the bandwidth of rack-level switches (4-8 ports) is used for inter-rack connectivity via cluster-level switches. This inter-rack communications bottleneck can be addressed through the use of higher-capacity interconnects such as Infiniband. Infiniband can scale up to several thousand ports but costs significantly more than even high-end Ethernet switches - approximately $500-$2,000 per port. Ethernet fabrics of similar scale are beginning to appear on the market, but these still cost hundreds of dollars per server. Very large scale WSCs can exhaust even the capacity of a two-layer hierarchy with high port-count switches at the cluster level. In this case, a third layer of switches is required to connect clusters of racks.

In this model, large-scale cloud infrastructures can be viewed as four-level hierarchies with an approximate resource count at each level: L_0 is composed of individual servers. L_1 is composed of

[1] The terms *cluster* and *cell* are used interchangeably.

racks containing $S \approx 40$ servers connected by a relatively high-speed local network. L_2 consists of clusters of $R \approx 100$ racks of lower bandwidth compared to the intra-rack connection bandwidth. Finally, L_3 consists of a set of $C \approx 24$ clusters, for a total WSC server count of $S \cdot R \cdot C \approx 96,000$.

Analysis of the size and organization of the largest scale cloud infrastructures currently in existence provides insight into real-world values of S, R and C. For example, Amazon's largest data center, US East, is located in Virginia, USA. A 2014 analysis used IP ranges to estimate that US East contained 6,382 racks of 64 servers each for a total server count of 408,448 [89]. US East is composed of three availability zones, each of which can be regarded as an independent WSC. Assuming an even distribution of racks between availability zones, we arrive at figure of approximately 136,000 servers per availability zone. However, it is unclear whether the distribution of resources between hierarchical levels corresponds with our model. It is also possible that a flatter or deeper networking hierarchy is used. Nevertheless, we can assume a disparity between intra- and inter-rack connectivity as full-bisection 10 Gbps bandwidth within placement groups is advertised as a selling point for HPC instance types [2].

The disparity between intra- and inter-rack connectivity necessitates that the concept of rack locality is accommodated during the architecture of software systems that are deployed to WSCs. For example, the HDFS filesystem used by Hadoop incorporates rack locality information for redundancy and performance purposes. Swift, the OpenStack object storage service, incorporates locality information so that location-aware services such as HDFS can be run efficiently on top of it. The inter-rack bottleneck also has implications for the management of the infrastructure itself, in particular monitoring.

Given the hierarchical topology of WSC networks and inter-rack bandwidth constraints, a distributed hierarchical management scheme is a natural choice. Several hierarchical cloud management schemes have been discussed in the literature [1, 3, 104]. As the operational details of most large-scale public clouds are not public, the management schemes used by the largest providers – Amazon, Google, and Microsoft – can only be surmised.

However, there is at least one large-scale cloud infrastructure where some operational details are available. Rackspace use OpenStack to manage their public cloud infrastructure and are co-founders of and active contributors to the OpenStack project [125]. By default, OpenStack Nova uses the

Filter Scheduler, which is centralized [96]. However, Rackspace have contributed to the development of a distributed scheduler, noting in the specification document[2] that *"Rackspace currently uses a distributed model for both scalability and high availability and these same principles need to be brought into Nova to make it suitable for large scale deployments."*

The distributed scheduler is hierarchical in that it allows the infrastructure to be organized into trees of nested zones. At the time of writing, Rackspace advertise a total server count of 110,453 [138], roughly an order of magnitude fewer that those of the largest providers. The fact that Rackspace use a hierarchical management scheme lends support to the idea that the larger providers do also. We therefore postulate that hierarchical management is the *de facto* approach to the management of large-scale cloud infrastructures.

The hierarchical approach to managing WSCs necessitates that service requests arriving at the top level must pass through four levels before they can be satisfied. The decision making processes at each hierarchical level is performed by controllers that oversee the logical unit at each level:

WSC Controllers (L_3): maintain state information about the C cells beneath it.

Cell Controllers (L_2): maintain state information on the R racks beneath it.

Rack Controllers (L_1): maintain state information on S servers beneath it.

The hierarchical management of WSCs involves two distinct activities: monitoring and control. Monitoring is the process of gathering the information required for control decisions. Control is the execution of service requests, along with housekeeping activities such as VM migration for power efficiency purposes.

Controllers at Level 3 monitor relevant aspects of server load, such as CPU and memory usage, and communicate this information periodically via messages to the Level 2 controllers. In turn, the controllers at Levels 2 and Level 1 receive state information from the levels below, aggregate it, and communicate this average load information to the level above. The WSC controller maintains the average load of each Level 1 (cell) controller beneath it.

Monitoring messages are sent periodically. Monitoring information is by definition obsolete. The longer the monitoring period the more inaccurate the information is likely to be. However, shortening the monitoring period places more load on the controllers and their networking links.

[2] http://wiki.openstack.org/wiki/DistributedScheduler

The hierarchical control model assumes that all service requests arrive first at the WSC controller before being routed through controllers at Levels 1 and 2 before it is fulfilled on a server controller at Level 3. A controller can reject a service request if the logic unit that it represents is overloaded. The optimal case is when none of the controllers a service request is routed through are overloaded and three messages are sent: $L_3 \rightarrow L_2$, $L_2 \rightarrow L_1$ and $L_1 \rightarrow L_0$.

As the load on the system increases, the number of messages required to fulfill a service request (N_{msg}) increases as messages are rejected by overloaded controllers and service requests must be rerouted. If there are multiple WSCs in the cloud infrastructure, the worst case scenario is when no WSC can find a cell which can find a rack which can a server fulfill the request: $N_{msg} = N_M - 1$ where $N_M = W \cdot C \cdot R \cdot S$.

3.3
Alternative Mechanism For Cloud Resource Management

The need for alternative mechanisms for cloud resource management. The cloud ecosystem is evolving, becoming more complex by the day. Some of the transformations expected in the future add to the complexity of cloud resource management and require different policies and mechanisms implementing these policies. Some of the factors affecting the complexity of cloud resource management decisions are discussed in Section 1.1.

Autonomic computing and self-organization. In the early 2000s it was recognized that the traditional management of computer systems is impractical and IBM advanced the concept of *autonomic computing* [51, 75]. Progress in the implementation of autonomic computing has been slow. The main aspects of autonomic computing as identified in [75] are: *Self-configuration* - configuration of components and systems follows high-level policies, the entire system system adjusts automatically and seamlessly; *Self-optimization* - components continually seek opportunities to improve their own performance and efficiency; *Self-healing* - the system automatically detects, diagnoses, and repairs localized software and hardware problems; and *Self-protection* - automatically defend against malicious attacks and anticipate and prevent system-wide failures.

Autonomic computing is closely related to self-organization and self-management but practical implementation of cloud self-organization is challenging for several reasons discussed in 1.2.

Market-oriented cloud resource management. The model we propose is based on the coalition formation and combinatorial auction ideas discussed in Chapter 4. It uses a market approach based on combinatorial auctions. *Combinatorial auctions* [46, 144] allow participants to bid [66] on bundles of items or *packages* e.g., combinations of CPU cycles, main memory, secondary storage, I/O and network bandwidth. The auctions provide a relatively simple, scalable, and tractable solution to cloud resource allocation, eliminate the need for admission control policies, which require some information about the global state of the system and, most importantly, allow the service to be tailored to the specific privacy, security, and Quality of Service (QoS) needs of each application. At this time, AWS support the so called *spot instances*, based on a market-oriented pricing strategy.

The application of market-oriented mechanisms [15, 106, 144] and their advantages over the other basic mechanisms implementing resource management policies in large-scale systems have been analyzed in the literature. Control theory [77, 80], machine learning [78, 154], and utility-based methods require a detailed model of the system, are not scalable, and typically support. If no bid exists for a service request then the request cannot be accepted. This procedure acts as an effective admission control. When a bid is generated, the resources are guaranteed to be available. Delivering on that bid is based solely on bidder's local state, so the ability to quantify and to satisfy QoS constraints can be established with a higher degree of assurance.

Energy optimization decisions such as: how to locate servers using only green energy [88], how to ensure that individual servers operate within the boundaries of optimal energy consumption [49, 55, 56, 127], when and how to switch lightly loaded servers to a sleep state [56] will also be based on local information in an auction-based system. Thus, basing decisions on local information will be accurate and will not require a sophisticated system model for a large set of parameters that cannot be easily obtained in a practical environment.

3.4
Simulation of A Hierarchically Controlled Cloud Infrastructure

We first report on a series of simulation experiments designed to understand the effectiveness of hierarchical control. These experiments were conducted on the Amazon cloud using *c3.8xlarge*[3] EC2 instances. It is challenging to simulate systems with 4 to 8 WSCs efficiently, the execution time for each one of the simulation experiments reported in this section is about 24 hours and each simulation requires 5-6 days wall clock time.

We wanted to understand how the scale and the load of the system, as well as, several parameters of the resource management system affect the ability of the cloud infrastructure to respond to service requests. An important measure of the hierarchical resource management system effectiveness is the communication complexity for monitoring the system and for locating a server capable to process a service request. The communication complexity is expressed by the number of messages at each level of an interconnection infrastructure with different latencies and bandwidth at different levels.

We simulate hierarchical monitoring and control in a time-slotted system. In each time slot incoming service requests are randomly assigned to one of the WSCs. Each WSC periodically collects data from the cells, which in turn collect data from racks, which collect data from individual servers. The communication complexity for this monitoring process increases linearly with the size of the system. The more frequent the monitoring at each level takes place the more accurate the information is, but the larger the volume of data and the interference with the "productive communication", communication initiated by running applications. The communication bandwidth at each level is limited and when the system load increases the communication latency is likely to increase significantly, as many applications typically exchange large volumes of data.

We assume a slotted time; in each reservation slot a batch of requests arrive and the individual requests are randomly assigned to one of the WSCs. We experiment with two different systems, the first has 4 WSCs and the second 8WSCs. A WSC has the following configuration: 25 cells, 100 racks per cell, 40 servers in each rack, and 4 processors per server. The system is homogeneous, all servers have the same capacity of 100 vCPU. Thus, a WSC has $100,000$ servers and $400,000$

[3] Compute-optimized instance with 32 vCPU and 60 GiB memory.

processors. All simulation experiments are conducted for 200 reservation slots and a random batch of service request arrive in each slot.

Our simulation models a system where load balancers at each level monitor the system they control. When a request is assigned to a WSC, the load balancer directs it to the cell with the lowest reported load and the process repeats itself at the cell level; the request is directed to the rack with the lowest reported load, which in turn directs it to server with the lowest reported load. If this server rejects the request the rack load balancer redirects the request to the server with the next lower load. If the rack cannot satisfy the request it informs the cell load balancer which in turn redirects the request to the rack with the next lowest reported average load, and the process ends up with a rejection if none of the cells of the WSC are able to find a server able to satisfy the type, duration, and intensity of the service request.

Our simulation environment is flexible. One first creates a configuration file which describes the system configuration, the network speed and server load and the parameters of the mode, as shown below for the high initial load case.

```
----------------------------------------------------
High initial load simulation
----------------------------------------------------

% System configuration

 static const int serverNum        = 40;
 static const int cpuNum           = 4;
 static const int rackNum          = 100;
 static const int cellNum          = 25;
 static const int WSCsNum          = 4;
 static const int servers_capacity = 100;

% Network sppeds and load parameters

 static const int interRackSpeed = 1;
 static const int intraRackSpeed = 10;
 static const int MIN_LOAD       = 80;
 static const int MAX_LOAD       = 85;

% Model parameters

 static const int NUMBER_OF_TYPES     = 100;
 static const int vCPU_MAX_REQUES     = 800;
 static const int vCPU_MIN_REQUEST    = 10;
 static const int vCPU_PER_SERVER     = 10;
 static const int MAX_SERVICE_TIME    = 10;
 static const int MONITORING_PERIOD   = 10;
 static const int SIMULATION_DURATION = 200;
 static const int TYPES_FOR_SERVER    = 5;
 static const int TYPES_FOR_REQUEST   = 5;

 static const int RACK_CAP    = serverNum  * servers_capacity;
 static const int CLUS_CAP    = rackNum    * RACK_CAP;
 static const int WSC_CAP     = clusterNum * CLUS_CAP;
 static const int SYSTEM_CAP  = WSCsNum    * WSC_CAP;
----------------------------------------------------
```

Table 3.1: Hierarchical control - the simulation results for a system configuration with 4 WSCs. Shown are the initial and final system load for the low and high load, the initial and final coefficient of variation γ of the load, the rejection ratio (RR), and the average number of messages for monitoring and control per service request at WSC level, Cell level, and Rack level.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/19.78	0.007/0.057	2.2	14,335,992	0.98	3.18	271.92
	78.50/82.38	0.004/0.183	7.1	57,231,592	1.01	10.16	973.15
8	22.50/19.26	0.006/0.049	1.9	31,505,482	0.98	3.18	271.92
	78.50/81.98	0.005/0.213	8.7	94,921,663	1.01	11.36	1071.75

Table 3.2: Hierarchical control - instead of 500 different requests types the system supports only 100; all other parameters are identical to the ones of the experiment with the results reported in Table 3.1.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/21.15	0.003/0.051	1.9	16,932,473	1.00	3.53	337.34
	82.50/67.18	0.003/0.109	7.2	42,034,225	1.00	11.15	1,097.00
8	22.50/22.13	0.008/0.055	5.4	38,949,889	1.00	4.22	470.35
	82.50/81.63	0.006/0.155	4.2	84,914,877	1.00	10.72	1,038.96

The parameters of the simulation experiments have been chosen as realistic as possible; the system configuration is derived from the data in [25].

Table 3.3: Hierarchical control - instead of 5 different service types a server offers only 2; all other parameters are identical to the ones of the experiment with the results reported in Table 3.1.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/21.15	0.003/0.051	1.7	17,341,885	0.99	3.22	276.34
	82.50/74.27	0.006/0.059	14.6	52,206,014	1.00	12.12	1255.40
8	22.50/16.27	0.006/0.035	1.3	37,750,971	0.99	3.18	268.27
	82.50/74.55	0.007/0.081	2.9	99,686,943	1.00	10.77	1,036.64

Table 3.4: Hierarchical control - the service time is uniformly distributed in the range $(1 - 20)$ reservation slots; all other parameters are identical to the ones of the experiment with the results reported in Table 3.1.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/22.41	0.005/0.047	0.20	12,352,852	1.00	3.13	261.11
	82.50/80.28	0.003/0.063	2.10	43,332,119	1.00	3.41	1108.12
8	22.50/22.77	0.005/0.083	1.30	25,723,112	1.00	3.11	236.30
	82.50/79.90	0.005/0.134	4.10	88,224,546	1.00	10.63	1029.56

Table 3.5: Hierarchical control - the monitoring interval is increased from 10 to 50 reservation slots all other parameters are identical to the ones of the experiment with the results reported in Table 3.1.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/21.07	0.003/0.033	1.00	12,335,103	0.99	3.21	270.07
	82.50/83.46	0.007/0.080	1.80	51,324,147	1.01	10.87	1040.63
8	22.50/19.16	0.005/0.030	1.30	29,246,155	1.00	3.37	304.88
	82.50/84.12	0.002/0.041	2.30	93,316,503	1.00	3.66	1005.87

The amount of resources in a service request has a broad range, between 10 and 800 vCPUs, while a single server can provide 10 vCPUs. The spectrum of service types offered is quite large, initially 500 types and then reduced to 100. The duration of simulation is limited to 200 reservation slots by practical considerations regarding costs and time to get the results.

A service request is characterized by three parameters: (1) The service type; (2) The service time expressed as a number of time slots; and (3) The service intensity expressed as the number of vCPUs needed.

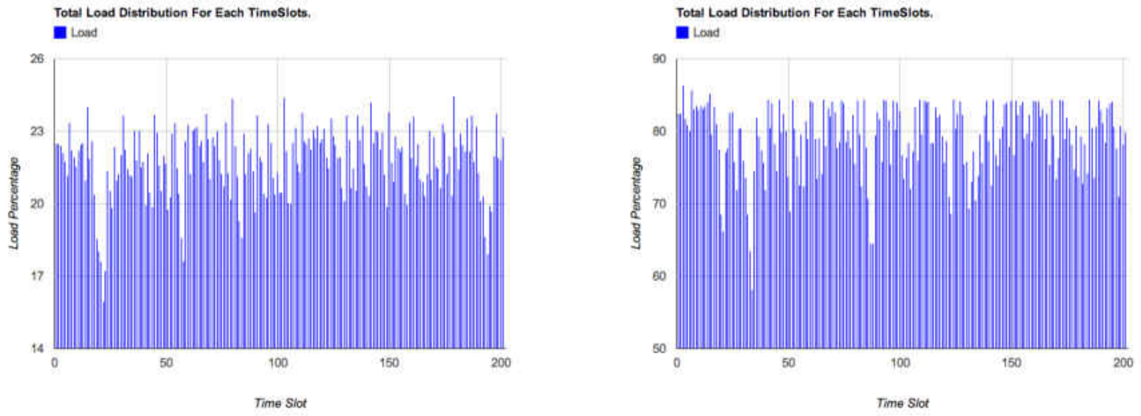


Figure 3.1: Hierarchical control - time series of the average load of a cloud with eight WSCs. The monitoring interval is 20 reservation slots and the service time is uniformly distributed in the range $1 - 20$ reservation slots. The initial average system load is: (Left) 20%; (Right) 80% of system capacity.

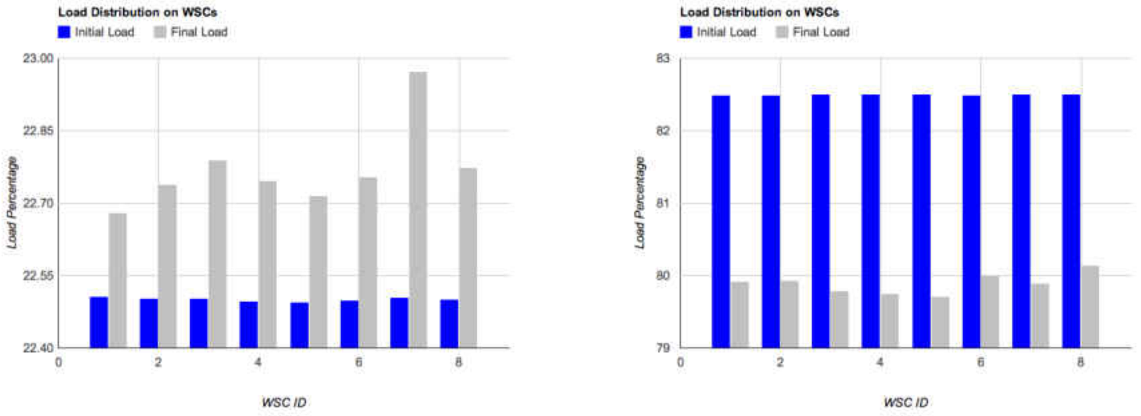


Figure 3.2: Hierarchical control - initial and final average load of a cloud with eight WSCs. The monitoring interval is 20 reservation slots and the service time is uniformly distributed in the range $1 - 20$ reservation slots. The initial average system load is: (Left) 20%; (Right) 80% of system capacity.

We study the impact of the system size, the system load, the monitoring interval, the number of service types, the number of service types supported by a server and of the service time on important systems parameters such as:

- a. The number of messages exchanged at different levels for mapping the service requests. These numbers reflect the overhead of the request processing process.

-b. The ability of the system to balance the load measured as the coefficient of variation (the variance versus the average) of the system load per reservation slot.

-c. The rejection ratio, the ratio of service requests rejected because there could be found no server able to match the service type, the service intensity, and the service duration demanded by the clients.

The simulation is conducted for two average initial system loads: low, around 20% and high, around 80% of the system's capacity. The total number of service requests for 4 WSCs and for low and high initial system load are around $(12-17) \times 10^6$ and $(42-57) \times 10^6$, respectively. In each case we show the number of WSCs, the initial and final system load for the low and high load, the initial and final coefficient of variation γ of the load, the rejection ratio (RR), and the number of messages for monitoring and control per service request at WSC level, Cell level, and Rack level.

For the first experiment the attributes of service requests are uniformly distributed and the ranges are: $(1-100)$, $(1-10)$, and $(10-800)$ for service type, service time, and service intensity, respectively. A server supports 5 different service types randomly selected from a total of 500 possible service types. The monitoring interval is 10 reservation slots; for later experiments it will increase to 20 and then to 50 reservation slots.

The results of our first simulation experiment in Table 3.1 show that the rejection ratio, the coefficient of the variation of the final load, and the average number of messages required to map a service request to a server are more than three fold larger in the case of higher load; indeed, $7.1/2.2 = 3.22, 0.183/0.057 = 3.22$, and $984/276 = 3.2$. At higher load more requests are rejected, load balancing is less effective, and the overhead for mapping a request is considerably higher. The increase in the number of messages means a substantial increase of the communication costs and also a longer waiting time before a request enters the service.

Doubling the size of the system does not affect the statistics for the same average system load. For example, when the initial average load is 22.50% the average number of messages exchanged per service request is the same at the three levels of the hierarchy for both system configurations. The rejection ratio varies little, 2.2% versus 1.9% and 7.1% versus 8.7% for 4 and 8 WSCs, respectively.

Next we explore the effects of changing various parameters of the system model. Our experiments investigate the effects of: (1) Doubling the number of WSCs from 4 to 8; (2) Reducing the number of types of services from 500 to 100; (3) Reducing the number of types of services offered by each server from 5 to 2; and (4) Increasing the monitoring interval to 50 time slots and changing the distribution of the service time; initially it was uniformly distributed in the interval $(1 - 10)$ time slots and this interval will be $(1 - 20)$ time slots.

Table 3.2 presents the results after reducing the total number of service request types from 500 to 100. We see a reduction of the rejection ratio and of the number of messages at high load for the larger configuration of 8 WSCs compared to the case in Table 3.1. We also notice that in this case the rejection ratio decreases from 7.4% to 4.2% when we increase the size of the system from 4 to 8 WSCs.

Table 3.3 presents the results when the number of service types offered by a server is reduced from 5 to just 2. We see again a reduction of the rejection ratio at high load when we double the size of the system. The drastic reduction of this ratio, from 14.6 to 2.9 can be attributed to the fact that we randomly assign an incoming service request to one of the WSCs and the larger the number of WSCs the less likely is for the request to be rejected. The number of messages at the rack level is considerably larger for the smaller system configuration at high load, 1255 versus 973 in the first case presented in Table 3.1.

Next we set the monitoring interval to 20 reservation slots and the service time is now uniformly distributed in the range $1 - 20$ reservation slots. The results in Table 3.4 show that the only noticeable effect is the reduction of the rejection rate compared with the case in Table 3.1. In the following experiment we extended the monitoring interval from 10 to 50 reservation slots. Recall that the service time is uniformly distributed in the range 1 to 10 reservation slots; even when the monitoring interval was 10 reservation slots, this interval is longer than the average service time thus, the information available to controllers at different levels is obsolete. The results in Table 3.5 show that increasing the monitoring interval to 50 slots has little effect for the 4 WSC configuration at low load, but it reduces substantially the rejection ratio and increases the number of messages at high load. For the 8 WSC configuration increasing the monitoring interval reduces the rejection ratio at both low and high load, while the number of messages changes only slightly.

Figures 3.1 and 3.2 refer to the case presented in Table 3.1 when the monitoring interval is 20 time slots and the service time is uniformly distributed in the $1-20$ slots range and there are 8 WSCs. Figures 3.1(Left) and (Right) show the time series of the average system load for the low and the high initial load, respectively. We see that the actual system workload has significant variations from slot to slot; for example, at high load the range of the average system load is from 58% to 85% of the system capacity. Figure 3.2 shows the initial and the final load distribution for the 8 WSCs; the imbalance among WSCs at the end of the simulation is in the range of $1-2\%$.

The results of the five simulation experiments are consistent, they typically show that at high load the number of messages, thus the overhead for request mapping increases three to four fold, at both cell and rack level and for both system configurations, 4 and 8 WSCs.

3.5
Simulation of An Economic Model of Cloud Resource Management

In this section we discuss the simulation of an economic model for cloud resource management based on a straightforward bidding scheme. There is no monitoring and in each reservation slot all servers of a WSC bid for service. A bid consists of the service type(s) offered and the available capacity of the bidder. The overhead is considerably lower than that of the hierarchical control; there is no monitoring and the only information maintained by each WSC consists only of the set of unsatisfied bids at any given time. The servers are autonomous but act individually, there is no collaboration among them while self-organization and self-management require agents to collaborate with each other.

At the beginning of a reservation slot servers with available capacity at least as large as a given threshold τ place bids which are then collected by each WSC. A bid is persistent, if is not successful in the current reservation slot it remains in effect until a match with a service request is found. This strategy to reduce the communication costs is justified because successful bidding is the only way a server can increase its workload.

We investigate the effectiveness of this mechanism for lightly loaded, around 20% average system load, and for heavily loaded, around 80% average system load. The thresholds for the two cases are different, $\tau = 30\%$ for the former and $\tau = 15\%$ for the latter. The choice for the lightly loaded case is motivated by the desire to minimize the number of messages; a large value of τ, e.g., 40% would lower the rejection ratio but increase the number of messages. In the heavily loaded system case increasing the threshold, e.g., using a value $\tau = 20\%$, would increase dramatically the rejection rate; indeed, few servers would have 20% available capacity when the average system load is 80%.

Table 3.6: Market-based mechanism - simulation results for a system configuration with 4 WSCs. Shown are the initial and final system load for the low and high load, the initial and final coefficient of variation γ of the load, the rejection ratio (RR), and the average number of messages for monitoring and control per service request at WSC level, Cell level, and Rack level.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/23.76	0.007/0.067	.22	15,235,231	0.002	0.011	0.987
	82.50/80.32	0.004/0.115	5.44	63,774,913	0.003	0.042	4.155
8	22.50/22.47	0.006/0.033	.18	30,840,890	0.002	0.011	0.987
	82.50/81.30	0.005/0.154	7.23	89,314,886	0.003	0.054	5.761

Table 3.7: Market-based mechanism - instead of 500 different requests types the system supports only 100; all other parameters are identical to the ones of the experiment with results reported in Table 3.6.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/22.3	0.004/0.050	.18	15,442,372	0.002	0.011	0.987
	82.50/79.88	0.004/0.098	6.01	56,704,224	0.002	0.059	5.968
8	22.50/23.0	0.007/0.049	.3	31,091,427	0.002	0.011	0.987
	82.50/80.91	0.009/0.127	5.81	85,322,714	0.003	0.0.51	5.845

To provide a fair comparison we repeat the measurements reported in Section 3.4 under the same conditions but with bidding replacing the monitoring and hierarchical control. We use the same performance indicators as the ones reported in Section 3.4, those measuring communication complexity, the efficiency of load balancing, and the rejection ratio as shown in Tables 3.6 - 3.9.

