5-2013

Elastic Resource Management in Cloud Computing Platforms

Upendra Sharma
University of Massachusetts Amherst, upendra.sharma@us.ibm.com

ELASTIC RESOURCE MANAGEMENT IN CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

UPENDRA SHARMA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2013

Computer Science

ELASTIC RESOURCE MANAGEMENT IN CLOUD COMPUTING PLATFORMS

A Dissertation Presented

by

UPENDRA SHARMA

Approved as to style and content by:

Prashant Shenoy, Chair

Don Towsley, Member

Arun Venkatramani, Member

Michael Zink, Member

Sambit Sahu, Member

Lori A. Clarke, Department Chair
Computer Science

Dedicated to my Gurus ...

ACKNOWLEDGMENTS

I would like to take this opportunity to express my heart felt thanks to my respected PhD advisor Prof. Prashant Shenoy without whose patience and guidance the doctorate would have remained a dream. I have learnt a great deal from him, starting from conducting systems research to things like time management, patience and tolerance.

Next, I am deeply indebted to Dr. Sambit Sahu who has been my mentor for the last four years. I have closely worked with him since the beginning of my graduate career. His guidance and invaluable comments have led me to finish my PhD work in a timely fashion. I am very grateful to Prof. Don Towsley from whom I learnt a great lot, specifically about queueing theory and in general about the method of conducting research. I am grateful to my other committee members, Prof. Arun Venkatramani, and Prof. Michael Zink, for their valuable inputs during the course of PhD. I also want to thank my collaborator Dr. Anees Shaikh for all his help, guidance and motivation during the early part of my PhD and also during my internships at IBM Watson.

I would like to thank my colleagues Rahul Singh, Tim Woods, Emmanuel Cecchet and Tian Guo with whom I have worked on several research projects. Rahul and I have developed a system that performs dynamic capacity planning and provisioning taking into account the non-stationarity in the workload mix. Emmanuel, Rahul and I have designed and developed Dolly, a virtualization driven database replication system. Tian, Tim and I developed Seagull, a system for cloud bursting applications from a private cloud to public and back.

I am very grateful to my buddy Rahul Singh for all the help he gave me during the course of PhD as well as for all the stimulating and thought provoking discussions either in

the cubicle or over the tea sessions. I am also grateful to other members of the LASS group, particularly to Navin Sharma, Himanshu Agarwal and Akshat Kumar with whom I have spent countless hours discussing a range of topics from philosophy to technical research. I also want to thank Jeremy Gummeson, Aditya Mishra and Gaul Niv with whom I have had numerous philosophical and technical discussions over tea and coffee. I am indebted to Tyler Trafford for his assistance in managing the machines used in my work, to Leeanne Leclerc for all the help in keeping my department and graduate school requirements on track, and to Karren Sacco for handling all administrative work; with their help all the non-research problems never looked like problems. My time in Amherst was made enjoyable by the incredible group of friends I developed over the years. Specifically, I am grateful to Siddharth Srivastav, Parthasarthi Valluri, Kshitij Neroorkar and Lokesh and also their families for the fun filled sessions of racquet ball, squash, badminton and various other social gatherings.

I am deeply obliged to my parents, my sister Shruti and brother in law Saurabh for patiently motivating me to pursue PhD and all the other assistance since the beginning; I also thank my adorable younger brother Subodh for always being there when needed and also for being a patient listener. I want to thank my old friends from India who motivated me to pursue PhD, my fellows from IBM India Research Lab for all the good time I had with them during their visits to US and during my home trips.

Last but not the least, I want to express my heartfelt thanks to my dear wife Ruchita, who has helped, supported and motivated me throughout the course of PhD. This journey would have been impossible without her patience and sacrifice.

ABSTRACT

ELASTIC RESOURCE MANAGEMENT IN CLOUD COMPUTING PLATFORMS

MAY 2013

UPENDRA SHARMA

B.S., BOMBAY UNIVERSITY, MUMBAI, INDIA

M.S., INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant Shenoy

Large scale enterprise applications are known to observe dynamic workload; provisioning correct capacity for these applications remains an important and challenging problem. Predicting high variability fluctuations in workload or the peak workload is difficult; erroneous predictions often lead to under-utilized systems or in some situations cause temporarily outage of an otherwise well provisioned web-site. Consequently, rather than provisioning server capacity to handle infrequent peak workloads, an alternate approach of dynamically provisioning capacity on-the-fly in response to workload fluctuations has become popular.

Cloud platforms are particularly suited for such applications due to their ability to provision capacity when needed and charge for usage on pay-per-use basis. Cloud environments enable elastic provisioning by providing a variety of hardware configurations as well as mechanisms to add or remove server capacity.

The first part of this thesis presents Kingfisher, a *cost-aware* system that provides a generalized provisioning framework for supporting elasticity in the cloud by (i) leveraging multiple mechanisms to reduce the time to transition to new configurations, and (ii) optimizing the selection of a virtual server configuration that minimize cost.

Majority of these enterprise applications, deployed as web applications, are distributed or replicated with a multi-tier architecture. SLAs for such applications are often expressed as a high percentile of a performance metric, for e.g. 99 percentile of end to end response time is less than 1 sec. In the second part of this thesis I present a model driven technique which provisions a multi-tier application for such an SLA and is targeted for cloud platforms.

Enterprises critically depend on these applications and often own large IT infrastructure to support the regular operation of these applications. However, provisioning for a peak load or for high percentile of response time could be prohibitively expensive. Thus there is a need of hybrid cloud model, where the enterprise uses its own private resources for the majority of its computing, but then "bursts" into the cloud when local resources are insufficient. I discuss a new system, namely Seagull, which performs dynamic provisioning over a hybrid cloud model by enabling cloud bursting.

Finally, I describe a methodology to model the configuration patterns (i.e deployment topologies) of different control plane services of a cloud management system itself. I present a generic methodology, based on empirical profiling, which provides initial deployment configuration of a control plane service and also a mechanism which iteratively adjusts the configuration to avoid violation of control plane's *Service Level Objective* (SLO).

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

The cloud computing paradigm has become very popular primarily because of proliferation of on-demand web applications, commoditization and virtualization of compute as well as storage technology and *pay as you go* pricing model. Various kinds of cloud computing paradigms have gained foothold, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Each of these computing paradigms are fast evolving with new higher level services, for e.g. monitoring, auto scaling etc. Cloud management systems behind these services face a formidable challenge of building an elastic resource management system for meeting service SLA while efficiently managing the underlying infrastructure. Administrators of enterprise web applications also face the same challenge even if they are are leveraging cloud computing platforms underneath. This thesis discusses challenges in making these large systems elastic in a cloud computing environment and presents solution-systems for the same.

What is Elasticity? The concept of elasticity has its origin in physics, where a material is called elastic if it regains its original shape when the subjected stress is reduced to zero. A more abstract definition of elasticity is found in economics [43], where it is defined as a ratio of percentage change in one variable, say y, to percentage change in another, say x. It is also understood as a measure of responsiveness of y with respect to change in x. From a cloud-computing point of view, elasticity is the responsiveness of quantified system-capacity (i.e. capacity of a system either in terms of resources allocated or in terms or number of requests serviced) with respect to its workload change.

1.1 Background and Motivation

Enterprise web applications often sprawl over a very large Infrastructure Technology (IT) resource. The performance management system of such enterprise applications requires elastic scaling of allocated IT resources, in accordance to the application workload. To be able to support elasticity, enterprises either own dedicated datacenters or rent infrastructure in commercial datacenters. In either of the cases there is a dedicated management service which understands the enterprise application's elasticity needs and translates into lower level IT requirements; on the other hand there is a data center management service, which manages data center's physical resource to satisfy the application's resource requirements. Many owners of large scale data centers employ their datacenter management service to create an application hosting platform and rent infrastructure to multiple applications (or customers). Cloud computing platform refers to such a hosting platform that rents data center resources and also offers programmatically consumable resource management and monitoring services to the end consumers.

Clouds have become popular IT delivery platforms as they offer benefits like elasticity, low operational cost, and ease of IT management. But managing elasticity of a large distributed system (such as a large enterprise web application or the cloud management system itself) is a very challenging task. IT administrators of such large distributed system need to constantly manage its capacity to avoid SLA violations as the workload supported by such a system is often very dynamic. The dynamic demand for resources, complexity of enterprise applications and management systems, coupled with heterogeneity of resources in a cloud environment and their non-linear pricing result in many challenging distributed systems and resource management problems.

In addition to this multiple similar cloud computing platforms by different service providers have opened up the possibility of hybrid cloud platforms, which can leverage services across multiple cloud platforms. This raises new challenges in making an enterprise application elastic.

This thesis investigates the problems of elastic provisioning of enterprise applications and also that of cloud platforms in a cloud computing environment. I propose autonomic systems/solutions that minimize the impact of management operations on enterprise applications both in terms of cost and time, while adhering to a specified Service Level Objective (SLO).

1.2 Thesis Contributions

This thesis focuses on problems concerning elasticity (or dynamic resource management) of enterprise applications as well as that of cloud management systems. Various aspects of elasticity studied in this thesis are not new research problems by themselves, but addressing them in a cloud computing context brings in new challenges. In each case, I propose novel solutions that combine various cloud-environment specific methods with modeling and optimization techniques to build intelligent systems which achieve elasticity with minimal impact on application's performance SLAs.

1.2.1 Contribution Summary

The key systems and contributions of this thesis are:

- *Cost aware elasticity in cloud*: An approach for supporting elasticity in the cloud in a cost effective manner. It accounts for the resource cost as well as the operational/transition cost – i.e. cost of reconfiguring an application – while computing the application reconfiguration steps.

- *Provisioning for the tail*: A queueing model driven approach for computing provisioning capacity of multi-tier cloud applications so as to achieve an SLA, which is expressed as a percentile bound on the end to end response time.

- *Cloud bursting*: A system to facilitate the use of hybrid cloud platforms for an application hosting environment by determining which applications can be transitioned

into the cloud most economically, and automating the process of transitioning at the proper time. It optimizes the deployment of applications into the cloud thereby lowering the bursting time from hours to minutes.

- *Flexible control plane services*: A system, based on hybrid approach of empirical profiling and modeling, for deciding architectural configurations of control plane services of a private cloud management system.

These systems address a variety of challenges in providing elasticity to applications in single as well as hybrid cloud environments.

1.2.2 Cost Aware Elasticity in Cloud

Cloud environments enable elastic provisioning by providing a variety of server configurations as well as mechanisms to add or remove server capacity. This provides flexibility to the customer but also makes the decision process challenging. Non-linear pricing of compute resources in cloud and dependence of transition cost, of elasticity mechanisms, on the application configuration increases the complexity of the problem.

I propose a generalized provisioning framework for supporting elasticity in the cloud, which is able to account for pricing differences of various resource configurations to suggest the most economic solution to an application. It can also account for the transition latency of available elasticity mechanisms to find a solution that will minimize transition overhead. Because the overheads of different elasticity mechanisms are largely dependent on the available cloud interfaces and implementation, evaluation on a real cloud platform is crucial. Hence, I demonstrate the effectiveness of the system through an experimental evaluation in both private and public clouds, which provide different elasticity methods.

1.2.3 Elastic Provisioning for the Tail

Enterprise web applications are known to observe highly variable workload and provisioning correct capacity for these applications is a challenging problem. Administrators,

because of high variability in load and response times, prefer SLA expressed for the peak workload or for a very high percentile of the response time distribution – for instance at Amazon the service SLAs are measured and expressed at the 99.9^{th} percentile of the distribution [32]. Provisioning for a SLA expressed as a high percentile of response time distribution is a challenging problem. The problem becomes even more complex because of i) multi-tiered nature of modern enterprise web applications, ii) hardware heterogeneity of cloud platforms, and iii) non-linear pricing in cloud platforms.

In this thesis, I present a new model driven provisioning approach targeted for cloud platforms. My approach focuses on i.) allocating capacity based on peak (high percentile) of the workload, ii) taking a holistic view of the entire multi-tier application by considering bounds on on end to end response times while making provisioning decisions and iii) accounting for cloud server configs and pricing models when determining the most cost effective config to provision a certain amount of capacity.

1.2.4 Cloud Bursting

Enterprises often have significant investments in their own IT data centers that house compute and storage systems for their applications. But provisioning applications for peak workload is very expensive and wasteful as they are infrequent. Rather than incurring capital expenditures for additional server capacity to handle infrequent workload peaks, a hybrid model has emerged where an enterprise leverages its local IT infrastructure for the majority of its computing needs, and supplements it with public cloud resources whenever local resources are stressed (a.k.a. *cloud bursting*)

Commercial and open-source virtualization tools are beginning to support basic cloud bursting functionalities but their primary focus is on the underlying enabling mechanisms. These systems leave significant policy decisions in the hands of system administrators, who must manually determine when to invoke cloud bursting and which applications to "burst".

In this thesis I have developed Seagull, a system to alleviate the above challenges. It automatically detects when local infrastructure is becoming overloaded, decides which applications can be migrated to the cloud at lowest cost, and then performs the migrations needed to dynamically expand capacity as efficiently as possible. By automating these processes, Seagull is able to respond quickly and efficiently to workload spikes.

1.2.5 Flexible Adaptive Control Plane for Private Clouds

Enterprises with existing IT infrastructure are beginning to employ private clouds to manage their infrastructure. Often the early deployments of private clouds are small and they eventually observe a demand of scaling to a large infrastructure. This poses a huge challenge to the private cloud administrators as individual control plane components need to support a larger scale deployment without violating control plane specific *Service Level Objectives* (SLOs). I identify the challenge as a three pronged problem; i) framework for dynamic scaling of control plane services ii) identifying configuration patterns for each services and, iii) dynamic reconfiguration of services to address demand specific changes. In this thesis, I present a virtualization-based approach to address the issue of dynamic scaling of control plane components and present a generic two step solution, based on empirical profiling, for finding a suitable configuration for any control plane service. Finally, we demonstrate the efficacy of the approach on two important control plane services, namely messaging and monitoring, on a prototype developed on OpenStack.

1.3 Thesis Outline

Chapter 2 provides background on cloud platforms and dynamic provisioning to set the context of my work. Chapter 3 describes *Kingfisher*, a cost aware auto scaling system for clouds. Chapter 4 describes a technique to compute correct provisioning for multi-tier cloud applications when the SLA is expressed as a threshold on a high percentile of end to end response time. Chapter 5 discusses *Seagull*, a system which performs dynamic

provisioning on hybrid cloud platforms by enabling cloud bursting and its proposed future work. Chapter 6 presents the work on scaling the control plane services of a private cloud management service. Finally, Chapter 7 concludes with status of completed work and lists remaining milestones for this thesis.

CHAPTER 2

RELATED WORK

This chapter presents a survey of literature relevant to the area of dynamic provisioning and monitoring to set the perspective for our contributions. Each chapter presents a more detailed related work relevant to the chapters.

2.1 Cloud Computing

Cloud computing refers to the software services delivered to end users over the internet as well as the backend hardware and systems software supporting them [5]. Vendors (or systems) which provide cloud computing to end consumers can be partitioned into three broad classes based on the level of abstraction exposed to the consumer, namely Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

Software as a Service (SaaS) refers to a model where a high level functionality (i.e an application) is delivered as a service over the internet [89]; SalesForce, Google Apps are some of the commercial SaaS offerings. The idea is to expose interfaces to an application software (like a spread-sheet or an HR-management solution etc). The challenge is of maintenance and management of the software are hidden from the client.

Platform as a service (PaaS) is a model where an application execution platform for a specific programing language is made available to the consumers, for instance a JVM platform for executing various Java APIs. PaaS providers often enforce a structure on the applications which they support on their platform. This limits the freedom of the platform consumer (i.e. an application developer) but helps in making various platform management

problems tractable. Chohan et al. in [25] outline the design and implementation of Appscale, an open source PaaS for Java; it is an effort to replicate the PaaS offering provided by Google AppEngine. Microsoft Windows Azure, Google AppEngine, Force.com, Heroku are some of the commercial PaaS offerrings.

Infrastructure as a Service (IaaS) model, essentially, refers to a facility that provisions compute, storage and network resources and charges the customers for the provisioned resources per-unit time. Nurmi et al. in [72] discuss the basic principles of such systems and provide details of Eucalyptus, an open source software for IaaS. OpenNebula [74], and OpenStack [79] are two popular open source IaaS platforms, while Amazon EC2, NewServers, Simple Storage Service (S3) are some of the examples of commercial IaaS platforms.

Vendors of these cloud computing offerings face challenges of IT infrastructure capacity planning, installation and management to meet the application performance guarantees of software systems (or services) running on top of them. These challenges are intensified by the massive scale of backend data centers, and unpredictable workload dynamics. Some of the key research challenges in the area of cloud computing are: i) resource and performance management, ii) energy management, iii) security and privacy and iv) business resiliency.

Cloud computing platforms offer very low cost services due to economies of scale. Thus the first technical challenge is to architect a large, efficient, modular and reliable datacenter. The trend is to use less reliable but cheaper commodity hardware for the datacenter, which is managed by efficient and redundant infrastructure management software system [46, 3, 56].

An infrastructure management software system, thus, is the most important component for cloud computing vendors as it provides all the necessary reliability and efficiency. Efficient *resource management* is of prime importance to cloud vendors as it enables them to reduce costs without violating Service Level Agreements (SLAs). Current cloud man-

agement systems are extensively using virtualization technologies [70] to increase resource utilization [22, 34, 37, 53]. Virtualization also offers performance isolation between multiple instances running on the same hardware [112, 63, 45]. The majority of resource management solutions are implemented in different management layers stacked on top of the abstractions exposed by the hypervisors, which enable virtualization [18].

Large data centers also require considerable amount of energy for its running as well as cooling, and this is one of the very important concerns from environmental as well as monetary point of view; research is being pursued under the umbrella of *green computing* for better solutions of power and workload management and in data centers [19, 106, 84].

Security and *data lock-in* are viewed as primary barriers to the faster adoption of cloud computing by enterprises [65, 82]. Although many cloud computing security problems are not new, but they require new solutions in terms of mechanisms [23, 100] and research is being aggressively pursued to find new solutions.

This thesis focuses on the resource and performance management of Infrastructure as a service cloud platforms. The problems have been formulated from both cloud computing client as well as service provider's perspective.

2.2 Dynamic Resource Provisioning

The problem of dynamically varying the resource allocation according to the workload variation such that a system is able to meet its SLA targets is called *dynamic resource provisioning*. It has become an important area of research and product development for the past few years. Armbrust et al. [6] specifically list *resource elasticity* as one of the key opportunities in cloud computing.

Virtualization technologies like Xen [8], KVM [59], VMWare [113] have played a significant role in aiding to dynamic management of server resources by using techniques of predictable allocation of processor [37, 53] disk bandwidth [51] and network bandwidth [22, 41]. Supplementing these are the virtual machine and data migration technologies,

which migrate workloads between servers [44, 92]. Resource management at the level of large data centers leverage these virtualization and migration technologies to manage and balance load across data center [117, 115]. In this thesis we use the virtualization tools for VM provisioning and migration either directly or indirectly via a private cloud management system, like OpenNebula or OpenStack.

Elastic scaling of application capacity, deployed in data centers or private clouds has been a topic of considerable current interest. There is an extensive body of work on dynamic provisioning of web applications in data centers; several projects have adopted an empirical approach of estimating SLO violations [94, 125, 92] and performing elastic scaling of applications, while other efforts have used a wide variety of analytic models of both applications and underlying IT infrastructure. A large number of researchers have used *queueing theory* tools to develop performance models [86, 107, 36, 104, 99, 10, 103], while others have leveraged *classical feedback control theory* or *machine learning* in their work [60, 81, 1, 58, 21]. In this thesis we have leveraged both empirical as well as modeling techniques to construct intelligent autoscaling systems, but unlike other research efforts, our work accounts for the non-linear pricing as well as the heterogeneity of cloud platforms.

2.3 Hybrid cloud

A hybrid cloud model is where the cloud management system transparently supplements local infrastructure with computing capacity from external cloud environment. It is increasingly common for businesses and service providers to own multiple data centers so managing resources across data centers is an increasingly challenging [97, 87, 14]. We believe that as data centers become connected by increasingly high bandwidth links, automated resource management techniques will naturally expand to include cross data center management approaches like cloud bursting (i.e. transitioning applications across clouds). The idea of hybrid clouds was proposed by Amazon's Jeff Barr as a way to allow enter-

prises, who already own significant amounts of IT infrastructure to still make use of the cloud during periods of high demand [27]. Researchers have been investigating the potential economic savings by using cloud bursting in specific domains such as medical image processing [57] and publishing [55]. Cloud bursting generally assumes that a private data center is connected to a public cloud, producing what is known as a hybrid-cloud. Hybrid clouds have become a popular service offering for hosting and data center companies [79, 74, 111], and also have been the subject of research [62, 96] but their primary focus is on the underlying enabling mechanisms. My work in this thesis uses some of the low level migration technologies available in the open cloud platforms to create necessary migration tools in a hybrid cloud and builds an autonomic system to address the problem of overload in a private cloud environment.