We notice a significant reduction of the communication complexity, more than two orders of magnitude in case of the market-oriented mechanism. For example, at low average load the average

Table 3.8: Market-based mechanism - instead 5 different service types a server offers only 2; all other parameters are identical to the ones for the experiment with results reported in Table 3.6.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/20.94	0.007/0.056	.1	15,295,245	0.002	0.011	0.987
	82.50/77.83	0.008/0.133	10.1	49,711,936	0.003	0.063	6.734
8	22.50/22.33	0.007/0.063	.02	31,089,191	0.002	0.011	0.987
	82.50/78.18	0.008/0.142	3.61	71,873,449	0.002	0.059	6.098

Table 3.9: Market-based mechanism - the service time is uniformly distributed in the range $(1-20)$ reservation slots; all other parameters are identical to the ones of the experiment with results reported in Table 3.6.

WSCs	Initial/Final load (%)	Initial/Final γ	RR (%)	# of service requests	WSC Msg/Req	Cell Msg/Req	Rack Msg/Req
4	22.50/23.31	0.002/0.064	2.27	13,445,186	0.001	0.011	0.988
	82.50/84.05	0.007/0.101	3.75	57,047,343	0.002	0.042	6.329
8	22.50/18.93	0.007/0.038	2.94	28,677,012	0.001	0.011	0.988
	82.50/85.13	0.008/0.072	4.38	88,342,122	0.002	0.029	4.078

number of messages per reservation request at the rack level is 0.987 (see Table 3.6) versus 271.92 reported in Table 3.1 for the 4 and for the 8 WSCs case. At high average load the same figures are: 4.155 versus 973.14 for the 4 WSC case and 5.761 versus 1071.75 for the 8 WSC case. A second observation is that when the average load is 20% of the system capacity the communication complexity is constant, 0.987, for both configurations, 4 and 8 WSCs, regardless of the choices of simulation parameters. At high average load, the same figure is confined to a small range, 4.078 to 6.734.

At low average load we also see a reduction of the average rejection ratio for both 4 and 8 WSCs, $0.1 - 0.3$ (see Tables 3.6 - 3.8). The results in Table 3.9 show that increasing the distribution of the service time from the $1-10$ to $1-20$ range increases the rejection rate; this is most likely due to the fact that we simulate only the execution over 200 reservation slots and requests with a large service time arriving during lather slots do not have time to complete. At high average system load the average rejection ratio is only slightly better in case of market-based versus hierarchical control. Lastly, the market-based mechanism performs slightly better than hierarchical control in terms of slot-by-slot load balancing, the coefficient of variation of the system load per slot $\gamma \leq 1.115$.

The number of different service types offered by the cloud does not seem to affect the performance of the system and neither does the number of services supported by individual servers, as we can see from the results reported in Tables 3.7 and 3.8. The organization is scalable, the results for 4 and for 8 WSCs differ only slightly. This is expected because of the distributed scheme where each WSC acts independently, it receives an equal fraction of the incoming service requests and matches them to the bids placed by the servers it controls.

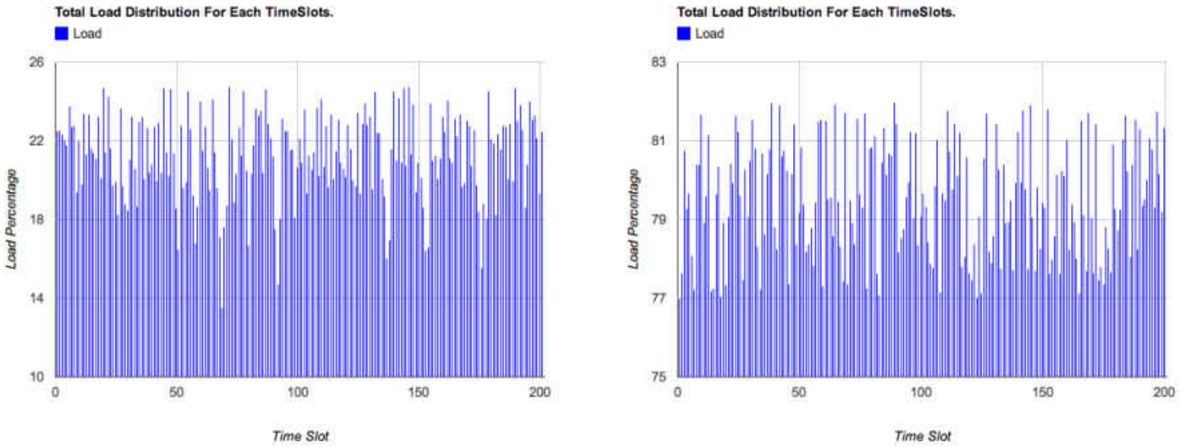


Figure 3.3: Hierarchical control - time series of the average load of a cloud with eight WSCs. The monitoring interval is 20 reservation slots and the service time is uniformly distributed in the range $1 - 20$ reservation slots. The initial average system load is: (Left) 20%; (Right) 80% of system capacity.

Figures 3.3 and 3.4 refer to the case presented in Table 3.9 when the monitoring interval is 20 time slots and the service time is uniformly distributed in the $(1 - 20)$ slots range and there are 8 WSCs. Figures 3.3 (Left) and (Right) show the time series of the average system load for the low and the high initial load, respectively. The actual system workload has relatively small variations from slot to slot; for example, at high load the range of the average system load ranges from 77% to 82% of the system capacity. Figures 3.4 (Left) and (Right) show the initial and the final load distribution for the 8 WSCs; the imbalance among WSCs at the end of the simulation is in the range of $1 - 3\%$ at low load and between 80.1% and 80% at high load.

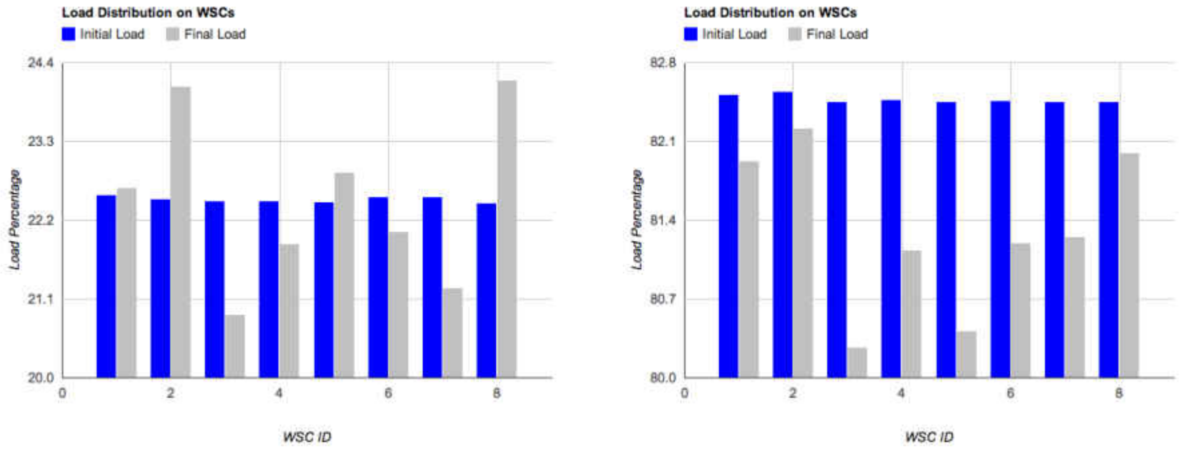


Figure 3.4: Hierarchical control - initial and final average load of a cloud with eight WSCs. The monitoring interval is 20 reservation slots and the service time is uniformly distributed in the range $1 - 20$ reservation slots. The initial average system load is: (Left) 20%; (Right) 80% of system capacity.

3.6 Conclusions And Future Work

Low average server utilization [148] and its impact on the environment [30] make it imperative to devise new resource management policies to optimize cloud resource utilization and significantly increase the average server utilization, as well as, the computational efficiency measured as the amount of computations per Watt of power used by the cloud infrastructure. The simulation experiments reported in Section 3.4 confirm our intuition that the effectiveness of hierarchical control decreases significantly as the system load increases.

In some sense a hierarchical cloud organization is scalable; our results show that at low and at high load a system with 8 WSCs has a similar behavior as the one with only 4 WSCs.

However, at high load both configurations suffer from a substantially increased overhead. It is significant the fact that at high load the number of messages as the cell level increases three fold. Typically, computer clouds use 10 Gbps Ethernet networks; the contention for cell-level and WSC-level connectivity will further limit the ability of the hierarchical control model to perform under stress.

The simulation time is prohibitive and would dramatically increase for a heterogeneous cloud infrastructure model where individual servers provide one type of service rather than several. Such a more realistic model is likely to make a hierarchical control model even less attractive.

CHAPTER 4
COALITION FORMATION AND COMBINATORIAL AUCTIONS

The work reported in this chapter is based on [111]. Sometimes an incoming request might not fit into any of the servers in our system and we need to split it among a group of them. In this chapter we propose a two-stage protocol for resource management in a hierarchically organized cloud. The first stage exploits spatial locality for the formation of coalitions of supply agents; the second stage, a combinatorial auction, is based on a modified proxy-based clock algorithm and has two phases, a clock phase and a proxy phase. The clock phase supports price discovery; in the second phase a proxy conducts multiple rounds of a combinatorial auction for the package of services requested by each client. The protocol strikes a balance between low-cost services for cloud clients and a decent profit for the service providers. We also report the results of an empirical investigation of the combinatorial auction stage of the protocol.

4.1
Introduction

Nowadays large farms of computing and storage servers are assembled to support several cloud delivery models including Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). In such systems users pay only for computing resources they use, similarly to other utilities such as electricity and water.

Computer clouds raise the question of how far we can push the limits of composability of computing and communication systems, while still being able to support effective policies for resource management and their implementation mechanisms. The software, the glue allowing us to build increasingly more complex systems, consists of more and more layers thus, the challenge of controlling large-scale systems is amplified.

Control theory tells us that accurate state information and a tight feedback loop are the critical elements for effective control of a system. In a hierarchical organization the quality of state information degrades as we move from the bottom to the top; *only local information about the state of a server is by definition accurate.* Moreover, this information is volatile, it must be acted upon promptly because the state changes rapidly. Our recent results [112] confirm that hierarchical control has considerably larger overhead than a simple economic model for cloud resource management. The communication complexity of hierarchical control based on monitoring is more than two orders of magnitude higher and consumes a significant fraction of the available bandwidth at all levels of the interconnection network.

As discussed in Section 3.1, existing solutions for cloud resource management are neither effective nor scalable and they require detailed models of the system and accurate information about the state of individual servers.

Informally, self-organization means synergetic activities of elements when no single element acts as a coordinator and the global patterns of behavior are distributed. Self-management means that individuals can effectively set their own goals, make decisions on how to achieve those goals, plan and schedule their activities independently, and evaluate the progress towards these goals. Self-management can lead to faster and more accurate resource management decisions.

Self-management as a result of auctions eliminates the need for a system model and requires only local thus, more accurate information about the state of individual components. This approach has the potential of optimizing the use of resources and allow Cloud Service Providers (CSPs) to offer services at a lower cost for the consumers [52, 91]. Though the virtues of self-management have long been recognized, there is, to our knowledge, no cloud computing infrastructure, based on self-organizing principles and self-management. This is in itself proof of the difficulties to apply these concepts in practice.

Self-management has to be coupled with some mechanisms for coalition formation allowing autonomous agents, the servers, to act in concert. Autonomous systems have to cooperate to guarantee QoS by distributing and balancing the workload, replicate services to increase reliability, and implement other global system policies. Cooperation means that individual systems have to partially surrender their autonomy.

Self-organization cannot occur instantaneously in an adaptive system. It is critical to give the autonomous cloud platforms interconnected by a hierarchy of networks the time to form coalitions in response to services requests thus, self-management requires an effective reservation system. Reservations are ubiquitous for systems offering services to a large customer population, e.g., airline ticketing, chains of hotels, and so on. Existing clouds, e.g., the Amazon Web Services, offer both reservations and spot access, with spot access rates lower than those for reservations.

The solution discussed in this chapter involves concepts, policies, and algorithms from several well-established areas of economics and computer science: self-organization and self-management of complex systems; coalition formation and virtual organizations; auction theory and practice; and system organization and computer architecture. We discuss related work and our contributions in Section 4.2 and in Section 4.3 we describe the system model. Algorithms for the formation of sub-coalitions and for clock-proxy auction are the subjects of Sections 4.4 and 4.5, respectively. The results of a simulation experiment and the conclusions of our work are presented in Sections 4.6 and 4.7.

4.2
Related Work

The present and future challenges outlined in Section 4.1 motivate the search for effective and scalable policies and mechanisms for cloud resource management [34, 40, 90, 113, 119, 145, 165]. In this section we survey some of the research in this area focused on market mechanisms.

4.2.1

Coalition Formation

Informally, a *coalition* is a group of entities which have agreed to cooperate for achieving a common goal. A *virtual organization* involves entities that require a communication infrastructure and dedicated software to support their activities. *Coalition formation* is a widely used method for increasing the efficiency of resource utilization and for providing convenient means to access these resources [116]. In recent years, the emergence of large-scale electronic markets, grid and cloud computing, sensor networks, and robotics have amplified the interest in coalition formation and virtual organizations [98, 99, 146]. For example, self-organization of sensor networks through bottom-up coalition formation is discussed in [107, 147].

Different aspects of resource management in computational grids including load balancing, job-allocation, and scheduling, as well as revenue sharing when agents form coalitions or virtual organizations are discussed in [39, 71, 83, 136, 149, 167]. Grid resource allocation is modeled as cooperative games [83] or non-cooperative games [136]. Resource co-allocation is presented in [167].

There is little surprise that the interest in coalition formation migrated in recent years from computational grids to cloud resource management. The vast majority of on-going research in this area is focused on game-theoretic aspects of coalition formation for cloud federations. A *cloud federation* is of a set of CSPs collaborating to provide services to a cloud user community.

A stochastic linear programing game model for coalition formation is presented in [119]; the authors analyze the stability of the coalition formation among cloud service providers and show that resource and revenue sharing are deeply intertwined. An optimal VM provisioning algorithm ensuring profit maximization for CSPs is introduced in [40].

A cloud federation formation described as a hedonic game and focused on the stability and the fairness of the game is discussed in [113]. The profit maximization for each federation is formulated as an integer programming problem (IP) and the game is augmented with a preference relation over the set of federations. The paper assumes that the Virtual Machines (VMs) contributed by each CSP to a federation are characterized by several attributes, $a \in \mathcal{A}$ including the number of cores, the amount of memory and of secondary storage. The IP problem for CSP $\mathcal{C}_i$ in federation

$\mathcal{F}$ is formulated as $\max \sum_{\mathcal{C}_i \in \mathcal{F}} \sum_{j=1}^{n} n_{i,j}(p_j - c_{i,j})$ subject to the set of conditions $\sum_{j=1}^{n} q_j^a n_{i,j} \leq A_i, \ \forall a \in \mathcal{A}$ and $\sum_{\mathcal{C}_i \in \mathcal{F}} n_{i,j} = r_j$ with: $n_{i,j}$ - the number of VMs of type j; p_j - the price for a VM running an instance of type j; $c_{i,j}$ - the cost of an instance of type j provided by $\mathcal{C}_i$; q_j^a - the quantity of resource of type a in a VM of type j; A_i - the total amount of resource of type a offered by $\mathcal{C}_i$; and r_j - the number of VMs of type j requested. The paper adopts a payoff division based on the Banzhaf value [113].

A combinatorial coalition formation problem is described in [99]. The paper assumes that a seller has a price schedule for each item. The larger the quantity requested, the lower is the price a buyer has to pay for each item; thus, buyers can take advantage of price discounts by forming coalitions. A similar assumption is adopted by the authors of [98] who investigate systems where the negotiations among deliberate agents are not feasible due to the scale of the system. The paper proposes a macroscopic model and derives a set of differential equations describing the evolution in time of coalitions with a different number of participants. The results show that even a low rate of leaving away participants allows a coalition to achieve a steady state.

An algorithm to find optimal coalition structures in cooperative games by searching through a lattice like the one in Figure 4.2, was introduced by [153]. A more refined algorithm is described in [139]; in this algorithm the coalition structures are grouped according to the so-called *configurations* reflecting the size of the coalitions.

4.2.2

Auctions

Auctions are a widely used mechanism for resource allocation [45, 62]. Among the numerous applications of auctions are: the auctioning of airport take-off and landing slots, spectrum licensing by the Federal Communication Commission (FCC), and industrial procurement. An online auction mechanism for resource allocation in computer clouds is presented in [167].

4.2.3

Combinatorial Auction

A combinatorial auction is one where a buyer requires simultaneous access to a *package* of goods. An auction allows the seller to obtain the maximum feasible profit for the auctioned goods; it is organized by an *auctioneer* for every *request* of a consumer. A *proxy* is an intermediary who collects individual bids from the buyers participating at an auction, computes the total cost of the package from the bids, and communicates this price to the auctioneer. A vast literature including [12, 13, 14, 150] covers multiple aspects of combinatorial auctions including bidding incentives, stability, equilibrium, algorithm testing, and algorithm optimality.

Package bidding assumes that a seller offers $\mathcal{N}$ different types of items. A buyer bids for packages of items. A *package* is a vector of integers $\mathcal{Z} = \{z_1, z_2, \ldots, z_{\mathcal{N}}\}$ which indicates the quantity of each item in the package; the price of items is given by $\mathcal{M} = \{m_1, m_2, \ldots, m_{\mathcal{N}}\}$.

Package bidding can be traced back to generalized Vickerey auctions based on the Vickerey-Clarke-Groves mechanisms [45, 62]. In Vickerey auctions a bidder reports its entire demand schedule. The auctioneer then selects the allocation which maximizes the total value of the package and requires a bidder to pay the lowest bid it would have made to win its portion of the final allocation, considering all other bids.

In an *ascending package auction (APA)* there are $\mathcal{K}$ participants identified by an index, $k = 0$ is the seller and $k = 1, 2, \ldots, \mathcal{K}$ are the buyers [12]. Each buyer has a *valuation* vector $v_i = (\nu_i(z), z \in [0, \mathcal{M}])$; $\nu_k(z)$ represents the value of package z to the bidder k. Some of the rules for this type of auction are: all bids are firm, a bid cannot be reduced or withdrawn; the auctioneer identifies after each round the set of the bids that maximize the total price, the so-called *provisional winning bids*. The auction ends when a new round fails to elicit new bids; then the provisional winning bids become the winners of the auction.

In an ascending package auction a bidder can be deterred from bidding for the package she really desires by the threat that competitors could drive prices up; this would threaten the equilibrium. This problem does not exist in *ascending proxy auctions* when each bidder instructs a proxy agent to bid on her behalf [12]. The proxy accepts as input the bidder's valuation profile and bids following a “sincere strategy”. Nash equilibrium can be reached when the bid increments are negligibly small [12].

In a *clock auction* the auctioneer announces prices and the bidders indicate the quantities they wish to buy at the current price. When the demand for an item increases, so does its price until the there is no excess demand. On the other hand, when the offering exceeds the demand, the price decreases [12]. In a clock auction the bidding agents see only aggregate information, the price at a given time, and this eliminates collusive strategies and interactions among bidding agents. The auction is *monotonic*, the amounts auctioned decrease continually and this guarantees that the auction eventually terminates. When the price of a package can be computed as the sum of products of prices and quantities it is said that auction benefits from *linear pricing*.

The *clock-proxy-auction* is a hybrid auction based on an iterative process with two phases [14]. A *clock* phase is followed by a *proxy* round. During the proxy round the bidders report the values they have submitted to the proxy which in turn submit bids for the package to the auctioneer. A bidder has a single opportunity to report the quantity and the price to the proxy, bid withdrawals are not allowed, and the bids are mutually exclusive. The auctioneer then selects the winning bids that maximize the seller’s profit.

4.2.4
The Novelty of The Approach Described In This Chapter

The reservation system we propose has two stages; coalitions of servers are formed periodically during the first and in the second the coalitions participate in combinatorial auctions organized in each allocation slot. To our knowledge this is the first attempt to address cloud self-organization and resource management based on coalition formation and combinatorial auctions when individual servers learn from past behavior, see Figure 4.1.

We discuss coalition formation for a realistic model of the cloud infrastructure, hierarchical organization, while most of the research reported in the literature is focused on coalition formation for cloud federations. The coalition formation problem has different formulations and different constraints in the two cases.

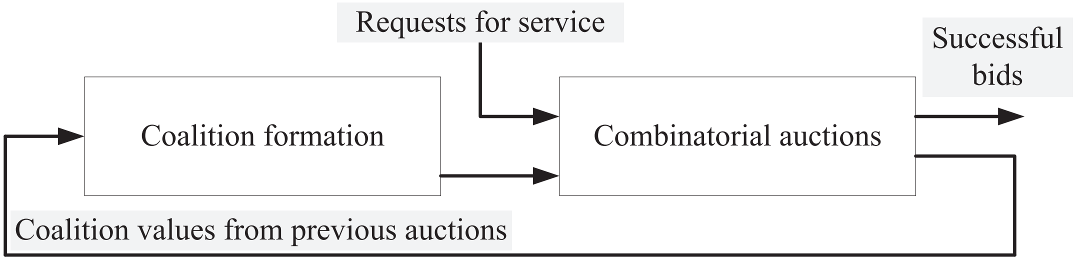


Figure 4.1: A protocol with two stages; feedback about past values of individual coalitions is used to determine the value of individual coalition structures as shown in Section 4.4.

At this time individual CSPs believe that they have a competitive advantage due to the unique value of their services and are not motivated to disclose relevant information about the inner working of their systems as we have re-discovered when investigating the energy consumption of AWS instances [130]. Thus, the practical realization of cloud federations seems a rather remote possibility [110].

A rare glimpse at the architecture of a cloud is provided in [25] and we are taking advantage of it to base our research on a realistic model of the cloud infrastructure. We investigate coalition formation subject to the physical constraints of the hierarchical cloud organization model. As more diverse applications, including Big Data applications, are likely to use computer clouds, the demand for computing resources allocated to a single application will increase and could be considerably larger than any server can provide; only coalitions of servers will be capable to offer such resources. It is critical for the members of a coalition to communicate effectively; this requires coalition member to be in close proximity of each other in a system consisting of a hierarchy of networks with different bandwidth and latency. This adds additional constrains to the coalition formation protocol.

To respond to the needs of increasingly more complex applications consisting of multiple phases and requiring workflow management, CSPs are already offering workflow management services such as SWF (Simple Workflow Management) and EBS (Elastic Beanstalk) at AWS. Different phases of an application may require coalitions of servers with different types of resources and this is the reason why we decided to investigate combinatorial auctions where packages of items are auctioned.

4.3
System Model

System architecture. We assume a hierarchical organization of the cloud infrastructure similar to the one described in [25] and Section 1.3. A data center consists of multiple warehouse-scale computers (WSCs), each WSC has multiple cells, each cell has multiple racks and each rack houses multiple servers. A WSC connects $50,000$ to $100,000$ servers and uses a hierarchy of networks. The servers are housed in racks; typically, the 48 servers in a rack are connected by a 48 port Gigabit Ethernet switch. The switch has two to eight up-links which go to higher level switches in the network hierarchy [25]. The bandwidth to communicate outside the rack is much smaller than the one within the rack; this has important implications for resource management policies and becomes increasingly difficult to address in systems with a large number of servers.

4.3.1
Model Assumptions

For simplicity we assume that the racks are homogeneous, they have identical processors with the same number of cores and an identical configuration of GPUs, FPGAs, workflow engines, or other hardware, the same amount of main storage, cache, and secondary storage. We also assume that all servers in a rack are identically configured and support the same type of services. The same service may be offered by multiple racks; for example, multiple racks could offer configurations with GPUs.

The system we envision supports a *reservation system* and *spot* resource allocation. The reservation system has two stages: (A) coalition formation, and (B) combinatorial auctions. The spot allocation is done through a bidding process for each type of service. The time is quantified, reservations are made as a result of auctions carried out at the beginning of each *allocation slot* of duration τ; for example, a allocation slot could be one hour.

4.3.2
Coalition Formation

The rational for coalition formation is that applications may need resources beyond those provided by an individual server. For example, a Map-Reduce application may require a set of 20 servers during the Map phase to process a data set of several PB (Petabytes). If the algorithm requires the servers to communicate during this phase then the application should start at the same time on all servers and run at the same pace, a condition known as *co-scheduling.* Co-scheduling is only feasible if the set of 20 servers form a coalition dedicated to the application; moreover, the hardware configuration of the coalition members should be optimal for the algorithms used by the application, e.g., have attached GPUs.

4.3.3
Combinatorial Auctions

Combinatorial auctions allow cloud users making the reservations to acquire packages consisting of coalitions of servers with different types and amounts of resources. Combinatorial auctions are necessary because different phases of an application may require systems with different configurations or systems supporting different functions. In our previous example the Reduce phase of the Map-Reduce application may require several servers with a very large amount of secondary storage.

4.4
Coalition Formation

First, we discuss the formulation of the coalition formation problem as a cooperative game. Then we introduce the algorithms for determining the optimal coalition structure and for coalition formation in the context of our model.

4.4.1

Coalition Formation As A Cooperative Game

The coalition formation is modeled as a cooperative game where the goal of all agents is to maximize the reward due to the entire set of agents. We consider a set of N servers $\{s_1, s_2, \ldots, s_N\}$, located in the same rack.