CHAPTER 3

COST-AWARE ELASTICITY IN THE CLOUD

Cloud computing enables application providers to allocate resources purely on-demand. This ability to allocate resources on an as-needed basis which we refer to as *elasticity*, can yield significant cost savings, but also raises new challenges for the application providers, particularly in an Infrastructure as a Service (IaaS) cloud. In this chapter we present Kingfisher, a *cost-aware* system that provides efficient support for elasticity in the cloud by (i) leveraging multiple mechanisms to reduce the time to transition to new configurations, and (ii) optimizing the selection of a virtual server configuration that minimizes the cost.

3.1 Introduction

Cloud computing is very attractive because of its usage-based pricing model – organizations only pay for the resources that are actually used, and can flexibly increase or decrease the resource capacity allocated to them at any time. This *elasticity* provided by Cloud computing can yield significant cost savings when compared to the traditional approach of maintaining an expensive IT infrastructure that is provisioned for peak usage – organizations can instead simply rent capacity, and grow and shrink it as the workload changes.

Cloud environments enable flexible, elastic provisioning by supporting a variety of hardware configurations and mechanisms to add or remove server capacity. However this flexibility also raises new challenges for application providers: (i) given several available resource configurations for a particular workload, which one to choose, and (ii) how best to transition from one resource configuration to another to handle changes in workload.

The first decision arises from the availability of a number of server configurations, each with a different amounts of virtual CPU cores, memory, and disk space to satisfy the same resource requirements. The array of available hardware configurations leads to a number of different ways to configure a typical multi-tier Web application. Further, these server configurations are typically not priced linearly with server capacity. For instance, a quad-core server may not be four times as expensive as a single-core server. As shown in Table 3.1, depending on the exact configuration, the price per core of a server may be higher or lower than the cost of a single-core system, and a careful choice of configuration may lower the total infrastructure cost. The second decision arises when adding more server capacity to accommodate an increase in the application request volume, for example. There is a similar array of choices in determining the new resource configuration (e.g., adding a new replica or move the application to a larger server), as well as different costs or overheads based on the mechanism used to make the transition to the new target configuration.

This chapter, we present a new approach for dynamically provisioning virtual server capacity that exploits pricing models and elasticity mechanisms to select resource configurations and transition strategies that optimize the incurred cost. In this chapter makes the following contributions:

- **Cost-aware elasticity.** We present Kingfisher, a cost-aware system that integrates multiple elasticity mechanisms such as replication and migration and computes both a cost-optimized configuration for the desired capacity as well as a plan for transitioning the application from its current setup to its new configuration. Kingfisher's algorithms can take into account price differentials in the per-core cost of different server types to minimize the *infrastructure cost* of provisioning a certain capacity. Kingfisher also minimizes the time to add extra capacity using different elasticity mechanisms (we call this time as *transition cost*). We formulate our provisioning problem as an integer linear program (ILP) to account for both infrastructure and transition cost for deriving appropriate elasticity decisions.

- **Prototype implementation and experimentation on public and private clouds.** We implement a prototype of the Kingfisher cloud provisioning engine, using the OpenNebula cloud toolkit [75], that incorporates our optimizations, and evaluate its efficacy on both a private laboratory-based Xen cloud and the public Amazon EC2 cloud. Our experimental results (i) demonstrate that cost-aware elasticity can reduce infrastructure costs by 24% , and by 35% in EC2 in comparison to cost-oblivious provisioning approaches, (ii) demonstrate that integrating multiple mechanisms such as migration and replication into a unified approach can double the cost savings, and (iii) demonstrate how our transition-aware approach can be employed to quickly provision capacity in scenarios where an application workload surges unexpectedly. In our experiments, we observed transition time improvements of 2x in the private cloud and up to 6x in EC2 using transition-aware elasticity.

- **The Case for Cost-aware Elasticity:** While there has been significant research on dynamic capacity provisioning for data center applications, there are three key differences between prior work and capacity provisioning in the cloud. First, some of prior work on dynamic provisioning has been *cloud provider centric*, where the data center provider attempts to maximize resource utilization by dynamically allocating a set of servers across hosted applications with varying workload demands (and attempts to statistically multiplex as many applications as possible on the data center). In contrast, the problem articulated in this chapter requires a *customer-centric* view, where each customer ("application provider") individually optimizes their capacity needs by choosing the best server configuration that matches their needs. Cloud provider centric approaches attempt to maximize revenue while meeting an application's SLA in the face of fluctuating workloads, while a customer-centric approach attempts to minimize the cost of renting servers while meeting the application's SLA.

  Second, the prior work on dynamic provisioning has not been *cost-aware*. By being cost-oblivious, prior approaches assume that so long as the desired capacity is allo-

cated to the application, the choice of exact hardware configuration is immaterial. That is, the unit cost per core is assumed to be identical, making an N-core system equivalent, from a provisioning perspective, to an N-core systems with single cores. In the cloud context, however, the choice of the configuration matters, since pricing per core is not uniform. Hence, Kingfisher must take server infrastructure costs into account during provisioning.

Third, much of the prior work on provisioning has employed replication as the primary means to increase an application's capacity. The application is assumed to be replicable, and workload increases are handled by adding additional server instances to the application's pool of servers. An alternative method for capacity provisioning is to employ migration, where an application and its data are migrated to larger capacity server (e.g., a server with more cores) to handle workload growth. As we will show in this chapter, Kingfisher considers both replication and migration when choosing the best method of transition the application from one capacity configuration to another.

3.2 Cloud Background and Problem Statement

Consider a cloud computing platform that offers N heterogeneous server configurations for rent, each with a different rental cost (infrastructure cost). The pricing of servers is assumed to be arbitrary. Thus, the pricing can be convex, where the cost per-core increases sub-linearly with the number of cores, or concave where more the cost of more capable servers increases super-linearly with the number of cores, or arbitrary where some other pricing formula is employed. As noted in Table 3.1, both the EC2 cloud and the NewServer (NS) cloud platform employ a convex function for their most popular choices (e.g., small, medium, large) and the pricing model becomes arbitrary when the "high-CPU" or "fast CPU" configurations are taken into account.

Amazon EC2 Cloud Platform			
Server size	Configuration	Cost/hr	$/core
Small	1 ECU, 1.7GB RAM, 160GB disk	$0.085	$0.085
Large	4 ECUs, 7.5GB RAM, 850GB disk	$0.34	$0.085
Med-Fast	5 ECUs, 1.7GB RAM, 350GB disk	$0.17	$0.034
XLarge	8 ECUs, 15GB RAM, 1.7TB disk	$0.68	$0.085
XLarge-Fast	20 ECUs, 7GB RAM, 1.7TB disk	$0.68	$0.034
New Server's NS Cloud Platform			
Small	1-core 2.8GHz, 1 GB RAM, 36GB disk	$0.11	$0.11
Medium	2-core 3.2 GHz, 2 GB RAM, 146GB disk	$0.17	$0.085
Large	4-core 2.0GHz, 4GB RAM, 250 GB disk	$0.25	$0.063
Fast	4 core 3.0 GHz, 4 GB RAM, 600GB disk	$0.53	$0.133
Jumbo	8 core 2.0GHz, 8GB RAM, 1TB disk	$0.60	$0.075

Table 3.1: Cloud server configurations and their prices. For EC2, 1 ECU= 1.2 GHz Xeon or Optron circa 2007.

We assume that these servers can be allocated or deallocated on-demand by a customer for her application. From an application standpoint, these capacity changes can be made either via *replication*—by adding or removing replicas—or via *migration*—by moving the application to a larger or a smaller server. If a specific cloud platform exposes a subset of these mechanisms (e.g., the EC2 cloud does not presently support live migration), then our approach must take these constraints into account when provisioning resources. We assume that an application is distributed with k interacting components (e.g., k tiers in multi-tier applications); each tier has an SLA associated with it that must be met by provisioning sufficient capacity to service that tier's workload.

Given such a cloud platform, the goal of our work is to develop a system that supports elasticity for applications by (i) choosing the most cost-effective elasticity mechanism (e.g., replication, migration) when adding or removing capacity, and (ii) choosing the most cost-effective server configuration. The elasticity problem arises both when initially provisioning/deploying an application in the cloud as well as during any subsequent reconfiguration.

3.2.1 Initial Provisioning

Assuming an application with k independent components/tiers, let λ_i denote the peak estimated workload seen at tier i. Then, the initial deployment problem is one of determining *how many* cloud servers to provision for each tier and of *what type* such that the infrastructure cost is minimized and a peak workload of λ_i can be sustained at each tier while meeting per-tier response time SLAs. Since the desired capacity can be satisfied using multiple hardware configurations, the goal is to choose the cheapest configuration that meets the needs of each tier. We compute the initial configuration and deploy the application.

3.2.2 Subsequent provisioning

Once an application has been deployed on the cloud, its workload demands may change over time—due to incremental growth or sudden change in popularity. In such cases, the application will need to be reconfigured by dynamically increasing (or decreasing) the capacity at each tier. The problem of subsequent re-provisioning is one where, given a certain server configuration that is already in use, we must determine a new configuration that specifies how many cloud servers and of what types to use for each tier to sustain the new peak workloads of λ_i' at tier i. Furthermore, we must also specify a *plan* for morphing each tier from its current configuration to the new configuration using mechanisms such as resizing, migration or replication. Thus, for subsequent provisioning decisions, we are interested in minimizing two types of costs: (i) the *infrastructure cost* of the servers, and (ii) the *transition cost*, defined as the latency, to move the current to the new configuration.

Depending on the scenario, a customer may be interested in optimizing the infrastructure cost, the transition cost or some combination of the two. For instance, steady growth in workload volume can be handled by computing a new configuration that minimizes the infrastructure cost of servers. In contrast, a sudden surge in workload—caused by a flash crowd—will require additional capacity to be brought online as quickly as possible. In this

scenario, it is more important to reduce the latency to bring additional capacity online even if it implies choosing a configuration that incurs a somewhat higher infrastructure cost. Such a transition cost aware approach must consider different configurations that offer the same capacities and pick the one that offers the fastest migration path.

3.3 Cost-aware Elasticity Algorithms

Any dynamic provisioning algorithm involves two steps: (i) *when* to invoke the provisioning algorithm, and (ii) *how* to provision capacity so as to minimize infrastructure or transition cost.

3.3.1 When to provision?

The provisioning algorithm can be triggered in a proactive or a reactive manner. A proactive approach uses workload forecasting techniques to determine when the future workload will exceed currently provisioned capacity and invokes the algorithm to allocate additional servers before this capacity is exceeded [49]. In contrast a reactive approach uses thresholds on resource utilization or on SLA violations to trigger the need for additional capacity. A combination of predictive and reactive approach is also employed to handle prediction inaccuracy and also to avoid oscillations in provisioned capacity due to oscillations[1] in workload [104]. The issue of proactive or reactive invocation is orthogonal to that of cost-aware provisioning, and hence we choose perfect forecaster, i.e. a forecaster that knows the workload in advance. Next we discuss *how* to provision for optimizing infrastructure/transition cost.

[1] Inaccurate workload prediction can lead to a rapid oscillation of the workload forecast between its increase and decrease.

3.3.2 Infrastructure Cost-aware Provisioning

Given the estimated peak workload $\lambda_1, \lambda_2, \ldots \lambda_k$ that must be sustained at each tier, the goal of our approach is to compute which type of cloud server to use and how many at each tier so as to minimize infrastructure cost; the provisioned servers must have the collective capacity to service at least λ_i request/s at tier i while meeting tier's response time SLAs.

Our cost-aware provisioning algorithm involves two steps: (1) for each type of cloud server, compute the maximum request rate that the server can service at each tier, and (2) given these server capacities, compute a least-cost combination of servers that have an aggregate capacity of at least λ_i.

3.3.2.1 Step 1. Empirical Determination of Server Capacities.

For each server configuration supported by the cloud platform (e.g., small, medium, large), we must first determine the maximum request rate that each configuration can sustain for this application. This information is used in the subsequent step by our provisioning algorithm to determine how many servers of a particular type will needed to service the peak workload λ_i. Clearly, the maximum request rate (i.e., the server capacity) depends on the nature of the application, its workload mix and the server type.

One possible approach for estimating the maximum workload that can be serviced by a particular server type is to employ queuing theory [103], where the server is modeled as a queuing system and queuing theoretic results are used to derive a relationship between the request rate, service times of requests, and the response time SLA. This approach can not account for software artifacts that limit the application capacity from scaling with the number of cores, causing the queuing-based model to overestimate the capacity of multi-core systems.

To overcome this drawback, we employ a systems approach that uses empirical profiling—Kingfisher estimates the maximum server capacity by running the application on different hardware configurations, subjecting them to gradually increasing workloads, and determin-

ing the point where the server saturates. Such an empirical approach is more accurate since capacities are computed using actual measurements on real hardware and can account for software artifacts since the actual application behavior is used when estimating capacities. The approach, however, requires an application provider to carefully set up and profile the application on various hardware configurations supported by the cloud platform, and such profiling is more involved than the simple measurements required by the queuing approach. We note, however, that a system such as JustRunIt [125] that can clone virtual machines and run the cloned application on a sandboxed server can be exploited to reduce the overheads of such an empirical approach. Once the maximum request rates of the various servers supported by the cloud platform have been determined, this information is subsequently used by the provisioning algorithm.

3.3.2.2 Step 2. Determining Server Configurations.

Consider a cloud platform with M different types of servers (e.g., small, medium, large). Let C_j and p_j denote that capacity (maximum request rate) and the infrastructure cost of server type j. Let λ denote the peak workload request rate for which capacity needs to be provisioned at a tier. The problem of infrastructure cost-aware provisioning is stated as

$$\text{minimize} \sum_{j=1}^{M} n_j p_j, \tag{3.1}$$

such that

$$\sum_{j=1}^{M} n_j C_j \geq \lambda, \tag{3.2}$$

where n_j denotes the number of servers of each type that is chosen. This optimization problem can be formulated and solved as an integer linear program, as discussed later in this section. The ILP solution yields $(n_1, n_2, \ldots, n_M)$ — which tells the application provider how many servers of each type should be chosen for the application tier. Notice that the ILP can handle both the capacity increase and capacity decrease.

3.3.3 Transition Cost-aware Provisioning

While our infrastructure cost-aware provisioning algorithm minimizes recurring infrastructure costs, it does not account for (or optimize) the transition latency to move from the old configuration to the new—a factor that depends on the size of the application's disk and/or memory state. In many cases, this latency is important, especially when additional capacity needs to be added to the application quickly (e.g., during a flash crowd).

To be able to handle such scenarios, the provisioning approach must be able to estimate the latency of using different provisioning mechanisms, such as replication, migration and resizing. By taking into account the latency of such mechanisms, a configuration that minimizes such overheads is chosen. We estimate the overhead of these mechanisms as follows:

- *Local resizing:* Local resizing involves using the hypervisor API on a machine to modify the resource allocation of a virtual machine (e.g., to give it more RAM or to allocate it additional cores or CPU shares). This can be done efficiently with minimal overheads (the latency is on the order of tens of milliseconds). Hence, local resizing is always the most desirable option to scale a VM's capacity. However, since the physical server may lack sufficient idle capacity for resizing, the algorithm must frequently resort to other options.

- *Replication:* Starting up a new instance (replica) of an application tier involves copying the machine image of the OS/application from central storage to the disk on the new server, starting up the OS and the application replica, and reconfiguring the application to make it aware of the new replica. The latency can be estimated as $\frac{D}{r} + b$, where D is the size of the disk image, r is the network bandwidth available for the copy operation and b is a constant representing the OS and application startup time.

- *Live migration:* Live migration of a virtual machine from one server to another involves copying the memory state of the VM to a new server while the application is

running (memory pages that are dirtied during the copy phase are iteratively resent). Typically live migration mechanisms assume that the disk state of the VM is maintained on a shared file system. Hence, the latency of the live migration is $w \cdot \frac{R}{r}$, where R is the size of the VM's RAM, r is the network bandwidth available for the copy operation, and w is a constant that captures the mean number of times a memory page is (re)sent over the network (due to dirtying of pages during the migration process).

- *Shutdown-migrate.* While live migration is a implemented in most popular hypervisors such as Xen and VMware, some public clouds such as Amazon's EC2 do not currently expose this option. Migration can be "simulated" in a public cloud by suspending the application, converting its disk state into a new machine image, copying the machine image to a new server and restarting the image on the new machine. Since the disk state may need to be copied twice, once to construct a new machine image and then to copy the machine image to the new server[2], the latency of this approach is $2\frac{D}{r} + b$.

The transition-aware approach then attempts to minimize this overhead by preferring mechanisms that incur the lower copying overheads (and hence, lower latencies). Like before, this can be stated and solved as an ILP optimization problem as discussed next.

3.3.4 An ILP-based Elasticity Algorithm

Both infrastructure and transition cost-aware provisioning problems can be stated using the following integer linear program (ILP). Let M denote the number of server types supported by the cloud platform; Let p_j denote the infrastructure cost[3] for server type j and let C_j denotes its maximum capacity. Let λ denote the peak workload for which the applica-

[2]In Amazon's EC2, the disk state must be uploaded to its S3 storage system as a machine image and then copied over to the new server, resulting in two copy operations

[3]Price changes are handled currently by updating the pricing parameters and recomputing the provisioning solution.

tion needs to be provisioned, and let N denote the maximum number of servers that could be needed to satisfy λ (any large number can be chosen as N). Let T denote the number of the provisioning mechanisms supported by the platforms (e.g., replication, migration, resizing). Then the objective function for minimizing *infrastructure cost* is

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} p_j x_{ijk} \tag{3.3}$$

subject to the constraints

$$\sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} C_j \geq \lambda \tag{3.4}$$

$$\sum_{j=1}^{M} \sum_{k=1}^{T} x_{ijk} = 1, \forall i \tag{3.5}$$

The terms x_{ijk} is an integer variable in the ILP that can take values of 0 or 1; A value of 1 indicates that server i is transformed into server-type j using a provisioning mechanism k (e.g., replicate or migrate); a value of 0 indicates that that option was not chosen by the ILP. The output of the ILP is set of values x_{ijk} that denotes which server types are chosen and also specifies a plan for transitioning for each server i to the new server type j using method k (replicate. migrate etc). If this is the first time the application is being deployed onto the cloud, the current configuration is empty; for subsequent (re)provisioning, the plan specifies how the current configuration is to be morphed into the new configuration (e.g., using replication, migration etc); note that the cost, p_j, in (3.3), is independent of the mechanism k, which means that all reconfiguration mechanisms are considered equal as long as they provide the same final capacity. However, this formulation becomes useful in capturing the transition cost as described below.

The ILP for *transition-aware* provisioning is identical to the previous one except for the optimization criteria which must minimize the transition cost rather than infrastructure cost, and thus Equation (3.3) changes to:

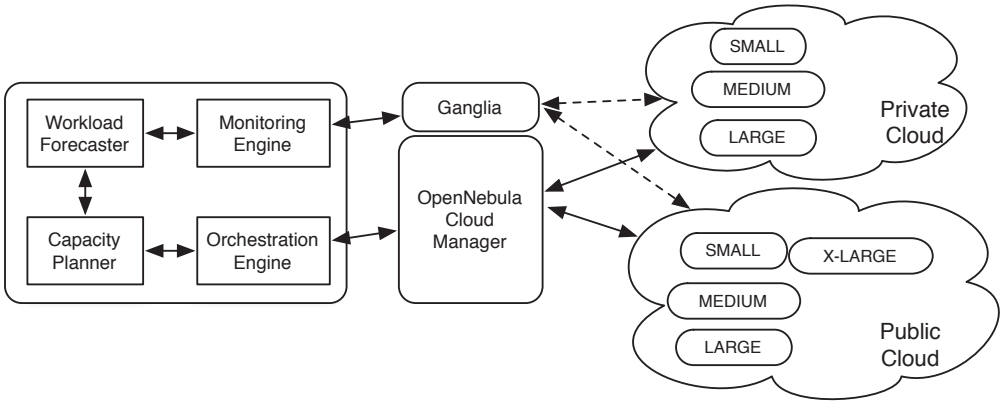


Figure 3.1: Architecture of our Kingfisher prototype

$$\min \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{k=1}^{T} m_{ijk} x_{ijk}. \tag{3.6}$$

Here m_{ijk} is the cost of transforming server i to server-type j using mechanism k. This cost is estimated using the mechanism-models mentioned in section 3.3.3 that capture the overhead of replication, live migration etc[4]. Like before, $x_{ijk} \in \{0, 1\}$ indicate whether the final solution will employ technique k to transition server i to server type j.

Although for a small problem (with nodes less than 10) a perfect solution can be obtained by solving the above formed ILP, as the size of the problem increases finding the optimal solution becomes hard. We have implemented a greedy-type heuristic with a worst case bound of 2 for an approximate solution of the above ILP [31]. The basic idea of the heuristic is to sort $x_{i,j,k}$ in increasing order of p_j/C_j and then find the smallest list of $x_{i,j,k}$'s which satisfy Eq. (3.4). Once an $x_{i',j',k'}$ has been chosen for a particular $i = i'$, we skip the remaining $x_{i',j,k}$; this ensures that we satisfy the constraint in Eq. (3.5).

3.4 Kingfisher System Implementation

We have implemented a prototype of Kingfisher, a system that supports elasticity in today's public and private cloud computing platforms. Kingfisher presently supports both Amazon's EC2 public cloud and Xen-based private clouds. Kingfisher combines an application-centric provisioning engine with a cloud management platform. It assumes a virtualized cloud platform and provides support for virtual machine (VM) deployment, VM image management, in conjunction with elastic provisioning. Kingfisher uses a modified version of the OpenNebula toolkit to implement its cloud management mechanisms—e.g., to deploy/undeploy VMs on a set of servers in a private-cloud, create/terminate instances on Amazon's EC2, and to reconfigure applications with more or less capacity. We use the XML-RPC APIs exposed by OpenNebula deploy, terminate, or reconfigure servers allocated to an application.

The architecture of Kingfisher and its relationship to the cloud orchestration framework is shown in Figure 3.1. We briefly describe below the key components of our architcture, the details are given in [91].

3.4.1 Monitoring engine

Our monitoring engine tracks application-workload and system resources. The monitoring data is stored in a round-robin database[5] [78]. We have implemented our monitoring engine by enhancing Ganglia [40]. Each VM image is pre-configured with the reporting agent; thus, when new virtual machines are dynamically deployed, the Ganglia server automatically recognizes new servers and begins to monitor them without the need for any additional configuration. In scenarios where the cloud platform provides monitoring ca-

[4]Using the model of mechanisms described in section 3.3.3, we pre-compute a matrix, say $M' = [m'_{ijk}]_{i,j=1...M;k=1...K}$, which represents the cost of migration from server-type i to server-type j using mechanism k. We use M' to compute $m_{i,j,k}$ of (3.6)

[5]In a round-robin database (RRD) time-series data like network bandwidth, temperatures, CPU load etc. is stored. The data is stored in a way that system storage footprint remains constant over time. This avoids resource expensive purge jobs and reduces complexity

pabilities (e.g., Amazon EC2 CloudWatch), our monitoring engine can directly query the cloud platform APIs, rather than Ganglia databases, to obtain these metrics.

3.4.2 Workload Forecasting

The workload forecasting component in Kingfisher uses the workload statistics gathered by the monitoring engine to derive estimates of future workloads. We use the open-source R statistical package to forecast workloads. In our experiments (in Section 3.5), we focus on evaluating the cost benefits of Kingfisher, hence we assume a perfectly accurate forecaster but any other forecaster can be seamlessly used in its place.

3.4.3 Capacity planner

The capacity planner is at the heart of Kingfisher's provisioning engine. It implements our ILP-based algorithm for optimizing the infrastructure cost for an application or the transition cost of moving to a new configuration. We employ an *lpsolve*, an open-source LP solver that is invoked via a JNI interface from Kingfisher.

Our ILP-based planner requires several inputs before it can begin computing cost-optimized configuration for an applications. First, the various types of servers supported by the cloud platform and their infrastructure prices need to be specified. Second, all provisioning mechanisms supported by the cloud platform (e.g., migration, replication etc) must be specified, and a model for estimating the cost/overhead of each mechanism must also be specified. Finally, the empirically derived application capacities for each server hardware type must be specified.

Given these configuration parameters, Kingfisher's planner can be invoked by specifying (i) the tier-specific peak request rate λ for which capacity must be provisioned, (ii) the current configuration for the application, which can be empty if this is the initial deployment of the application, and (iii) the optimization objective, which can be infrastructure cost or transition cost.

3.4.4 Orchestration engine

Once an initial or new configuration has been computed, Kingfisher's orchestration engine instantiates the configuration using the transition plan. This component uses the interfaces exposed by the cloud management platform to resize VMs, startup new instances, or migrate existing VMs. The orchestration engine merely specifies the server type to use (e.g., small, medium, large) for each configuration step, and leaves the problem of placement of these VMs onto physical servers to the cloud manager. Thus, the management platform (OpenNebula or EC2) is assumed to track which physical servers are available to create a VM of the desired type for the application. Migrations were implemented by the VM-manipulation capabilities provided by the underlying hypervisor or by EC2.

3.5 Experimental Study for Elasticity: Methodology and Setup

In our experimental investigation for cost-aware elasticity, we consider two environments: (i) Private Cloud - a setup based on our prototype design in a lab setting, and (ii) Public Cloud - conducting our study on Amazon EC2 with some adaptation of our prototype. We conduct experiments with a number of mechanisms for achieving elasticity in the cloud, starting with cost-awareness with replication, and adding migration and transition-cost awareness. Our goal is to understand whether these mechanisms can further improve cost-aware elasticity support beyond the traditional replication-only approach. Our evaluation metrics are the overall infrastructure cost of the virtual servers supporting the application deployment, the cost in terms of latency to change or scale the configuration, and the latency to achieve target application response time after a configuration change.

3.5.1 Cost-aware elasticity mechanisms

We conducted following experiments to study the impact of mechanisms of elasticity on cost:

- **Cost-aware vs Cost-oblivious with Replication:** First, we consider replication-only as the method for supporting elasticity - the typical method that is available in public clouds to support elasticity. Here we compare between resource cost-oblivious (CO-R) and cost-aware (CA-R) approaches to illustrate the benefit of cost-aware approaches.

- **Migration:** Second, we introduce migration in addition to replication as the means for supporting elasticity to investigate benefit from such additional mechanisms beyond base level replication based elasticity[6]. We refer them as CA-RM and CO-RM.

- **Transition cost-aware:** Third, we account for transition cost, defined as the time taken to execute the configuration change to understand its effect on supporting elasticity. We compare the transition cost aware (TA-RM) and transition-cost oblivious (TO-RM) approach to explicitly account for such costs as part of elasticity study.

3.5.2 Experimental Testbed and Workload

For the private cloud, leveraging the prototype we discussed earlier, we use a laboratory-based cloud system built on virtualized Xen/Linux-based cluster, while our evaluation on the public cloud uses Amazon's EC2. We use the java implementation of TPCW [101] for our experiments. TPC-W is a multi-tier web benchmark that represents an e-commerce web application comprising of a Tomcat application tier and a mysql database tier. The workload used to trigger the provisioning the algorithm was browsing mix of the TPC-W specification; that was generated using TPC-W clients. We have tested each approach on two types of workload patterns: 1) smoothly increasing workload (*small-jump* workload) 2) Sharply increasing workload (*large-jump* workload).

---

[6]We implemented the migration of a server-instance to another instance using the *live-migration*, *vcpu-set* and *mem-set* facilities of Xen to perform migration. *Live-migration* migrates a virtual machine (server-instance) to a new host-machine (which has more CPU and MEM), while *vcpu-set* and *mem-set* change the number of virtual-cpus and memory of the virtual-machine.

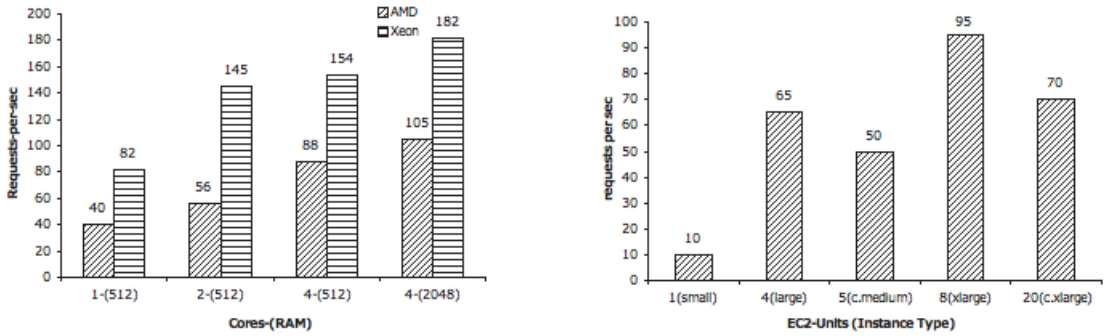
3.5.3 Profiling Server Capacities

Earlier, we have argued that real-world applications will not scale linearly with the number of cores due to software artifacts and differences in processor hardware across different systems. As our decision algorithms will have to determine the amount of resource required to meet the desired application level performance, we resort to empirical profiling to determine the application's capacity on each server type.

We configured TPC-W with both tiers in a single VM, and ran this VM on various server instances of both private and public cloud. In the case of public cloud we used the following EC2 server instances: m1.small (S), c1.medium (M), m1.Large (L), c1.xlarge (XL) and m1.xlarge (XLM); these instances have 1, 4, 8, 5 and 20 EC2 compute units (ECUs), respectively. On the private cloud, it was not possible to have instances equivalent to those of public cloud, nonetheless, we created 1, 2 and 4 core systems; we refer to single-core system as "small" dual-core as "medium" while the quad-core as "large". In each case, we gradually increased the workload seen by the TPC-W application until the server saturated and began dropping requests. Fig. 3.2a plots the empirically derived capacities for various multi-core configurations on our Intel and AMD systems on our *private cloud*. It is quite apparent that server configurations on each processor have a very different capacity and in both cases they scale non-linearly. Fig. 3.2b plots the derived capacities for various EC2-instances.

3.6 Evaluation on a Private Cloud

Our private cloud platform is built on two types of servers: 8-core 2GHz AMD Opteron 2350 servers and 4-core 2.4 GHz Intel Xeon X3220 systems. All machines run Xen 3.3 and Linux 2.6.18 (64bit kernel). Our platform is assumed to support small and large servers, comprising 1, 2 and 4 cores, respectively. These are constructed by deploying a Xen VM on the above servers and dedicating the corresponding number of cores to the VM (by pinning the VM's VCPUs to the cores).



(a) Non-linear scaling of TPC-W on Intel and AMD servers

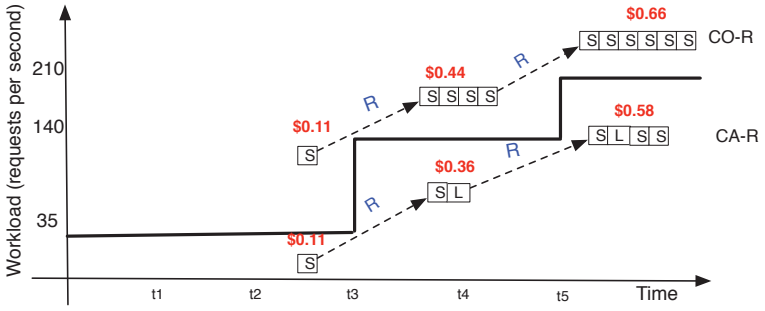
(b) Non-linear scaling behavior of TPC-W on EC2-instances

Figure 3.2: Profiling server instances for private and public cloud

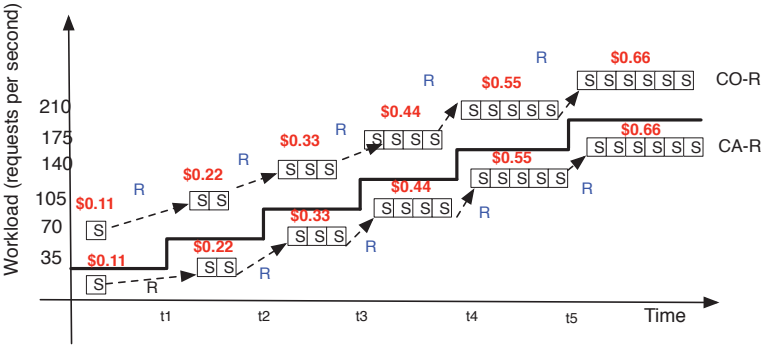
We created a virtual appliance of TPC-W on CentOS 5.2. We have used a modified version Tomcat-5.5.27 as the servlet container and mysql-5.0.45 as the backend database-server; our modified Tomcat server logs the service time of each request, in addition to other default per-request statistics. We also created a dispatcher appliance using the HAProxy load balancer; the dispatcher is used to distribute and load balance across all TPC-W replicas.

3.6.1 Cost-aware versus Cost-oblivious Provisioning

We first compare the cost-aware approach to a cost-oblivious approach (which ignores infrastructure costs when provisioning servers) in a restricted setting where only "replication" is used to modify the deployment. We denote these two approaches as CA-R (cost-aware with replication) and CO-R (cost-oblivious with replication). In these experiments, for simplicity we used two types of server-classes, small and large, with the NS-cloud platform's pricing model, detailed in Table-3.1. We increase the request rate (λ) from 35 to 210 req/s. Fig. 3.3a depicts the server configurations chosen by the CA-R and CO-R approaches (and the resulting infrastructure cost) when the workload increases sharply in a few large steps. We see that, even for this relatively small deployment, cost-aware shows up to 12% reduced infrastructure cost for the same provisioned capacity.



(a) *large-jump* workload

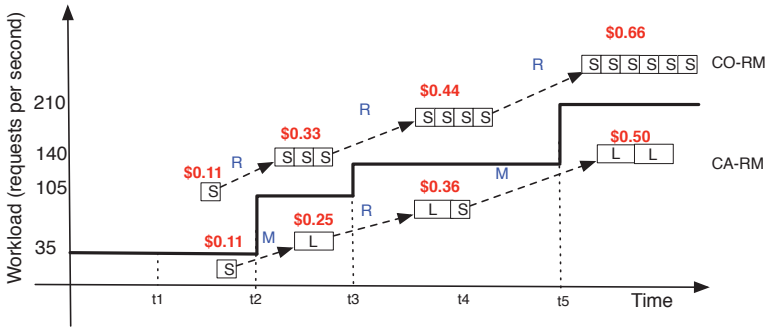


(b) *small-jump* workload

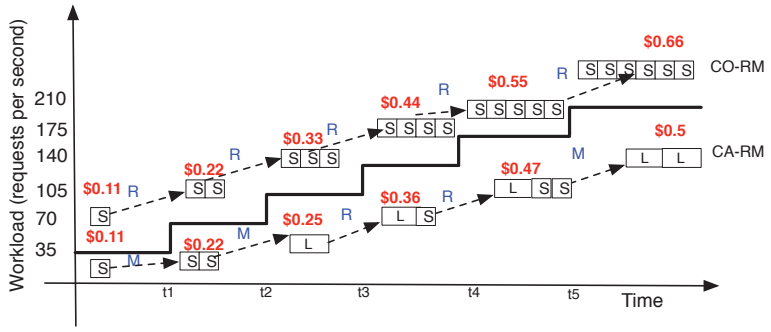
Figure 3.3: Cost-aware versus cost-oblivious provisioning

If the workload increases more steadily, as shown in Fig. 3.3b, both approaches choose *identical* configurations, i.e., an increasing number of small servers. With replication as the only elasticity mechanism, and slowly increasing workload, the cost-aware approach is not able to find opportunities for further cost improvement.

3.6.2 Benefits of adding Migration mechanism



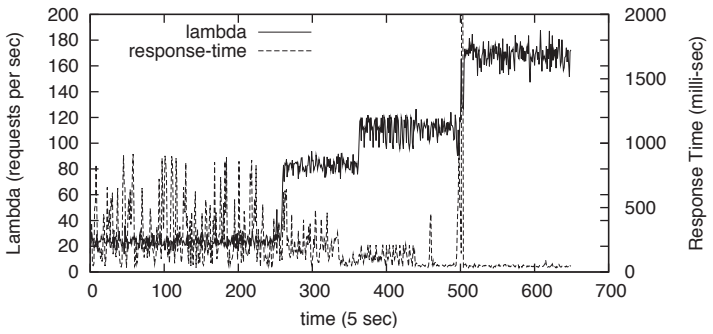
(a) *large-jump* workload



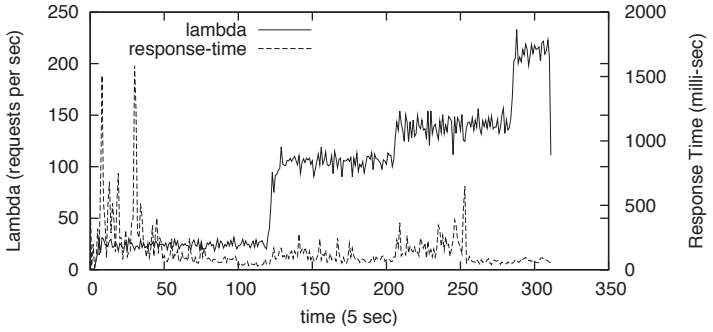
(b) *small-jump* workload

Figure 3.4: Benefits of using replication and migration in a unified provisioning approach.

We next consider the benefit of the cost-aware approach compared to cost-oblivious when migration is added as an additional elasticity mechanism to allow relocation of an application to a more cost-effective server configuration. By enabling both mechanisms to modify the deployment, our provisioning algorithms are able to consider a larger set of feasible configurations, which can yield higher savings in the infrastructure cost. Figure 3.4a compares the two approaches as the workload grows in large jumps. The cost-aware ap-



(a) Cost Oblivious



(b) Cost Aware

Figure 3.5: Application Performance during cost-aware and cost-oblivious provisioning for *large-jump* workload.

proach (CA-RM) shows a benefit as high as 24% over cost-oblivious (CO-RM), twice the relative benefit as with using replication-based elasticity alone. For the steadily growing workload, shown in Figure 3.4b, the cost-aware algorithm shows a similar benefit over cost-oblivious. Recall that with replication alone, the cost-aware approach produced an equivalent solution as cost-oblivious for the slowly increasing workload – in this case, by adding the migration mechanism, cost-aware provisioning is able to improve the infrastructure cost by 24%. Though the cost-oblivious approach uses both migration and replication as well, its choices are frequently more expensive than those of the cost-aware approach.

Figure 3.5 shows the changing request-rate applied to the TPC-W application, and the corresponding average response-time during the experiments. By leveraging migration elasticity, the CA-RM approach is able to be much more responsive to provisioning re-

quests. For example, for the first large increase in workload, CA-RM chose a migration while CO-RM selected 2 replications, hence cost-aware finished the task in 10 sec as opposed to 1000 sec for cost-oblivious. This is because *live-migration* copies only the RAM-image of the VM, which is an order of magnitude faster than copying the disk image in replication.

3.6.3 Transition cost-aware Provisioning

Our experiments thus far have focused on optimizing infrastructure cost and have ignored the overhead of transitioning the application deployment from one configuration to another. By making elasticity decisions based on the time overhead of various options, Kingfisher's transition cost-aware approach can quickly provision additional capacity in the cloud when the workload surges suddenly. However, by focusing on rapid reconfiguration, transition cost-aware provisioning may not produce the minimal infrastructure cost.

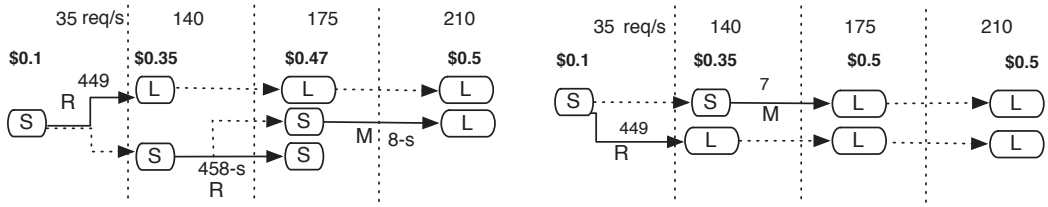
To demonstrate the benefits of our approach, we increased the TPC-W application workload in a series of large steps. At each step, we invoked Kingfisher's transition cost-aware provisioning and compared the decisions made by this approach with its infrastructure cost-aware provisioning method (i.e., which ignores the transition cost when making decisions). We assumed a cloud platform with two server types, small (S) and large (L), with infrastructure costs of $0.11 and $0.25 per hour, respectively (as in Table-3.1).

Figure 3.6 shows that the transition and infrastructure costs resulting from the chosen configuration after each workload step (i.e., from 35 req/s to 175 req/s). The transition cost-aware approach is able to pick lower transition time configurations, while the other approach opts for a lower infrastructure cost configuration but takes an order of magnitude more time. For example, when the workload increases from 140 to 175 req/s, the transition cost-oblivious approach performs a replication requiring 458s, while transition cost-aware opts for migration to a large server which requires 7 seconds, but results in a slightly higher infrastructure cost. Over the course of the experiment, the figure shows that transition cost-

oblivious chooses replication twice, while transition cost-aware replicates once, resulting in a much quicker response at the expense of some added infrastructure cost.

Figure 3.7(a) and (b) show the applied workload on the TPC-W application and the average response time as the workload increases. Between 150 and 200 seconds the workload increases to 175 req/s and, after a small spike in response time corresponding to the migration to a large server, the transition cost-aware solution settles to the target response time. In contrast, in (a) the transition cost-oblivious approach take significantly longer to reach the desired response time as the replication operation proceeds.

The experiment demonstrates that since copying memory state during live migration incurs lower latencies than copying disk images during replication live migration may be preferred, whenever feasible, to reduce transition costs. However, migration is not always feasible (e.g., if the application is already on the largest possible server) and replication may be needed in such cases.