A *coalition* $\mathbb{C}_i$ is a non-empty subset of N. A *coalition structure* is set of m coalitions $\mathbb{S} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_m\}$ satisfying the following conditions:

$$|\bigcup_{i=1}^{m} \mathbb{C}_i |= N \quad \text{and} \quad i \neq j \;\Rightarrow\; \mathbb{C}_i \bigcap \mathbb{C}_j = \emptyset. \tag{4.1}$$

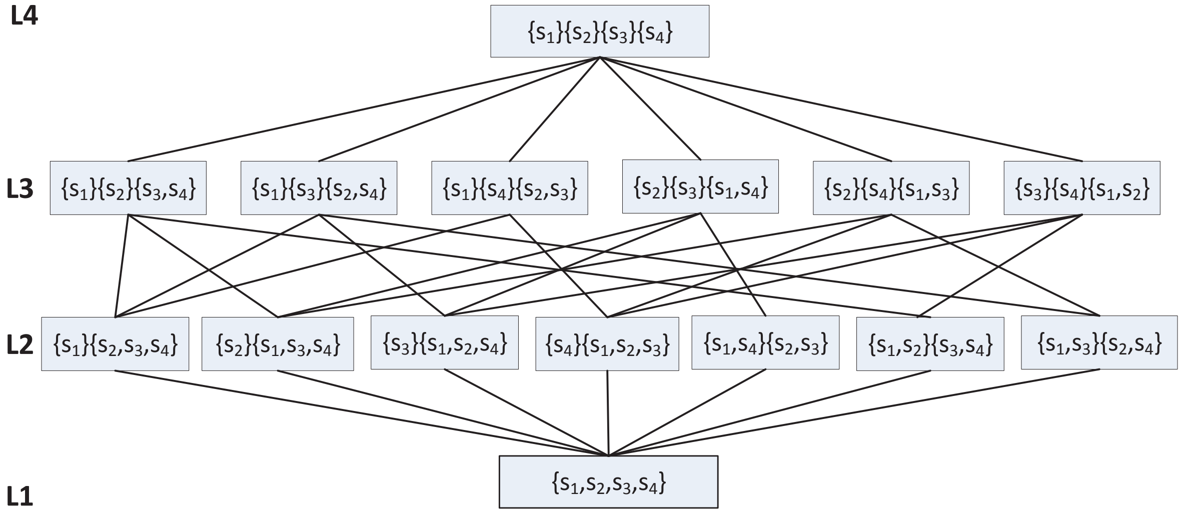


Figure 4.2: A lattice with four levels $L1, L2, L3$ and $L4$ shows the coalition structures for a set of 4 servers, s_1, s_2, s_3 and s_4. The number of coalitions in a coalition structure at level L_k is equal to k.

Figure 4.2 shows a lattice representation of the coalition structures for a set of four servers s_1, s_2, s_3 and s_4. This lattice has four levels, $L1, L2, L3$ and $L4$ containing the coalition structures with $1, 2, 3$ and 4 coalitions, respectively. In general, the level k of a lattice contains all coalition structures with k coalitions; the number of coalitions structures at level k for a population of N agents is given by the Sterling Number of Second Kind:

$$\mathcal{S}(N,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^N. \tag{4.2}$$

In the case illustrated in Figure 4.2 $N = 4$ and the number of coalition structures at levels $L1 - L4$ are $1, 7, 6, 1$, respectively.[1] The total number of coalition structures with N agents is called the Bell number

$$\mathcal{B}(N) = \sum_{k=0}^{N}\mathcal{S}(N,k) = \sum_{k=0}^{N}\frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^N. \tag{4.3}$$

The number of coalitions structures increases exponentially with the number of agents. For example, for $N = 40$, a typical number of servers in a rack, the logarithm of the number of coalition structures is close to 10^{35} and $\mathcal{S}(40,14) = 3.58598722556218034914285 54E + 34$. The logarithm of number of coalitions is close to $E + 10$.

Searching for the optimal coalition structure $\mathbb{C}$ is computationally challenging due to the size of the search space. The first step for determining the optimal coalition structure is to assign a *value* v reflecting the utility of each coalition. The second step is the actual coalition formation.

4.4.2

Rack-level Coalition Formation

Recall from Section 4.3 that in our model a rack is homogeneous, all servers have an identical configuration. This realistic assumption simplifies considerably the complexity of the search for an optimal coalition structure as the servers are indistinguishable from one another.

[1] For $N = 5$ and $N = 6$ the Stirling Numbers of the Second Kind are respectively $1, 15, 25, 10, 1$ and $1, 31, 90, 65, 15, 1$.

The second important observation is that we have a system with two stages and feedback, see Figure 4.1. In the second stage the coalitions created during the first stage are included in successfully auctioned packages thus, we can determine precisely the value of all coalitions structures. The third important observation is that only *available servers*, servers with no commitments for the current slot, can participate to coalition formations and then to the auction organized in that slot; call $N_a \leq N$ the number of available servers.

An elected *rack leader* collects information about all *successful coalitions* - coalitions that have been included in packages auctioned successfully during a window of w successive past allocation slots. The current rack-leader records an entry for the corresponding *partial coalition structure (PCS)* including n_k - the *coalition size*, m_k - the *multiplicity* of occurrence, the value $\bar{v}_k$ calculated as the average price over all auctions when a PCS including a coalition of size n_k was part of a *package* successfully auctioned during the past w allocation slots.

Call $\mathcal{L}$ the *PCL-list.* For a window of size w the list $\mathcal{L}$ is the list of all triplets $\mathcal{L}_k = [n_k, m_k, \bar{v}_k]$ ordered first by $1 \leq n_k \leq N_a$ then by m_k. The list includes only entries $\mathcal{L}_k$ with $\bar{v}_k > 0$. Given N_a a *coalition structure (CS)* $\mathbb{S}_k$ among the entries $\mathcal{L}_{k1}, \mathcal{L}_{k2}, \ldots \ldots, \mathcal{L}_{kn}$ is *feasible* if $\sum_j n_k \times m_k = N_a$. Then the value of the coalition structure $\mathbb{S}_k$ is $v_k = \sum_j \bar{v}_j$. Note that we force the formation of coalitions involving all available servers. An example of a PCS list $\mathcal{L}$ follows

```
a  [1,4,35]     \* 4 PCS of 1-server {s}
b  [1,15,682]  \ *15 PCS of 1-server {s}
       .........
c  [2,3,78]      \* 3 PCS of  2-servers {s,s}
       ........
d  [3,2,502]   \* 2 PCS of 3-servers {s,s,s}
e  [3,4,812]   \* 4 PCS of 3-servers {s,s,s}
       .........
f  [16,1,751]  \* 1 PCS of 16-servers {s,...s}
g  [16,2,740]  \* 2 PCS of 16-servers {s,...s}
       .........
```

In this example some of the feasible coalitions structures when $N_a = 16$ are: $\mathbb{S}_g$ with $v_g = 751$; $\mathbb{S}_{a,b}$

with $v_{a,b} = 35 + 682 = 712$; $\mathbb{S}_{a,e}$ with $v_{a,e} = 35 + 812 = 837$; $\mathbb{S}_{a,c,d}$ with $v_{a,c,d} = 35 + 78 + 502 = 615$, and so on. Note that the value of a coalition reflects also the length of time the coalition was active in response to successful auction. We see that a PCS of 15 coalitions of 1 server have been active for larger number of slots than a PCS of 4 coalitions of 1 server. The value attributed to a coalition of k servers is a distributed equally among the servers; the value of a package of several coalitions auctioned successfully is divided among the coalitions based on the resource supplied by each one of them.

4.4.3
Coalition Formation

The protocol for coalition formation proceeds as follows:

1. Server s_i sends to the current rack leader:

   (a) A vector $([\nu_i^1, \beta_i^1], [\nu_i^2, \beta_i^2], \ldots\ldots [\nu_i^N, \beta_i^N])$ with $\nu_i^k, 1 \leq k \leq N$ the total value due to the participation of s_i in successful coalitions, of k servers and β_i^k a bit vector with w components with $\beta_i^{k,j} = 1$ if s_i was included in a successful coalition of k servers in slot j of window w.

   (b) Availability, $a_i = 1$ if available, 0 otherwise.

2. After receiving the information from all servers the current rack leader:

   (a) Determines $N_a = \sum_{i=1}^{N} a_i$.

   (b) Computes $m_k = \sum_{i=1}^{N_a} \sum_{j=1}^{w} \beta_i^{k,j}, 1 \leq k \leq N$.

   (c) Computes $\bar{v}_k = \sum \nu_i^k$

   (d) Computes the optimal coalition structure.

   (e) Assigns a server to coalition of size k a based on the values ν_i^k.

   (f) Chooses the best performer as the next coalition leader. The best performer is the one with the largest value $\sum_j \nu_i^j$.

Finding the optimal CS requires at most L operations with L the size of the PCL-list. The system starts with a predetermined coalition structure and coalition values.

4.5
A Reservation System Based On A Combinatorial Auction Protocol

The protocol introduced in this section targets primarily the *IaaS* cloud delivery model represented by AWS. Reservation systems are currently used by CSPs. For example, AWS supports reservations as well as spot allocation and offers a limited number of instance families, including M3 (general purpose), C3 (compute optimized), R3 (memory optimized), I2 (storage optimized), G2 (GPU) and so on. An instance is a package of system resources; for example, the `c3.8xlarge` instance provides 32 vCPU, 60 GiB of memory, and 2×320 GB of SSD storage. The resources auctioned are supplied by coalitions of servers in different racks and the cloud users request packages of resources.

The combinatorial auction protocol is inspired by the clock-proxy auction [14]. The clock-proxy auction has a clock phase, where the price discovery takes place, and a proxy phase, when bids for packages are entertained. In the original clock-proxy auction there is one seller and multiple buyers who bid for packages of goods.

For example, the airways spectrum in the US is auctioned by the FCC and communication companies bid for licenses. A package consists of multiple licenses; the quantities in these auctions are the bandwidth allocated times the population covered by the license. Individual bidders choose to bid for packages during the proxy phase and pay the prices they committed to during the clock phase.

Our protocol supports auctioning service packages; a packages consist of combinations of services in one or more time slots. The items sold are services advertised by coalitions of autonomous servers and the bidders are the cloud users. Each service is characterized by

1. A *type* describing the resources offered and the conditions for service,

2. The time slots when the service is available.

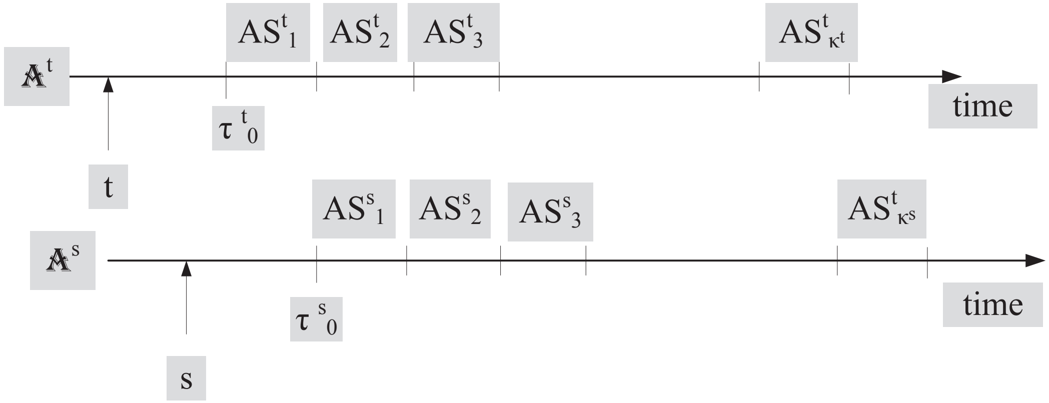


Figure 4.3: Auctions $\mathbb{A}^t$ and $\mathbb{A}^s$ conducted at times t and s, respectively. τ_0^t and τ_0^s are the start of the first allocation slots, AS_1^t and AS_1^s of the two auctions. The number of slots auctioned in each case are κ^t and κ^s, respectively.

4.5.1

Protocol Specification

The terms used to describe the protocol are discussed next. An *allocation slot* (AS) is a period of fixed duration, e.g., one hour, that can be auctioned. An *auction*, $\mathbb{A}^t$, is organized at time t if there are pending reservation requests which require immediate attention. Figure 4.3 shows two consecutive auctions at times t and s; during the first slot of auction $\mathbb{A}^t$ new reservation requests are received and the allocation slot AS_2^t is not fully covered; this slot becomes AS_1^s for $\mathbb{A}^s$.

A service $\mathcal{A}$ is described by a relatively small number of *attributes*, $\{a_1, a_2, \ldots, \}$. Each attribute a_i can take a number of distinct values, $v_i = \{v_{i,1}, v_{i,2}, \ldots\}$. The first attribute is the coalition size or equivalently the number of vCPS provided; other attributes could be the type of service and server architecture with two values “32-bit” and “64-bit;” another attribute could be “organization” with values “vN” (von Neumann), “DF” (data-flow), or “vN-GPU” (vN with graphics co-processor).

Call $\mathcal{S}^t$ the set of services the clients want to reserve during auction $\mathbb{A}^t$

$$\mathcal{S}^t = \{S_1^t, S_2^t, \ldots, S_{\nu^t}^t\} \quad \text{with} \quad S_i^t = [sId, (a_j, v_{j,k})] \tag{4.4}$$

A *reservation bundle,* $\alpha_{i,j}^t \subset \mathcal{S}^t$, is the set of services requested by client i in slot j of auction $\mathbb{A}^t$

$$\alpha_{i,j}^t = \{(S_{i,j,1}^t, r_{i,j,1}^t), (S_{i,j,2}^t, r_{i,j,2}^t), \ldots\} \tag{4.5}$$

with $r_{i,j,l}^t$ a measure of the quantity; for example, if the attribute is "service intensity" the quantity is the number of vCPUs.

An *advertised bundle,* $\beta_{k,j}^t \subset \mathcal{S}^t$, is the set of services advertised by coalition k in slot j of auction $\mathbb{A}^t$

$$\beta_{k,j}^t = \{(S_{k,j,1}^t, q_{k,j,1}^t, p_{k,1}^t), (S_{k,j,2}^t, q_{k,j,2}^t, p_{k,2}) \ldots\} \tag{4.6}$$

with $q_{k,j,l}^t$ a measure of the quantity of service l and $p_{k,l}$ the price per ECU of service S_l^t determined by coalition k. A *package*, $\mathcal{P}_i^t$ is a set of reservations for services requested by client i for slots $j_1, j_2, \ldots$ during auction $\mathbb{A}^t$.

$$\mathcal{P}_i^t = \{\alpha_{i,j_1}^t, \alpha_{i,j_2}^t, \ldots\} \tag{4.7}$$

4.5.2
The Clock Phase

Figure 4.4 illustrates the basic idea of a clock phase: the auctioneer announces prices and the bidders indicate the quantities they wish to buy at the current price. When the demand for an item increases, so does its price until there is no excess demand; on the other hand, when the offering exceeds the demand, the price decreases.

During the clock phase of auction $\mathbb{A}^t$ the price discovery is done for each time slot and for each type of service; a clock runs for each one of the κ^t slots and for each one of the ν^t services. Next we describe the clock phase for service S_l^t in slot j. Assume that there are n coalitions $\mathbb{C} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_n\}$ offering the service and m requests for reservations from clients $\mathbb{D} = \{\mathbb{D}_1, \mathbb{D}_2, \ldots, \mathbb{D}_m\}$.

A clock auction starts at clock time $t = 0$ and at price per unit of service for S_l

$$p_l^0 = \min_{\mathcal{C}_k}\{p_{k,l}\} \tag{4.8}$$

Call $\mathcal{C}_0$ the available capacity at this price and $\mathcal{D}_0$ the demand for service S_l^t offered at price p_l^0 in slot j

$$\mathcal{C}_0 = \sum_{k=1}^{n} q_{k,j,l}^t \text{ and } \mathcal{D}_0 = \sum_{i=1}^{m} r_{i,j,l}^t. \tag{4.9}$$

If $\mathcal{C}_0 < \mathcal{D}_0$ the clock c advances and the next price per unit of service is

$$p_l^1 = p_l^0 + \mathcal{I} \tag{4.10}$$

with $\mathcal{I}$ the price increment decided at the beginning of auction. There is an ample discussion in the literature regarding the size of the price increment; if too small, the duration of the clock phase increases, if too large, it introduces incentives for gaming [14].

The process is repeated at the next clock value starting with the new price. The clock phase for service S_l^t and slot j terminates when there is no more demand.

4.5.3

The Proxy Phase

In a traditional clock-proxy auction the bidders do not bid directly, they report to a proxy the price and the quantity of each item in the package they desire.

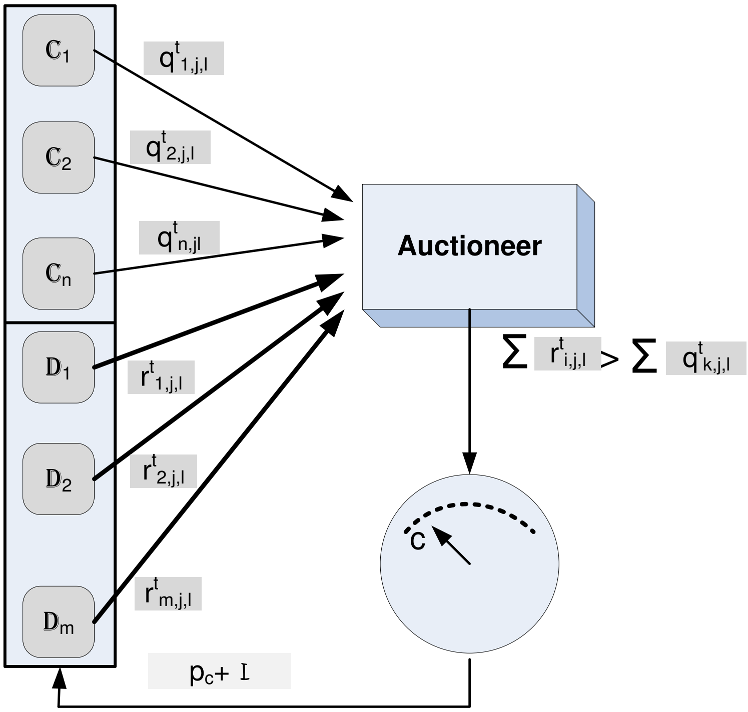


Figure 4.4: The clock phase for service S^t_l and slot j. The starting price is p^0_l given by Equation 4.8. The clock advances and the price increases from p_c to $p_c + \mathcal{I}$ when the available capacity at that price given by Equation 4.9 is exhausted; the demand is given by Equation 4.9.

The proxy then bids in an ascending package auction. In our application, the proxy phase of the auction consists of multiple rounds. The auction favors bids for long runs of consecutive slots when the service is provided by the same coalition. This strategy is designed to exploit temporal and spatial locality.

The auction starts with the longest runs and the lowest price per slot and proceeds with increasingly shorter runs and diminished incentives. Once a run of consecutive slots is the subject of a provisional winning bid, all shorter runs of slots for that particular service are removed from the coalition offerings.

During the first round only the longest run of consecutive slots for each one of the services offered by the participating coalitions is auctioned and only bidders that have committed to any of the slots of the run are allowed to bid. The price per slot for the entire run is the lowest price for any slot of the run the bidder has committed to during the clock phase of the auction.

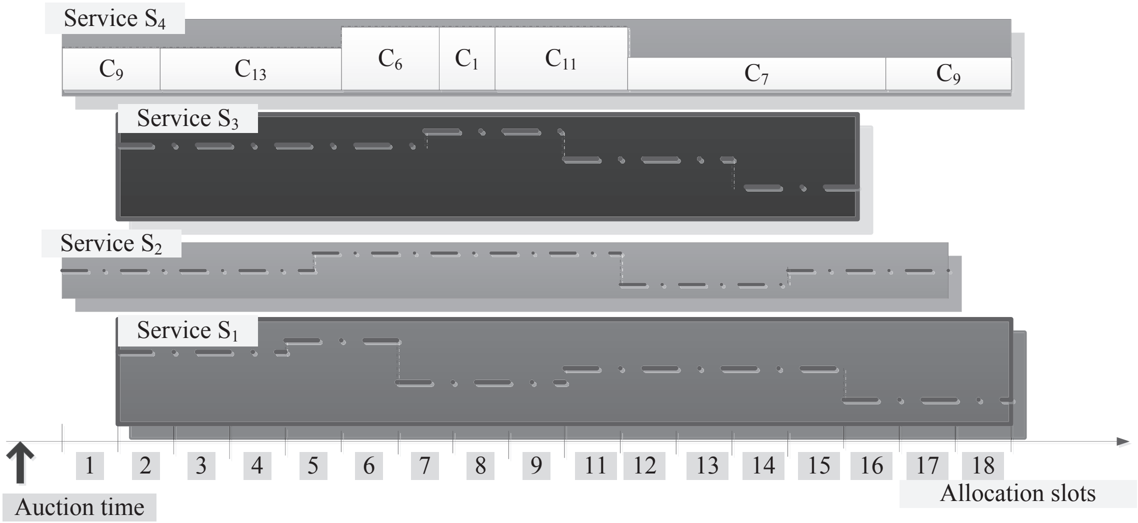


Figure 4.5: A snapshot at the end of the preliminary rounds of the proxy phase when there are four services offered and the auction covers 18 allocation slots. Dotted lines represent the quantity of service with provisional winners. Only the provisional winners for S_4 are shown, the clients labeled as $C_9, C_{13}, C_6, C_1, C_{11}, C_7$ and C_9.

If there are multiple bids for service S_l^t the *provisional winner* is the one providing the largest revenue for the coalition offering the service.

If κ_l^t is the longest run of consecutive slots for service S_l^t auctioned in the first round then, in the second round, a shorter run of $\kappa_l^t - 1$ slots is auctioned. The price for the entire run equals the second lowest price for any slot of the run the bidder has committed to during the clock phase of the auction times the number of the time slots in the run.

The length of the consecutive slot runs auctioned decreases and the incentives diminish after each round. The preliminary rounds end with the auction of a single slot for each service. At the end of the preliminary round each bidder is required to offer the price for the slot committed to during the clock phase. Figure 4.5 depicts a plausible snapshot at the end of the preliminary rounds of the proxy phase when four services $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ and $\mathcal{S}_4$, are offered and shows the provisional winners for service $\mathcal{S}_4$.

During the final round the bidders reveal the packages they want to reserve; these packages include only the provisional winners from the preliminary slots. Once all provisional winning bids for services in a reservation request are known, the auctioneer chooses the package that best matches the consumer's needs and, at the same time maximizes the profit for the cloud service provider. The *coalition* for a reservation request consists of the set of coalitions that provide the services in the winning package.

In this auction all bids are firm, they cannot be withdrawn. The auction is monotonic, the length of runs of consecutive slots auctioned decreases continually; this guarantees that the auction eventually terminates. Linear pricing guarantees that the price of any package can be computed with ease.

The effectiveness of the protocol is captured by several metrics including:

1. The *customer satisfaction index* - percentage of reservation requests fully or partially satisfied in each allocation slot given the total number of requests.

2. The *service mismatch index* - percentage of services requested but not offered in each allocation slot given the total number of services in that slot.

3. The *service success index* - percentage of services used in each allocation slot given all services offered in that slot.

4. The *capacity allocation index* - percentage of the capacity offered but not auctioned in each allocation slot given the capacity offered in that slot.

5. The *overbidding factor* - percentage of slots with a provisional winner that have not been included in any package given all slots offered at the beginning of the auction.

6. The *temporal fragmentation index* - percentage of services successfully auctioned in non-consecutive slots given all services successfully auctioned.

7. The *additional profit index* - percentage of additional profit of coalitions involved in the auction (the difference of the actual price obtained at the auction and the price demanded by the coalition) relative to the price demanded by the coalition.

4.5.4
Limitations And Vulnerabilities

The protocol is fairly complex and has at least one vulnerability. A bidder may be the provisional winner of services in slots not included in its winning package; such services will remain unassigned during the current auction. A solution is to penalize *excess bidding activity* and charge the bidder a percentage of the costs for these services. Another alternative is to include, in a reservation request, a set of "substitute services" for a service S_i. Then, during the last round of the proxy phase, the auctioneer could try to match services having provisional winners with unsatisfied requests for services.

The capacity offered, but not auctioned in each slot is available for *spot allocation* thus, it has the potential to be used, rather then being wasted. The capacity of a coalition left uncommitted at the end of the auction $\mathbb{A}^t$ for AS_1^t, the first slot of the auction, is then available for *spot allocation* at a price equal to $p_{k,l}$, while the free capacity in slots starting with AS_2^t can be offered at the next auction if this auction takes place before the beginning of the slot. This capacity is measured by the *spot allocation opportunity index.*

4.6
Protocol Analysis And Evaluation

We report on the results of our simulation experiments to gain some insight into the proxy phase of the clock-proxy auction of the PC^2P protocol. The system we wish to evaluate requires the description of the environment in which the auction takes place, the reservation requests, and the services offered:

1. The environment elements: n - the number of coalitions offering services in this round; m - the number of clients; and κ - the number of slots auctioned.

2. The package j requested by client i: α_i^n - the number of services in the package; the slots desired by the service S_k, ordered by the length of the run of consecutive slots; $r_{k,j}$ - the

intensity of service S_k in slot j; $p_{i,j}$ - the price per unit of service for slot j if client i was a provisional winner of that slot during the clock phase.

3. The service S_k provided by coalition $\mathbb{C}_k$ includes: γ_k - the largest run of consecutive slots for each offered service S_k; the profile of the service S_k - the slots offered ordered by the length of consecutive slots, when it is available; $q_{k,j}$ - the quantity of service S_k offered in slot j; and p_k - the price per unit of service offered by coalition $\mathbb{C}_k$.

For simplicity, we assume that a coalition offers one service only and the number of services is $\nu < n$. We also assume that all platforms have a maximum capacity of 100 vCPUs and that $q_{k,j}$, the quantity of service S_k offered for auction, and $r_{k,j}$, the quantity of S_k requested in slot j are the same for all the slots of an offered/requested run. The number of slots auctioned is fixed, $\kappa = 50$.

The range and the distribution of parameters for the protocol evaluation are chosen to represent typical cases. The parameters of the simulation are random variables with a uniform distribution:

(a) The number of coalitions and clients requesting reservations, n and m, respectively; the interval is $[200 - 250]$.

(b) The number of services offered and requested ν; the interval is $[10 - 20]$.

(c) The number of clients bidding for each service in a given slot; the interval is $[0 - 4]$.

(d) The capacity offered for auction for a service in a given slot; the interval is $[60 - 90]$ vCPUs.

(e) The services offered by a coalition; the interval is $[1 - \nu]$.

(f) The number of consecutive slots a service is offered in; the interval is $[1 - \kappa]$.

(g) The number of services in the package requested by a client; the interval is $[1 - 3]$.

(h) The number of consecutive slots of the services in the package requested by a client; the interval is $[1 - \kappa]$.

We also randomly choose the slots when the client is the provisional winner. The evaluation process consists of the following steps:

A. Initialization.

B. Preliminary rounds. Carry out γ preliminary rounds with $\gamma = \max_k \gamma_k$.

- In the first preliminary round auction κ_1 slots of service S_1, κ_2 slots of service S_2, and so on.
- Identify the first slot of each run and the reservation request that best matches the offer.
- Identify the provisional winners if such matches exist and remove the corresponding runs from the set of available runs. A match exists if the run consists of the same number of slots or is one slot longer than requested and if the capacity offered is at least the one required by the reservation request. For services without a match, remove the last slot, add both the shorter run and the last slot to the list of available runs.
- Continue this process until only single slots are available.

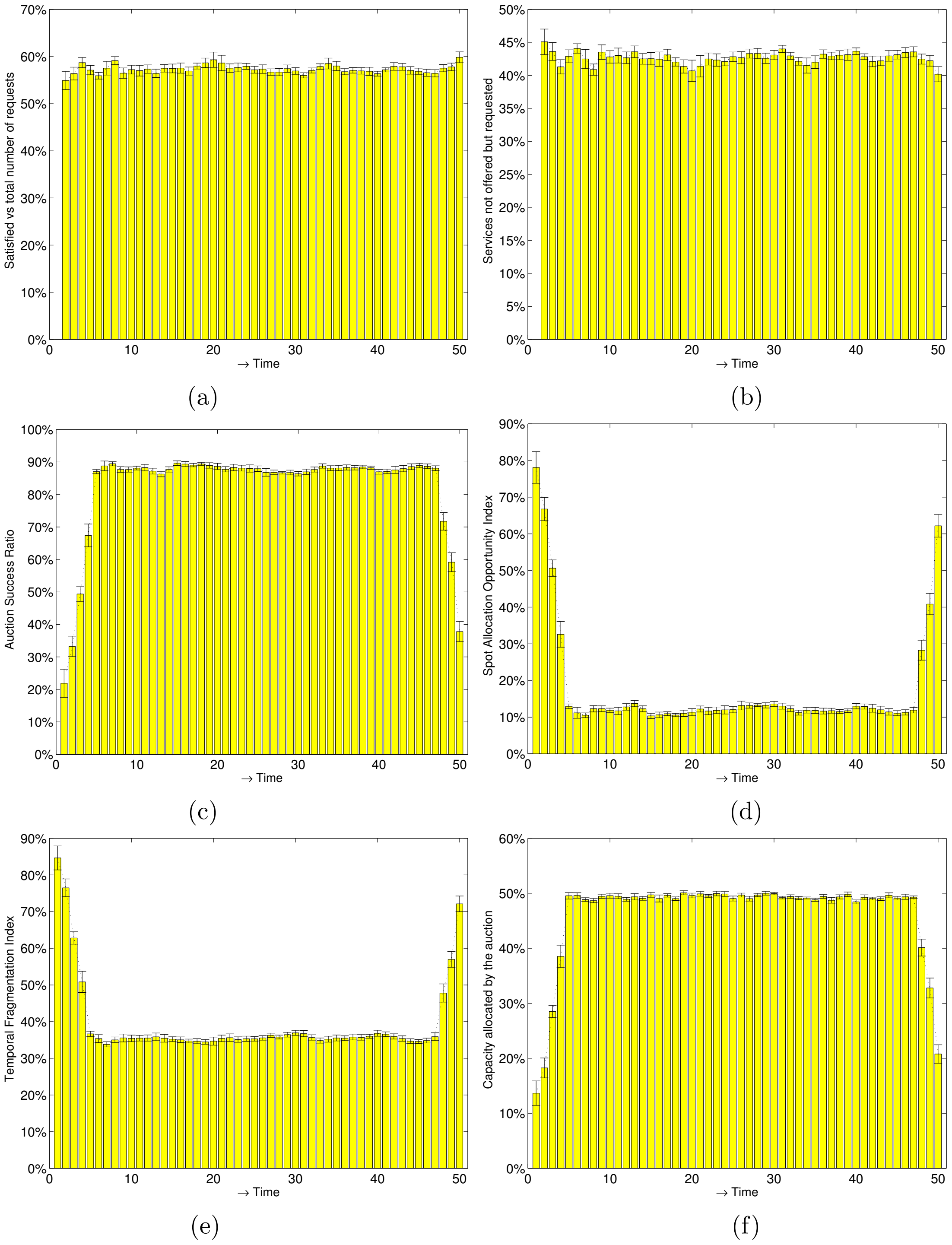
C. Final round. In this round we:

- Identify the packages for each client and if multiple packages exist determine the one which best matches the request.
- Compute the cost for the winning package for each client.

Figures 4.6(a)-(e) show several performance metrics including the customer satisfaction index, the service mismatch index, the auction success ratio, the spot opportunity index, the temporal fragmentation index, and the capacity allocation index. The simulation covers 50 time slots.

The 95% confidence intervals for the mean of all performance metrics are computed for 25 batches each one of 200 realization of each random variable. The simulation times are 6.4 seconds for $2,000$ runs and 11.7 seconds for $5,000$ runs. The confidence intervals are rather tight; this indicates that the performance of the protocol is relatively stable for the range of parameters explored in this evaluation.

The auction success rate is high, typically above 80%. The initial low auction success rate is an artifact of the manner we conducted the simulation; we picked up randomly the service start up time. The spot allocation opportunity index is in turn correlated with the auction success rate and shows that a significant fraction of the capacity is available for spot allocation.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 4.6: Proxy phase of an auction with 50 time slots. Indices of: (a) Customer satisfaction; (b) Service mismatch; (c) Auction success; (d) Spot allocation opportunity; (e) Temporal fragmentation; (f) Capacity allocation.

This result is correlated with the one in Figure 4.6(f) which shows that on average some 50% of the server capacity is not allocated by the reservation system and so is available for spot contention.

A reservation system covering 50% of the server capacity is probably the most significant result; it shows that self-management based on auctions can drastically improve server utilization. We live in a world of limited resources and cloud over-provisioning is not sustainable either economically or environmentally.

The service mismatch index is fairly high, typically in the 50% range and it is above 60% in a few slots. The customer satisfaction is correlated with the service mismatch and typically is in the region of 50%. In a realistic scenario, when coalitions maintain statistics regarding the services offered and avoid offering services unlikely to be demanded by the cloud users, the service mismatch would not affect the performance of the algorithm. Temporal fragmentation, though rather low, is undesirable. The overbidding factor 64 ± 2.93% is another indication that the protocol needs to be fine tuned.

Self-organization cannot occur instantaneously in an adaptive system and this simple observation has important consequences. It is critical to give autonomous cloud platforms, interconnected by a hierarchy of networks, the time to form coalitions in response to services demanded. Thus, self-management requires an effective reservation system and our results indicate that the reservation protocol is working well.

4.7 Conclusions

Self-organization and self-management offer an appealing alternative to existing cloud resource management policies; they have the potential to significantly alter the cloud computing landscape. So far, pragmatic means for the adoption of self-organization principles for large-scale computing and communication systems have eluded us. A main reasons for this state of affairs is that self-management has to be coupled with some mechanisms for cooperation; these mechanisms should allow autonomous servers, to act in concert towards global system goals. Cooperation means that individual systems have to partially surrender their autonomy. Striking a balance between autonomy and cooperation is a challenging task, it requires a fresh look at the mechanics of self-organization and the practical means to achieve it.

Practical implementation of cloud self-organization is challenging for several reasons including the absence of a technically suitable definition of self-organization, a definition that could hint to practical design principles for self-organizing systems and quantitative evaluation of the results. Computer clouds exhibit the essential aspects of complexity and it is inherently difficult to control complex systems.

We started our investigation with a realistic model of the cloud infrastructure, the hierarchical organization reported in [25] which seems inherently tied to hierarchical control. First, we compared hierarchical control which based on monitoring with a market model in which the servers of a WSC place bids for service requests and found out that the latter is much more effective than hierarchical control [112]. In the simple market model the servers act individually, rather than cooperating with each other, a fundamental aspect of self-organization. But cooperation is clearly needed because individual servers may not be able to supply the resources demanded by many data-intensive applications. Thus, we concluded that servers have to form coalitions to offer larger pools of resources. At the same time, it seemed obvious to us that complex applications with multiple phases would require packages of resources offered by different coalitions.

Algorithms for coalition formation based on combinatorial auctions are at the heart of the cloud ecosystem we propose. The path we chose seems logical as auctions have been successfully used for resource management in the past. Auctions do not require a model of the system, while traditional resource management strategies do. The auction-based protocol is scalable, and the computations can be done efficiently, though the computational algorithms involved are often fairly complex.

The results reported in Section 4.6 indicate that the performance of the protocol is relatively stable for the range of parameters explored in our evaluation. The protocol leads to a higher server utilization and it seems reasonable to expect that a fine tuned version of the protocol could further improve this critical performance measure.

The future work should address several problems revealed by this investigation. We have to address the effects of overbidding, the process which allows a client to become a provisional winner of one or more service slots and then, in the final round failing to acquire some of them. This situation is critical for the first slot of an auction as the next auctions could find clients for these slots. A more difficult problem is the temporal fragmentation which does not seem to have an obvious solution. We are also investigating a software architecture which supports cloud interoperativity and re-configuration.

CHAPTER 5
A MODEL FOR A SELF-ORGANIZING CLOUD ARCHITECTURE

The work reported in this chapter is based on [108]. The three traditional cloud delivery models – IaaS, PaaS, and SaaS – constrain access to cloud resources by hiding their raw functionality and forcing us to use them indirectly via a restricted set of actions. Can we introduce a new delivery model, and, at the same time, support improved security, a higher degree of assurance, find relatively simple solutions to the hard cloud resource management problems, eliminate some of the inefficiencies related to resource virtualization, allow the assembly of clouds of clouds, and, last but not least, minimize the number of interoperability standards?

We sketch a self-organizing architecture for very large compute clouds composed of many-core processors and heterogeneous coprocessors. We discuss how self-organization will address each of the challenges described above. The approach is *bid-centric.* The system of heterogeneous cloud resources is dynamically, and autonomically, configured to bid to meet the needs identified in a high-level task or service specification. When the task is completed, or the service is retired, the resources are released for subsequent reuse.

Our approach mimics the process followed by individual researchers who, in response to a call for proposals released by a funding agency, organize themselves in groups of various sizes and specialities. If the bid is successful, then the group carries out the proposed work and releases the results. After the work is completed, individual researchers in the group disperse, possibly joining other groups or submitting individual bids in response to other proposals. Similar protocols are common to other human activities such as procurement management.

5.1
Introduction

As discussed in Chapter 4, from the beginning, CSPs (Cloud Service Providers) made their cloud offerings available all based on the three delivery models, SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service), that represented the state of the art. As cloud computing gained traction, cloud vendors adapted their products and services to fit into the three delivery models available to them.

As a result, as the models became more used, they became the dominant design patterns and, more subtly, they became the dominant way of thinking about cloud computing. This has given rise to a constrained view of the possibilities afforded by the cloud. In this chapter, we attempt to break with the traditional mode of thinking and to look afresh at how cloud services are delivered.

Today's cloud computing landscape is partitioned into SaaS, PaaS, IaaS. These models provide very different types of services and have different capabilities and internal structure. The emergence of the three models during the first decade of the new millennium was well-justified, as they respectively targeted different types of application and distinct user groups. This gave rise to a strong ecosystem in which these models were the only alternatives.

With SaaS, software becomes an appliance, the administration of which is outsourced. Typically, little or no technical expertise is required to consume a SaaS offering. This makes it easy to sell, the end-user simply needs to trust the software vendor and the cloud provider to deliver the service competently. PaaS places some responsibility on the vendor/consumer to manage the service lifecycle, but without requiring them to engage in low-level systems administration and provisioning. IaaS provides the consumer with resources from which he can create familiar environments. These can be provisioned and customized to meet specific requirements. A high-level of customization can yield competitive advantage without associated costs of ownership, however, IaaS demands a high degree of administrative competence and places the onus on the consumer to shoulder the ensuant responsibility. Restricting access to services supported by software developed in-house, as in the case of SaaS, or to software installed after the verification and approval of the CSP, as in the case of PaaS, limits the security exposure of the CSP. The internal resource management policies and mechanisms to implement these policies are simpler for both SaaS and PaaS than the ones for IaaS.

Typically, a cloud stack is constructed in layers; each layer being composed of building blocks from the layer on which it sits. At the bottom of the stack is the infrastructure layer that employs virtualization to deliver physical resources to consumers. The virtualization process, by its nature, leads to resource fragmentation and hence to inefficiencies. Consider two virtual machines running on the same physical host, where one is overloaded and the other is underloaded. The boundary defined by the virtualization process prohibits the overloaded machine from exploiting the physical resources allocated to the unloaded machine and hypervisor. Since the clustering of virtual machines is the most common method of achieving horizontal scalability, this inefficiency propagates to the higher layers in the stack. Moreover, new inefficiencies arise in the upper layers of the stack in the form of contention for shared resources in a multi-tenancy environment. Current approach for dealing with shared resource contention, such as VM migration to achieve an optimal heterogeneous mix, are purely reactive. Could a more proactive approach be taken to shared resource management?

The Openstack's baremetal driver attempts to address the inefficiencies caused by virtualization fragmentation and hence could be argued to represent an emerging fourth cloud delivery model - MaaS (Metal-as-a-Service) [100]. However, even more so than IaaS, MaaS requires a high degree of technical skill to deploy and manage and this makes it, at least at present, difficult to widely consume. Furthermore, unless a physical resource is utilized to its fullest, there is still a resource utilization issue. However, it is now passed from the CSP to the consumer. Is it possible to deliver these physical resources without precipitating these inefficiencies?

The proliferation of CSPs has given rise to many different APIs for performing similar tasks, such as IaaS provisioning. Different providers innovate in different ways, however, these differences are not conducive to implementing portable or interoperable applications. Would a more declarative means of specifying services free us from the mechanics of how they are provisioned and hence foster portability and interoperability?

The "walled Gardens" of the current CSPs are analogous to the fragmented state of computer networks before the advent of the Internet. Since the introduction of the Internet Protocol, hardware and software has undergone a dramatic evolution allowing the Internet to became a critical infrastructural component of society. Today the Internet supports the Web, electronic commerce, data streaming, and countless other applications including cloud computing.

With the glue provided by the Internet Protocol, the Internet was free to develop organically; unconstrained by top-down regulation. Today, clouds are islands among which communication is difficult. Could a new delivery model bridge these islands and accelerate the development of the cloud ecosystem?

The work reported in this chapter attempts to answer some the questions posed above. It is motivated by the desire to improve and enrich the cloud computing landscape and by the desire to identify a disruptive technology that addresses some of the very hard problems at the core of today's cloud computing infrastructure and service delivery models. The architecture we propose has its own limitations, it cannot eliminate all the inefficiencies inherent to virtualization, requires the development of new families of algorithms for resource management and the development of new software. On balance this approach has compelling advantages as we shall see in Section 5.2.

5.2
A Cloud Architecture Model Based On Self-management And Auctions

As discussed in Section 3.3, the model we propose is based on the coalition formation and combinatorial auction ideas discussed in Chapter 4. The basic tenets of the model we propose are: autonomy of individual components, self-awareness, and intelligent behavior of individual components. Self-awareness enables servers to create coalitions of peers working in concert to respond to the needs of an application, rather than offering a menu of a limited number of resource packages, as is the case with AWS. Again, *global order results from local interactions.*

Cloud organization. We distinguish between a core server and a cloud periphery server, Figure 5.1. The *core* consists of a very large number of computational and storage servers, CS, dedicated to the cloud mission - the provision of services to the user community. The core servers are typically heterogeneous; co-processors and/or GPUs are attached to multi/many core processors.

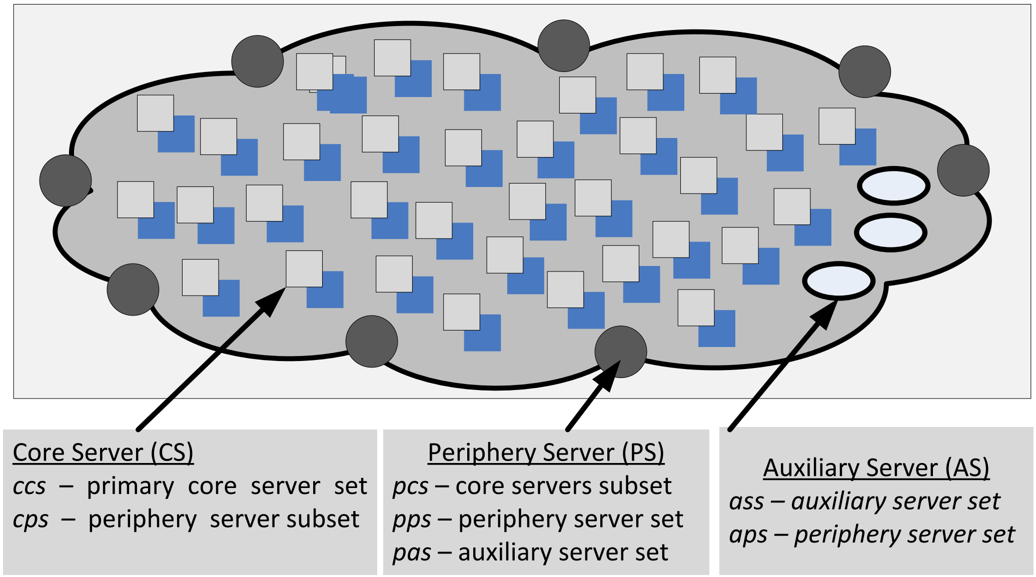

Core Server (CS)
ccs – primary core server set
cps – periphery server subset
Periphery Server (PS)
pcs – core servers subset
pps – periphery server set
pas – auxiliary server set
Auxiliary Server (AS)
ass – auxiliary server set
aps – periphery server set

Figure 5.1: The cloud consists of a core and a periphery populated by core and periphery servers, respectively; a group of auxiliary servers support internal services. Some of the data structures used for self-awareness are shown: (i) *ccl* - the set of primary contacts for a core server and *cps* - the subset of periphery servers known to core server; (ii) *pcs* - the subset of core servers known to the periphery server, *pps* - the set of all periphery servers, and *pas* - the set of all auxiliary servers; and (iii) *aas* - the set of auxiliary servers and *aps* -the set of periphery servers known to an auxiliary server. These data structures are populated during the initial system configuration and updated throughout its lifetime.

A relatively small number of *periphery servers*, PS, known to the outside world, act as access points to the core and as the nerve center of the system. There are also a few, auxiliary servers, AS, providing internal cloud services.

Operation modes. The cloud computing infrastructure is reconfigurable; a core server is an autonomous entity that can operate in several modes:

(i) M1: the server is configured to support multiple virtual machines, as in existing clouds.

(ii) M2: the server is either the leader or a member of coalition of core servers running under the control of one OS designed to support Single System Image (SSI) operation. This mode will be able to support data-intensive applications which require a very large number of concurrent threads/processes of one application.

(iii) M3: the server is either the leader or a member of coalition of core servers where each server runs under the control of one of the operating systems supported by the cloud. This mode is of particular interest for data-intensive applications with a complex workflow involving multiple applications and under strict privacy and security constraints; it is inspired by the clusters supported by AWS and by the MaaS (Metal as a Service) model [100].

(iv) M4: operation as a storage server.

Cloud core; the self-configuring monitor (ScM). The cloud core consists of computational and storage servers providing services to the user community. To address the natural tension between self-awareness and the desire to maintain minimum state information, each core server has a relatively small set of *primary contacts*, and, when necessary, it has access to a much larger pool of *secondary contacts. Secondary contacts*, are core servers that can be accessed via the periphery servers known to a core server. The ScM is a Virtual Machine Monitor (VMM) [137]; its main function is to configure and supervise the system, rather than tightly control the virtual machines (VMs) running on the physical server, as is the case for Xen [24], Xoar [44], or other VMMs. The ScM is a component of the software stack running on each core server at all times; its main functions are:

1. To respond to external events following on a set of policies and goals; e.g., choose the operation mode of the server in response to a service request and configure the server accordingly.

2. To evaluate the performance of the server relative to its long- and short-term goals after the completion of a service request. To maintain information about the environment and the past system history such as: the primary contacts, the energy consumption, the ratio of successful versus failed bids for service, the average system utilization over a window of time, the effectiveness of the cost model used for bidding, and all other data relevant for the future behavior of the server.

3. To evaluate the behavior of the application, whether the information in the service request is adequate and the resulting SLA adopted after the bid was successful.

4. To support system security and control communication with the external world.

The effectiveness of the model depends on the ability of the ScM to make most decisions using local information thus, minimizing communication with other systems. Minimal intrusiveness is another critical requirement for the ScM; this implies that the ScM should monitor system events by observing control data structures, rather than being directly involved in the flow of control and reacting only to prevent policy violations.

Cloud periphery. A self-organizing cloud includes a relatively small number of periphery servers. The cloud periphery plays a critical role in a self-organizing system, *it acts like the nervous system of the cloud* and it links the core to the outside world and with the internal services. This strategy allows the core servers to maintain a minimum amount of data about the external and the internal environment and should be dedicated to their main mission of providing services to the user community. At the same time, they should be more agile and able to respond to unforeseen events, to accommodate software updates, and to balance the load placed on the auxiliary servers (AS). The main functions of a periphery server are:

1. To provide an interface of the cloud with the outside world.

2. To aid in self-organization, self-management, and self-repair.

3. To act as a broker - sending service requests, to a subset of core servers and then forwarding bids from core servers to the entity requesting service. Finally, they could be used to mediate the creation of a Service Level Agreement (SLA) between the two parties.

Auxiliary servers. Support internal services such as: accounting, billing, policy and goal management, system statistics, patch management, software license management, and reputation and trust services.

Distinguishing features of the model. Some of the mechanisms in our model have been discussed in the literature e.g., [11, 26, 27, 47, 58, 91], others have been incorporated in different cloud architectures [9, 143]. We believe that enough progress has been made on a range of topics related to the model we propose; the next step is to combine existing ideas with new ones in a coherent new architecture. Several elements distinguish the model we propose from others:

1. The system is reconfigurable - the autonomous computational and storage servers can operate in several modes to maximize QoS.

2. Coalitions of core servers are created dynamically to respond to service requests. Winning coalitions are determined by the results of auctions.

3. Virtualization simulates the interface of an object by several means including multiplexing, aggregation, emulation, and multiplexing combined with emulation. While exiting clouds are based

exclusively on multiplexing - multiple virtual machines sharing the same server - in our model, in addition to multiplexing, we consider aggregation - the creation of a single virtual server from several physical servers, and federation of servers.

4. A service request is formulated in a Cloud Service Description Language (CSDL). Once a bid is selected a client-specific, legally-binding Service Level Agreement (SLA) can be negotiated between the cloud service provider and the cloud client.

5. At the completion of a service request the system evaluates the performance of the service provision, the client and the application. This information can be fed back into future service negotiations.

We now examine the manner that the self-organization model addresses some of the challenges discussed in Section 1.2. First, it allows the three service delivery models to coexist. For example, an organization interested in the SaaS or PaaS delivery models, could request servers operating in the M2 or M3 modes depending on expected load. In the case of a request for IaaS the periphery would contact core servers already operating in Mode M1 or it would initiate the formation of a new coalition. The lifetime of a contract varies widely; the SLAs for a SaaS, could be rather long, months, years, while the one for PaaS could be just for the time to complete a specific job, or for multiple jobs extended over a period of time. QoS guarantees for systems operating in Modes M2 and M3 can be provided due to performance isolation; indeed, the leader of a coalition has access to accurate information about the resource consumption and the available capacity of the coalition members.

The inefficiencies inherent to virtualization by multiplexing cannot be eliminated; at the same time, we expect that virtualization by aggregation will introduce other inefficiencies. Some of them will be caused by distributed shared memory, which, depending on the interconnect, can be severe, others by the interactions between the guest OS and distributed hypervisor. Virtualization plays no role for coalitions of servers operating in Mode M3 and the only overhead is due to the ScM's role in shielding the group from outside actions.

5.3
Virtualization By Aggregation

Single System Image is a computing paradigm where a number of distributed computing resources are aggregated and presented via an interface that maintains the illusion of interaction with a single system [20, 28]. The concept of seamlessly aggregating distributed computing resources is an attractive one, as it allows the presentation of a unified view of those resources, either to users or to software at higher layers of abstraction. The motivation behind this aggregation process is that by hiding the distributed nature of the resources the effort required to utilize them effectively is diminished. Aggregation can be implemented at a number of levels of abstraction, from custom hardware and distributed hypervisors through to specialized operating system kernels and user-level tools. Single system image embodies a rich variety of techniques and implementations with a history going back over three decades. Three of these are most relevant to the self-organizing clouds concept: distributed hypervisors, kernel-level SSI and APIs that aggregate coprocessors. A brief overview of each is presented next, before the suitability of each for implementing Mode M3 is examined.

Single System Image. There have been several implementations of distributed hypervisors that provide a single guest operating system instance with a single system image of an entire cluster [42, 76, 132]. This approach has the advantage of being largely transparent to the guest OS, eliminating the need for far-reaching kernel modifications and hence allowing for a wider selection of guest operating systems. These systems build a global resource information table of aggregated resources such as memory, interrupts and I/O devices. A virtualized view of these hardware resources is then presented to the guest operating system. Standard distributed shared memory techniques are used to provide a global address space. The single system image is maintained by alternating as necessary between the guest OS and the hypervisor via trap instructions.