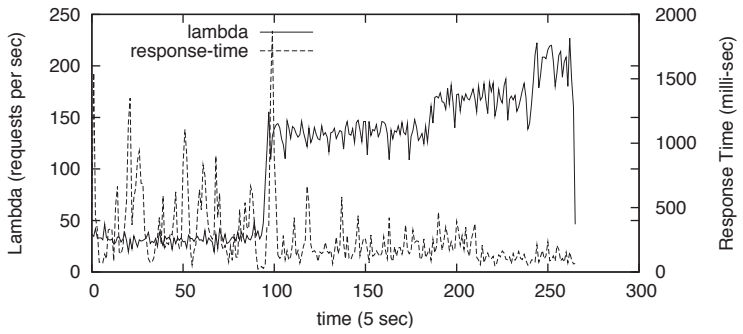
(a) Transition-cost oblivious

(b) Transition-cost aware

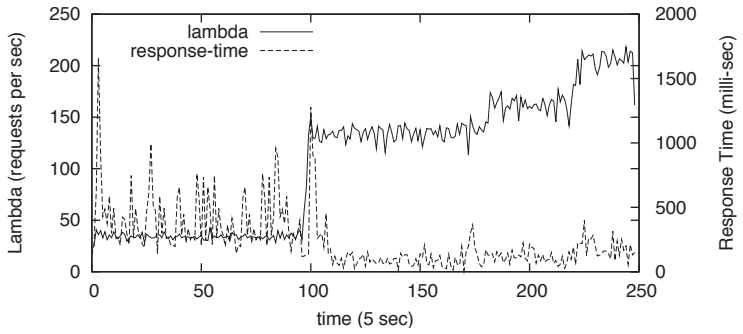
Figure 3.6: Comparison of a transition-cost aware system with a transition-cost oblivious system. Solid lines denote a configuration change, while dotted lines indicate no change.

3.6.4 Impact of the Pricing Model

Prior experiments have assumed a convex pricing model where the cost-per-core decreases as the number of cores increases. Since our ILP can handle arbitrary pricing models, we demonstrate how different pricing models can impact the choice of the configuration.



(a) Transition-cost Oblivious



(b) Transition-cost Aware

Figure 3.7: Workload and Response Times of a transition-cost aware system with a transition-cost oblivious system.

We consider the TPC-W application and wish to deploy it on a cloud platform with different initial capacities (workload, and corresponding required capacity, is increased from λ to 6λ). We assume that the cloud supports three types of servers, small, medium and large. For comparison, we also show the results of the cost-oblivious approach, which always chooses small servers regardless of the pricing model. First, we assume a convex pricing model, which resembles those employed in current clouds. In this case, larger servers have lower cost-per-core, causing our approach to prefer medium and large servers over small ones, when possible. Next, we employ concave pricing model, where the cost-per-core increases for larger systems. In this case, since the small server has the cheapest price per code, our cost-aware approach uses only small servers to provision capacity, effectively choosing the same configuration as cost-oblivious. Finally, we choose an arbitrary pricing

model, where the medium server is the cheapest, and the large server is the next cheapest on a cost-per-core basis. This causes our approach to prefer medium servers when possible and occasionally some large server instances.

Provisioning Algorithm	λ	3λ	4λ	5λ	6λ
Convex pricing model (S=0.11,M=0.15,L=0.25)					
Cost Aware	S	2M	M,L	2L	4M
Cost Oblivious	S	3S	4S	5S	6S
Concave pricing model (S=0.11,M=0.24,L=0.5)					
Cost Aware	S	3S	4S	5S	6S
Arbitrary pricing model (S=0.11,M=0.15,L=0.44)					
Cost Aware	S	2M	S,2M	2S,2M	4M

Table 3.2: Provisioning with different pricing models

3.7 Evaluation on Public Cloud: Amazon EC2

In this section we conduct our experimental evaluation on public cloud using Amazon EC2. We compare cost-aware with cost-oblivious elasticity methods for both infrastructure and transition costs. We need careful examination of the migration support in Amazon EC2 as the steps and the associated costs are quite different based on whether image provisioning is based on EBS or instance-store based [38]. We compare three different scenarios for transition cost-aware approach - these differ in the way the transition cost is accounted as well as the storage is used for image provisioning.

Note that Amazon EC2 supports eight EC2-instance types [38]. We have used 5 of these server-types of EC2, namely S, M, L, XL and XLC, each of which are profiled offline and the results are shown in Fig. 3.2b. EC2 allows creation of instances of each of these server types; these instances can be created either from instance-store or from EBS-volume snapshots, where an EBS-volume is a persistent storage. Amazon offers snapshoting capability on these EBS-volumes and these snapshots can be used to create new EC2-instances.

3.7.1 Determining Transition Costs in EC2

Kingfisher's transition-aware provisioning method needs to accurately account for the overheads of different replication / migration mechanisms available in EC2. We conducted a sequence of experiments to empirically determine these costs that we require for Kingfisher provisioning step.

We determine the transition costs for both EBS and instance-store based provisioning approach as the associated process and costs are quite different. EC2 provides two mechanisms from starting up a new new replica: (1) using an EBS-volume image (2) using the instance-store. Unlike private cloud, which supports live migration, the EC2 system supports only *shutdown-and-migrate* on EBS-volume based instances, while on instance-store based EC2-instances it only supports *replication*. Nevertheless, it is possible to simulate a migrate operation for instance-store based instances (i.e. those created using instance store) in two different ways. If the application does not maintain any state on its local disk (e.g., if the persistent state is stored on the S3 and on a separate EBS-volume, which is mounted on EC2-instance during instance-creation time), then we can emulate migration by starting a new instance on a larger server (via replication) and simply shutting down the old server and attaching the disk state to the new server (called *replicate-shutdown*). In contrast, if the state of the local disk needs to be migrated as well, then a *shutdown-copy-migrate* operation can be performed, where an application is shutdown, a machine image of its disk state is created and uploaded to S3, and a new replica is started with this image; on EBS-snapshot based instances, one can stop the instance and restart it as a different EC2-instance; we call this as *stop-and-start* operation.

In order to capture the cost of each of the provisioning operation, we break down the each operation into its component steps and capture the cost of each of the component steps. The *shutdown-copy-migrate* option, in a non-EBS volume instance involves following five steps 1.) copy the complete disk-image 2.) compress it 3.) uploading it onto S3 4.) register

it as an AMI[7] 5.) create an instance using this new AMI. Table 3.3(a) shows the time taken to complete each component steps for different size-images. Note that the total time is linearly varying with the size of compressed image. Similarly for EBS, there are three distinct steps. Table 3.3(b) depicts the time it takes to take a snapshot of volume which contains data which cannot be compressed any-further. The time to take the snapshot of an EBS volume can also be modeled as a linear function of size of compressed image size. As shown in 3.3c, the time it takes to boot an instance from EBS-snapshot is nearly constant– our measured average value is 85 sec The average instance registration time is 7 sec. The *replicate-shutdown* option incurs a similar overhead as that of a pure replicate operation. In our experiments we have used the time to be 800 sec (since our instance gets compressed to 3GB). Finally, the *stop-and-start* operation is estimated to have mean overhead of 65 sec.

Volume Size (GB)	Compressed Image (GB)	Snapshot	upload time(s)	boot time (s)
10	1.22	675	175	190
10	1.60	710	210	246.5
10	2.34	927	310	345
10	2.99	1160	314	407.1
10	3.08	1308	435	424
10	3.54	1466	490	494.3

(a) Time measurements of steps involved in shutdown-copy-migrate operation

Volume Size (GB)	Used Space	Compressed Image (GB)	Zone	Snapshot time
10	2	2	us-east-1a	491
10	4	4	us-east-1a	915
10	6	6	us-east-1a	2064
10	8	8	us-east-1b	2596

(b) Time Measurements of taking snapshot of an EBS volume

Volume Size (GB)	Used-up space	Zone	Startup Time (s)
10	5	us-east-1a	82.7
10	6	us-east-1a	84
10	7	us-east-1a	82
10	8	us-east-1b	85.7
10	9	us-east-1a	88

(c) Time measurements of start-up time of an image from EBS-volume

Policy	λ	2λ	3λ	6λ
CO-RM	4S(.34)	S,L (.425)	2L (.68)	3L,2S (1.19)
CA-RM	4S (.34)	2M (.34)	S,2M (.425)	4M,S (.765)
TA-RM-1	4S (.34)	2S,M (.34)	S,2M (.425)	XL,L,M (1.19)
TA-RM-2	4S (.34)	2S,M (.34)	S,2M (.425)	S,4M (0.765)
TA-RM-3	4S (.34)	2S,M (.34)	S,2M (.425)	3M,L (0.85)

(d) Provisioning with different methods ($\lambda = 35$). Choice of provisioning mechanism for each transition, i.e. from $\lambda \rightarrow 2\lambda, 2\lambda \rightarrow 3\lambda$ and $3\lambda \rightarrow 6\lambda$, are described in section 3.7.3

Table 3.3: Measurements and Provisioning on EC2

[7]An Amazon Machine Image (AMI) is a virtual machine image which is used by EC2 to create server instances

3.7.2 Infrastructure-cost aware Provisioning

To evaluate the efficacy of Kingfisher in taking infrastructure and transition costs into account, we repeated our TPC-W experiment on the public EC2 cloud. We assume an initial configuration of four small servers serving an initial workload of $\lambda = 35$. The infrastructure cost of servers is summarized in Table 3.1 and transition cost is discussed above. Like before we varied the workload in steps and Table 3.3(d) depicts the configurations generated by the cost-oblivious and Kingfisher's cost-aware methods. The cost-aware (CA-RM) method is able to provision the same capacities at 35% lower cost.

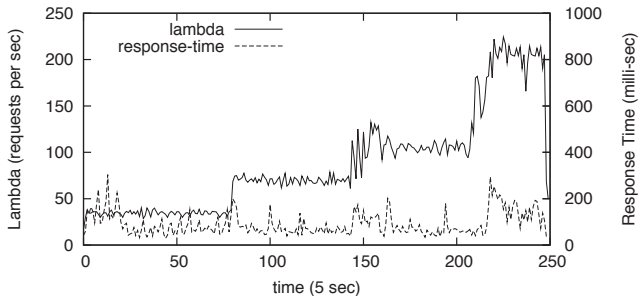
3.7.3 Transition-cost aware Provisioning

Using the empirically determined transition costs, we next evaluate transition cost-aware elastic provisioning. We consider three transition cost scenarios based on usage pattern and constraints in EC2: (i) TA-RM-1, which only takes into account the number of transitions and cost of each transition and also the infrastructure cost of final configuration; (ii) TA-RM-2: that considers transition costs and final infrastructure costs for non-EBS instances in EC2, and (iii) TA-RM-3 that distinguishes between 32-bit small EC2 instances, and 64-bit larger EC2 instances, and assumes that 32-bit and 64-bit applications are not mixed across the corresponding server types.

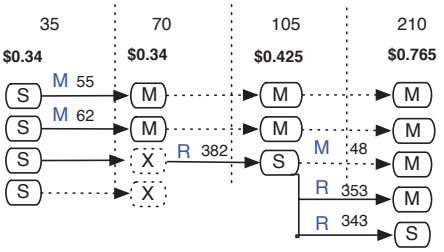
As shown in Table 3.3(d), when workload jumps to 2λ, TA-RM-* chooses to perform only one *stop-and-start* operation as opposed to two chosen by CA-RM; notice that both configurations have the same dollar cost however CA-RM policy tries to maximize capacity, while TA-RM-* schemes minimize the number of reconfigurations. When the workload increases from 2λ to 3λ, the CA-RM method resorts to *replication*, while the TA-RM-* chooses the faster *stop-and-start* provisioning. In the final step, CA-RM chooses to perform two replications, however, TA-RM-1 initiates two *stop-and-start* operations for faster provisioning. Since TA-RM-2 provisions non-EBS instances, it chose the faster *replication*

option (over the slower *shutdown-copy-migrate*). TA-RM-3, on the other-hand, performs a *stop-and-start* from S to M instances and then initiates another *replication*.

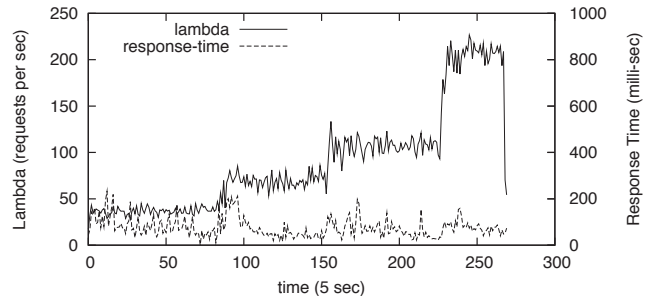
Figure 3.8 show the result of provisioning experiment conducted using Kingfisher for TA-RM-3 scheme. Figure 3.8a and Figure 3.8c show the response-times of the of the corresponding configurations, indicating the responsiveness of the system using the end-to-end response time of the configuration under workload. The benefit of transition-cost aware approach is apparent from Figure 3.8b,3.8d: in the first and last step it approximately takes the same time[8], however in the second jump the transition-cost aware system achieves the new configuration in 60 sec as opposed to 382 sec.



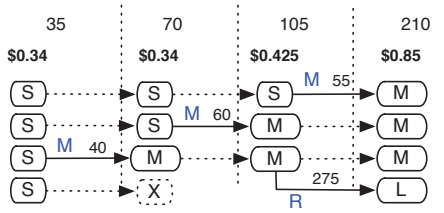
(a) Transition-cost oblivious



(b) Transition cost oblivious provisioning



(c) Transition-cost aware



(d) Transition cost-aware provisioning

Figure 3.8: Comparison of a transition-cost aware system with a transition-cost oblivious system.

[8]The large variation in similar operations is because the copy operation is dependent on the load on the backend network and disk systems of EC2

3.8 Related Work

This chapter focuses on optimizing the use of elasticity mechanisms and is applicable in commercial cloud service offerings (exemplified by Amazon EC2 and others) and cluster management systems such as OpenNebula or Eucalyptus. In particular, this study is the first work to propose cost-aware provisioning in a cloud, along with algorithms to optimize how additional mechanisms beyond replication can be leveraged to support elasticity.

There is a significant amount of related work, however, in the area of dynamic capacity provisioning in data centers, grids, or compute clusters, starting with earlier work such as [39] and [20]. Much of this work is platform-centric, while our work considers a customer-centric view of provisioning and resource optimization. Other work has considered migration as a means of dynamic provisioning [42], while we consider replication with different types of migrations and assign cost to each of them. There is also an extensive body of work on dynamic provisioning of web applications using analytic models [104, 99, 122, 126]. Classical feedback control theory has also been used to model the bottleneck tier for providing performance guarantees for web applications [1, 107]. The approach in [107] formulates the application tier server provisioning as a profit maximization problem and models application servers as M/G/1/PS queueing systems. The work in [103] provides a model-driven approach for adapting resources for a multi-tier application. Finally, machine learning techniques have also been used for provisioning, such as the k-nearest neighbor approach to provision the database tier [21].

In contrast to these efforts, our work automates the process of characterizing the workload mix and uses empirical models as a basis for provisioning system capacity. Further, while I employ analytic models of infrastructure and transition costs, my approach involves full prototype implementation and experiments on an actual Linux cluster.

3.9 Concluding Remarks

Since today's cloud platforms offer a plethora of different server configurations for rent and price them differently on a cost-per-core basis, we argued that these pricing differentials can be exploited by an application provider to minimize the infrastructure cost of provisioning a certain capacity. We proposed a new cost-aware provisioning approach for cloud applications that can optimize either the infrastructure cost for provisioning a certain capacity or the transition cost of reconfiguring an application's current capacity. Our approach exploits both replication and migration to dynamically provision capacity and uses an integer linear program formulation to optimize cost. We prototyped a cloud provisioning engine, using OpenNebula, that implements our approach and evaluated its efficacy on a laboratory-based Xen cloud. Our experiments demonstrated the cost benefits of our approach over prior cost-oblivious approaches and the benefits of unifying both replication and migration-based provisioning into a single approach. We also presented a case study of how our approach can be employed in a public cloud such as Amazon EC2. In future we plan to extend kingfisher by integrating it with systems which employ queuing theory based model for capacity estimation for provisioning on cloud.

CHAPTER 4

ELASTIC PROVISIONING OF MULTI-TIER CLOUD APPLICATIONS USING STATISTICAL BOUNDS ON SOJOURN TIME

Web applications use a tiered architecture to afford dynamic scaling of capacity according to the workload. How tiers are provisioned is not only critical for providing a compelling user experience but is also critical for provider's profit margin. It becomes even more significant in a cloud kind of environment, which adopts a usage based pricing model. Traditional provisioning algorithms either provision for average response time or assume knowledge of per tier response-time and provision for the same. Tier wise response times cannot be predicted in advance. On the other hand average response time is not a very useful QoS metric as service providers are often more interested in the percentile bound on end to end response time. In the previous chapter we addressed the problem of cost aware elastic provisioning of an application on a cloud platform, when the application SLA is expressed on the average response time. In this chapter we present a simple and effective approach for resource provisioning to achieve a percentile bound on the end to end response time of a multi-tier application. We first model the multi-tier application as an open tandem network of M/G/1-PS queues and develop a method that produces a near optimal application configuration, i.e, number of servers at each tier, to meet the percentile bound in a homogeneous server environment – using a single type of server. We then extend our solution to a K-server case and our technique demonstrates a good accuracy, independent of the variability of service-times.

4.1 Introduction

Enterprise applications are known to observe dynamic workload and provisioning correct capacity for these applications remains an important and challenging problem. High workload variability is caused by a variety of reasons, such as flash crowds, short term sustained surges, or long-term fluctuations based on change in business or underlying IT infrastructure etc. Predicting these workload fluctuations or the peak workload is challenging. Erroneous predictions often lead to under-utilized systems or in some situations cause temporarily outage of an otherwise well provisioned web-site; e.g. in November 2000 Amazon.com site suffered a forty-minute outage due to overload. Consequently, rather than provisioning server capacity to handle infrequent (and hard to predict) peak workloads, an alternate approach of dynamically provisioning capacity on-the-fly in response to workload fluctuations has become popular. Dynamic provisioning is especially well suited to the cloud due to the ability of cloud platforms to provision capacity when needed and charge for usage on pay-per-use basis.

Numerous efforts that have addressed the issue of dynamic provisioning of server capacity to distributed applications [36, 66, 104, 103] . These efforts fall into two categories - *proactive*, where a model of the application is used to compute the capacity needed to service a particular workload at a certain performance level and *reactive*, where additional capacity is allocated *after* a workload spike arrives and causes significant performance degradation.

In the case of proactive approaches, application models have been derived to predict how much capacity is needed to provide a certain *mean* response time for a given workload [103, 104]. However, typical service level agreement (SLAs) for the application are specified in terms of the worst case (or peak) response times [32] (e.g. 99% of the requests should see no more than a 1-sec response time). Consequently, there is a mismatch between the provisioning models which allocate capacity for a target mean response, time and the

SLA, which dictates that the capacity should be allocated based on a high percentile (peak) response time.

Second, many enterprise applications possess a multi-tier architecture. Typically SLAs are specified on an end-to-end basis for the entire application. The few provisioning efforts that focus on allocating capacity for the tail of the work translate the end-to-end SLA to a per-tier one [104]; provisioning for per-tier SLA can result in large errors in provisioning the capacity if the tier response time estimates are incorrect.

Third, most provisioning techniques to-date are cost oblivious – they determine *how much* server capacity to allocate but do not consider the *cost* of allocating the server capacity. In a cloud platform, different server configurations are available at different prices. Server capacity does not scale linearly across configurations and nor does the price. Since multiple combinations of servers can provision a certain capacity C for an application, a cloud specific provisioning scheme must take the cloud costs into account when making provisioning decisions.

In this chapter I present a new model driven provisioning approach targeted to cloud platforms. The approach focuses on i.) allocating capacity based on peak (high percentile) of the workload, ii) takes a holistic view of the entire multi-tier application by considering bounds on on end to end response times while making provisioning decisions and iii) takes cloud server configs and pricing models when determining the most cost effective config to provision a certain amount of capacity.

4.1.1 Research Contributions

This chapter makes the following contributions:

- **Cost aware provisioning subject to a percentile response time SLA.** I present an algorithm for resource provisioning for a multi-tier cloud application, subject to an SLA expressed in terms of high percentile of end to end response time, that mini-

mizes the total cost of compute resources required by the application. The formulation models the application as an open tandem queue network of M/G/1-PS queues.

- **Service time and response-time approximations.** I present an approximation of the response time distribution of the M/G/1 processor sharing queue based on the distribution of conditional expected response times given the service times and show it to be accurate for the purposes. In addition, I present a new service time characterization based on a mixture of shifted exponential distributions.

- **Cost-efficient configuration with heterogeneous servers subject to percentile SLA.** I extend the above approach to account for the presence of multiple types of servers with different costs and computational capabilities. This is achieved by formulating an integer optimization problem with the constraint that per-tier capacity should be at least as much as that computed by the queueing theoretic model.

- **Prototype implementation and experimentation.** I have implemented an analytical model in MATLAB and tested it using a multi-tier application, i.e. java implementation of TPC-W, over a private cloud. For comparison, we also implemented a baseline case using M/M/K-FCFS queues. The experimental results show that the approach is able to provision the application to meet the SLA specified on 99 percentile of end-to-end response time with less than 3% provisioning error, while the baseline techniques provisioned with an error as large as 140%. In the case of heterogeneous provisioning, the approach shows, as high as, 81% savings in server cost as compared to that of the corresponding optimal homogeneous configuration. In case of private cloud experiments we found that heterogeneous approach showed around 11% cost saving (using Amazon EC2 pricing) over homogenous configurations.

4.2 Background and Problem Formulation

In this section, we present the system model and a high level problem description. We describe the SLA performance metric, and thereafter formulate the provisioning problem that we address in this work.

4.2.1 Multi-tier Application

Modern large scale web applications are developed as multiple tiers for reasons pertaining to scalability. A multi-tier architecture offers flexibility for development as well as deployment of applications. Each application tier, typically, provides a specific functionality and the various tiers form a processing pipeline. In a typical multi-tier application various tiers participate in the processing of an incoming request; each of the participating tier receives partially processed requests from the previous tier and feeds these requests into the next tier after local processing (see Figure 4.1). The tiers are replicated to scale according to the processing demand; a load balancer is used to distribute the load over all replicas of such a tier. Figure 4.1 depicts a two-tier application where both tiers are replicated. This is a commonly employed architecture by e-commerce web applications where, both, web-server and database tiers are clustered to scale up according to increase in the incoming workload.

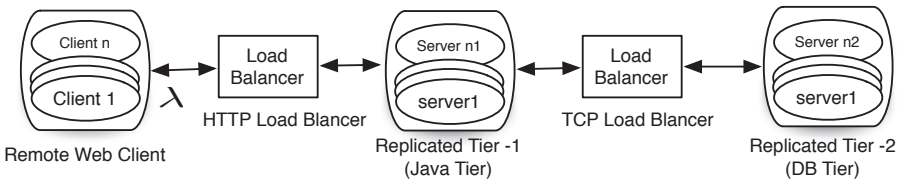


Figure 4.1: Topological configuration of a typical replicated two-tier web application

We assume that each tier is placed on a dedicated server and that replicating a tier essentially means replicating the server. Each clustered tier is also assumed to employ a protocol-session aware load balancer responsible for distributing requests to replicas in that tier. We assume that the each tier's capacity (number of servers), can be varied dy-

namically without disturbing the application's normal functioning, and that each tier can be independently provisioned for capacity.

4.2.2 Cloud Platforms

Cloud computing has emerged as a new IT delivery model. The Infrastructure as a Service (IaaS) cloud-model is being seriously evaluated by enterprises to deploy their web applications that support dynamic capacity resizing. In this model, an organization/client can rent remote compute and storage resources to host networked applications and resources can be dynamically added or removed on an as-needed basis. We consider a cloud computing platform that allows compute servers to run hosted applications. We assume that the platform offers N heterogeneous server configurations for rent, each with a different rental-cost and configuration.

We assume that the cloud platform has an infinite pool of servers and that servers can be provisioned by invoking server-instance creation APIs; servers may be requested and terminated at any time and billing is based on the amount of time for which each server is used (e.g., based on the number of hours for which each server is used). We also assume that the cloud platform employs virtualization—each physical server is assumed to run a hypervisor that controls the allocation of physical resources on the machine and offers performance isolation to each of its virtual servers.

4.2.3 Problem Formulation

Let N and M denote the number of tiers and server-types respectively. Let tier j be jointly served by $\sum_{i=1}^{M} n_{ij}$ servers, where n_{ij} denotes the number of servers of type i present at tier j. Let $\overline{n}_j = [n_{1j}, n_{2j}, \ldots n_{Mj}]$ be a vector representing the server configuration of tier j and $\overline{p} = [p_1, p_2, \ldots p_M]$, where p_k denotes the cost of a server of type k. Let T be the end-to-end response time of requests to the multi-tier application and $F_T(t)$ be its CDF, i.e.

$F_T(t) = P(T \leq t)$. Then for a given percentile bound θ, and response-time threshold T_D, the cost minimization problem becomes:

$$\text{minimize} \sum_{j=1}^{N} \sum_{i=1}^{M} n_{ij} p_i, \tag{4.1}$$

subject to the constraint

$$F_T(T_D) \geq \theta. \tag{4.2}$$

It should be noted that F_T, also depends on n_{ij}, since n_{ij} specifies the application configuration that determines the end-to-end response time of the application. In the next section we present a model of a multi-tier application which enables us to capture the effect of n_{ij} on F_T.

4.3 Application Model

In this section we model the multi-tier application as a network of queues. Our first model of multi-tier application is a chain of tiers where each tier is modeled as single M/G/1-PS queue (see Figure 4.2). Each tier carries out a specific function, for instance, a web-application server or a database server etc. In this work we assume single customer class.

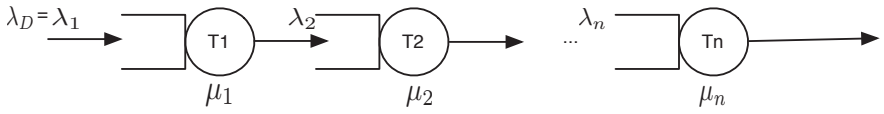


Figure 4.2: Multi-tier application model

Let A_i denote the i^{th} tier of the application, λ_i the average arrival rate of incoming requests at the i^{th} tier, and μ_i the average service rate $\forall i = 1 \ldots N$. We define the total response time of a request as the time between when it enters the first tier and the time when it leaves the last tier. Note that different λ_i for each tier handles the case where one tier issues multiple requests to the lower tier.

Let T_j be a random variable representing the response time for tier j, then the end-to-end response time of a request is

$$T = \sum_{j=1}^{N} T_j. \tag{4.3}$$

Let $f_T(t)$ be the probability density function (PDF) of the response time T and $L_T(s) = \mathcal{L}(f_T(t))$ be the Laplace transform of the PDF of response time T then

$$L_T(s) = \prod_{j=1}^{N} L_{T_j}(s), \tag{4.4}$$

where $L_{T_j}(s)$ is the Laplace transform of the PDF of T_j. Thus the PDF of end-to-end response time, $f_T(t)$, can be computed by taking the Laplace inverse of (4.4)

$$f_T(t) = \mathcal{L}^{-1}\left(\prod_{j=1}^{N} L_{T_j}(s)\right). \tag{4.5}$$

To solve (4.5) we require the PDF of the random variable T_j. Unfortunately there are no exact formulas for response time distributions of an M/G/1-PS queue. We, therefore, present an approximation for the same in the next section.

4.4 Estimating End-to-end Response Times

In this section we describe our approach to estimate the PDF of end-to-end response time of a chain of M/G/1-PS queues. In order to do that we estimate the PDF of response time of a single M/G/1-PS queue and then leverage (4.5) to compute the end to end response time.

Section 4.4.1 describes our method of approximating the response time distribution of a M/G/1-PS queue. The result depends of the definition of the PDF of service-time distribution of the queue and we describe a mechanism to approximate the same for any real-life system in section 4.4.2. Section 4.4.3 provides a closed form equation of the end-to-end response time of the chain of queues.

4.4.1 Approximate Response Time Distribution

The exact form of the response time distribution for the M/G/1-PS is not generally known [120]. Thus we approximate it with the expected conditional response time distribution as described below. Let T denote the job response time, and X its service time; then the expected conditional response time, conditioned on the service time being x is

$$\tau = E[T|X = x] = \frac{x}{1-\rho}, \tag{4.6}$$

where $\rho = \lambda/\mu$ is the average load.

We approximate T by τ. Since τ is a function of X,

$$F_\tau(t) = P[\tau \leq t] = P[\tfrac{X}{1-\rho} \leq t] = P[X \leq t(1-\rho)],$$
$$F_T(t) \approx F_\tau(t) = F_X(t(1-\rho)), \tag{4.7}$$

It has been observed in real-life systems that job service time distributions exhibit heavy tailed behavior [30]. Heavy tailed distributions have very high variance; high variance in service time distribution of jobs makes it a dominant factor in determining the behavior of response time distribution. Approximation proposed in (4.7) captures the variability of service time and will be particularly useful in such situations. We discuss the impact of variability of service time in section 4.7 and demonstrate that our approach shows significant improvement.

4.4.2 Approximate Service Time Distribution

In real systems, like computer clusters and web servers, there is a strong evidence that job service times are highly variable [30]. Some heavy tailed distributions do not have a closed-form Laplace transforms, e.g., the Pareto distribution, while those possessing convenient Laplace transforms might lead to an intractable complex function after undergoing an N^{th} order convolution in (4.4). We, thus, need a distribution function, which can closely

approximate a service time distribution observed by a real world application and leads to an easily invertible Laplace transform even after undergoing higher order convolutions. In this section we describe such a distribution function and also present an algorithm to approximate the service time distribution from the service-time histogram; service time histograms can be easily collected from the server either through logs or through off-line profiling.

We express the service time distribution as a mixture of K shifted exponentials, as shown in (4.8). The motivation behind this is two fold: i.) the web application workload is a mix of different job types [67, 24]. Capturing the service time distribution as sum of shifted exponentials, essentially, means that job-size of each job-type is exponentially distributed but each job-type has a different mean job-size. ii.) The formulation leads to a Laplace transform that is easy to invert.

Formally, we want to fit a mixture of shifted exponentials,

$$f_X(x) = \sum_{k=1}^{K} \alpha_k \mathbf{1}\{x \geq t_k\} \mu_k e^{-\mu_k(x-t_k)}, \quad x \geq 0 \tag{4.8}$$

to data $x_1, x_2, \ldots, x_n$, where $\mathbf{1}\{P\}$ is one if predicate P is true and zero otherwise. This involves inferring the number of shifted exponentials, K, the shifts of each exponential, $\{t_k\}$, the mix proportion of the shifted exponential, $\{\alpha_k\}$, and their average rates $\{\mu_k\}$ from the data. Let us begin by assuming that K and $t_1, \ldots, t_K$ are already known. In other words we want to find the best fit for $\{\mu_k\}$ and $\{\alpha_k\}$; we perform maximum likelihood estimation using the expectation-maximization algorithm (EM).

4.4.2.1 EM algorithm for estimating mixture parameters

Suppose we know which shifted exponential distribution each observation x_i belongs to, in other words suppose we have $y_i \in \{1, \ldots, K\}$ available to us where $y_i \in \{1 \ldots K\}$ represents the particular shifted exponential distribution. Then the parameter values that maximize the log likelihood function can be computed as:

$$\alpha_k = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{y_i = k\}/n, \quad k = 1, \ldots, K \tag{4.9}$$

$$1/\mu_k = \frac{\sum_{i=1}^{n} \mathbf{1}\{y_i = k\}x_i}{\sum_{i=1}^{n} \mathbf{1}\{y_i = k\}}, \quad k = 1, \ldots, K \tag{4.10}$$

EM is an iterative algorithm that infers y_i as needed. Suppose μ_k^j and α_k^j are the estimates at the end of the j-th iteration. The next iteration consists of an expectation step followed by a maximization step as given below.

Expectation. Let $y_{i,k}$ denote the probability (expectation) that sample x_i belongs to the k-th shifted exponential. It is given as

$$\begin{aligned} y_{i,k} &= P[Y_i = k | X = x_i] \\ &= \frac{\alpha_k^j \mathbf{1}\{x_i \geq t_k^j\}\mu_k e^{-\mu_k^j(x_i - t_k^j)}}{\sum_{l=1}^{K}(\alpha_l^j \mathbf{1}\{x_i \geq t_l^j\}\mu_l e^{-\mu_l^j(x_i - t_l^j)})} \end{aligned} \tag{4.11}$$

$\forall i = i, \ldots, n$ and $k = 1, \ldots K$. Note that $y_{i,k} = 0$ when $x_i < t_k^j$.

Maximization. Having computed $y_{i,k}$, we now update our estimates of α_k and μ_k. This is done by using modified versions of (4.9) and (4.10).

$$\alpha_k^{j+1} = \frac{1}{n}\sum_{i=1}^{n} y_{i,k}, \quad k = 1, \ldots, K \tag{4.12}$$

$$1/\mu_k^{j+1} = \frac{\sum_{i=1}^{n} y_{i,k}x_i}{\sum_{i=1}^{n} y_{i,k}}, \quad k = 1, \ldots, K \tag{4.13}$$

This is referred to as the maximization step because the above estimates maximize the likelihood given the current values of $\{y_{i,k}\}$.

These steps are repeated until the parameters converge; $\{\alpha_k^0\}$ and $\{\mu_k^0\}$ are the initial values, which can be computed as mentioned in the section below.

4.4.2.2 Algorithm for approximating service-time distribution

We use an iterative approach to determine the best number of exponentials K, and then determine t_k, μ_k^0, and α_k^0, to initialize the EM algorithm, (4.11), (4.12) and (4.13).

The basic idea underlying the algorithm, as outlined in as mentioned in [67], is to iteratively run a *k-means* clustering algorithm for every value of $k = 1 \ldots K_{max}$ and compute the following three metrics[1]: coefficient of variation[2] of intra-cluster distance (C_{intra}), coefficient of variation of inter-cluster distance (C_{inter}), and ratio of intra-cluster to inter-cluster coefficient of variation (β_{cv}). The value of β_{cv} drops as number of clusters increase and will be minimum (i.e. zero) when number of clusters is equal to the total number of points. We find that K, where the rate of decrease of β_{cv} falls below a threshold (or the slope goes above a negative threshold value).

Having computed K, and the cluster centers e_k, we compute initial estimates of the mean service rate $\{\mu_k^0\}$ and mixture fraction (α_k^0) as follows:

$$\mu_k^0 = \frac{1}{e_k - t_k},\ \alpha_k^0 = \frac{\text{number of points in cluster}}{\text{total number of points}}. \tag{4.14}$$

We set the shifts to be equidistant from from two neighboring cluster centers, i.e., $t_i = (\mu_{i-1} + \mu_i)/(2\mu_{i-1}\mu_i), \forall i = 2 \ldots K$. However, $t_1 = 0$, i.e., the shift for the first exponential is zero (details of the algorithm can be found in [90].

4.4.3 Approximate Application Response Time Distribution

The PDF of the end to end response time of N-tier application is obtained using (4.8) and (4.7) in (4.5) as

$$f_\tau(t) = \mathcal{L}^{-1}\left(\prod_{j=1}^{N}\sum_{k=1}^{K_j}\frac{\alpha_{jk}\mu'_{jk}e^{-st'_j}}{(s+\mu'_{jk})}\right), \tag{4.15}$$

where for each tier $j = 1, \ldots, N$, service times are modeled as mixtures of K_j shifted exponentials and their density functions are expressed using (4.8); we rewrite the result for the j^{th} tier for the sake of completeness:

---

[1] the metrics are computed as mentioned in [67]

[2] Coefficient of variation or variation coefficient is defined as a ratio of the standard deviation to the mean, i.e. $C_v = \sigma/\mu$;

$$f_{X_j}(x) = \sum_{k=1}^{K_j} \alpha_{jk} \mathbf{1}\{x \geq t_{jk}\} \mu_{jk} e^{-\mu_{jk}(x - t_{jk})}. \tag{4.16}$$

After inverting (4.15), the final expression of $f_\tau(t)$ takes the following form:

$$f_\tau(t) = \sum_{i_1=1}^{K_1} \cdots \sum_{i_N=1}^{K_N} \Big(\mathbf{1}\{t \geq t'\} \prod_{j=1}^{N} \alpha_{ji_j} \mu'_{ji_j} \times \sum_{l=1}^{N} r_l e^{-\mu'_{li_l}(t-t')} \Big), \tag{4.17}$$

where $\mu'_{ji_j} = \mu_{ji_j}(1 - \rho_j)$, $t' = \sum_{j=1}^{N} t_{j,i_j}/(1 - \rho_j)$, and $r_l = 1/\left(\prod_{k \neq l}^{N} (\mu'_{ki_k} - \mu'_{li_l}) \right)$.

Note that α_{ji_j} and μ_{ji_j} are the parameters of the k^{th} shifted exponential of the j^{th}-tier (as shown in (4.16)); ρ_j is the average utilization of the j^{th} tier, and r_j is the j^{th} residue, where $j = 1, \ldots, N$.

Note that the expression in (4.15) does not involve higher order poles[3] because none of the rates μ_{li_l} is ever equals any of the μ_{ji_j}. This becomes especially helpful in inverting the Laplace transform as absence of higher order terms in denominator leads to a simple computation of partial fractions.

The final expression of $f_\tau(t)$ in (4.17) is, essentially, a product of sums of the shifted exponentials, which is easily readable in (4.15). This means that the $f_\tau(t)$ will be expressed, in total, by $\prod_{j=1}^{N} K_j$ terms; for example let $K_j = a, \forall j = 1 \ldots N$, then $f_\tau(t)$ will be expressed as a sum of a^N terms. It is easy to see that number of terms grow exponentially with number of tiers. Fortunately, real life systems do not have more than three or at most four tiers and thus $f_\tau(t)$ is easily computable.

---

[3]If for some l, j, $\mu_{li_l} = \mu_{ji_j}$, we slightly perturb the starting $\mu^0_{li_l}$ for tier-l by adding a small random number and re-run the EM algorithm for that tier-l

4.5 Finding Near optimal Homogeneous Configuration

In this section we present a solution to the the resource optimization problem, as expressed by (4.1) and (4.2), but with only one type of server, $M = 1$ (homogeneous setting).

We substitute the approximate response time of an M/G/1-PS queue, i.e. $f_\tau(t)$ as shown in (4.17), in (4.2) to obtain:

$$F_\tau(T_D) = \sum_{i_1=1}^{K_1} \cdots \sum_{i_N=1}^{K_N} \left(\mathbf{1}\{T_D \geq t'\} \prod_{j=1}^{N} \alpha_{ji_j} \mu'_{ji_j} \sum_{l=1}^{N} \frac{r_l(1-e^{-\mu'_{li_l}(T_D-t')})}{\mu'_{li_l}} \right) \geq \theta, \tag{4.18}$$

where μ'_{jk_j} and r_j are the same as in (4.17) while $t' = \sum_{j=1}^{N} t_{j,i_j}/(1-\rho_j)$.

Thus the problem of minimizing (4.1) reduces to the problem of maximizing ρ_j ($\forall j = 1, \ldots, N$) such that $F_\tau(T_D) \geq \theta$, where $F_\tau(T_D)$ is given by (4.18). As this is an N-dimensional non-linear maximization problem, it is not easy to solve. However, the problem complexity is significantly reduced by assuming same utilization at each tier[4], i.e.,

$$\rho_1 = \rho_2 = \ldots = \rho_N = \rho.$$

It should be noted that it is desirable to have a balanced utilization at each tier in real-life systems. In practice, administrators often use a rule of thumb to bound the max utilization of servers of all tiers to avoid performance problems and outages [80].

Consequently, (4.18) reduces to an inequality in a single variable, namely ρ.

$$F_\tau(T_D) = \sum_{i_1=1}^{K_1} \cdots \sum_{i_N=1}^{K_N} \left(\mathbf{1}\{T_D \geq t'\} \prod_{j=1}^{N} \alpha_{ji_j} \mu_{ji_j} \sum_{l=1}^{N} \frac{r'_l(1-e^{-\mu'_{li_l}(T_D-t')})}{\mu_{li_l}} \right) \geq \theta, \tag{4.19}$$

[4]The constraint reduces the solution search space and thus the final solution is not guaranteed to be an optimal solution as it could result into a slightly over-provisioned system.

where, $t' = \sum_{j=1}^{N} t_{j,i_j}/(1-\rho)$, and $r'_l = 1/\prod_{k\neq n}^{N}(\mu_{ki_k} - \mu_{li_l})$. We solve for the maximum value of ρ, say ρ^*, by numerically solving (4.19) as an equality.

4.5.1 Computing the Application Configuration

In practice, large scale applications have each of their tier replicated for scalability as depicted in Figure 4.3. The idea is to be able to handle increasing number of requests while conforming to the SLA. In an ideal situation an application-tier's ability to process the number of requests increases linearly with number of its replicas, which means that if an application or application-tier with a single replica had a service rate of μ then K replica version of application-tier will have a request rate of $K\mu$. We have assumed a linear scaling in this work but that is not a limitation and any kind of scaling function can used in the technique to obtain the number of replicas at each application-tier. We have used replicas and servers interchangeably because we have assumed dedicated hosting model.

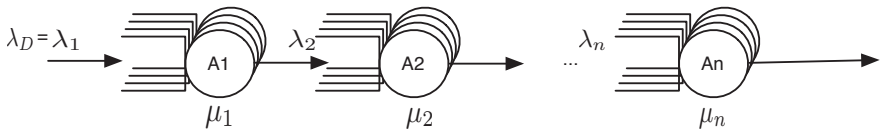


Figure 4.3: Multi-tier application model

We use ρ^* to compute the number of servers at each tier, i.e. n_{ij}. In the homogenous setup $i = 1$ and thus we solve for $n_j, j = 1 \ldots N$. Let λ_j and μ_j be arrival and service rates respectively, at tier j then $n_j = \lceil \lambda_j/(\zeta\rho^*\mu_j) \rceil$, where ζ is the scale factor, which can be chosen heuristically and

$$\mu_j = \sum_{i=1}^{K_j} \alpha_{ji} \frac{(1+\mu_{ji}t_{ji})e^{-\mu_{ji}t_{ji}}}{\mu_{ji}}. \tag{4.20}$$

The pseudo code of the algorithm for finding the application configuration in homogenous setup is outlined in [90].

4.6 Cost Efficient Heterogenous Configuration

We extend the solution approach described in Section 4.5 to be able to generate a cost efficient configuration in a heterogenous setting.

The basic idea underlying our approach is to greedily search for a low cost configuration which has a high utilization. At a high level the algorithm is iterative involving the following three steps at each iteration: 1.) creating a single hybrid-server from a given hybrid-configuration for each tier, 2.) solve the homogeneous configuration problem for the hybrid-server, 3.) translate the solution for hybrid-server into a heterogenous configuration, and the iterations are used to search for new hybird-configuration with lower cost and higher utilization. Figure 4.4 shows the block diagrammatic representation of the cost effective heterogeneous configuration algorithm.