There have been numerous implementations of kernel-level SSI, with the oldest dating back to the 1970s. There are two basic approaches in terms of design philosophy: dedicated distributed operating systems and adaptations of existing operating systems. The latter approach can be further divided into over-kernel and under-kernel implementations. Under-kernel (or *"under-ware"*

[163]) systems seek to transparently preserve existing APIs, such as POSIX. In contrast, over-kernel systems provide additional or extended APIs that allow for more efficient utilization of distributed resources. The under-kernel approach is of most interest here as it allows existing application code to run without modification.

Notable implementations of kernel-level SSI include MOSIX [23], OpenSSI [164] and Kerrighed [115]. Of these, Kerrighed provides the greatest level of transparency, supporting a unified file system, process space (including migration) and memory space. Uniquely for a Linux-based system, Kerrigheds distributed memory model allows for thread migration, albeit with the attendant inefficiencies caused by OS-managed remote paging; early experiments on commodity hardware resulted in a significant slowdown compared to a true SMP machine [105]. The extent of the transparency provided by Kerrighed is highlighted by an experiment where detailed load balancing results for Kerrighed could not be obtained because it was not possible to determine the load on individual cluster nodes as commands such as `ps` display all processes running cluster-wide [123].

A number of abstraction models have been proposed for co-processors that present a single system image to the application developer or runtime environment. For example, RASCAL (Reconfigurable Application-Specific Computing Abstraction Layer) is a software library developed by SGI that provides functionality for device allocation, reconfiguration, data transport, error handling and automatic wide scaling across multiple FPGAs [43]. Virtual OpenCL (VCL) [19] provides a similar API-level abstraction for GPUs distributed across a cluster. Depending on the size of the request, the VCL runtime environment may allocate one or more GPUs installed in a single machine or a collection of devices spanning several machines. The Symmetric Communication Interface (SCIF) API performs a similar role for Intel Many-Integrated-Core (MIC) coprocessors [72].

Cloud resource aggregation. Mode M3 of the self-organizing cloud architecture is characterized by the aggregation of physical computational resources into virtual servers. At this speculative stage of development, it is too early to definitively identify the most appropriate aggregation mechanisms. Nevertheless, a review of the strengths and weaknesses of candidate techniques and technologies is instructive.

Creating virtual cloud servers by abstracting at the hypervisor level has the compelling advantage that existing virtual machine images, operating systems, and application code can run unmodified. Under this approach, a self-configuration monitor would examine the role assigned to it as part of a bid and determine if it needs to scale horizontally. If so, it pulls in other servers which merge with it to create an aggregated virtual server using a distributed hypervisor. As with all distributed shared memory systems, the performance of the overall system is closely bound with that of the interconnect used, and applications with unsuitable memory access patterns can undergo significant performance degradation compared to a true SMP machine. For those applications that are suitable, however, the transparent aggregation would allow for a significant degree of horizontal scaling without any changes to virtual machine image, operating system or application code. A drawback of the hypervisor-level approach is the inability to scale the virtual server up or down after it has been created given the current state of distributed hypervisor and operating system functionality.

The operating system approach to aggregation would be similar to that for hypervisors: in response to a bid, a number of virtual machine instances, potentially across a number of physical machines. These virtual machine instances would run a single system image operating system, such as Kerrighed, and as such could be combined into clusters, each of which would aggregate the underlying virtual resources in a transparent fashion. The benefits of this approach are that using higher-level operating system abstractions, such as processes and file systems, may result in more efficient distributed execution of some workloads compared to the low-level shared memory approach taken by distributed hypervisors. Furthermore, advanced SSI implementations such as Kerrighed allow nodes to join and leave the cluster dynamically, facilitating elastic scaling. The most obvious drawback is the need to use a specific SSI operating system, which may lead to software compatibility issues. More fundamentally, the design assumptions that underpin software packages may preclude the SSI approach. For example, a scalable web serving cluster might utilize multiple instances of the Apache web server running on multiple physical or virtual machines, which are clustered together to load balance incoming requests. Replicating this arrangement using process migration would be difficult, as a host of issues, such as filesystem contention, would arise if multiple instances of Apache were run simultaneously [133].

The various single system image abstraction of distributed coprocessing resources could also have a role in self-optimizing clouds. Several cloud providers support the creation of VM instances with attached GPU coprocessors, with support for FPGAs and MICs likely to become available in the future. This heterogeneity of compute resources would provide self-organizing clouds with a rich palette in terms of how bids could be configured, at the cost of a much larger number of potential resource combinations. One approach would be to statically associate coprocessing resources with virtual machines when responding to bids. So, if a VM is expected to run an application that can benefit from GPU acceleration then a suitable number of GPU cores can be assigned to the VM as a single virtual GPU. The obvious drawbacks to this approach are efficiency and scalability. Granting virtual machines exclusive access to coprocessors prevents those coprocessors being used by other VMs when there is no work available. Similarly, static coprocessor allocations limit the horizontal scalability available to individual VMs by preventing them from taking advantage of idle coprocessors in the resource pool that are not assigned to them. The ideal solution, therefore, is API support at the application level so that the global resource pool of coprocessors can be utilized by application running in VMs. However, this approach would require support at the application level.

5.4 Simulation Experiments

It is impractical to experiment with a very large number of physical systems, so we chose numerical simulation to investigate the feasibility of some of the ideas discussed in this chapter. We believe, that at this stage, the emphasis should be on qualitative rather than quantitative results. Thus, our main concern when choosing the parameters of the simulation was to reflect the scale of the system and "typical" situations. The very large number of core servers we chose to experiment with forced us to consider simpler versions of the protocols for bidding.

The simulation experiments reported in this section run on the Amazon cloud. Several storage optimized `hi1.4xlarge` instances[1] running on a 64-bit architecture were used.

[1] An `hi1.4xlarge` instance can deliver more than 120,000 4 KB random read IOPS and between 10,000 and 85,000 4 KB random write IOPS. The maximum throughput is approximately 2 GB/sec read and 1.1 GB/sec write.

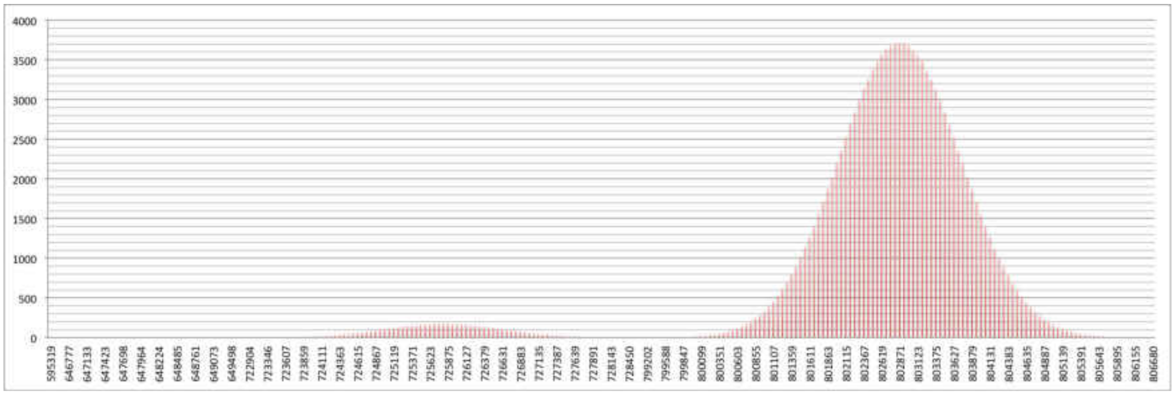


Figure 5.2: Histogram of the number of secondary contacts for $m = 10$.

This choice was motivated by the scale of the system we simulated; the description of the very large number of core servers requires a very large address space thus, systems with access to a very large physical memory. The simulation ran concurrently on 16 virtual cores (vCPUs) delivering 35 ECUs (Elastic Cloud Units) and used 60.5 GB of main memory. The instances were launched on the same cluster and servers were connected by a non-blocking 10 Gbps Ethernet. The simulation, implemented in *C++*, used extensively multi-threading to reduce the execution time; each one of the 16 virtual cores ran a single thread.

First, we report on experiments for the self-organization stage. The self-organization algorithm consists of the following steps: (i) the M periphery servers broadcast their identity and address; (ii) a core server randomly selects m of the M periphery servers, includes them in its *cps* list, and informs each one of them about its selection; (iii) a periphery server constructs the list of core servers known to it, *pcs*.

We chose several parameters to configure the system : (a) the number N=8,388,608 of core servers was randomly chosen in the $10^6 - 10^7$ range; (b) the number of periphery servers was chosen to be $M = 1,000$; and (c) the size $n = 500$ of the primary contact list, *pcl*, maintained by each core server. To determine the number of secondary contacts for the N core servers, we experimented with several values for the size of *pl*, the list of periphery servers known to a core server, $m = 5, 10, 20, 50$. We study p, the size of the *pcs* list maintained by each one of the M periphery servers; this list gives the number of the core servers known to each periphery server. As expected, $\bar{p}$, the average p, increases when m increases, the dependence is nearly linear:

m	$\bar{p}$	$\bar{p}/N$
5	$41,859$	0.494%
10	$83,593$	0.996%
20	$166,363$	1.983%
50	$409,707$	4.884%

(5.1)

We are also interested in the distribution of the number of secondary contacts S that can be reached by each one of the N core servers when required. The histogram of the distribution of the secondary contacts, shows that when $m = 5$ some $7,200$ core servers have a number of secondary contacts close to $207,423$, the average number of secondary contacts. On the other hand, when $m = 50$ we have a multi-modal distribution; for example, there are some $1,500$ core servers claiming 93% of the total number of core servers as a secondary contact. When $m = 50$ each core server is able to claim a very large fraction of all other core servers, and there are $8,388,608$ of them, as secondary contacts. The average number of secondary contacts, $\bar{S}$, increases with m but much faster than $\bar{p}$:

m	$\min(S)$	$\max(S)$	$\bar{S}$	$\bar{S}/N$
5	$124,303$	$210,086$	$207,423$	2.472%
10	$490,466$	$807,182$	$799,403$	9.529%
20	$2,289,093$	$2,797,363$	$2,768,772$	33.006%
50	$7,241,339$	$7,748,921$	$7,701,688$	91.811%

(5.2)

Based on these results we concluded that $m = 10$ is a good choice for the number of periphery servers known to each core server, see Figure 5.2. This choice provides each core server with a sufficient number of potential secondary contacts - about 10% of the total number of core servers. The execution time for the self-organization stage was about two hours.

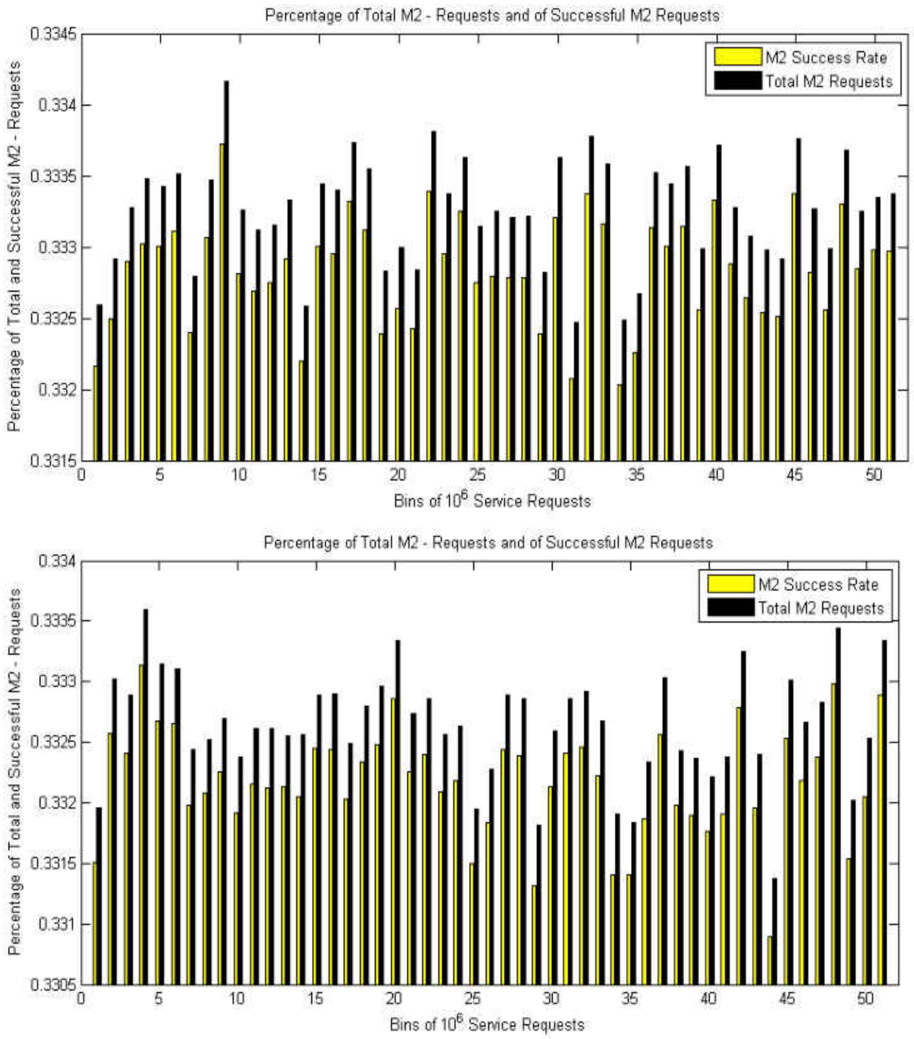


Figure 5.3: The ratio of requests and the success rate for M2 mode. (Left) *Exp1*; (Right) *Exp2*.

Table 5.1: Summary of results for experiments *Exp1* and *Exp2*.

Mode	Exp	Requests	Percent of total	Failed	Success rate
		in a bin	$\times 10^6$	requests	(%)
M1	*Exp 1*	17,001,928	0.33370	0	100
	Exp 2	16,871,479	0.33049	0	100
M2	*Exp 1*	16,974,809	0.33260	21,254	99.875
	Exp 2	16,972,653	0.33049	23,630	99.61
M3	*Exp 1*	17,001,928	0.33370	0	100
	Exp 2	17,180,818	0.33655	0	100

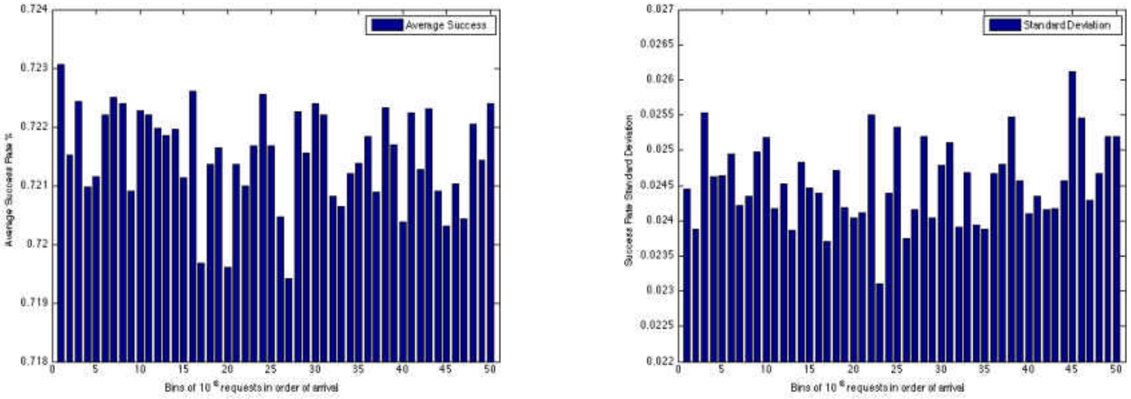


Figure 5.4: *Exp3* - coalitions initiated by a core server. (left) Average and (right) standard deviation of the success rate for M2 service requests.

The second group of experiments is designed to show if the auctions, central to the architecture we propose, enables the system to respond effectively to outside requests. There are two modes to create a coalition:

C1. Coalitions initiated by a periphery server. After receiving a service request a periphery server sends an invitation to a subset of the core servers in its pcs list to join a coalition. The message invites bids and includes the details of the service request. When one or more coalitions is formed, a leader is elected and the periphery server transmits the bid(s) to the client. Finally the periphery server may negotiate the generation of an SLA.

C2. Coalitions initiated by a core server. After receiving a service request, a periphery server sends an invitation to a subset of core the servers known to it and provides the details of the service request. The Zookeeper (*http://zookeeper.apache.org/*) coordination software based on the Paxos algorithm [94, 97] can then be used to elect a leader among the candidates. The leader then uses its primary contacts and if necessary its secondary contacts to build a coalition. If successful it generates a bid. In our experiments a periphery server selects a subset of 0.1% of the core servers connected to it and invites them to become a leader.

The first mode is likely to require less communication overhead thus, shorter time to generate a coalition, but also considerably more intensive participation of the periphery servers which could then become a bottleneck of the system. The second approach seems more aligned to self-organization principles thus, more robust, as the periphery servers play only the role of an intermediary.

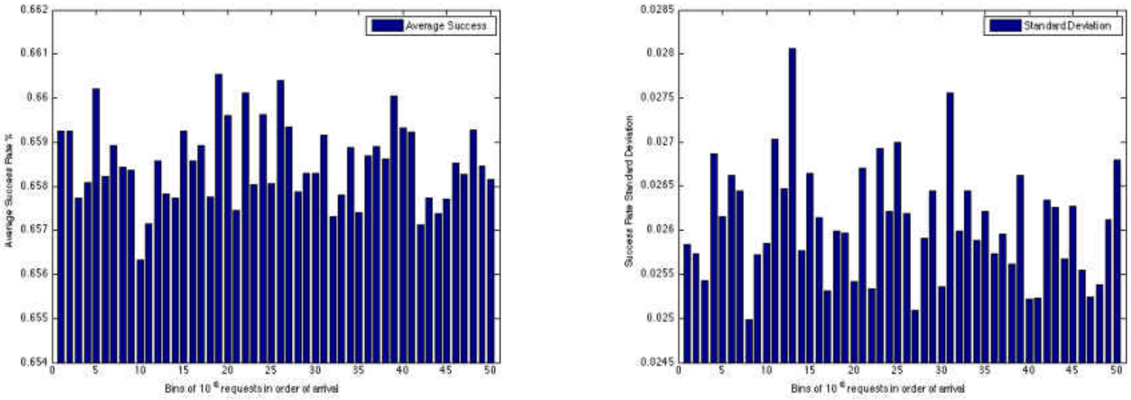


Figure 5.5: *Exp4* - coalitions initiated by a periphery server. (left) Average and (right) standard deviation of the success rate for M2 service requests.

The large number of core servers force us to consider several simplifying assumptions: (i) we assume that all core servers have the same capacity, but they are heterogeneous and thus the cost for providing services can be different; (ii) a request is characterized only by the amount of resources needed, while in a realistic scenario other elements such as deadlines, energy consumption, costs, privacy and security would be considered; (iii) in a realistic scenario a bidder would offer several alternatives with different levels of compliance to user requirements and cost; sophisticated bidding algorithms would choose the optimal bid, while in our simulation a bid consists only of the cost of providing the service and the lowest bid is always selected; (iv) if there is no bid for a request then that request is added to a queue of unsatisfied requests, rather than being used to trigger a renegotiation of the service. This added level of complexity could be addressed in a future iteration of the system.

We experimented with different: number of core servers, number of service requests, initial system load, demand for resources, and the modes to create a coalition.

For the first four experiments we considered the initial organization resulting from the previous experiment with $m = 10$, a cloud with $8,388,608$ core servers and $M = 1000$ periphery servers subjected to 50×10^6 service requests. A service request arrives at a randomly chosen periphery server and has an equal probability of being of modes M1, M2, or M3.

1. *Exp1* - a lightly loaded system with coalitions initiated by a core server. Initially, the state of each core server is randomly selected such that the fraction of servers in each state is: 20% - sleep state, 40%, 15%, and 25% running, in M1, M2, and M3 modes, respectively. The initial load of a core server is uniformly distributed in the range $30\% - 80\%$ of the server's capacity. The arrival and service processes have exponential distributions with inter-arrival times $\lambda = 1.5$ and, respectively, $\mu = 1.2$ units. The workload required is uniformly distributed in the range $[0.1 - 8.0]$ SCUs (Server Compute Units) and only primary contacts are used to assemble a coalition.

2. *Exp2* - similar to Exp1 but with the workload uniformly distributed in the $[0.1 - 40.0]$ SCUs and secondary contacts are used to assemble a coalition when the primary list is exhausted.

3. *Exp3* - highly loaded system with coalitions initiated by a core server. Initially, the state of each core server is randomly selected; the fraction of servers in M1, M2, and M3 modes is the same, 33%. The initial load of the core servers is uniformly distributed in the range $50\% - 80\%$ of the server capacity. The arrival processes has an exponential distributions with inter-arrival times $\lambda = 1.0$ and the service process has a heavy-tail distribution, a Pareto distribution with $\alpha = 2.0$. The workload required is uniformly distributed in the $[0.1 - 40.0]$ SCUs and only primary contacts are used to assemble a coalition.

4. *Exp4* - similar to *Exp3* but a coalition is initiated by a periphery server.

5. *Exp5*- heavily loaded, small scale cloud with the number of core servers, $N = 100$, and the number of periphery servers, $M = 2$, and coalitions are initiated by the periphery servers and subjected to $1,000$ service requests. The initial load of the core servers is uniformly distributed in the range $70\% - 90\%$ of the server capacity. The arrival and service processes have exponential distributions with inter-arrival times $\lambda = 1.5$ and, respectively, $\mu = 1.2$ units. The workload required is uniformly distributed in the $[0.1 - 8.0]$ SCUs and only primary contacts are used to assemble a coalition.

6. *Exp6* - same as *Exp5* but the workload required is uniformly distributed in the $[0.1 - 40.0]$

SCUs. Secondary contacts are used when needed.

The results of *Exp1* and *Exp2* for the three delivery modes, M1, M2, and M3 are summarized in Table 5.1 and are illustrated in Figure 5.3. We construct bins of 10^6 requests and compute the success rate for each bin; this success rate is computed as the ratio of the number of successful bids to the number of service requests in the bin. We call this quantity a *rate* because the requests are ordered in time thus, the ratio reflects the number of successes in the interval of time correspond to the arrival of 10^6 requests of a certain type. The results in Table 5.1 show that the workload required expressed as the number of SCUs has little effect on a lightly loaded system. The first simulation experiment when the core servers use only their primary contacts took slightly less than 10 hours, while the second one, when the load was higher and the core servers had to use their secondary contacts, took almost 24 hours.

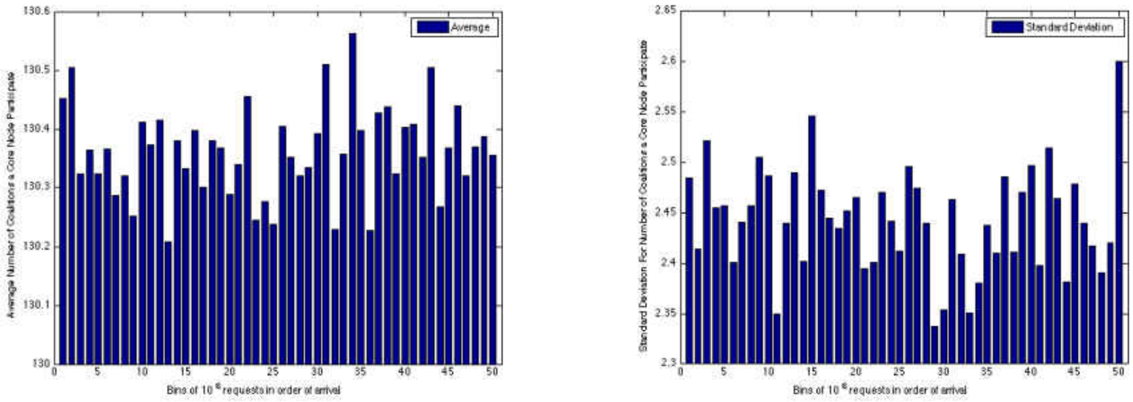


Figure 5.6: Average and standard deviation of number of coalitions a core server is a member of. Coalitions initiated by a core server.

For the last four experiments we only show the results for mode M2 because the simulation does not distinguishes between the three modes. To illustrate the system behavior for *Exp3* and *Exp4* we show histograms of the average success rate and its variance for each of the three modes in Figures 5.4 and 5.5, respectively. We split a 10^6 bin into 10^3 sets with 10^3 requests in each set and compute the standard deviation using the success rate in each set. The success rates range from 65% to close to 80% with relatively small standard deviation for each bin. The core-initiated coalitions have slightly larger success rates because in this mode the leader uses secondary contacts after exhausting the set of primary contacts, therefore has access to a larger population. The average number of coalitions a core server participates in and the standard deviation for these two experiments are shown in Figures 5.6 and 5.7. The averages for the core-initiated mode are slightly larger. The executions times for each one of the two experiments are about 24 hours.

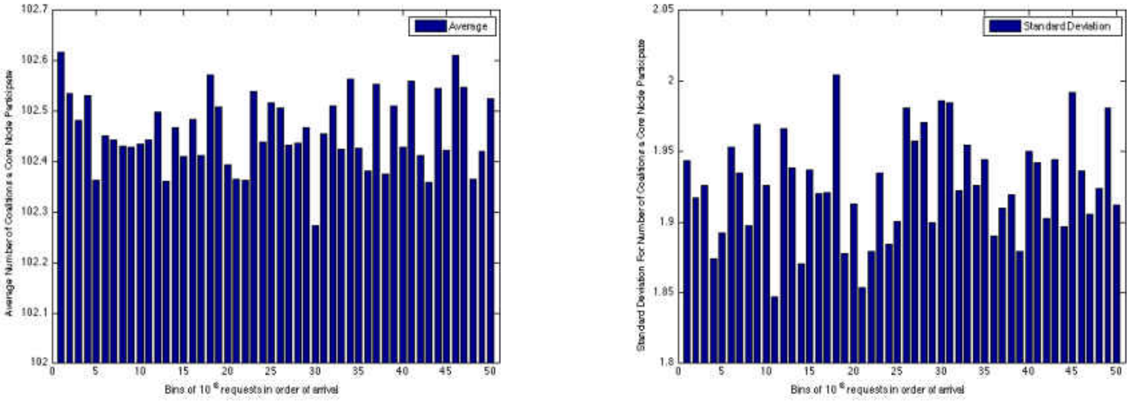


Figure 5.7: Average and standard deviation of number of coalitions a core server is a member of. Coalitions initiated by a periphery server.

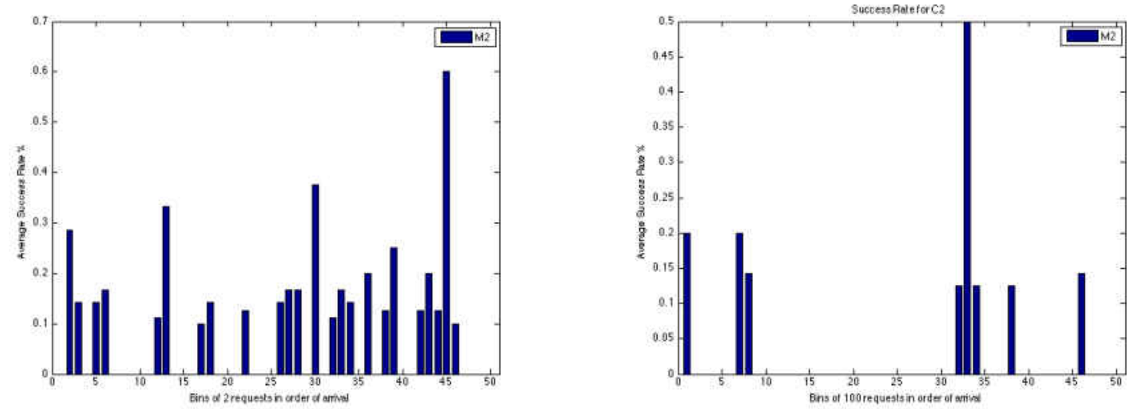


Figure 5.8: The ratio of requests and the success rate for M2 mode; (Left) *Exp5*; (Right) *Exp6*.

Lastly *Exp5* and *Exp6* show what we expected, namely that in a small, overloaded system, the success rate is low, there are no bids for many requests thus, the admission control works well, Figures 5.8.

5.5
Conclusions And Future Work

The IaaS cloud delivery model, and the Amazon Web Services in particular, support not only a cost-effective, but also a very convenient and elastic computing environment.

The fact that we are able to simulate the behavior of a complex system with almost ten million components at a cost of slightly more than \$100 (the time for the three simulation experiments are $2 + 10 + 24) = 36$ hours and the cost per hour is \$3.0. gives us an indication of the appeal of cloud computing for scientific and engineering applications.

Any proposal for a novel cloud computing delivery model has to address the feasibility, the performance, and the cost involved. The first question is whether the state of the art of the technologies involved allow the development of systems based on the new principles. The next, and the more difficult question, is whether the performance of the proposed systems justifies the investment in the new ideas. This is a more challenging task because qualitative arguments such as security and performance isolation, user-centric organization, self-organization and self-management, support for aggregation, assembly of systems of systems – clouds of clouds in our context – have to be balanced against quantitative measures such as revenue, utilization, QoS, and so on.

Numerical simulation is widely used in science and engineering for getting insights into physical phenomena, for checking the accuracy and limitations of theoretical models, for testing hypothesis, or for comparing different design options and parameters of the systems we plan to build. In all these cases, we start with a model, an abstraction of a physical system or a phenomena, then we carry out the simulation based on this model and, finally, we validate the simulation results by comparing them with theoretical predictions and, whenever feasible, with measurements of the physical system.

How to validate the simulation results for a new model of a complex system, a heterogeneous system with a very large number of components and with many interaction channels among them? In this case we have only the model, there is no physical system, and no comprehensive theory describing the system as a whole. One can only validate the algorithms which control the evolution of the system; we can do that by showing that the model of the system subject to different stimuli reacts according to the behavior prescribed by the algorithm. Admittedly, this is a weak form of validation of simulation results, but the only one available. Of course, one could test the algorithms on a small-scale system, but scalability is an insidious problem, and there is no guarantee that an idea working beautifully on a small-scale system will work at all at a larger scale.

For this reason in this chapter we limit our discussion to qualitative rather than quantitative results and two aspects of the feasibility of the self-organizing model: the initial system organization and the bidding mechanisms. The model is rather complex and an in-depth analysis in necessary to determine optimal parameters such as: the ratio between the core and periphery servers, number of primary and secondary contacts, the size of the population invited to place bids in response to a service requests. We use several simplifying assumptions discussed in Section 5.4, as well as, parameters we believe reflect a "typical" behavior for the simulation experiments. While, during the initial system configuration, we used a random selection process, more intricate learning algorithms are likely to lead to a more effective selection of primary contacts by each core server.

The simulation experiments we conducted show that the initial system organization phase can be tuned to provide each core server with a balanced number of primary and secondary contacts. The bidding mechanisms seem to work well when the workload is relatively low. In spite of the larger overhead, core-initiated coalition formation seems more effective than periphery-initiated coalitions.

As our objective is a qualitative, rather than a quantitative analysis we do not compute confidence intervals for our simulation results. Our primary goal was to test the scalability of the self-organizing architecture and the simulation experiments for a system with a close to nine million core servers took almost 36 hours on a very powerful cloud configuration. The cost for a large number of runs necessary to report confidence intervals was prohibitive.

The disruptive approach we propose poses a number of new questions to be addressed by future research. For example, some of the open research questions related to the role of single system image techniques in self-organizing clouds are: (i) Can the distributed hypervisors be modified to support dynamic horizontal scaling? (ii) What are the performance differences between distributed hypervisors and best-of-breed SSI operating systems? (iii) How do SSI techniques, such as process and thread migration, perform when used over with low-latency networking interconnect technologies such as RDMA Verbs [70]? (iv) Can abstract cloud service descriptions be efficiently mapped to static heterogeneous clusters of multicore, GPU, FPGA and MIC processing resources? (v) Can dynamic horizontal scaling of coprocessor resources be integrated into abstract cloud service descriptions?

Further work includes the development of a cloud service description language and of efficient bidding algorithms which take into account the workload, deadlines, energy consumption, costs, and possibly other parameters of a service request. An interesting concept developed by Papadimitriou and his collaborators is that of "mixability" the ability of an entity to interact with others [86]. The authors argue that mixability is a critical element in the evolution of species, a good indicator of the fitness and of the ability of an individual to transmit the positive traits to its descendants. In our system mixability will quantify the ability of a server to successfully interact with other servers in each of the three modes of operation.

Lastly, some of the highly desirable features of the architecture we introduce, such as security and privacy due to isolation of the servers assigned to a specific application, come at a higher cost for providing the service. The main appeal of the solution we propose is the potential to satisfy a very broad range of user preferences and allow service level agreements that reflect the specific user requirements and the contractual obligation of the CSP for a specific service and user. This becomes increasingly more important as information privacy laws differ from country to country and a CSP has to obey the security and privacy rules and laws demanded by each user.

CHAPTER 6
A CLUSTERING ALGORITHM FOR SCALE-FREE ORGANIZATIONS OF SCALE-FREE NETWORKS

The work reported in this chapter is based on [128]. In this chapter we introduce an algorithm for clustering around the nodes with a high connectivity in a scale-free network and discuss results of a simulation experiments to evaluate the algorithm. The clustering algorithm can be used for self-organization of large-scale systems and can be applied to social networks where scale-free organization appears naturally.

6.1
Motivation And Related Work

The possible interactions of the entities in complex biological, social, economic, or computing systems can be described by a graph where vertices represent active entities and the edges represent the communication channels between them. In this chapter we investigate algorithms for grouping together, or clustering, the nodes of a special type of graphs, graphs used to represent scale-free networks.

Clustering of computer and communication systems can be viewed as a form of virtualization by aggregation. A number of physical systems are grouped together to achieve a well-defined goal e.g., to increase reliability as is the case of RAID storage systems, to support effective management of resources in a large-scale system, or to satisfy QoS requirements.

Clustering algorithms have been investigated for some time [74]. They have many applications; for example, in a social network such algorithms can be used to group together individuals with similar interests or to identify individuals who are most influential in some area and their followers. In a sensor network the sensors can be organized in clusters and transmit their measurements to a nearby cluster leader which could then filter the data and report the results to monitoring stations; this organization increases the lifetime of the network as the radio-frequency component of the majority of sensors are only required to transmit at low power level thus, it takes a longer time to exhaust their power reserves [107].

Many other large-scale computing systems are organized hierarchically; in each cluster a *leader* is responsible for monitoring and controlling the activity of the cluster nodes and interacts with other cluster leaders to achieve common system objectives. Such an organization is more efficient as communication delays are shorter, the overhead for resource management is lower, and the information used to control the cluster better approximates the state of the cluster.

Hierarchical organization is a common approach to accommodate the complexity of many large-scale social, economical, and computer and communication systems [122]. Scalability becomes a critical concern as the systems are increasingly more complex e.g., social networks are projected to reach a billion users, today's computer networks connecting hundreds of million of computers, computer clouds with millions of servers, the future smart power grid infrastructure expected to have a similar number of nodes including customers, power generators, and transmission lines, and so on.

Several models of graphs have been investigated starting with the Erdös-Rény model [48] where the number of vertices is fixed and the edges connecting vertices are created randomly. This model produces a homogeneous network with an exponential tail; connectivity follows a Poisson distribution peaked at the the average degree $\bar{k}$ and decaying exponentially for $k >> \bar{k}$. An evolving network, where the number of vertices increases linearly and a newly introduced vertex is connected to m existing vertices according to a preferential attachment rule is described by Barabási and Albert in [5, 6, 21]. Networks whose degree distribution follows a power law are called scale-free networks.

The term "scale-free" means an organization with properties invariant to the scale of the system. In a scale-free organization the probability $p(k)$ that an entity interacts with k other entities decays as a power law $p(k) \approx k^{-\gamma}$ with γ a constant and k a positive integer. This probability is independent of the type and the function of the system, the identity of its constituents, and the relationships between them.

The graph of a scale-free network is non-homogeneous, there are a few nodes with a high degree of connectivity and the majority of the nodes are only connected with few other nodes. A number of studies have shown that scale-free networks have remarkable properties such as: (a) robustness against random failures [22]; (b) favorable scaling [5, 6]; (c) resilience to congestion [53]; (d) tolerance to attacks [156]; and (e) small diameter [36] and small average path length [21].

Networks with a power-law distribution of node degrees, such as scale-free networks, may appear naturally in social networks and other virtual organizations which are inherently heterogeneous, there are a few highly connected individuals and a very large number of individuals with few connection [121]. Many complex systems enjoy a *scale-free organization* [21, 22]. Though, sometimes the statistics used to identify the organization of a system as scale-free is questionable, several instances of virtual organizations, as well as man-made systems, seem to enjoy this type of organization.

Empirical data for several man-made systems confirm the existence of scale-free organization. Examples abound, e.g., the power grid of the Western US has some $5,000$ vertices representing power generating stations; in this scale-free network $\gamma \approx 4$. The scale-free organization appears naturally in social networks. For example, the collaborative graph of movie actors where links are present if two actors were ever cast in the same movie follows the power law with $\gamma \approx 2.3$. The probability that q pages of the World Wide Web point to one page is $p(k) \approx k^{-2.1}$ [22]. Recent studies indicate that $\gamma \approx 3$ for the citation of scientific papers. When the scale-free network is generated using the preferred attachment model [21] and when $\gamma \approx 3$, then the larger the number of the nodes the better the approximates the theoretical distribution. .

When scale-free organization occurs naturally we can apply clustering algorithms to group together individuals around the highly connected ones. In other instances we have first to construct a virtual interconnection network and only then we can apply clustering algorithms. For example, the servers of a large-scale computing system are interconnected by a physical interconnection network with a topology dictated by consideration such as physical proximity, power supply, and so on. In this case one can construct an overlay network, or a virtual network, with a scale-free topology and then apply the clustering algorithms to group together the computer systems around highly connected nodes.

A major advantage of an overlay network with a scale-free topology is that it provides a path to self-organization. Indeed, the highly connected nodes in this overlay network can act as the nerve centers of the system and control the clusters formed around them. This can be done by individual nodes which are autonomous and use only local information. As each node of the scale-free networks is aware of its own degree the ones with a degree higher than a given threshold, say 10, could act as leaders and all the other nodes can be clustered around them; for example, a node will join the cluster formed around a leader at the minimum distance.

6.2
Scale-free Organization

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

$$p(k) = \frac{1}{\zeta(\gamma, k_{min})} k^{-\gamma}. \tag{6.1}$$

In this expression k_{min} is the lowest degree of any node, and for the applications we discuss in this grant request $k_{min} = 1$; ζ is the Hurvitz zeta function[1]

$$\zeta(\gamma, k_{min}) = \sum_{n=0}^{\infty} \frac{1}{(k_{min} + n)^{\gamma}} = \sum_{n=0}^{\infty} \frac{1}{(1 + n)^{\gamma}}. \tag{6.2}$$

A scale-free network is *non-homogeneous*; the majority of the nodes have a low degree and only a few nodes are connected to a large number of links, Figure 6.1. The average distance d between the N nodes, also referred to as the diameter of the scale-free network, scales as $\ln N$; in fact it has been shown that when $k_{min} > 2$ a lower bound on the diameter of a network with $2 < \gamma < 3$ is $\ln \ln N$ [36].

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

The moments of a power law distribution play an important role in the behavior of a network. It has been shown that the giant connected component (GCC) of networks with a finite average vertex degree and divergent variance can only be destroyed if all vertices are removed; thus, such networks are highly resilient against faulty constituents. These properties make scale-free networks very attractive for interconnection networks in many applications including social systems [121], peer-to-peer systems, and sensor networks [107].

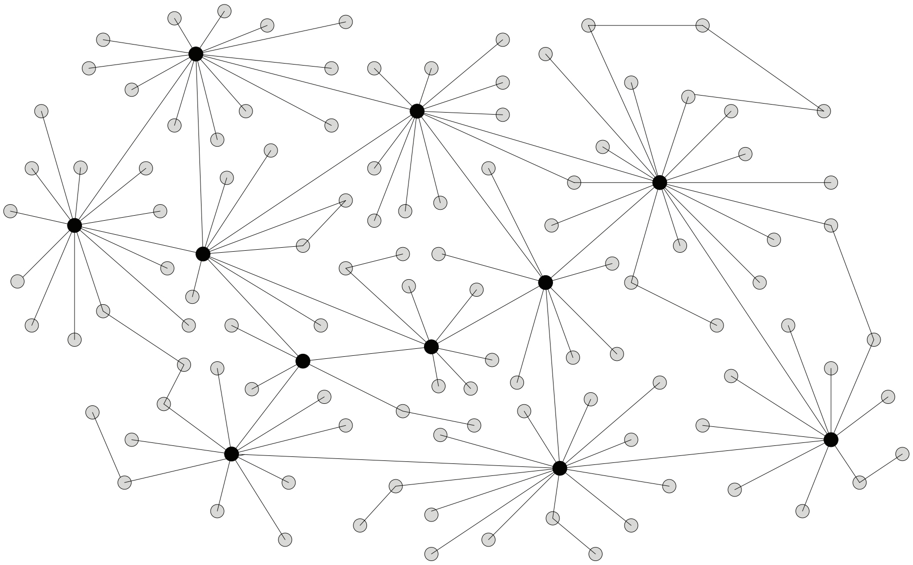


Figure 6.1: A scale-free network is non-homogeneous; the majority of the vertices have a low degree and only a few vertices are connected to a large number of edges; the majority of the vertices are directly connected with the vertices with the highest degree.

Another important property is that the majority of the nodes of a scale-free network are directly connected with the nodes of higher degree, see Figure 6.1. For example, in a network with $N = 130$ nodes and $m = 215$ links 60% of the nodes are directly connected with the five nodes with the highest degree, while in a random network fewer than half, 27%, have this property [6]. Thus, the nodes of a scale-free network with a degree larger than a given threshold T could assume the role of "core nodes" and assume management functions; the other nodes assume the role of computational and storage servers. This partition is autonomic; moreover, most of the server nodes are at distance one, two, or three from a core node which could gather more accurate state information from these nodes and with minimal overhead. In the next example if $k_{lim} = 4$ then 92.5% of the nodes are servers.

As an example, consider the case $\gamma = 2.5$ and the minimum node degree, $x_{min} = 1$; we first determine the value of the zeta function $\zeta(\gamma, x_{min})$ and approximate $\zeta(2.5, 1) = 1.341$ thus, the distribution function is $p(k) = k^{-2.5}/1.341 = 0.745 \times (1/k^{2.5})$, where k is the degree of each node. The probability of nodes of degree $k > 10$ is $\text{Prob}(k > 10) = 1 - \text{Prob}(k \leq 10) = 0.015$. This means that at most 1.5% of the total number of nodes will have more than 10 links connected to them; we also see that 92.5% of the nodes have degree $1, 2$ or 3. Table 6.1 shows the number of nodes of degrees 1 to 10 for a very large network, $N = 10^8$.

Table 6.1: A power-law distribution with degree $\gamma = 2.5$; the probability, $p(k)$, and N_k, the number of nodes with degree k, when the total number of vertices is $N = 10^8$.

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

The four graph models mentioned in this section are sometimes abbreviated as: ER (Erdös-Rény), BA (Barabási - Albert), WS (Watts-Strogatz), and SF (Scale-free) models, respectively [53]. Throughout this chapter we shall use the terms networks, nodes, and links when we discuss a physical system; we shall use the terms graphs, vertices, and arcs when we discuss the model of a system.

Given an Erdös-Renyi (ER) graph G_{ER} with N vertices we wish to rewire this graph and produce a new graph G_{SF} where the degrees of the vertices follow a power-law distribution. The procedure proposed in [95] consists of the following steps:

1. Given $0 < \alpha < 1$ we assign to each node i a probability

$$p_i = \frac{i^{-\alpha}}{\sum_{j=1}^{N} j^{-\alpha}} = \frac{i^{-\alpha}}{\zeta_N(\alpha)} \quad \text{with} \quad \zeta_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha}. \tag{6.3}$$

2. We select a pair of vertices i and j and create an edge between them with probability

$$p_{ij} = p_i p_j = \frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)} \tag{6.4}$$

and repeat this process n times.

Then the probability that a given pair of vertices i and j is not connected by an edge h_{ij} is

$$p_{ij}^{NC} = (1 - p_{ij})^n \approx e^{-2np_{ij}} \tag{6.5}$$

and the probability that they are connected is

$$p_{ij}^{C} = (1 - p_{ij}^{NC}) = 1 - e^{-2np_{ij}}. \tag{6.6}$$

Call k_i the degree of vertex i; then the moment generating function of k_i is

$$g_i(t) = \prod_{j \neq i} [p_{ij}^{NC} + t p_{ij}^{C}]. \tag{6.7}$$

The average degree of vertex i is

$$\bar{k}_i = t \frac{d}{dt} g_i(t)|_{t=1} = \sum_{j \neq i} p_{ij}^{C}. \tag{6.8}$$

Thus,

$$\begin{aligned} \bar{k}_i &= \textstyle\sum_{j \neq i} (1 - e^{-2np_{ij}}) = \sum_{j \neq i} \left(1 - e^{-2n \frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)}} \right) \\ &\approx \textstyle\sum_{j \neq i} 2n \frac{(ij)^{-\alpha}}{\zeta_N^2(\alpha)} = \frac{2n}{\zeta_N^2(\alpha)} \sum_{j \neq i} (ij)^{-\alpha} \end{aligned} \tag{6.9}$$

This expression can be transformed as

$$\bar{k}_i = \tfrac{2n}{\zeta_N^2(\alpha)} \textstyle\sum_{j \neq i} (ij)^{-\alpha} = \frac{2ni^{-\alpha} \sum_{j \neq i} j^{-\alpha}}{\zeta_N^2(\alpha)} = \frac{2ni^{-\alpha} \left(\zeta_N(\alpha) - i^{-\alpha} \right)}{\zeta_N^2(\alpha)}. \tag{6.10}$$

The moment generating function of k_i can be written as

$$\begin{aligned} g_i(t) &= \prod_{j\neq i}[p_{ij}^{NC} + tp_{ij}^{C}] = e^{(1-t)\bar{k}_i} = \prod_{j\neq i} e^{-(1-t)p_{ij}^{C}} \\ &\approx \prod_{j\neq i}[1-(1-t)p_{ij}^{C} = e^{(1-t)\sum_{j\neq i} p_{ij}^{C}} = e^{(1-t)\bar{k}_i} \end{aligned} \tag{6.11}$$

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

$$p_{d,i}(k) = \frac{1}{k!}\frac{d^k}{dt^k} g_i(t)|_{t=0} \approx \frac{\bar{k}_i}{k!} e^{-\bar{k}_i}. \tag{6.12}$$

When $N \to \infty$ then $\zeta_N(\alpha) = \sum_{i=1}^{N} i^{-\alpha}$ converges to the Riemann zeta function $\zeta(\alpha)$ for $\alpha > 1$ and diverges as $\frac{N^{1-\alpha}}{1-\alpha}$ if $0 < \alpha < 1$. For $0 < \alpha < 1$ Equation 6.3 becomes

$$p_i = \frac{i^{-\alpha}}{\zeta_N(\alpha)} = \frac{1-\alpha}{N^{1-\alpha}} i^{-\alpha} \tag{6.13}$$

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

$$k = p_i \times mN = 2mN\frac{1-\alpha}{N^{1-\alpha}} i^{-\alpha} = 2m(1-\alpha)\left(\frac{i}{N}\right)^{-\alpha} \tag{6.14}$$

Indeed, the total number of edges in graph is mN and the graph has a power law distribution. Then

$$i = N\left(\frac{k}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} \tag{6.15}$$

From this expression we see that there is a one-to-many correspondence between the unique label of the node i and the degree k; this reflects the fact that multiple vertices may have the same degree k. The number of vertices of degree k is

$$n(k) = N\left(\tfrac{k}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} - N\left(\tfrac{k-1}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}} = N\left(\tfrac{k-1}{2m(1-\alpha)}\right)^{-\frac{1}{\alpha}}\left(\left(1+\tfrac{1}{k}\right)^{-\frac{1}{\alpha}} - 1\right) \tag{6.16}$$

We denote $\gamma = 1 + \frac{1}{\alpha}$ and observe that

$$\left(1+\tfrac{1}{k}\right)^{-\frac{1}{\alpha}} = \quad 1+\left(-\tfrac{1}{\alpha}\right)\left(\tfrac{1}{k}\right)^{-\frac{1}{\alpha}}+\tfrac{1}{2}\left(-\tfrac{1}{\alpha}\right)\left(-\tfrac{1}{\alpha}-1\right)\left(\tfrac{1}{k}\right)^{-\frac{1}{\alpha}-1}+.... \tag{6.17}$$

We see that

$$n(k) = \quad N\left(\tfrac{(k-1)(\gamma-1)}{2m(\gamma-2)}\right)^{-\gamma+1} \times \left((1-\gamma)\left(\tfrac{1}{k}\right)^{-\gamma+1} - \tfrac{\gamma(1-\gamma)}{2}\left(\tfrac{1}{k}\right)^{-\gamma} + ...\right) \tag{6.18}$$

We conclude that to reach the value predicted by the theoretical model for the number of vertices of degree k, the number of iterations is a function of N, of the average degree $2m$, and of γ, the degree of the power law.

6.3
Construction of Scale-free Networks

A distributed algorithm to construct scale-free overlay topologies with an adjustable exponent was proposed in [152]. The algorithm adopts the equilibrium model discussed in [53]. The algorithm is based on random walks in a connected overlay network $G(V,E)$ viewed as a Markov chain with state space V and a stationary distribution with a random walk bias configured according to a Metropolis-Hastings chain. We assign a weight $p_i = i^{-\alpha},\ 1 \leq i \leq N, \alpha \in [0,1)$ to each vertex and add an edge between two vertices a and b with probability $p_a / \sum_{i=1}^{N} p_i \times p_b / \sum_{i=1}^{N} p_i$ if none exists; the process is repeated until mN edges are created and the mean degree is $2m$. Then the degree distribution is

$$p(k) \ \sim k^{-\gamma}, \quad \text{with} \quad \gamma = (1+\alpha)/\alpha. \tag{6.19}$$

The elements of the transition matrix $P = [p_{ij}]$ are

$$p_{ij} = \begin{cases} \frac{1}{k_i} \min\left\{ \left(\frac{1}{j}\right)^{\frac{1}{\gamma-1}} \frac{k_i}{k_j}, 1 \right\} & (i,j) \in E \\ 1 - \frac{1}{k_i} \sum_{(l,i)\in E} c_{il} & i = j \\ 0 & (i,j) \notin E \end{cases} \tag{6.20}$$

with k_i the degree of vertex i.

6.3.1
A Distributed Algorithm

The algorithm to generate the scale-free network Γ with N nodes and $\mid E \mid$ edges proposed in [152] assumes that each node has a unique ID, nId, $1 \leq nId \leq N$. Our implementation of the algorithm consists of the following steps:

1. Set L the random walk length, e.g., $L = 10$.

2. Set the number of nodes already rewired, $n_{rewired} = 0$.

3. Select at random a node e.g., node **a** and check if it has any edge that has not been rewired yet.

   (a) If NO repeat step 3.

   (b) If YES pick up one of the edges at random and save both endpoints of that edge.

4. Check which one of the endpoints has higher degree, if they were same pick one of at random.

5. Set the number of hops for random walk $n_{hop} = 0$.

6. Draw a random number $0 < \kappa < 1$.

7. Pick up at random a node in the neighborhood of the current node **a**, e.g. node **b**.

8. Given the degree d_a of node **a** with vId_a and the degree d_b of node **b** with vId_b calculate

$$h = \frac{d_a}{d_b} \left[\frac{nId_a}{nId_b} \right]^{\frac{1}{\alpha\gamma-1}}. \tag{6.21}$$

   (a) If $h > \kappa$ choose node **b**.