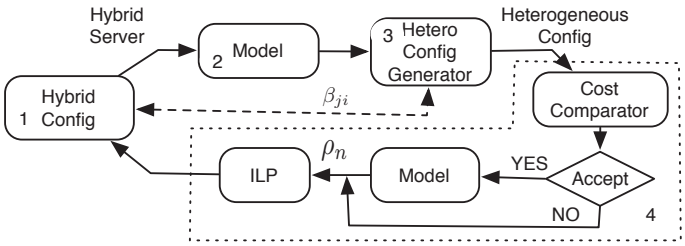


Figure 4.4: Functional block diagram of heterogeneous configuration algorithm

4.6.1 Hybrid server

Inorder to reuse our methodology for finding the near optimal number of servers in homogeneous setting, it is imperative that we approximate each hybrid configuration at each tier by a single server; we call it a hybrid-server. We construct the service time distribution of the hybid-server for each tier as a proportional mixture of the service time distributions of the servers involved in the heterogeneous configuration. Let $\overline{n} = \{n_i\}$ denote the hybrid-configuration where n_i, $i = 1, \ldots, M$, is the number of servers of type i. Then the hybrid-server's service-time distribution function for tier-j is expressed as

$$f'_j = \sum_{m=1}^{M} \beta_{jm} f_{jm}, \tag{4.21}$$

where f_{jm} is the service-time probability density function (PDF) of the m^{th}-server-type at j^{th}-tier and f'_j is the PDF of the hybrid-server for tier-j; β_{jm} is the mixing proportion of the component server m for tier-j and is computed using the formula

$$\beta_{jm} = \frac{n_m \mu_{jm}}{\sum_{j=1}^{M} n_m \mu_{jm}}. \tag{4.22}$$

We explain our procedure of creating a hybrid-server with the following example: suppose we have two servers, say s_1 and s_2, with corresponding average service rates at tier-j as $\mu_{j1} = 50$ and $\mu_{j2} = 100$, respectively. We construct a single hybrid-server, say s_h, by proportionally mixing the component shifted-exponentials of each s_1 and s_2. Let the configuration be one-server of each type, i.e. $\overline{n} = [1, 1]$; then the mixing proportions using (4.22) is $\beta_{j1} = 1/3$ and $\beta_{j2} = 2/3$, and the final service-time distribution of the hybrid-server for the j^{th} tier is $f'_j = \left(\frac{f_{j1}}{3} + \frac{2f_{j2}}{3}\right)$.

4.6.2 Heterogeneous configuration

Once we obtain the optimal configuration for a given hybrid-server, and given workload and percentile, we translate this solution configuration to the corresponding heterogenous server configuration; this is done by reversing the steps of creating the hybrid-server. Let us assume that the servers are indexed in increasing order of their average service rate; i.e. $\mu_1 \leq \mu_2 \leq \ldots \leq \mu_M$; let n'_j be the number of hybrid-servers at tier-j, then the number of servers of type-i for tier-j is $n_{ji} = \beta_{ji} n'_j / (\mu_i / \mu_1)$.

4.6.3 Searching for a new hybrid-configuration

The cost of the new heterogenous configuration, computed in the step above, is evaluated using the prices of the servers. If the cost is less than that of the current solution

configuration, then this new configuration is accepted else it is dropped. The new configuration is again fed to the model, and its utilization ρ^* is evaluated for the desired arrival rate λ_D. We then try to search for a new hybrid-configuration which has higher utilization but lower cost then the current-configuration; the new utilization $\rho_n = (\rho_{max} + \rho_l)/2$, where ρ_{max} is maximum utilization of the hybrid-server and $\rho_l = \rho^*$. The new hybrid configuration is searched for using the following ILP solved for each tier:

$$\text{minimize} \sum_{i=1}^{M} n_{ji} p_i, \tag{4.23}$$

subject to the constraint

$$\sum_{i=1}^{M} n_{ji} \mu_{ji} > \lambda_D / \rho_n. \tag{4.24}$$

Note that if the currently suggested configuration is not accepted we continue to search for higher ρ^*. The algorithm stops when $\rho_n - \rho_{max}$ is less than a pre-decided threshold; the pseudo code is outlined in [90].

4.7 Experimental Evaluation

In this section we demonstrate the efficacy of our approach. We have implemented our analytical method using MATLAB®. For solving the ILP, we have used lpsolve version 5.5.2.0 and have used mxlpsolve MATLAB Interface version 5.5.0.6 for calling lpsolve from within the MATLAB environment.

We begin by showing the effectiveness of the service-time approximation algorithm on lognormal[5] distribution with different coefficient of variations (C_v). Thereafter we evaluate the goodness of the approximation of the response-time distribution for a 1-tier and a 2-tier system by comparing the response times computed using (4.17) with those obtained using a multi-tier application-simulator described below. Finally we do a case-study of

[5]PDF of a log normal distribution is expressed as $f(x, \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-((ln(x)-\mu)/(\sqrt{2}\sigma))^2}$, where mean is $e^{\mu+\sigma^2/2}$

provisioning of a two-tier application for a SLA specified as a threshold on the 99^{th} percentile of response time. We evaluate the effectiveness of our approach by computing the 99 percentile of response times obtained using a two-tier application-simulator configured according to the capacity decisions provided by our approach; note that the simulator depicts an ideal version of a multi-tier application which we analytically model as a chain of M/G/1-PS queues. We also evaluate the effectiveness of our approach, using a metric called provisioning error (described in Section 4.7.4), by comparing against the two other baseline approaches, which model the multi-tier application as an open tandem network of M/M/K-FCFS queues.

4.7.1 Multi-tier Application Simulator

We implemented a simulator for the PS queue in MATLAB®. It takes as input an array of request arrival instants and size of each request (in terms of service time) and outputs the request departure instants. We used this queue simulator to simulate a multi-tiered application by feeding the output of first queue to the input of the next queue.

To simulate an application with replicated tiers, we have implemented a loadbalancer, as shown in Figure 4.1, which takes the incoming requests from the previous tier and distributes it to the next tier according to a specific load distribution policy. It also does the necessary book-keeping to track each request across various tiers for computing the end-to-end response time. We have implemented a random loadbalancing policy, i.e. loadbalancer distributes the requests at random but ensures that each server gets the same load, i.e. $\rho*$ as computed in section 4.5. We have assumed an ideal loadbalancer, which means that it introduces no queueing and processing delay. Note that this is not a limitation of our approach, as our approach can easily account for loadbalancer by considering it as another tier and its capacity can also be computed, which is often needed in a real setup.

4.7.2 Service Time Approximation

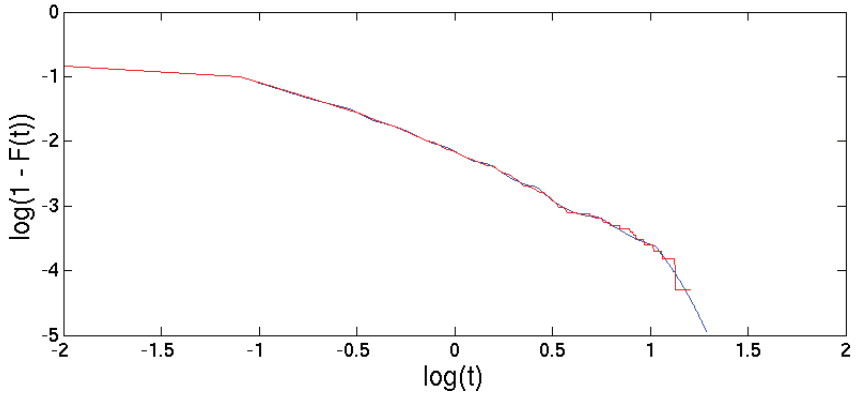
We have implemented the EM algorithm (in MATLAB) for finding the parameters of mixture of shifted exponentials, namely α_i, μ_i in (4.8), using the E and M steps mentioned in Section 4.4.2.1. The shifts and initial values of parameters are estimated using the algorithm outlined in Section 4.4.2.2. We use MATLAB's implementation of KMeans and have kept $K_{max} = 20$ in all our experiments, which means that we search for the number of shifted exponents from 3 till 20. We evaluate the accuracy of CDF approximation using relative percentage error defined as $\epsilon(x) = (F_{aprx}(x) - F_{sim}(x))/(F_{sim}(x))$, where $F_{aprx}(x)$ and $F_{sim}(x)$ are the values of approximate and actual CDFs, respectively, evaluated at x.

To evaluate the effectiveness of our approach in approximating highly variable distribution, we approximated the PDF a log-normal distribution with same mean rate of 20 but with a coefficient of variation of 100 ($C_v = 100$); it was expressed using 10 shifted exponentials. Figure 4.5a shows the CCDF of the actual distribution in red while our approximated distribution is shown in blue.

The CCDF in Figure 4.5 highlight the approximation of the tail of the distribution by plotting the $1 - F(x)$ in a log log scale. We observed that the approximation shows a relatively high error at low percentiles (as high as 21%) but displays low errors at the tail, with errors less than 1% at 95 percentile. This is because, that at low percentiles the number of exponentials available to approximate the distribution are less but as we approach the tail of the distribution a large number of exponentials contribute towards the approximation of the PDF and thus we observe much greater accuracy.

Another aspect of our algorithm is K, i.e. the number of exponentials required to approximate a distribution. We conducted a large number of experiments on various data sets with C_v ranging from 1 to 100. In our experiments we found that K its average values starts at 14 for $C_v = 1$ and slightly decreases to an average value of 10 for $C_v = 100$. It should be noted that we are testing our approximation scheme for a smooth distribution function, but the scheme has been designed keeping a web application in vision, which has

only a limited number of request types at each tier and our approach is tuned to estimate this number as K.



(a) Lognormal with $C_v = 100$

Figure 4.5: Figure shows the log log plot of 20,000 data points sampled from lognormal distribution with $C_v = 100$; the simulated CDF is shown in red and approximate in blue.

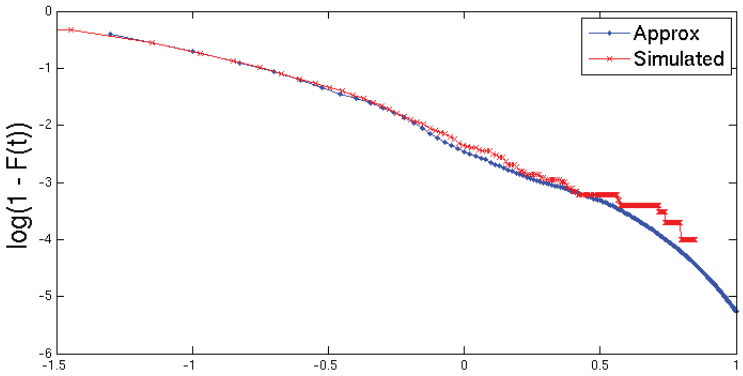
In summary, the service time approximation approach offers very low errors, i.e. less than 1%, in estimating the tail of a distribution.

4.7.3 Response Time Approximation

In this section we describe the effectiveness of our approach of approximating the end to end response time of an application modeled as a chain of M/G/1-PS queues.

We evaluated the goodness of the response-time approximation we compare the response time computed by our approach with that obtained from the simulator described above. We show the results for for a 2-tier setup by plotting the response time CDFs for our approximation and simulation. We have sampled service times from a lognormal distribution with a $C_v = 10$.

We generate the workload which has exponential inter-arrival times with $\lambda = 25$ and service times sampled from a lognormal distribution with $\mu = 50$ at each tier. The simulation results are considered exact since the simulation model is an exact representation of the queueing network under study.



(a) CDF RT of 2-tier app

Figure 4.6: Figure shows the CDF plot of actual response time distribution in red and approximated using our approach in blue for a heavy-tailed service-time distribution with $\mu = 50$ and $c_v = 10$

The approximated response time, using our approach, exhibits high accuracy, as can be seen from Figure 4.6. The jagged tail of simulation result is because of less number of data points.

4.7.4 Provisioning in a Homogenous Setup

In this section we evaluate the effectiveness of our approach, outlined in section-4.5, in finding the homogenous configuration for a two-tier application, where each tier is replicated using same type of servers.

For a given SLA, expressed as a cutoff threshold T_D on the 99^{th} percentile of end to en response time, we fix a service time distribution and for different arrival rates compute the number of servers required at each tier of the application. We, then, run the replicated application simulator with these number of servers and obtain the end to end response time distribution for the provisioned application. To evaluate the goodness of provisioning decisions made, we define a metric called *provisioning error*, which essentially calculates the error in the 99^{th} percentile response time observed from the simulator, i.e. T_{scheme}, and T_D. Formally, $\epsilon_{scheme} = (T_{scheme} - T_D) * 100/T_D$. To do a comparative evaluation of our

technique, we have implemented two baseline provisioning algorithms based on M/M/1-FCFS queues, namely *per-tier-exp* and *end-to-end-exp*. The schemes are described below:

- *per-tier-exp (pte)* : In this scheme we assume the knowledge of average proportion of time spent by a request at each tier. In other words, let T be the total time spent by a request in the system and T_i be the time spent at tier i; then, *pte* assumes the knowledge of $E[\delta_i]$, where $\delta_i = T_i/T$. We model each tier as an M/M/K-FCFS queue and again approximate multiple servers by a single server, thus each tier can be approximated by an M/M/1-FCFS queue. For this system the response time is exponentially distributed with parameter $\mu(1-\rho)$. Finally, as in Section 4.5, for each tier j, we solve for ρ^* with $T_D = \delta_j T$ and compute $n_j = \lceil \lambda_j/(\rho^*\mu_j) \rceil$

- *end-to-end-exp (ete)*: We developed this scheme completely along the lines of our scheme, however assuming an M/M/K-FCFS queue based model instead of an M/G/K-PS queue based model. The corresponding version of (4.19) is:

$$F_T(t') = \sum_{j=1}^{N} r_j(1 - e^{-\mu'_j t'}) \geq \theta, \tag{4.25}$$

  where $t' = T_D$, $r_j = 1/\prod_{k\neq j}^{N}(\mu'_k - \mu'_j)$ and $\mu'_j = \mu_j(1-\rho)$. The provisioning algorithm for homogenous setting is outlined in [90].

We ran the experiment with $T_D = 0.4s$, $\mu = 50$ and $C_v = 3$. We increased the workload from $\lambda = 40$ rps to 240 rps and for each λ we computed application capacity using each of the three algorithms. For *pte* we used $\delta_1 = \delta_2 = 0.5$. The results are shown in Table 4.1.

A Positive value of ϵ means that some or all of the tiers of the application were provisioned with fewer servers than required (we call it under-provisioning); however, a negative value means the opposite (we call it over-provisioning). Thus a positive ϵ is an SLA violation, while a negative ϵ is not. However, a negative ϵ does suggests a possibility of finding a more cost efficient solution. Note that our scheme reports a worst case provisioning error

λ	% ϵ_{our}	% ϵ_{ete}	% ϵ_{pte}	Config$_{our}$	Config$_{ete}$	Config$_{pte}$
40	-3.63	16	15.2	[3;3]	[2;2]	[2;2]
80	-6.17	48.1	27.9	[5;5]	[3;3]	[4;3]
120	-0.235	94.3	38.5	[8;7]	[4;4]	[5;5]
160	-2.25	91.6	49.9	[9;9]	[5;5]	[6;6]
200	-2.17	140	40.7	[12;12]	[6;6]	[8;8]
240	2.57	91.3	53.7	[15;15]	[8;8]	[9;9]

Table 4.1: Homogeneous configuration suggested by the three schemes and their provisioning errors. Note that, unlike the positive error, negative value of ϵ is not an SLA violation.

of 2.57% as opposed to the worst case under-provisioning of 140% by *ete* and 53.7% by *pte*.

In summary: for a single server type scenario (i.e. homogeneous setup), application provisioned by our scheme reports worst case provisioning error of 2.57%, while the baseline approaches shows as high as 140% *provisioning error*

4.7.5 Effect of Variability of Service Time

In this section we evaluate the effect of variability of service-time distribution on three provisioning schemes, namely *ours, pte* and *ete*.

For a fixed $\lambda = 160$, we computed the capacity of the two tier application, in a homogenous setting, using all the three schemes. We obtained the service times for both the tiers by sampling from a lognormal distribution with a fixed μ of 50 rps, while a varying the standard deviation σ. We vary σ so that we can control C_v, ranging from 1 to 10.

The computed capacities, by each of the schemes, were again tested using the application-simulator. Their percentage provisioning errors were computed and plotted in Figure 4.7.

Figure 4.7, shows that percentage provisioning error for *ete* and *pte* increases as a function of C_v, while maintaining the average service-rate constant, as opposed to our scheme, which shows a worst case provisioning error of 11%. The main reason behind this is that both *ete* and *pte* schemes are unable to capture the tail of the service-time distribution and thus cause severe under-provisioning.

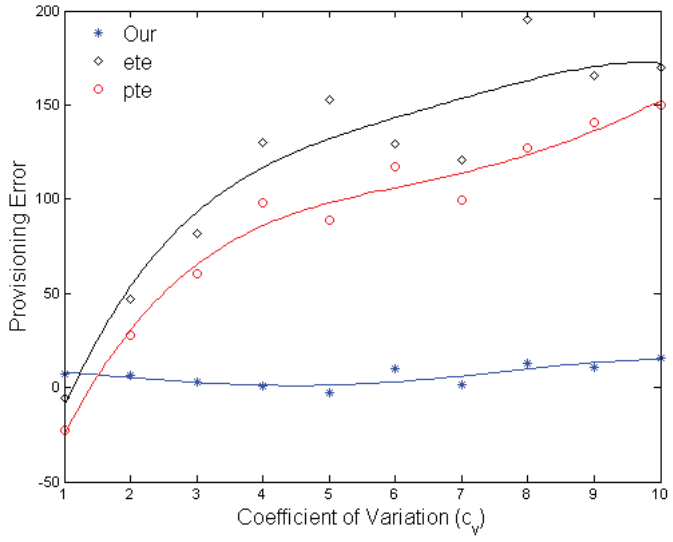


Figure 4.7: Variation in provisioning error with c_v

Thus we conclude that our scheme captures the tail of service-time distribution and is able to provision for the 99-percentile capacity with a max provisioning error less than 11% as opposed to other schemes which severely under-provision the capacity with the max-provisioning error of 196 %

4.7.6 Cost Efficient Server Configuration in a Multiple Server-type Environment

Here we demonstrate the effectiveness of our heterogenous provisioning algorithm in finding a cost-efficient solution when multiple types of servers are available. We have kept the time threshold $T_D = 0.4$-sec and varied the desired load from $\lambda = 40$-rps to $\lambda = 240$-rps. We have considered four types of servers, namely small (S), medium (M), large (L), and extra-large (XL), with their corresponding average service rates being 50, 100, 150 and 200 rps, respectively. The coefficient of variation of service times for requests at each of the tiers is $C_v = 9$.

We assume linear pricing as depicted in Table 4.2a. The results of provisioning algorithms in homogenous and heterogenous settings are shown in Table 4.2b. We call

ServerType	Small	Medium	Large	XLarge
Price	0.02	0.04	0.06	0.08

(a) server prices

λ	% ϵ_{homo}	% ϵ_{hetro}	Config$_{homo}$	Config$_{hetro}$	%Saving
40	1.63	-40.9	[9;9]	[0 1 0 0;0 1 0 0]	77.78
80	1.01	-35.7	[17;15]	[0 0 0 1;0 0 1 0]	78.13
120	1.16	-22.9	[26;23]	[0 0 2 0;0 1 1 0]	77.55
160	1.06	-23.5	[34;30]	[0 0 1 1;0 0 2 0]	79.69
200	1.09	-21.5	[43;47]	[0 0 3 0;0 1 2 0]	81.11
240	1.04	-9.82	[51;45]	[0 0 2 1;0 0 3 0]	80.21

(b) 99-percentile provisioning and cost benefit

Table 4.2: Heterogeneous configuration suggested by the three schemes and provisioning error of each scheme. Note that a negative ϵ only means over-provisioning and is not an SLA violation

the computed capacity configurations in the homogenous and heterogeneous settings as $Config_{homo}$ and $Config_{hetro}$, respectively. Only the "small" server-types were used in $Config_{homo}$, while all the available server types we used to obtain $Config_{hetro}$. As in previous evaluations, we again test the computed configuration using the multi-tier application simulator.

Each configuration is $N \times M$ dimensional matrix depicting the number of servers of each type; each row j depicts the configuration of the j^{th} tier, while each column tells the number of servers for each type: for e.g. Config$_{homo} = [9; 9]$ means 9-small servers at both the tiers, while Config$_{hetro} = [0\ 1\ 0\ 0;0\ 1\ 0\ 0]$ means 0-small, 1-Medium, 0-large and 0-x-large server at both the tiers. The "%Cost Saving" is computed as a percentage of cost of homogenous configuration, i.e. $\frac{Cost(Config_{homo}) - Cost(Config_{hetro})}{Cost(Config_{homo})}$.