   (b) If $h \leq \kappa$ choose node **a**.

9. Increment the number of hops $n_{hop} = n_{hop} + 1$.

   (a) If $n_{hop} \neq L$ and $n_{hop} < L$ go to Step 6.

   (b) If $n_{hop} = L$ save the node as the target node **c** then go to Step 6.

   (c) Else save the node as the second target node **d**.

10. Connect target nodes to each other.

11. Remove the edge found in Step $3b$.

12. Mark the edge you found as a rewired edge.

13. Increment the number of nodes already rewired, $n_{rewired} = n_{rewired} + 1$.

   (a) If $n_{rewired} \leq E$ go to Step 3.

   (b) Else, the algorithm terminates as we have rewired all edges.

6.3.2
A Modified Algorithm For Construction of Scale-free Networks

Stopping after mN iterations as suggested by [152] leads to an undesirable distribution of the degrees of vertices. We have to slightly modify the algorithm to allow for the revisiting of an already rewired node and to continue to iterate until the number of degree one vertices reaches the theoretical value calculated in Section 6.2 for the corresponding value of γ, the degree of the power law distribution.

The execution times required by this version of the algorithm to reach the distribution predicted by the theory are summarized in Table 6.2. The results are for $N = 1,000$ nodes are the averages over 10 runs. The experiments were conducted on an Intel Core 2 Duo E7500 system with 4 GB of memory and a clock rate of 2.93 GHz running under Fedora 12 64-bit.

Table 6.2: The time required by the algorithm to construct the scale-free network to converge to the theoretical value for degree one vertices is a function of N, the number of vertices. T is the mean execution time, *Std* is the standard deviation and CI is a 95% confidence interval for the mean execution time.

N	T	Std	95% CI
	(sec)	(sec)	(sec)
1,000	53.1	68.7	39.6 - 66.5
10,000	28.9	39.5	21.1 - 36.5
100,000	278.6	16.6	275.3 - 281.9
1,000,000	9,473.6	424	9390.54 - 9556.7

6.4 A Clustering Algorithm

We assume that a scale-free network has been created. The nodes with a degree equal or larger than a threshold known to all nodes $LeaderTsh$ will be called *cluster leaders* and the nodes with a degree lower than $LeaderTsh$ will be called regular nodes or simply nodes. Due to the scale of the system there is no single site holding the information about all the nodes instead, each node has the following information:

1. The the node Id, $nodeId$.

2. The degree of the node, $nodeDeg$.

3. The threshold $LeaderTsh$ allows each node can determine if it is a leader or a regular node, $nodeDeg \geq LeaderTsh$. We choose $LeaderTsh = 10$.

4. The $neighborList$, the list of pairs $(nodeId, netAddress)$, of all nodes directly connected to the node.

5. The time t_0 when the clustering processing should start.

The goal of the algorithm is to identify: (a) The nodes connected to a cluster leader; (b) The network connecting the cluster leaders. The algorithm uses two types of messages: (1) *Type 1 - cluster initiation*, messages sent by a leader to all its neighbors; *Type 2 - request to join a cluster*, message sent by a node to the leader node at the shortest distance. We shall use a modified version of an epidemic algorithm when a node s_i re-sends an incoming message from the node $s_{i,j}$ to all other nodes in its connected nodes list, but $s_{i,j}$.

The algorithm is asynchronous and, in absence of global knowledge, about the system a node should be able to determine if information provided by a leader is delayed due to a slower communication link and when it should proceed to making the decision which cluster it should join. All nodes share a time interval, τ_{end}, which can be used to set up a timer; when the timeout occurs, all service nodes start the decision process leading to selection of the cluster a service node decides to join.

At the end of this asynchronous algorithm each leader would have built two tables:

a. The cluster table, *clusterTab*, the list of service nodes which joined the cluster built around the core node. For each service node the list includes:

1. The *nodeId* of the node requesting to join the cluster.

2. The distance from the leader to a node, given by the hop count, $hopCount$.

3. The path from the leader to the node, a list of nodes traversed by the *Type 1* message to reach the node.

b. The Leader table, *LeaderTab*, the list of other leader nodes directly connected to this leader.

Each regular node maintains a table of all possible clusters it could join, the *tempTab*; at the end of the algorithm the node will record the cluster it intends to join.

At time t_0 all nodes start execution of the algorithm consisting of the following steps:

1. All nodes start their timer set to expire after τ_{end} seconds.

2. Each leader sends to all nodes in its *neighborList* a *Type 1* message containing its $LNodeId$ with a $hopCount = 1$.

3. Upon receiving a *Type 1* message, a regular node performs the following actions:

   - Parse the message and identify: (i) the leader sending the message, $LNodeId$; (ii) the path *msgPath*; and (iii) The number of hops, *hopCnt*.

   - Check the *tempTab* for entries from the same leader.

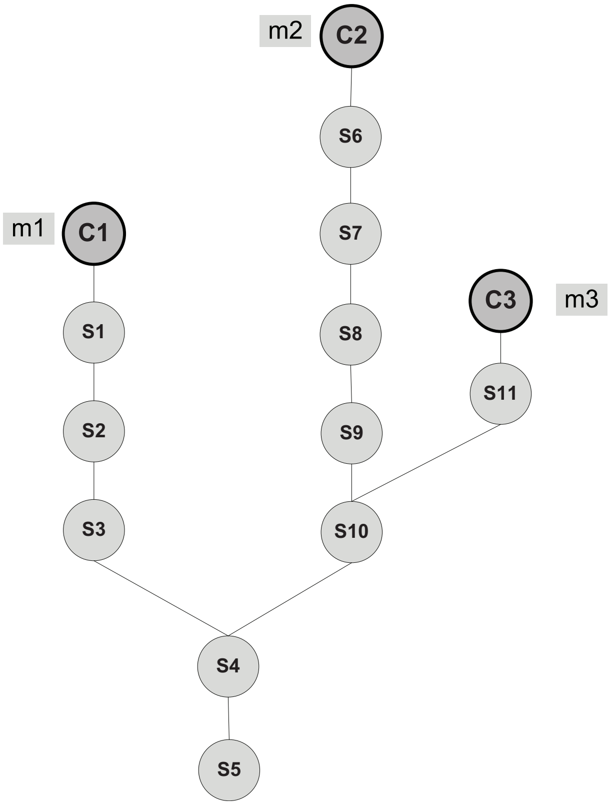


Figure 6.2: The view of the world of service node $S4$; it is only aware of its neighbors, $S3, S5, S10$. The core nodes $(C1, C2, C3)$ send *Type 1* messages, $(m1, m2, m3)$, respectively. The distances of $S4$ to the three leader nodes are: $d(S4, C3) = 3, d(S4, C1) = 4$, and $d(S4, C2) = 6$. node $S4$ will join the cluster of built around leader $C3$ at the minimum distance, $d = 3$, if and only if message $m3$ arrives before $S4$ starts processing the information in its $tempTab$.

(a) If no such entry exists:

i. Add to its $tempTab$ an entry consisting of: (i) Cluster Id, $clusterId = LNodeId$; (ii) the identity of neighbor delivering the message, $neigborId$; (iii) the $hopCnt$; and (iv) the path followed by the message, $msgPath$.

ii. Increment the $hopCnt$.

iii. Add itself to the $msgPath$.

iv. Resend the message to all neighbors, except the one which delivered the message.

(b) If such an entry exists compare $hopCnt_{entry}$ of the existing entry with the one in the message, $hopCnt_{msg}$.

– If $hopCnt_{msg} < hopCnt_{entry}$:

        i. Replace the entry in the table with one containing the information in the message.
        ii. Increment *hopCnt*.
        iii. Add itself to the *msgPath*.
        iv. Send the message to all neighbors except the one which delivered the message.
    - If $hopCnt_{msg} \geq hopCnt_{entry}$ drop the message.

4. When the timer τ_{end} expires a node processes its *tempTab*. If there are no entries then the node is isolated and cannot join any cluster. Note: nodes of degree one can proceed with the actions discussed next once they get a *Type 1* message; they do not need to wait for the timer to expire because their commitment cannot be changed by any other message as their distance to the *leader* is one. All other nodes should proceed as follows:

    (a) Identifies the leader at the minimum distance.

    (b) Retrieves from the entry:

        i. The identity of the leader, *LNodeId*;
        ii. The distance to the core node, *hopCnt*.
        iii. The path to the core node, *msgPath*.

    (c) Constructs a *Type 2* message including this information.

    (d) Sends the message to the core node.

    (e) Resets the two timers.

    (f) Stop.

5. Upon receiving a *Type 2* message originating from a regular node, a leader will:

    (a) Processes the message to identify:

        i. The sender *NodeId*;
        ii. The path to the node, *msgPath*; and
        iii. The distance to the service node, *hopCnt*.

    (b) Add a new entry to its *clusterTab*

6. Upon receiving a *Type 1* message originating from another leader, a leader:

    (a) Retrieves from the entry, the identity of the core node, $LNodeId$.

    (b) Adds to its *coreTab* a new entry.

Once the clusters are constructed, a node communicates only with the leader of the cluster. Leaders communicate with one another using an epidemic algorithm, each one forwards an incoming message to all its neighbors, except the one it has received the message from.

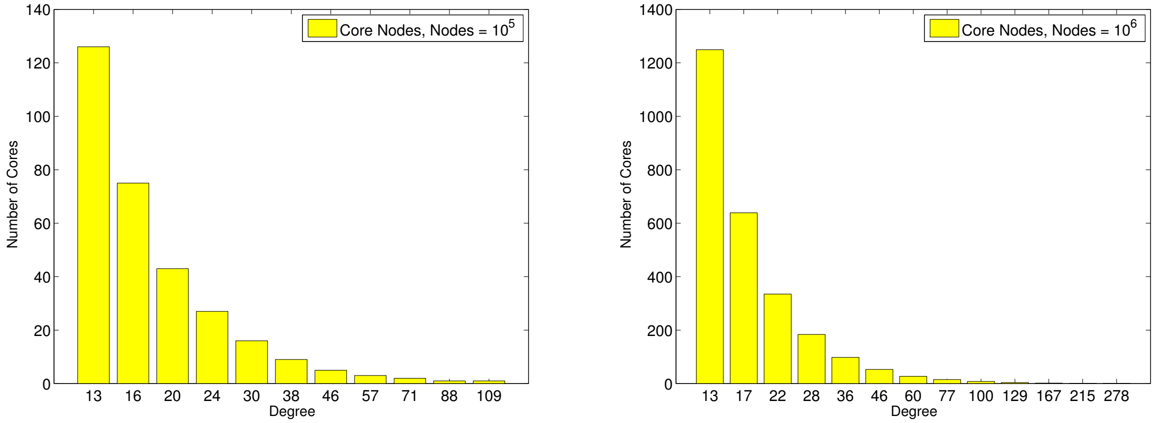
The algorithm requires a timer because individual nodes do not have global information. Indeed, each node has only local information, it is aware of its neighbors and of its own degree. A service node does not know if a *Type 1* message from a node at a smaller distance from a core node was delayed and it will come after it has already received *Type 1* messages from all its neighbors.

This situation is illustrated in Figure 6.2 where we assume that the message $m3$ from core node $C3$ is delayed. We see that after receiving *Type 1* messages $m1$ and $m2$ from both its neighbors, $S3$ and $S10$ then service node $S4$ would choose to join the cluster around core node $C1$ which is at distance 4 (core node $C2$ is at distance 5). On the other hand, if it waits for the message $m3$ then service node $S4$ makes the correct decision. Thus, the time $\tau_{decision}$ should be chosen to ensure that all healthy nodes could transmit the *Type 1* messages they receive from core nodes.

6.5
A Simulation Experiment

Table 6.3: Summary of the results for the creation of a scale-free network with the algorithm in Section 6.3 and for clustering using the algorithm in Section 6.4.

# of nodes (N)	Threshold (T)	Number of clusters (M)	Average size of cluster	SFN time (seconds)	Clustering time (seconds)	Total time (seconds)
10^5	10	316	315	2.94	3.31	6.25
	15	200	531	2.94	4.04	6.98
10^6	10	2,637	378	11.59	63.71	75.3
	15	1,367	684	11.59	69.35	80.94



(a)

(b)

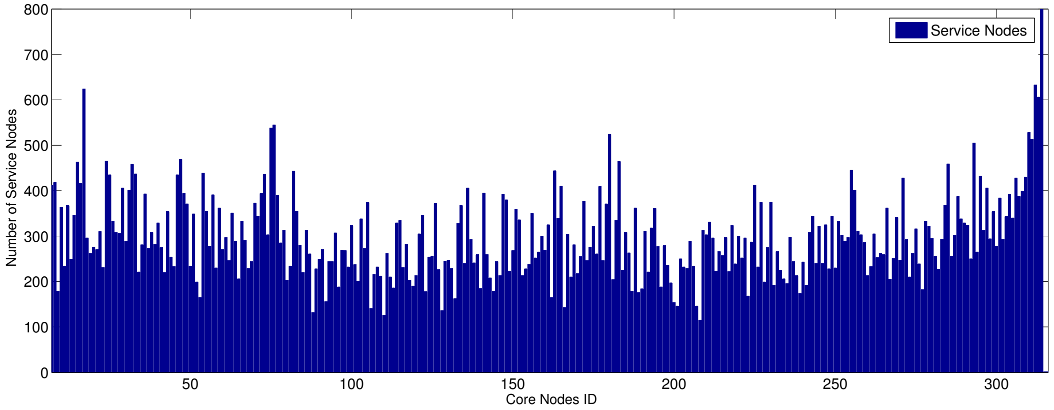
Figure 6.3: The algorithm to construct a scale-free network; distribution of core nodes based on their degrees with the degree threshold $T = 10$. (a) $N = 10^5$ nodes; (b) $N = 10^6$ nodes.

The algorithms for the construction of scale-free networks with $\gamma = 2.5$ and for clustering were implemented in C++. The number of nodes we experimented with are: $N = 10^5$, $N = 10^6$, and $N = 10^7$. The execution was carried out on the Amazon cloud using one *medium* EC2 instance, the lowest cost instance suitable for the problem we invetigate. The execution times on the cloud are comparable with the times when execution was done locally on a system with similar resources as the ones provided by the EC2 instance of AWS.

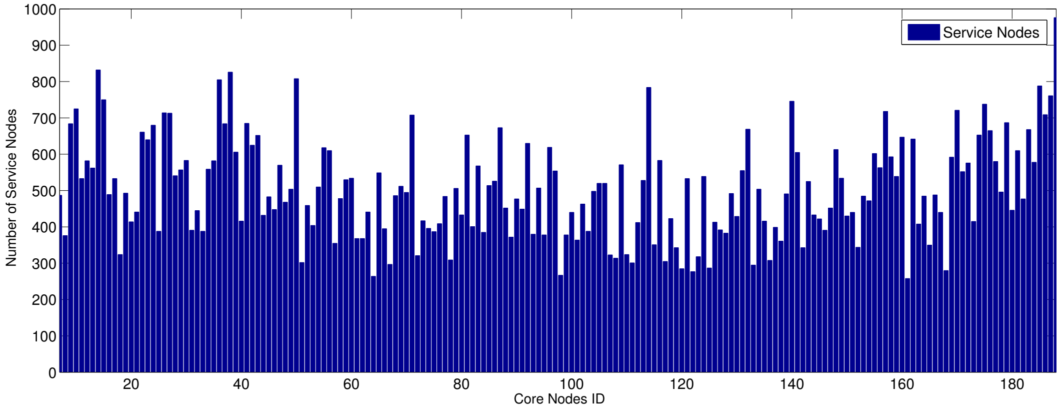
Table 6.4: Summary of the results for the creation of a scale-free network based on the algorithm discussed in Section 6.3. Graph density is defined as $D = \frac{2E}{N(N-1)}$

Nodes (N)	Edges (E)	Average degree	Distance (e)	Graph density (D)
10^5	149247	1.49	0.0205	2.98×10^{-5}
10^6	1500267	1.50	0.0137	3.00×10^{-6}
10^7	14369056	1.44	0.0278	2.87×10^{-7}

Figure 6.3 shows a histogram of the core nodes degree distribution when the number of nodes are $N = 10^5$ and $N = 10^6$ and the degree threshold is 10. These are typical parameters for the systems we investigate. The figure also shows the highest degree for each of the networks, which is 711 for $N = 10^7$. Table 6.4 summarizes the properties of the scale-free networks constructed with the algorithm in Section 6.3 for $N = 10^5$, $N = 10^6$, and $N = 10^7$.



(a)



(b)

Figure 6.4: The histogram of the cluster size for $N = 10^5$; (a) $T = 10$; (b) $T = 15$.

The average degree in all cases is slightly greater than 1, which means most of the nodes have degree 1. This is a characteristic of a scale-free network as discussed in Section 6.3. The distance e between the theoretical and the experimental degree distribution is $0.0205, 0.0137$ and 0.0278 for $N = 10^5, 10^6$ and 10^7 respectively.

Figures 6.4 (a) and (b) show the histogram of the the cluster size when the number of nodes is $N = 10^5$ for $T = 10$ and $T = 15$. When $N = 10^6$ due to the very large number of clusters the distribution of the cluster size $\mathcal{C}$ cannot be represented graphically. When $T = 10$ for $N = 10^5$, the average cluster size is $\mu_{\mathcal{C}} = 315$, the standard deviation is $\sigma_{\mathcal{C}} = 46.1$ and the variance $\text{Var}_{\mathcal{C}} = 2,125$. When $T = 15$, previous parameters will be 531, 156.98 and 2.464×10^4 respectively.

Table 6.3 summarizes our results. The time to construct the scale-free network increases logarithmically with the number of nodes. The average number of clusters increases by approximately one order of magnitude when the number of nodes increases by one order of magnitude, but the time for cluster construction increases much faster. Still, the total time for the construction of the scale-free network and clustering is of the order of minutes even for a network with several million nodes. We conclude that the clustering using the algorithm introduced in this chapter is entirely feasible even for very large social networks.

The average number of nodes in a cluster is sensitive to the threshold, it increases from 315 to 531 and from 378 to 684 when the threshold increases from 10 to 15 and the number of nodes is 10^5 and 10^6 respectively. This number is less sensitive to N, the number nodes.

6.6
Conclusions

The algorithms introduced in this chapter can be used to construct overlay networks for large-scale systems such a computer cloud, or a sensor network and, then cluster the nodes around the highly connected once which act as the nerve centers for self-organization and self-management. The scale-free organization seems to occur naturally in social networks and clustering can be used to facilitate the exchange of information among the individual members of the network.

Our results show that the algorithm for the construction of a scale-free overlay network achieves a relatively good approximation of the theoretical degree distribution, in a reasonably short time. The execution time of the algorithm increases as the logarithm of the number of nodes. On the other hand, clustering is more computationally intensive and the clustering time increases with the number of clusters and the number of nodes.

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

The main focus of this thesis was resource management in large-scale systems. Our primary concerns were energy management and practical principles for self-organization and self-management. Energy management concerns, models and experiments had been discussed in detail in Chapter 2. The model described in that chapter introduces the operating regimes of operation for processors and the conditions when to switch a server to one of the sleep states. Load balancing and scaling algorithms suitable for a clustered cloud organization based on the model are also presented in Chapter 2; these algorithms aim to optimize the energy efficiency and to balance the load.

In Chapter 3, we compared hierarchical organization based on system monitoring and control model with a simple bidding scheme in order to propose a model to minimize the cost and optimize the shared resources utilization when we connect large number of servers together. Then we conducted a series of simulation experiments to confirm our intuition and conclude that the effectiveness of hierarchical control decreases significantly as the system load increases.

In Chapter 4, we proposed a two-stage protocol for resource management in a hierarchically organized cloud in order to serve the requests that might not fit into a single server. The first stage exploits spatial locality for the formation of coalitions of supply agents; the second stage, a combinatorial auction, is based on a modified proxy-based clock algorithm and has two phases, a clock phase and a proxy phase. The results reported in this chapter indicated that the performance of the protocol is relatively stable for the range of parameters explored in our evaluation. The protocol leads to a higher server utilization and it seems reasonable to expect that a fine tuned version of the protocol could further improve this critical performance measure. However, there are still few unsolved problems such as the effects of overbidding, the process which allows a client to become a provisional winner of one or more service slots and then, in the final round failing to acquire some of them. This situation is critical for the first slot of an auction as the next auctions could find clients for these slots. A more difficult problem is the temporal fragmentation which does not seem to have an obvious solution.

In Chapter 5, we proposed a new cloud delivery model which has its own limitations, it cannot eliminate all the inefficiencies inherent to virtualization, requires the development of new families of algorithms for resource management and the development of new software. However, it had compelling advantages as we saw in Section 5.2. The simulation experiments we conducted showed that the initial system organization phase can be tuned to provide each core server with a balanced number of primary and secondary contacts. The bidding mechanisms seem to work well when the workload is relatively low. In spite of the larger overhead, core-initiated coalition formation seems more effective than periphery-initiated coalitions.

Finally, in Chapter 6, we propose a simple scheme for the organization of scale-free systems because scalability is a critical property of a large-scale systems. The algorithms and models proposed in this chapter can be used to construct overlay networks for large-scale systems such as a computer cloud, or a sensor network and, then cluster the nodes around the highly connected once which act as the nerve centers for self-organization and self-management. The algorithm for the construction of a scale-free overlay network achieves a relatively good approximation of the theoretical degree distribution, in a reasonably short time. The execution time of the algorithm increases as the logarithm of the number of nodes. However, clustering is more computationally intensive and the clustering time increases with the number of clusters and the number of nodes.

Our future work will cover practical implementation of the self-organization principles in larger scale. Also investigating the energy consumption of heterogenous systems both for hierarchical and traditional market model architectures as well as finding solutions to newly revealed problems discussed earlier.

REFERENCES

[1] B. Addis, D. Ardagna, B. Panicucci, M. Squillante, and Li Zhang. A hierarchical approach for the resource management of very large cloud platforms. *IEEE Transactions on Dependable and Secure Computing*, 10(5):253–272, 2013.

[2] Amazon Web Services, Inc. *Amazon Elastic Compute Cloud User Guide (API Version 2014-06-15)*, 2014.

[3] M. Andreolini, S. Casolari, and S. Tosi. A hierarchical architecture for on-line control of private cloud-based systems. In *Proc. of 10th World Wide Web Internet Conference (WWW-CONF2010)*, Timisoara, Romania, October 2010.

[4] R. Abbott. "Complex systems engineering: putting complex systems to work." *Complexity,* **13**(2):10–11, 2007.

[5] R. Albert, H. Jeong, and A.-L. Barabási. "The diameter of the world wide web." *Nature,* **401**:130–131, 1999.

[6] R. Albert, H. Jeong, and A.-L. Barabási. "Error and attack tolerance of complex networks." *Nature,* **406**:378–382, 2000.

[7] D. Ardagna, M. Trubian, and L. Zhang. "SLA based resource allocation policies in autonomic environments." *J. Parallel Distrib. Comp.*, **67**(3):259–270, 2007.

[8] D. Ardagna, B. Panicucci, M. Trubian, and L. Zhang. "Energy-aware autonomic resource allocation in multi-tier virtualized environments." *IEEE Trans. on Services Computing,* **5**(1):2–19, 2012.

[9] J. Y. Arrasjid et. al. *VMware vCloud Architecture Toolkit.* VMware Press, Upple Saddle River NJ, 2013.

[10] M. Auty, S. Creese, M. Goldsmith, and P. Hopkins. "Inadequacies of current risk controls for the cloud." *Proc. IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 659–666, 2010.

[11] M. Azambuja, R. Pereira, K. Breitman, and M. Endler. "An architecture for public and open submission systems in the cloud." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 513–517, 2010.

[12] L. Ausubel and P. R. Milgrom. "Ascending auctions with package bidding." *Frontiers of Theoretical Economics*, **1**(1):1–42, 2002.

[13] L. M. Ausubel and P. Cramton. "Auctioning many divisible goods." *Journal European Economic Assoc.*, **2**(2-3):480-493, 2004.

[14] L. Ausubel, P. Cramton, and P. Milgrom. "The clock-proxy auction: a practical combinatorial auction design." *Chapter 5,* in *Combinatorial Auctions*, P. Cramton, Y. Shoham, and R. Steinberg, Eds. MIT Press, 2006.

[15] X. Bai, D. C. Marinescu, L. Bölöni, H. J. Siegel, R. A. Daley, and I-J Wang. "A macroeconomic model for resource allocation in large-scale distributed systems." *Journal of Parallel and Distributed Computing (JPDC)*, **68**:182-199, 2008.

[16] I. Bradic. "Towards self-manageable cloud services." *Proc. 33 Int. Conf. on Computer Software and Applications,* pp. 128–133, 2009.

[17] J. Baliga, R.W.A. Ayre, K. Hinton, and R.S. Tucker. "Green cloud computing: balancing energy in processing, storage, and transport." *Proc. IEEE*, **99**(1):149-167, 2011.

[18] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. "A security analysis of Amazon's elastic compute cloud service." *Proc. 27th Annual ACM Symp. on Applied Computing*, pp. 1427–1434, 2012.

[19] A. Barak and A. Shiloh. "The Virtual OpenCL (VCL) cluster platform." *Proc. Intel European Re & Innovation Conference*, Leixlip, Ireland, pp. 196–200, 2011.

[20] R. Buyya. "Single system image: need, approaches, and supporting HPC systems." In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97).* Las Vegas, Nevada, USA, pp. 1106–1113, 1997.

[21] A-L. Barabási and R. Albert. "Emergence of scaling in random networks," *Science*, **286**(5439):509–512, 1999.

[22] A-L. Barabási, R. Albert, and H. Jeong. "Scale-free theory of random networks; the topology of World Wide Web." *Physica A*, **281**:69–77, 2000.

[23] A. Barak and A. Shiloh. *The MOSIX Cluster Operating System for High-Performance Computing on Linux Clusters, Multi-Clusters and Clouds*, 2012.

[24] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the art of virtualization." *Proc. SOSP'03, 19th ACM Symp. on Operating Systems Principles*, pp. 164–177, 2013.

[25] L. A. Barossso, J. Clidaras, and U.Hözle. *The Datacenter as a Computer; an Introduction to the Design of Warehouse-Scale Machines.* (Second Edition). Morgan & Claypool, 2013.

[26] D. Bernstein, D. Vij, and S. Diamond. "An Intercloud cloud computing economy - technology, governance, and market blueprints." *Proc. SRII Global Conference,* pp. 293–299, 2011.

[27] I. Brandic, S. Dustdar, T. Ansett, D. Schumm, F. Leymann, and R. Konrad. "Compliant cloud computing (C3): Architecture and language support for user-driven compliance management in clouds." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 244–251, 2010.

[28] R. Buyya, T. Cortes, and H. Jin. "Single system image." *International Journal of High Performance Computing Applications 15,* 2, 124–135, 2001.

[29] L. A. Barroso and U. Hözle. “The case for energy-proportional computing.” *IEEE Computer,* **40**(12):33–37, 2007.

[30] M. Blackburn and A. Hawkins. “Unused server survey results analysis.” *www.thegreengrid.org/media/WhitePapers/Unused%20Server%20Study_WP_101910_v1.ashx?lang= en*, accessed on December 6, 2013.

[31] A. Beloglazov, R. Buyya “Energy efficient resource management in virtualized cloud data centers.” *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Comp.*, 2010.

[32] A. Beloglazov, J. Abawajy, R. Buyya. “Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing.” *Future Generation Computer Systems*, **28**(5):755-768, 2012.

[33] A. Beloglazov and R. Buyya. “Managing overloaded hosts for dynamic consolidation on virtual machines in cloud centers under quality of service constraints.” *IEEE Trans. on Parallel and Distributed Systems*, **24**(7):1366-1379, 2013.

[34] D. Bruneo. “A stochastic model to investigate data center performance and QoS in IAAS cloud computing systems.” *IEEE Trans. on Parallel and Distributed Systems,* **25**(3):560–569, 2014.

[35] Cloud Security Alliance. “Security guidance for critical areas of focus in cloud computing V3.0.” *https://cloudsecurityalliance.org/guidance*, 2011.

[36] R. Cohen and S. Havlin. “Scale-free networks are ultrasmall.” *Phys. Rev. Lett.,* **90**(5):058701, 2003.

[37] A. Celesti, F. Tusa, M. Villari, and A.Puliafito. “How to enhance cloud architecture to enable cross-federation.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 337–345, 2010.

[38] A. Chandra, P. Goyal, and P. Shenoy. “Quantifying the benefits of resource multiplexing in on-demand data centers.” *Proc. 1-st Workshop on Algorithms and Architecture for Self-Managing Systems,* 2003.

[39] T. E. Carroll and D. Grosu. “Formation of virtual organizations in grids: a game-theoretic approach.” *Concurrency and Computation: Practice and Experience,* **22**(14):1972–1989, 2010.

[40] S. Chaisiri, B. Lee, and D. Niyato. “Optimization of resource provisioning cost in cloud computing.” *IEEE Trans. on Services Computing,* **5**(2):164–177, 2012.

[41] V. Chang, G. Wills, and D. De Roure. “A review of cloud business models and sustainability.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 43–50, 2010.

[42] M. Chapman and G. Heiser. “Implementing transparent shared memory on clusters using virtual machines.” In *Proc. USENIX Annual Technical Conference*, pp. 23–23, 2005.

[43] H. Cofer, M. Fouquet-Lapar, T. Gamerdinger, C. Lindahl, B. Losure, A, Mayer, J. Swoboda, and T. Utsumi. “Creating the world’s largest reconfigurable supercomputing system based on the scalable SGI® Altix® 4700 system infrastructure and benchmarking life-science applications.” *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 268–273, 2008.

[44] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. "Breaking up is hard to do: security and functionality in a commodity hypervisor." *Proc. Symp. Operating Systems Principles,* pp. 189–202, 2011.

[45] E. H. Clarke. "Multipart Pricing of Public Goods." *Public Choice*, **IX**:13–33, 1971.

[46] P. Cramton, Y Shoham, R. Steinberg Eds. *Combinatorial Auctions,* MIT Press, 2006.

[47] X. Dutreild, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. "From data center resource allocation to control theory and back." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 410–417, 2010.

[48] P. Erdös and A. Rényi. "On random graphs." *Publicationes Mathematicae*, **6**:290–297, 1959.

[49] A. Gandhi, and M. Harchol-Balter. "How data center size impacts the effectiveness of dynamic power management." *Proc. 49th Annual Allerton Conference on Communication, Control, and Computing, Urbana-Champaign*, pp. 1864–1869, 2011.

[50] M. Gell-Mann. "Simplicity and complexity in the description of nature." *Engineering and Science*, Caltech, **LI**(3):3–9, 1988.

[51] A. G. Ganek and T. A. Corbi. "The dawning of the autonomic computing era." *IBM Systems Journal*, **42**(1):5-18, 2003.

[52] J. O. Gutierrez-Garcia and K.- M. Sim. "Self-organizing agents for service composition in cloud computing." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 59–66, 2010.

[53] K. I. Goh, B. Kahang, and D. Kim. "Universal behavior of load distribution in scale-free networks." *Physical Review Letters,* **87**:278701, 2001.

[54] Google. "Google's green computing: efficiency at scale." *http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/green/pdfs/google-green-computing.pdf* accessed on August 29, 2013.

[55] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M.Kozuch. "AutoScale: dynamic, robust capacity management for multi-tier data centers." *ACM Trans. on Computer Systems*, **30**(4):1–26, 2012.

[56] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M.Kozuch. "Are sleep states effective in data centers?" *Proc. Int. Conf. on Green Comp.*, pp. 1–10, 2012.

[57] S. Garfinkel and M. Rosenblum. "When virtual is harder than real: security challenges in virtual machines based computing environments." *Proc. Conf. Hot Topics in Operating Systems,* pp. 20–25, 2005.

[58] D. Gmach, J. Rolia, and L. Cerkasova. "Satisfying service-level objectives in a self-managed resource pool." *Proc. 3rd. Int. Conf. on Self-Adaptive and Self-Organizing Systems,* pp. 243–253, 2009.

[59] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Tucricchi, and A. Kemper. "An integrated approach to resource pool management: policies, efficiency, and quality metrics." *Proc. Int. Conf. on Dependable Systems and Networks,* pp. 326–335, 2008.

[60] N. Gruschka and M. Jensen. "Attack surfaces: A taxonomy for attacks on cloud services." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 276–279, 2010.

[61] V. Gupta and M. Harchol-Balter. "Self-adaptive admission control policies for resource-sharing systems." *Proc. 11th Int. Joint Conf. Measurement and Modeling Computer Systems (SIGMETRICS'09)*, pp. 311–322, 2009.

[62] T. Groves. "Incentives in teams." *Econometrica,* **41**:617–631, 1973.

[63] K. Hasebe, T. Niwa, A. Sugiki, and K. Kato. "Power-saving in large-scale storage systems with data migration." *Proc IEEE 2nd Int. Conf. on Cloud Comp. Technology and Science,* pp. 266–273, 2010.

[64] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. "ElasticTree: saving energy in data center networks." *Proc. 7th USENIX Conf. on Networked Systems Design and Implementation*, pp. 17–17, 2011.

[65] J. Hopfield. "Neural networks and physical systems with emergent collective computational abilities." *Proc. National Academy of Science,* 79, pp. 2554-2558, 1982.

[66] P.Healy, S.Meyer, J.Morrison, T.Lynn, A.Paya, D.Marinescu "Bid-Centric Cloud Service Provisioning" *IEEE 13th Int. Symposium on Parallel and Distributed Computing*, pp. 73-81, 2014

[67] M. Hinchey, R. Sterritt, C. Rouff, J. Rash, W. Truszkowski. "Swarm-based space exploration." *ERCIM News* 64, 2006.

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture; A Quantitative Approach, 5th Edition.* Morgan Kaufmann, a division of Elsevier, Amsterdam, New York, 2012.

[69] Hewlet-Packard/Intel/Microsoft/Phoenix/Toshiba. "Advanced configuration and power interface specifications, revision 5.0" *http://www.acpi.info/ DOWNLOADS/ACPIs pec50.pdf*, accessed on November 10, 2013.

[70] J. Hilland, P. Culley, J. Pinkerton and R. Recio, "RDMA Protocol Verbs Specification." *RDMAC Consortium Draft Specification*, 2003

[71] L. He and T. R. Ioerger. "Forming resource-sharing coalitions: a distributed resource allocation mechanism for self-interested agents in computational grids." *Proc. ACM Symp. on Applied Computing,* pp. 84–91, 2005.

[72] Intel. *Intel® Xeon Phi^{TM} Coprocessor System Software Developers Guide*, 2013

[73] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, N. J. Wright. "Performance analysis of high performance computing applications on the Amazon Web services cloud." *Proc. IEEE Second Int. Conf. on Cloud Computing Technology and Science,* pp. 159–168, 2010.

[74] A. K. Jain and R.C. Dubes. *Algorithms for Clustering data.* Prentice Hall, 1988

[75] J. O . Kephart and D. M. Chase. "The vision of autonomic computing." *Computer,* **36**(1):41-50, 2003

[76] K. Kaneda, Y. Oyama, and A. Yonezawa. "A virtual machine monitor for providing a single system image." *Proc. 17th IPSJ Computer System Symposium (ComSys 05).* pp. 3–12, 2005.

[77] E. Kalyvianaki, T. Charalambous, and S. Hand. "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters." *Proc. 6th Int. Conf. Autonomic Comp. (ICAC2009),* pp. 117–126, 2009.

[78] J. Kephart, H. Chan, R. Das, D. Levine, G. Tesauro, F. Rawson, and C. Lefurgy. "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs." *Proc. 4th Int. Conf. Autonomic Computing (ICAC2007),* pp. 100-109, 2007.

[79] P.R. Krugman. "The Self-organizing Economy." *Blackwell Publishers,* 1996.

[80] D. Kusic, J. O. Kephart, N. Kandasamy, and G. Jiang. "Power and performance management of virtualized computing environments via lookahead control." *Proc. 5th Int. Conf. Autonomic Comp. (ICAC2008)*, pp. 3–12, 2008.

[81] J. G. Koomey. "Estimating total power consumption by servers in the US and world." *http://hightech.lbl. gov/documents/data_centerssvrpwrusecompletefinal.pdf*, accessed on May 11, 2013.

[82] J.G. Koomey, S. Berard, M. Sanchez, and H. Wong. "Implications of historical trends in the energy efficiency of computing." *IEEE Annals of Comp.*, **33**(3):46–54, 2011.

[83] S. U. Khan and I. Ahmad. "A cooperative game theoretical technique for joint optimization of energy consumption and response time in computational grids." *IEEE Trans. on Parallel and Distributed Systems,* **20**(3): 346-360, 2009.

[84] J. Kephart. "The utility of utility." *Proc. Policy 2011*, 2011.

[85] M. Litoiu, M. Woodside, J. Wong, J. Ng, and G. Iszlai. "Business driven cloud optimization architecture." *Proc. 2010 ACM Symposium on Applied Computing.* pp. 380–385.

[86] A. Livnat, C. Papadimitriou, J. Dushoff, and M. W. Feldman. "A mixability theory of the role of sex in evolution." *Proc. National Academy of Science,* **105**(50):19803-19807, 2008.

[87] Y.-Y. Lu, J.-J. Slotino, and A. L. Barabási. "Controllability of complex networks." *Nature* 473, 167, 2011.

[88] M. Lin, Z. Liu, A. Wierman, and L. L. H. Andrew "Online algorithms for geographical load balancing." *Proc. Int. Conf. on Green Computing*, pp. 1–10, 2012.

[89] H. Liu. Amazon EC2 grows 62% in 2 years. Blog post, February 2014.

[90] H. Li, C. Wu, Z. Li, and F. Lau. "Profit-maximizing virtual machine trading in a federation of selfish clouds." *Proc. of the IEEE INFOCOM,* pp. 25–29, 2013.

[91] H C. Lim, S. Babu, J. S. Chase, and S. S. Parekh. "Automated control in cloud computing: challenges and opportunities." *Proc. First Workshop on Automated Control for Datacenters and Clouds,,* ACM Press, pp. 13–18, 2009.

[92] C. Li, A. Raghunathan, and N. K.Jha. "Secure virtual machine execution under an untrusted management OS." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 172–179, 2010.

[93] B. Li; J. Li; J. Huai; T. Wo; Q. Li; L. Zhong. “EnaCloud: an energy-saving application live placement approach for cloud computing environments.” *IEEE Int. Conf. on Cloud Computing,* pp.17 - 24, 2009.

[94] L. Lamport. “The part-time parliament.” *ACM Trans. on Computer Systems*, **2**:133–169, 1998.

[95] D. S. Lee, K. I. Goh, B. Kahng, and D. Kim. “Evolution of scale-free random graphs: Potts model formulation.” *Nuclear Physics B.* 696:351–380, 2004.

[96] O. Litvinski and A. Gherbi. Experimental evaluation of open-stack compute scheduler. *Procedia Computer Science*, 19:116–123, 2013.

[97] L. Lamport. “Paxos made simple.” *ACM SIGACT News*, **32**(4):51–58, 2001.

[98] K. Lerman and O. Shehory. “Coalition formation for large-scale electronic markets.” *Proc. ICMAS 2000 - 4th Int. Conf on Multiagent Systems,* pp. 167–174, 2000.

[99] C. Li and K. Sycara. “Algorithm for combinatorial coalition formation and payoff division in an electronic marketplace.” *Proc. AAMAS02 - First Joint Int. Conf. on Autonomous Agents and Multi-agent Systems,* pp. 120–127, 2002.

[100] “Metal as a service.” *https://maas.ubuntu.com/*, accessed on October 19, 2013.

[101] M. W. Mayer. “Architecting principles for system of systems.” *Systems Engineering,* **1**(4):27Th67–274, 1998.

[102] M. Minsky. “Computation: Finite and Infinite Machines.” *Prentice Hall*, New York, 1967.

[103] C. Mastroianni, M. Meo, G. Papuzzo. “ Probabilistic consolidation of virtual machines in self-organizing cloud data centers.” *IEEE Trans. on Cloud Computing,* **1**(2):215–228, 2013.

[104] H. Moens and F. De Turck. A scalable approach for structuring large-scale hierarchical cloud management systems. In *9th International Conference on Network and Service Management (CNSM-2013)*, pp. 1–8, 2013.

[105] D. Margery, G. Vallée, R. Lottiaux, C. Morin, and J.-Y. Berthou. “Kerrighed: A SSI cluster OS running OpenMP.” Research Report RR-4947, INRIA, 2003.

[106] D. C. Marinescu, J. P. Morisson , and H. J. Siegel. “Options and Commodity Markets for Computing Resources.” *Market Oriented Grid and Utility Computing,* R. Buyya and K. Bubendorf, Eds., Wiley, ISBN: 9780470287682, pp. 89–120, 2009.

[107] D. C. Marinescu, C. Yu, and G. M. Marinescu. “Scale-free, self-organizing very large sensor networks.” Journal of Parallel and Distributed Computing (JPDC), **50**(5):612–622, 2010.

[108] D. Marinescu, A. Paya, J. Morrison, P. Healy “An Auction-driven Self-Organizing Cloud Delivery Model”, *http://arxiv.org/pdf/1312.2998v1.pdf* , December 2013.

[109] D. C. Marinescu. “High probability trajectories in the phase space and system complexity.” *Complex Systems,* **22**(3):233–246, 2013.

[110] D. C. Marinescu. *Cloud Computing; Theory and Practice.* Morgan Kaufmann, a division of Elsevier, Amsterdam, New York, 2013.

[111] D. C. Marinescu, A. Paya, and J. P. Morrison. “Coalition formation and combinatorial auctions; applications to self-organization and self-management in Utility Computing.” *http://arXiv.org/pdf/1406.7487.pdf*, 2014

[112] D. C. Marinescu, A. Paya, J. P. Morrison, and P. Healy. “Distributed hierarchical control versus an economic model for cloud resource management.” *http://arxiv.org/pdf/1503.01061v2.pdf*, March 2015.

[113] L.Mashayekhy, M.M.Nejad, and D.Grosu. “Cloud federations in the sky: formation game and mechanisms.” *IEEE Trans. on Cloud Computing,* 2015 (to appear).

[114] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. “Diagnosing performance overheads in Xen virtual machine environments.” *Proc. First ACM/USENIX Conf. on Virtual Execution Environments,* 2005.

[115] C. Morin, P. Gallard, R. Lottiaux and G. Vallée. ”Towards an efficient single system image cluster operating system.” *Future Generation Computer Systems 20,* 4, pp.505–521, 2004.

[116] I. Müller, R. Kowalczyk, and P. Braun. “Towards agent-based coalition formation for service composition.” *Proc. IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology,* pp. 73-80, 2006.

[117] P-A. Noël, C. D. Brummitt, and R. M. D´ Souza. “Controlling self-organizing dynamics on networks using models that self-organize.” *Phys. Rev. Lett.* **111**, 078701, 2013.

[118] NRDC and WSP 2012. “The carbon emissions of server computing for small-to medium-sized organization - a performance study of on-premise vs. the cloud.” *http://www.wspenvironmental.co m/media/docs/ ourlocations/usa/NRDC-WSP_Cloud_Computing.pdf* October 2012, accessed on November 10, 2013.

[119] D.Niyato, A.Vasilakos, and Z.Kun. “Resource and revenue sharing with coalition formation of cloud providers: Game theoretic approach.” *Proc. IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Comp.,* pp. 215–224, 2011.

[120] J. von Neumann. “Probabilistic logic and synthesis of reliable organisms from unreliable components.” In *Automata Studies*, C. E. Shannon and J. McCarthy, Editors. Princeton University Press, Princeton, NJ, 1956.

[121] M. E. J. Newman. “The structure of scientific collaboration networks.” *Proc. Nat. Academy of Science,* **98**(2):404–409, 2001.

[122] M. E. J. Newman. “Properties of highly clustered networks.” *Phys. Rev. E,* **68**:026121, 2003.

[123] P. Osiński and E. Niewiadomska-Szynkiewicz. “Comparative study of single system image clusters.” *Evolutionary Computation and Global Optimization*, pp. 145–154, 2009.

[124] N. Paton, M. A. T. de Arago, K. Lee, A.A.A. Fernandes, and R. Sakellariou, R. “Optimizing utility in cloud computing through autonomic workload execution. *Bulletin of the Technical Committee on Data Engineering,* **32**(1):51-58, 2009.

[125] S. Patterson. Openstack vs. closed clouds the AOL factor. White paper, 2011.

[126] A. Paya and D. C. Marinescu. "Clustering Algorithms for Scale-free Networks and Applications to Cloud Resource Management." *http://arxiv.org/pdf/1305.3031v1.pdf*, 2013.

[127] A. Paya and D. C. Marinescu. "Energy-aware application scaling on a cloud." *http://arxiv.org/pdf/1307.3306v1.pdf,* July 2013.

[128] A. Paya and D. C. Marinescu. "Evaluation of a Clustering Algorithm for Scale-free Organization of Large-scale Systems and Social Networks" *TBD*

[129] A.Paya, D.Marinescu "Energy-aware Load Balancing Policies for the Cloud Ecosystem" *IEEE 28th Int. Parallel & Distributed Processing Symposium Workshops*, pp.823-832, 2014.

[130] A.Paya, D.Marinescu "Energy-aware load balancing and application scaling for the cloud ecosystem." *IEEE Trans. on Cloud Computing.* doi: 10.1109/TCC.2015.2396059, 2015.

[131] S. Pearson and A. Benameur. "Privacy, secuitry, and trust issues arising from cloud computing." *Proc. Cloud Computing and Science,* pp. 693–702, 2010.

[132] J. Peng, X. Long, and L. Xiao. "DVMM: A distributed VMM for supporting single system image on clusters." *Proc. 9th International Conference forYoung Computer Scientists*, pp. 183–188, 2008.

[133] G. Pfister. "Multi-multicore Single System Image/Cloud Computing. A good idea?" *The Perils of Parallel (Blog).* 2009. Online; accessed September 26 2012.

[134] M. Price. "The paradox of security in virtual environments." *Computer,* **41**(11):22–28, 2008.

[135] C. Preist and P. Shabajee. "Energy use in the media cloud." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 581–586, 2010.

[136] S. Penmatsa and A. T. Chronopoulos. "Price-based user-optimal job allocation scheme for grid systems." *Proc of Parallel and Distributed Processing Symposium,* pp. 8-16, April 2006.

[137] M. Rosenblum and T. Garfinkel. "Virtual machine monitors: Current technology and future trends." *Computer,* **38**(5):39–47, 2005.

[138] Rackspace. Rackspace announces strong third quarter. Press Release, November 2014.

[139] T. Rahwan, S. D. Ramchurn, N. R. Jennings, and A. Giovannucci. "An anytime algorithm for optimal coalition structure generation." *Journal of Artificial Intelligence Research,* **34**:521–567, 2009.

[140] V. Sarathy, P. Narayan, and R. Mikkilineni. "Next generation cloud computing architecture." *Proc. IEEE Int. Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*, pp. 48–53, 2010.

[141] P. Schuster. "Nonlinear dynamics from Physics to Biology. Self-organization: An old paradigm revisited." *Complexity,* **12**(4):9-11, 2007.

[142] I. Sommerville, D. Cliff, R. Calinescu, J, Keen, T. Kelly, M. Kwiatowska, J. McDermid, and R. Paige. "Large-scale IT complex systems." *Communications of the ACM*, **55**(7):71–77, 2012.

[143] J. Sugerman, G. Venkitachalam, and B. Lim. "Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor." *Proc. USENIX Conf.*, pp. 70–85, 2001.

[144] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. “Using a market economy to provision compute resources across planet-wide clusters.” *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS 2009)*, pp. 1–8, 2009.

[145] N. Samaan. “A novel economic sharing model in a federation of selfish cloud providers.” *IEEE Trans. on Parallel and Distributed Systems,* **25**(1):12–21, 2014.

[146] S. Sen and P. S. Dutta. “Searching for optimal coalition structures.” *Proc. ICMAS 2000 - 4th Int. Conf on Multiagent Systems,* pp. 287–295, 2000.

[147] M. Sims, C. V. Goldman, and V. Lesser. “Self-organization through bottom-up coalition formation.” *Proc. Int. Conf. on Autonomous Agents and Multi Agent Systems,* pp. 867–874, 2003.

[148] B. Snyder. “Server virtualization has stalled, despite the hype.” *http://www.infoworld .com/print/146901*, accessed on December 6, 2013.

[149] R. Subrata, A. Y. Zomaya, and B. Landfeldt. “Game-theoretic approach for load balancing in computational grids.” *EEE Trans. on Parallel and Distributed Systems,* **19**(1):66–76, 2008.

[150] S de Vries and R. Vohra. “Combinatorial auctions; a survey.” *INFORMS Journal of Computing*, **15**(3):284–309, 2003.

[151] SPEC ”SPECpower_ssj2008 benchmark.” *https://www. spec.org/power_ssj2008/*, accessed on May 15, 2014.

[152] I. Scholtes. “ Distributed creation and adaptation of random scale-free overlay networks.” *Proc. 4th IEEE Int. Conf. of Self-Adaptive and Self-Organizing Systems, SASO-10*, pp. 51–63, 2010.

[153] T. W. Sandholm, K. S. Larson, M. Andersson, O. Shehory, and F. Tohm. “Coalition structure generation with worst case guarantees.” *Artificial Intelligence,* **111**(1-2):209–238, 1999.

[154] C. Tung, M. Steinder, M. Spreitzer, and G. Pacifici. “A scalable application placement controller for enterprise data centers.” *Proc. 16th Int. Conf. World Wide Web,* 2007.

[155] A.M. Turing. “The chemical basis of morphogenesis.” *Philosophical Transactions of the Royal Society of London,* Series B **237**:37–72, 1952.

[156] Z. Toroczkai and K. E. Bassler. “Jamming is limited in scale-free systems.” *Nature,* **428**:716, 2004.

[157] L. Tang, J. Dong, Y. Zhao, and L.-J. Zhang. “Enterprise cloud service architecture.” *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 27–34, 2010.

[158] B. Urgaonkar and C. Chandra. “Dynamic provisioning of multi-tier Internet applications.” *Proc. 2nd Int. Conf, on Automatic Computing*, pp. 217–228, 2005.

[159] H. N. Van, F. D. Tran, and J. M. Menaud. “Autonomic virtual resource management for service hosting platforms.” *Software Engineering Challenges of Cloud Computing, ICSE Workshop at CLOUD09.*, pp. 1–8, 2009.

[160] H. N. Van, F. D. Tran, and J.-M. Menaud. "Performance and power management for cloud infrastructures." *Proc. IEEE 3rd Int. Conf. on Cloud Computing*, pp. 329–336, 2010.

[161] A. Verma, G. Dasgupta, T. K. Nayak, P. De, and R. Kothari. "Server workload analysis for power minimization using consolidation." *Proc. USENIX'09 Conf.*, pp.28–28, 2009.

[162] S. V. Vrbsky, M. Lei, K. Smith, and J. Byrd. "Data replication and power consumption in data grids." *Proc IEEE 2nd Int. Conf. on Cloud Computing Technology and Science,* pp. 288–295, 2010.

[163] B. Walker and D. Steel. "Implementing a full single system image UnixWare cluster: Middleware vs. Underware." *Proc. Intl Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 1999.

[164] B. Walker. *Open single system image (openSSI) Linux cluster project.* Technical report, 2003.

[165] G. Wei, A. Vasilakos, Y. Zheng, and N. Xiong. "A game-theoretic method of fair resource allocation for cloud computing services." *The Journal of Supercomputing,* **54**(2):252–269, 2010.

[166] ZDNet. "Infrastructucture is not a differentiator." *http://www.zdnet.com/amazon-cto-werner-vogels-infrastructure-is-not-a-differentiator-7000014213*, accessed on August 29, 2013.

[167] H-J. Zhang, Q-H. Li, and Y-L. Ruan. "Resource co-allocation via agent-based coalition formation in computational grids." *Proc Second Int. Conf. on Machine Learning and Cybernetics,*, pp. 1936–1940, 2003.