We make following important observations: 1) the percentage provisioning error for the heterogeneous scheme is as low as -41%, which means that not-only is this configuration cost-efficient but it also provides low average response-times (because negative provisioning error means the system is probably over-provisioned). The small positive error in the case of homogeneous configurations is because of approximation used in section 4.5.1 and

can be easily corrected by setting $\zeta < 1$, i.e. a sublinear scaling. 2) it is better to use larger servers that fit the same cost and average service-rate; in other words its better to use a small number of large servers instead of a large number of small servers.

In summary, it is better to use a small number of large servers instead of a large number of small servers for high percentile provisioning ii) Cost efficient heterogenous algorithm offers server configurations with cost savings as high as 81% and also offer a configurations with lower average response-times.

4.8 Evaluation on Private Cloud

In this section we describe an experimental investigation for provisioning for a percentile SLA in a private cloud setup. Our goal is to evaluate our provisioning algorithm under situations which are typical to multi-tier web applications deployed in a datacenter or private/public cloud environment.

4.8.1 Private Cloud Setup

In this section we provide the necessary details of our experimental testbed, i.e private cloud, and necessary steps before we can perform server provisioning.

4.8.1.1 Web Application

We used TPC-W for our experiments. TPC-W is a multi-tier transactional web benchmark that represents an e-commerce web application – an online bookstore – comprising of a web server tier and a database tier. It simulates the activities of a retail store website using 14 different type of pages for web interactions; each of these pages are created dynamically by the web server using differing amounts of data stored in the database tables. TPC-W benchmark defines three different mixes of web interactions, namely browsing, shopping and ordering, each varying the ratio of inventory browsing related web pages and ordering related web pages. It applies the workload mixes via remote browser emulator (RBE).

We used the Java implementation of TPC-W [15]. The web application has following two-tiers: i.) Web server tier based on Apache Tomcat servlet container 5.5.26 ii.) database tier based on MySQL 5.0.77. We deployed each of the tiers on separate dedicated servers. We performed round robin load balancing between replicas of web server tier using a dedicated loadbalancer server on HAProxy [47] on a server as a dedicated load balancer. We used round robin load balancing at the database tier by setting up a master-slave replication configuration of MySQL servers; we instrumented TPC-W to use the replication aware MySQL JDBC connector version 3.1.12.

4.8.1.2 Private Cloud

We constructed private cloud using OpenNebula [75] on Xen/linux-based cluster consisting of two types of servers: 8-core 2GHz AMD Opteron 2350 servers and 4-core 2.4 GHz Intel Xeon X3220 systems. All machines run Xen 3.3 and Linux 2.6.18 (64bit kernel). Our platform is assumed to support *small* and *large* servers, comprising 1 and 4 cores, respectively. These are constructed by deploying a Xen VM on the above mentioned servers and dedicating the corresponding number of cores to the VM (by pinning the VM's VCPUs to the cores)

4.8.1.3 Profiling servers for web server tier

For profiling the servers for the first tier, i.e. web-server tier, we instrumented Tomcat such that it reports per-request service times, along with the other default stats. We profile each server type (e.g. *small* and *large*) by provisioning an instance of that server-type and deploying the first tier of TPC-W on it. We then connect it to an already installed TPC-W database installed on a *large* server type instance. We then issue the browsing workload using the TPC-W clients (i.e. RBEs) for a duration of 35 mins and collect the service times from the tomcat server logs.

4.8.1.4 Profiling servers for database server tier

Profiling the servers for the second tier of TPC-W (i.e. the database tier) was in two steps: firstly, we collect the 35-min query logs from MySQL server, executing the TPC-W workload; then for each server type we slowly replay each of the SQL query and record their execution time as service times.

4.8.2 Percentile Based Capacity Provisioning on Private Cloud

Given λ_D and T_D, we outline the high level steps required to compute application capacities for both homogenous and heterogenous setup. In both the cases we assume to require an SLA where 99^{th} percentile of the end-to-end response time must be less than 0.5 seconds. We follow the following sequence of steps

Step 1: Estimating service time distributions: We use the service times collected during the offline profiling step and use the service time approximation algorithm – as outlined in [90].

Step 2: Estimating capacity in a homogenous/Heterogenous setup: We used the single core virtual machines (i.e. *small*) in our homogenous setup. Load across multiple web-server replicas was distributed using a HAProxy based load-balancer, however, in the case of database tier, we used the master-slave setup. In this setup all the wites are sent to the master, whereas the reads are load-balanced.

We test our approach for both homogenous and heterogeneous environment. For homogenous setup, we choose *small* server type for this case and assume $T_D = 0.5$sec. To test the provisioning setup for large change in workload, we varied λ_D from 15 rps to 90 rps. For each λ_D, using our approach, we computed server capacities for each of the tier of TPC-W. We ran the setup for 35-mins and in the end we collected the end-to-end response times from the first tier (i.e. web-server tier). We ran our heterogeneous provisioning algorithm on the same setup and found that it gave a different configuration, only for $\lambda_D = 90$. Table 4.3a provides the details of the final configuration and also the 99^{th}-percentile of the

end-to-end response time details of the experiment. We compute the percentage provisioning error, ϵ_{our}, as mentioned in 4.7.4.

λ_D	99^{th} %	% ϵ_{our}	Config$_{our}$
15	0.361	-27.8	[1;1]
30	0.459	-8.2	[1;2]
45	0.488	-2.4	[1;3]
90	0.512	2.4	[2;7]
90	0.46	-8.0	[2,0;2,1]

(a) Server Provisioning

Server Type	*small*	*large*
Prices ($)	0.085	0.34

(b) Server prices

Table 4.3: Homogenous and heterogeneous provisioning decisions. Note that a -ve ϵ_{our} only means that the system is over-provisioned and thus SLA will not be violated

We found that server provisioning by our approach keeps provisioning error below 3%. The positive 2.4% error at $\lambda = 90$ for homogenous setup could be because database tier does not scale linearly as the master database server gets overloaded by replicating the updates to each of the 6 slaves. We see that the server provisioning for the heterogenous environment, is not only 11.11% cheaper than the corresponding homogenous server setup but also has a lower 99^{th} response time.

In summary, our algorithm effectively accurately captures the service time distributions and provisions the two-tier implementation of TPC-W with the worst provisioning error of 3%. Also, we, again, find that its better to use bigger server for high percentile provisioning.

4.9 Related work

A number of efforts have modeled internet applications. Modeling single tier has gotten much of the attention. Doyle *et al.* propose a queuing model for static content [36], Menasce uses a queuing model to model the web servers [66], while Abdelzaher *et al.* in [1] use classical feedback control theory to model the bottleneck tier for providing performance guarantees for web applications serving static content, while Chen *et al.* in [21] use a machine learning technique for provisioning the database tier.

Ranjan *et al.* [86] use a G/G/N queuing model to compute the number of servers necessary to maintain a target utilization level. This strategy is shown to be effective for sudden increases in request arrival rate. Other efforts have employed M/G/1 queuing models in conjunction with offline profiling to model service delay and predict performance [99] but they do not provision for response time percentile and neither do they address the problem in heterogenous environment. The approach in [107] formulates the application tier server provisioning as a profit maximization problem and models application servers as M/G/1/PS queuing systems; the approach only considers the impact of different number of end-clients (and thus, request volumes)

Benanni *et al.* in [10] employ approximate mean-value analysis (MVA) to develop an online provisioning technique for multiple request classes. Urgaonkar *et al.* in [103] develop a queuing network model for multi-tier Internet applications having request classes with differentiated QoS. Zhang et. al. [121] use a multi-class model to capture the dynamics of workload by employing a fixed set of 14 predefined transactions-types and leverage it to predict the performance of a multi-tier system.

There has been some work for finding the pdf of response time, for e.g. Muppula *et al.* in [68] derive the response time for a closed queuing network using pteri-nets and sojourn time distribution was calculated for large Markov chains in [48]. The approach leads to an inversion of a complex Laplace transform. Xiong *et al.* in [118] perform the provisioning of a multi-station setup for a given percentile bound. The model the system as a open tandem network of M/M/1-FCFS queues and compute the response time PDF by numerical inversion of its Laplace transform; they assume that each station is serviced by same type of servers.

In contrast to these efforts, our work automatically characterizes service time distribution as a mixture of shifted exponentials and leverages this to estimate the response time distribution. The estimated distribution is used to estimate the capacity of the system which assists in finding a near optimal solution to the provisioning problem in homogeneous en-

vironment. Further, while most of these efforts have focused on a single server type environment (i.e. homogeneous), we extend our approach for a cloud specific heterogenous environment as well. We developed a full prototype implementation and our experiments were conducted on an actual private cloud.

4.10 Conclusion

Multi-tier architecture is a preferred architecture for enterprise web applications and high response time percentile provisioning is more meaningful than mean response time based ones. We present an approach of optimizing server allocation for a multi-tier application to achieve a percentile bound on the end to end response time. We model the application as an open tandem network of queues and model each tier as an M/G/1-PS queue. We have developed an approximate model to compute the response time distribution and have also developed a technique to estimate the service time distribution from the service time histograms. We have developed an algorithm to compute per tier server allocation of the application and in a homogenous setup. We also have extended the homogenous setup solution to solve the server allocation problem in a heterogenous setup. We have tested the efficacy of our approach using a multi-tier application simulator and also compared it against two other baseline approaches developed using models based on M/M/K-FCFS queue. We have demonstrated superior performance of our approach as compared to the baseline approaches. Our experiments indicated that its better to use small number of large servers than large number of small servers. Finally we tested our approach using the multi-tier implementation of TPC-W benchmark over private cloud created using Xen over Linux.

CHAPTER 5

SEAGULL: INTELLIGENT CLOUD BURSTING FOR ENTERPRISE APPLICATIONS

The high cost of provisioning resources to meet peak application demands has led to the widespread adoption of pay-as-you-go cloud computing services to handle workload fluctuations. In the previous chapters we have addressed elasticity of a multi-tier application assuming that cloud has infinite capacity. But in the case of private clouds, the in-house IT infrastructure is limited and often enterprises employ a hybrid cloud model where the enterprise uses its own private resources for the majority of its computing, but then "bursts" into the cloud when local resources are insufficient. However, current commercial tools rely on the system administrator's knowledge to answer key questions about when a cloud burst is needed and which applications must be moved. In this chapter we describe Seagull, a system designed to facilitate cloud bursting by determining which applications can be transitioned into the cloud most economically, and automating the movement process at the proper time.

5.1 Introduction

Many medium and large enterprises have significant current investments in IT data centers that house compute and storage systems. This IT infrastructure is often sufficient for the majority of their computing needs, while offering greater control and lower operating costs than the cloud. However, workload spikes in hosted enterprise applications, both planned and unexpected, can sometimes drive the resource needs of enterprise applications above the level of resources available locally. Rather than incurring capital expenditures for

additional server capacity to solely handle such infrequent workload peaks, a hybrid model has emerged where an enterprise leverages its local IT infrastructure for the majority of its computing needs, and supplements with cloud resources whenever local resources are stressed.

This hybrid technique, which is referred to as "cloud bursting", allows the enterprise to expand its capacity as needed while making efficient use of its existing resources. While commercial and open-source virtualization tools are beginning to support basic cloud bursting functionalities [79, 74, 111], the primary focus has been on the underlying mechanisms to enable the transition of virtual machines between locations. These systems leave significant policy decisions in the hands of system administrators, who must manually determine when to invoke cloud bursting and which applications to "burst".

Manually performing these steps requires individual administrators to have significant knowledge of the data center applications. As a result, manual decision making may lead to poor choices in terms of minimizing cloud costs or reducing downtime during the transition. One of the *insights* of our work is that rather than naïvely moving an overloaded application to the cloud, it is sometimes be cheaper and faster to move *different* applications and then assign the freed-up server resources to the overloaded application. Judiciously making these choices manually is difficult especially when there are a large number of diverse applications in the data center and different cloud platform pricing models.

Typically, cloud bursting assumes that both local and cloud data centers employ virtualization. Bursting an application to the cloud involves copying its virtual disk image and any application data. Since this disk state may be large, consisting of tens or hundreds of gigabytes, a pure on demand migration to the cloud may require hours to copy this large amount of data. A second *insight* of our work is that periodic background precopying of virtual disk snapshots of candidate applications can significantly reduce the cloud bursting latency–since only the incremental delta of the disk state needs to be transferred to

reconstruct the disk image in the cloud. Our work also examines the impact of judiciously choosing the candidate set of applications for such precopying.

We have developed Seagull to alleviate the above challenges; Seagull automatically detects when local infrastructure is becoming overloaded, decides which applications can be moved to the cloud at lowest cost, and then performs the migrations needed to dynamically expand capacity as efficiently as possible. By automating these processes, Seagull is able to respond quickly and efficiently to workload spikes.

We make several contributions in this work: **(i)** a placement algorithm that determines which applications should be moved to minimize cost; **(ii)** a precopying algorithm that decides which applications should be proactively replicated to the cloud to enable much faster VM migrations; **(iii)** a prototype of Seagull using a Xen-based local data center and the Amazon EC2 cloud platform, and **(iv)** a detailed experimental evaluation of Seagull for different applications. Seagull supports live and non-live migration to enable cloud bursting and we show Seagull's placement algorithm can make intelligent decisions about which applications to move, lowering the cost of resolving an overloaded large scale data center by over 45%. We also demonstrate that our precopying algorithm can dramatically lower the time needed to move applications into the cloud while incurring only a small cost to retain replicated state in the cloud.

5.2 Background and Problem Statement

This section provides background information on existing cloud bursting tools and the types of applications which they can be used with. We then detail the challenges faced by a cloud bursting management system and describe the problem we seek to resolve.

5.2.1 Cloud Bursting Background

Employing cloud bursting can save enterprises a significant amount of money. Figure 5.1 illustrates a scenario where a business typically requires five "extra large" servers

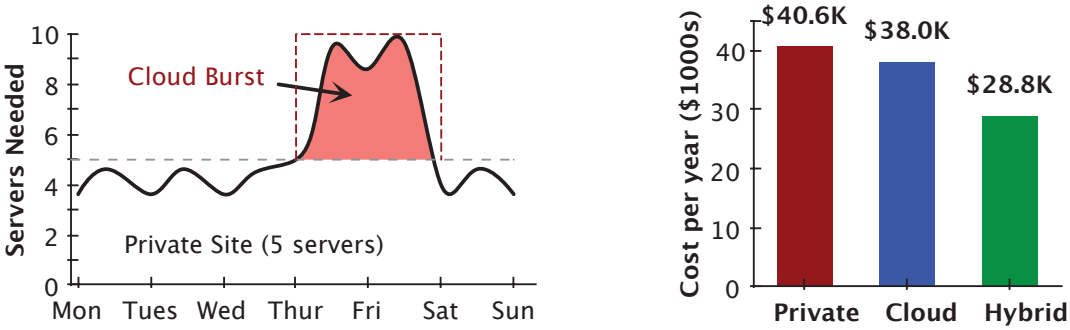


Figure 5.1: Hybrid clouds can utilize cheaper private resources the majority of the time and burst to the cloud only during periods of peak demand, providing lower cost than exclusively private or public cloud based solutions.

for its daily needs, but two days a week experiences a spike up to ten servers. Using Amazon's EC2 Cost Calculator [7], we find that using private resources for this would cost about $40K a year since the business would be required to pay for all ten servers up front. A cloud-only solution provides greater elasticity, allowing the business to pay for ten servers only during the two days a week that they are needed, but the premium paid for the cloud negates much of this benefit, only lowering the cost by $2,600. However, if a hybrid approach is used so five servers are kept in the private site at all times and the cloud is only used for an additional five servers on Thursdays and Fridays, the total price drops by a further $9,200, a saving of 27%.

These observations have resulted in new product offerings from software vendors and cloud providers such as Amazon, VMware, and Rackspace that help businesses connect and manage "hybrid" clouds that span both private and public resources. Cloud management tools such as OpenNebula and Eucalyptus have begun to support flexible placement models where new applications can be easily deployed into either a local or public cloud.

However, these existing solutions are generally designed to move resources between private and public clouds only at very coarse time scales. For example, Terremark's cloud bursting system designed for a government agency could take between two and ten days

to fully burst from the local to cloud site [16]. This slowdown is caused by the massive amount of application state that must be transferred over relatively slow links between a private data center and a public cloud. Our work seeks to enable more agile cloud bursting that can respond to moderate workload spikes within hours or even minutes.

A further limitation is the difficult high-level policy decisions of when to invoke these migration tools, which applications to move, and for how long still must be done manually by system administrators. These decisions are non-trivial, particularly in data centers with a large number of applications.

5.2.2 System Model and Problem Statement

Our work assumes a medium size or large enterprise that has its own backend IT infrastructure housed in one or more data centers. We assume that each application is virtualized and comprises one or more virtual machines. Each server may host one or more VMs for different applications. We assume that the data center is virtualized and is agile in terms of allocating server capacity to applications. Application capacity can be scaled in one of two ways. In horizontal scaling, the application is replicable so that new VM replicas can be started up on demand to increase application capacity. In vertical scaling, we assume that the application is not replicable, and the data center can only scale the application capacity by migrating the VM to a larger server. Typically horizontal or vertical scaling is performed locally within the data center by using any spare servers that may be available. When the local site becomes overloaded, cloud bursting is used to obtain additional capacity. We enforce the common constraint that all VMs that make up an application must be kept together either in the private site or the cloud.

The goal of our work is to design a system that can both automate and optimize cloud bursting tasks. We assume that the application resource needs and constraints such as whether an application is horizontally or vertically scalable are known and so is the cloud pricing model which dictates server rental and network I/O costs. Given this knowledge,

our system must answer the following questions: i) When to trigger a cloud burst? ii) Which applications to cloud burst so as to optimize cloud server and I/O costs? iii) Can judicious precopying reduce cloud bursting latency? and if so by how much?

In this work, we formulate the second question as an optimization problem and present an algorithm, that iteratively uses the solution to this problem to find an answer for the third question.

5.2.3 Problem definition and formulation

Let N denote the number of applications. Each application i is composed of M_i virtual machines. Let L denote the number of clouds (or datacenter locations), with the first being the private cloud location. Let H_l denote the number of hosts in location l. Each of these virtual machines require two resources (p_{ijkl}, r_{ijkl}), where p_{ijkl} denotes number of cores[1] and r_{ijkl} denotes the size of RAM required by the j^{th} VM of i^{th} application on the k^{th} host of l^{th} location. Let (P_{lk}, Q_{lk}) be the cores and RAM, respectively, of k^{th} host at l^{th} location. Let $\mathcal{C}_{ijlk}$ be the cost of moving the i^{th} VM of j^{th} application to k^{th} host at l^{th} location; within the same location we keep the cost to be zero, while across locations we compute the cost using (5.7) and (5.8). Let α_{ijkl} and β_{ijkl}be binary variables, such that:

$$\alpha_{ijkl} = \begin{cases} 1 \text{ if } j^{th} \text{ VM of } i^{th} \text{ app is on } k^{th} \text{ host of } l^{th} \text{ loc} \\ 0 \quad \text{otherwise} \end{cases}$$

$$\beta_{il} = \begin{cases} 1 \text{ if } i^{th} \text{ app is at the } l^{th} \text{ loc} \\ 0 \quad \text{otherwise} \end{cases}$$

The optimization problem is:

[1]The number of cores required on each host varies depending on the hardware of host; thus the number of cores also depends on the host k

$$\text{minimize} \sum_{i=1}^{N} \sum_{j=1}^{M_i} \sum_{l=1}^{L} \sum_{k=1}^{H_l} \alpha_{ijk} \mathcal{C}_{ijlk}$$

subject to

$$\sum_{l=1}^{L} \sum_{k=1}^{H_l} \alpha_{ijkl} = 1 \qquad \forall j = 1 \ldots M_i, \forall i = 1 \ldots N \qquad (5.1)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M_i} \alpha_{ijkl} p_{ijkl} \leq P_{lk} \qquad \forall k = 1 \ldots H_l, \forall l = 1 \ldots L \qquad (5.2)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M_i} \alpha_{ijkl} q_{ijkl} \leq Q_{lk} \qquad \forall k = 1 \ldots H_l, \forall l = 1 \ldots L \qquad (5.3)$$

$$\sum_{j=1}^{M_i} \sum_{l=1}^{L} \sum_{k=1}^{H_l} \alpha_{ijlk} = M_i \qquad \forall i = 1 \ldots N \qquad (5.4)$$

$$(1/H_l) \sum_{j=1}^{M_i} \sum_{k=1}^{H_l} \alpha_{ijlk} p_{ijlk} \leq \beta_{il} \qquad \forall i = 1 \ldots N, \forall l = 1 \ldots L \qquad (5.5)$$

$$\sum_{l=1}^{L} \beta_{il} = 1 \qquad \forall i = 1 \ldots N \qquad (5.6)$$

$$\alpha_{ijk}, \beta_{il} \in \{0, 1\} \qquad \forall i, j, l, k.$$

Constraint (5.1) ensures that each VM is on a single host. Constraints (5.2) and (5.3) ensure that resources used by VMs do not exceed the host capacity, while constraint (5.4) ensures that all the VMs of each application are placed on some host. Constraints (5.5) and (5.6) together ensure that all the VMs of an application stay in one location.

The optimization problem is a 2-dimensional bin packing problem [28, 35] and is a well know NP-hard problem. We solve this formulation using CPLEX but the approach will be unacceptable for large number of applications and VMs. We introduce a heuristic in the following sections and also compare its performance with the optimal solution for small setups in our evaluation section.

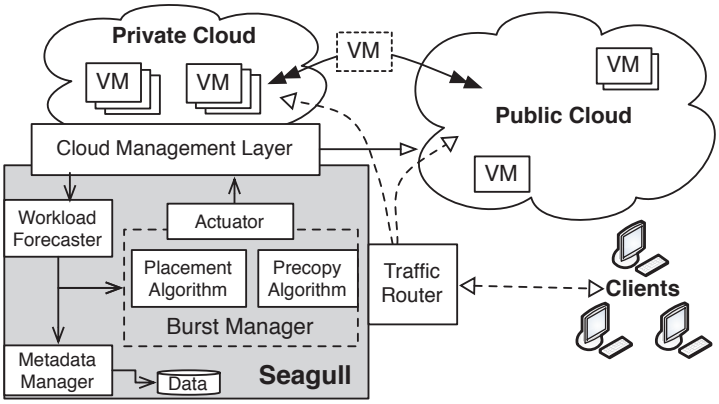


Figure 5.2: Seagull architecture

5.3 Seagull Design: Bursting to the Cloud

Seagull's architecture is divided into the core algorithms that control placement and precopying, the actuator that enacts the decisions of these algorithms, and the cloud management layer which translates generic Seagull orders into data center or cloud specific commands. This overall architecture is illustrated in Figure 5.2. This section describes the placement and precopying algorithms, we discuss how migrations are enacted in Section 5.4, and provide details of our EC2 specific implementation in Section 5.5.

5.3.1 Intelligent Placement Algorithm

Consider a simple scenario of a quad-core server Q and a dual-core one D. Application A is hosted on D with two cores and a disk state of 10GB while Application B is on Q with two cores but only 5GB disk state. Later A becomes overloaded and requires four cores. The naïve way to distribute this load is to directly burst A to the cloud even though there may exist a less expensive option, i.e., bursting B and move A to Q. The latter solution from Seagull saves not only the cloud charges but also reduces the data copying latency. The intuition behind Seagull is to maximize the utilization of local resources, which are cheaper than public resources, and migrate the cheapest applications when local resources are stressed.

Based on this intuition, Seagull uses the following algorithms to determine when a data center is overloaded, whether to use cloud bursting, and which applications to migrate.

5.3.1.1 Threshold based triggers

The decision of when to trigger a cloud burst involves monitoring performance metrics (like CPU utilization or load etc) and using a threshold on these metrics to trigger the cloud bursting algorithm. Depending on the scenario, we can use system-level metrics (such as CPU utilization, disk/network utilization or memory page fault rate) or application-level metrics such as response time. We assume that the system administrator makes a one-time decision on which metrics are relevant to the applications and specifies both the metrics that should be monitored and the thresholds that should trigger a cloud burst. In case of system-level metrics, the desired metrics can be monitored at the hypervisor-level, and for application-level metrics, we assume the presence of a monitoring system such as Ganglia that supports extensions to monitor any application metric of interest.

5.3.1.2 Use local resources first when possible

We assume that the capacity increase C necessary for each overloaded application can be determined using one of the many empirical or analytical methods proposed in the literature [102, 93]. Seagull first examines if any of the local servers have sufficient idle capacity to satisfy this desired capacity C. If so, the overloaded application can be live migrated to this server (for vertical scaling) or a new VM replica can be spawned on the server (for horizontal scaling). This is the simplest scenario for addressing the overload; Seagull also supports more sophisticated "repacking" of VMs to first free up the desired capacity C on a particular server and then move (or replicate) the application to that server. This is done in Seagull using a greedy technique that sorts all servers in decreasing order of free capacity. Starting with the first server on the list, if its idle capacity is less than C, the technique examines if one or more current VMs can be moved to a different server to

free up sufficient capacity C. If so, this sequence of VM moves can address the overload. Otherwise it moves on the next server with the most idle capacity and repeats.

5.3.1.3 Move the cheapest applications first

If the overload cannot be handled locally, Seagull must choose a set of applications to burst to the cloud. The objective is to select the cheaper option between bursting the overloaded app or one or more of the other applications based on cost. To do so in a cost-effective manner, the algorithm picks those applications to move that free up the most local resources relative to their cost of running in the cloud.[2]

To determine which applications should be moved, we assume the duration of the workload spike, L, and the desired capacity C for each virtual machine are known. Note that C is a vector representing the CPU, disk, network and memory capacity needs of the VM.

We then define the cost of bursting an application, say A, that is composed of n virtual machine in terms of the cost of transferring the memory and storage, storing the data, and the execution cost in public cloud:

$$Cost = \sum_{j=1}^{n} C\_tran_j + C\_stor_j + (C\_run_j * L), \tag{5.7}$$

$$C\_tran_j = C\_tran\_stor_j + C\_tran\_mem_j, \tag{5.8}$$

where $C\_tran\_stor_j$ and C_stor_j are calculated based on the amount of storage actively used by the j^{th} VM of A (i.e. VM_j); $C\_tran\_mem_j$ counts for the cost of transferring the memory in live migrating VM_j; C_run_j is determined based on the type of the virtual machine (e.g., the number of cores it requires) and must be multiplied by L to account for the length of time the VM would need to remain in the cloud before the workload spike passes. We sum the cost across all VMs in the application to account for the constraint that

[2]Additional administrative criteria such as security policies may also preclude some applications from being valid cloud burst targets; we assume that system administrators provide this information as a cloud bursting black list.

all virtual machines that comprise an application be grouped either in the local data center or on the cloud.

The virtual machines in the overloaded application are considered in decreasing order of their resource requirements. For each of these virtual machines, Seagull considers the potential hosts in the local data center sorted by their free capacity in descending order.

This approach is biased towards utilizing the free capacity in the local data center first, potentially reducing the number of applications that needs to be moved to the cloud.

The secondary sorting criteria(tie breaker) we consider is the total cost of moving all *applications* on a host to the cloud; this includes the cost of not only the VMs on that particular host, but all other VMs that would need to be moved in order to keep each application grouped together in the cloud. This causes the hosts that run primarily low cost applications to be considered first.

When hosts have been sorted in this way, the algorithm considers the first host and decides if a set of VMs on that host can be moved in order to create space for the overloaded VM. Each virtual machine, VM_j on the host is ranked based on:

$$num\_cores_j/Cost \tag{5.9}$$

where $Cost$ is the cost of moving the full application that VM_j is part of, and num_cores_j is the number of CPU cores currently in use by the virtual machine. The VMs on the host are considered in decreasing order of this criteria, and the first k VMs are selected such that the free capacity they will generate is sufficient to host the overloaded virtual machine. The intuition behind this greedy heuristic is that it *optimizes the amount of local capacity freed per dollar spent running applications in the cloud*.

Each of the overloaded applications is considered for bursting using this metric. When a solution is found, the total cost of moving all of the marked applications is compared to moving just the overloaded application; the cheaper of the two options is chosen in each case.

5.3.2 Opportunistic Precopying

In general, the application state, consisting of its code and data, may be large, of the order of tens or hundreds of gigabytes. Migrating all of this data at cloud bursting time can easily take hours or even days, significantly reducing the agility with which a data center can respond to rising workloads.

Seagull performs precopying by transferring an incremental snapshot of a virtual machine's disk-state to the cloud. Seagull's precopying technique makes two important decisions: (i) *which* applications to precopy, and (ii) *how frequently* to precopy each one. Each of these decisions lead to a cost-benefit tradeoff. The larger the set of candidate applications chosen for precopying, the greater the chances Seagull's cloud bursting algorithm will pick one of the precopied applications to burst to the cloud when the peak workload arrives, increasing the agility of the system to respond to local stress. Similarly, the more frequently each application is precopied to the cloud, the smaller the delta will be, leading to a smaller bursting latency. Thus a careful choice of the candidate set of applications to precopy and precopying frequency can both reduce the overheads.

Seagull supports multiple strategies to assist administrators in controlling the cloud storage and bandwidth costs incurred during precopying. We have implemented a strategy which computes a set of candidate applications that balances the benefits precopying against its cost and also two baseline strategies for comparison.

In our cost-benefit tradeoff strategy, we first generate an *overload list*, i.e. a list of applications likely to become overloaded[3]. Seagull, then, runs the cloud bursting algorithm, described in the previous section, in an *offline* mode over the *overload list*. That is, for each application A on the overload list, Seagull runs its algorithm to see which application(s) get chosen for bursting if A were to become overloaded. These applications form the *precopy list*.

[3]Seagull can generate such a list based on the history of prior cloud-burst instances and system administrators can alter it, based on their expert knowledge of which overload scenarios are still likely in the future.

The disk state of applications in the *precopy list* is replicated to the cloud based on a frequency strategy. In the simplest case the precopy frequency can be chosen statically — say once a day or once a week. However, Seagull can analyze the write rates to the virtual disks of each precopy application and compute how much data is being modified, which in turn allows for an estimate of both the monetary cost of transferring data and the time needed to perform a cloud burst. Based on these estimates, the system administrators can then "tune" the precopy frequency for each application in the precopy list. Doing so enables them to manage overall cloud costs (and avoid incurring large cloud bills) while retaining agility.

The two other strategies that we use as the baseline strategies for our comparitive evaluation are:

- *Random Precopying:* selects a random set of applications to be precopied based on the maximum expected overload (e.g., randomly precopy 20% of the data center's applications).

- *Naïve Precopying:*selects the set of applications that is predicted to become overloaded for precopying.

5.4 Cloud Migration

Once Seagull determines a plan for which applications to move to the cloud, the Actuator must execute this plan. Bursting an application is accomplished by copying its virtual disk image to the cloud. The virtual disk image consists of the application code, configuration files, and the application's data; once this has been moved to the cloud, a new VM can be started using the transferred disk for its storage. In practice, current cloud platforms require several additional steps to prepare a disk image for booting within the cloud, thus Seagull uses the following procedure, shown in Figure 5.3, to migrate a virtual machine to the cloud:

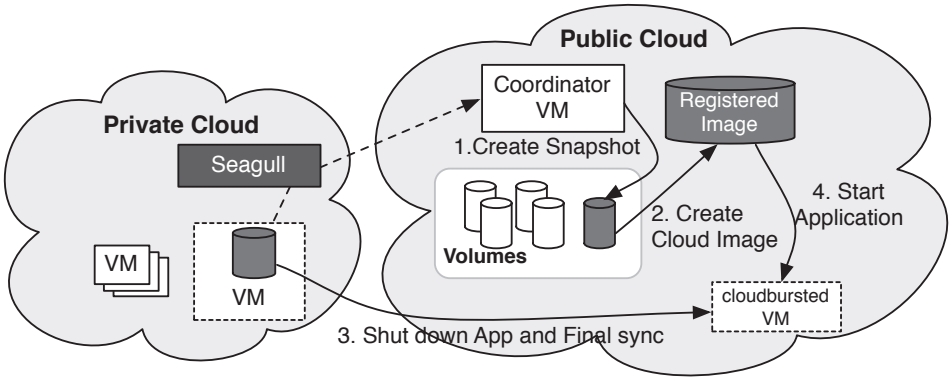


Figure 5.3: Seagull Cloud Bursting Procedure

1. Seagull creates a snapshot of the VM's file system as it is executes and transfers this to the cloud to be stored in a new disk image.
2. Seagull then takes the image snapshot and transforms it to the public cloud's usable image format. Seagull then registers the image with the public cloud management system and boots the VM.
3. Next Seagull shuts down the local instance of the application so that it can resynchronize any file system changes that have occurred at the origin site since the snapshot was made.
4. Finally, Seagull restarts the application within the cloud VM and redirects the application workload to the public cloud.

Seagull contains modules to implement this functionality for Amazon EC2, as discussed in the subsequent section. This approach is designed to minimize the amount of downtime incurred during a migration; the application being moved is only completely stopped during step 3. The process above can be easily enhanced to support precopying by preemptively performing steps 1 and 2, and then periodically copying a snapshot to the cloud. Thus when a burst is required, only the final stages of steps 3 and 4 need to be performed, reducing the total time of the cloud migration.

5.4.1 Supporting Live Migration

Since today's public cloud platforms such as EC2 and Azure do not presently support live VM migration, Seagull's cloud bursting mechanisms employ migration strategies that involve VM and application downtimes. However, if public cloud platforms were to support live migration in the future, Seagull's cloud bursting mechanisms can be easily adapted to take advantage of such a feature. For instance, we recently proposed the CloudNet system to support live migration of VMs over a WAN from one data center site to another [116]. Seagull can employ CloudNet technology to perform live cloud bursting. CloudNet employs VPN and VPLS protocols to enable transparent migration of a VM's IP address from one WAN location to another and uses several optimization techniques such as content-based redundancy elimination and block deltas to efficiently transfer memory (and disk) state of the VM over slow WAN links. In Section 5.6.2.3, we experimentally demonstrate how Seagull can employ such WAN migration mechanisms from CloudNet to support live cloud bursting from a private cloud to a public cloud site.

5.5 System Overview and Implementation

This section describes Seagull's five main components, namely i.) cloud management layer, ii.) workload forecaster, iii.) burst manager iv.) metadata manager v.) actuator. The functional block diagram of the architecture of Seagull is shown in Figure 5.2.

5.5.1 Cloud Management Layer

This layer exposes cloud management APIs used for managing and monitoring VMs and their resources across both private and public cloud. This layer offers a common abstract interface to the public cloud for all the other functional blocks of Seagull and adapts according to the destination cloud used for cloud bursting and monitoring.

We extended OpenNebula [74] to implement the *Cloud Management Layer* in our prototype. We implemented the mechanism of cloud bursting and precopying a VM from private cloud to public cloud and exposed it as an XML-RPC API.

Our cloud bursting implementation supports two different modes, namely *live* and *non-live*. The steps for VM migration are outlined in Section 5.4. We have implemented the migration tasks as python wrappers around the basic VM operations.

5.5.2 Precopier

We have implemented a filesystem level precopyer using *rsync* to replicate VMs across clouds. Our implementation supports live precopying, which means we precopy a VM image while the VM is active. Our implementation has two modes 1) *Initial Copy*: In this mode the precopier follows the following control sequence: i) create a volume on public cloud and attach it to the Coordinator VM, ii) rsync the local volume to the remote volume, iii) update the local database with local volume information and remote volume id. 2) *Subsequent Copy*: In this mode, the precopier performs only step (ii) of the initial copy stage.

5.5.3 Monitoring

We have extended OpenNebula to support sophisticated monitoring capabilities like that of EC2 cloudwatch. Our monitoring engine is implemented using the Ganglia monitoring system. Ganglia consists of a monitoring agent (gmond), which runs inside each host machine and virtual machines, and a gathering daemon (gmetad), that aggregates monitoring statistics from multiple monitoring/gathering daemons. Each VM image used by applications is pre-configured with a monitoring agent; thus, when new virtual machines are dynamically deployed, the Ganglia system automatically recognizes new servers and begins to monitor them. When the VMs are cloud bursted, we tune the monitoring agents to report data according to the destination cloud setting, e.g., using EC2's cloudwatch service.

5.5.4 Metadata Manager

We assume that all applications are a collection of VMs connected to each other over LAN. Each VM is of particular type, which is captured by a specific VM image, and an application may be composed of VMs with several different types. In addition to this, each application is either migratable or replicable. We capture all information about applications, hosts, and network configuration in a database.

In our prototype, we have implemented the metadata manager as a collection of python classes for each application, which store their data in the backend MySQL database. This offers the functionality of safe retrieval and updating of application metadata.

5.5.5 Workload Forecaster

The workload analyzer uses the workload statistics to derive estimates of future workloads. It obtains the application resource list using the application metadata and obtains the workload statistics from the *cloud management layer* for each of the application components. It then coalesces this to generate application level workload data. A *forecaster* module pulls this application level monitored data for forecasting the future application workload.

The design of Seagull is generic and any time series based forecaster [49],[105], conforming to the interface, can be used. In our current implementation, we have implemented an ARIMA *forecaster*. The ARIMA *forecaster* obtains a time-series of workload observations from the monitoring engine and models it as an ARIMA time-series. We use the ARIMA forecasting libraries of open-source statistical package R for generating the model and predicting the future peak workload.

5.5.6 Burst Manager

This is the core of Seagull that must i) find which applications/VMs to burst over to public cloud or which to bring back into the private cloud and ii) find which applications to precopy and schedule their periodic precopying operations. Using the workload forecast,

obtained from *workload forecaster*, and the application metadata, it runs the placement algorithm outlined in section 5.3. The output of the placement algorithm is a list of applications/VMs and their final destinations, i.e. either public cloud or hosts within the private cloud. If precopying is enabled, burst manager also computes a list of applications to precopy, using a precopying algorithm outlined in section 3.2, and creates a schedule to synchronize these applications to the cloud

We have implemented *burst manager* as a python daemon, which periodically wakes up and performs both the above mentioned operations. The periodicity is adjusted by a configuration file.

Actuator executes all the commands issued to it by the *burst manager*. Actuator takes the list of VMs from *burst manager* and calls the cloud bursting API exposed by the *cloud management layer*. It also executes periodic tasks of precopying issued by *burst manager*.

5.6 Experimental Setup and Evaluation

In this section we describe the experimental setup used to evaluate the performance of Seagull. We have created a private cloud environment on a lab cluster using OpenNebula [74]; for public cloud we have used Amazon EC2. We conducted experiments to illustrate the effectiveness of our algorithms and the intuition behind them. We also present an analysis of the costs and benefits when moving applications to the cloud and using precopying.

We have created a private cloud environment using two types of servers: 8-core 2GHz AMD Opteron 2350 servers and 4-core 2.4 GHz Intel Xeon X3220 systems. All machines run Xen 3.3 and Linux 2.6.18 (64bit kernel). We deployed OpenNebula on these machines to create a private cloud and manage a total of 44 cores distributed across 9 physical hosts. Our private cloud supports small, medium and large servers, comprising 1, 2 and 4 cores, respectively.

We have used Amazon EC2 as the public cloud environment in our experiments. EC2 offers two type of storage solutions for their instances, namely S3 and EBS. We have used

the latter for our experiments, primarily to simplify the implementation of replication of data across cloud boundaries.

5.6.1 Application appliances

We use three applications, TPC-W, Wikibooks and CloudStone for our evaluation. We have created private-cloud as well as public-cloud appliances for each of these three applications and their respective client applications. An appliance instance creates the virtual machine(s) which house the complete application. We warm up each application, using its clients, for two minutes before collecting data.

TPC-W is a multi-tier transactional web benchmark that represents an online bookstore [101]. We use the Java implementation of TPC-W which has two-tiers: a Web server tier based on Apache Tomcat servlet container and a database tier based on MySQL. In our appliance we have deployed both the tiers on the same VM.

Wikibooks is an open-content textbooks collection website for which an http replay tool has been developed [17]. It uses a MySQL database with a front-end PHP application. We have created two separate VMs for this application, one containing Wikimedia software and the other hosting the database.

CloudStone is a multi-platform, multi-language benchmark, which represents Web 2.0 applications in a Cloud Computing environment [95]. It implements a social online calendar as an AJAX application, called Olio. Olio uses MySQL as the backend database and supports a memcached tier which can be horizontally scaled. We use CloudStone both in a single VM deployment and in a multi-node, replicated setup. We again use the http replay tool as a workload generator.

5.6.2 Migration and Precopying Tools

This section evaluates the tools used by Seagull to burst applications to the cloud and perform precopying.

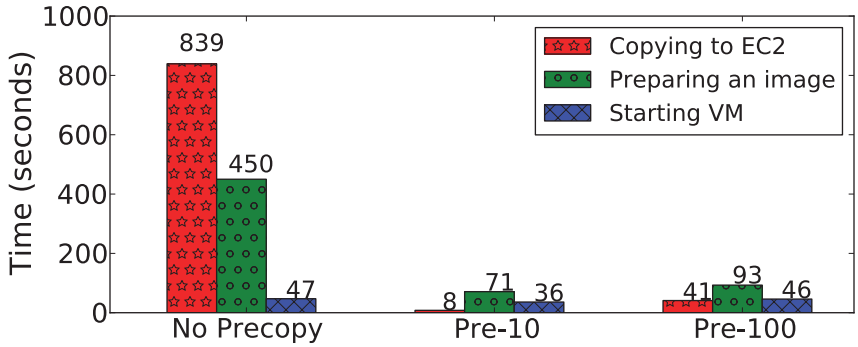


Figure 5.4: Impact of size of application on cloud bursting operation

5.6.2.1 Burst Operation Time Costs

We analyze the amount of time needed for each of the steps to burst an application to the cloud with and without precopy. The total cloud bursting time can be decomposed into three major parts: copying data to the cloud, preparing an application image and booting up the virtual machine. We migrate a virtual machine running the CloudStone application with a disk-state size of 5GB.

As shown in Figure 5.4, the total time to migrate an application with even a very small 5GB disk state, is 1336 secs ($\sim$ 22 mins); this clearly illustrates the need for precopying in real applications that may have ten or more times as much state. We then precopy the application and reduce the delta (i.e. difference between the original and precopied snapshot) to 10MB or 100MB; the total time to burst the application significantly reduces to less than 200 secs for a delta of 100 MB. Note that as delta reduces the image preparation time and boot time start to flatten around 120 secs and become the prime component of total bursting time.

5.6.2.2 Performance Impact of Precopying

To measure the impact of precopying on application's performance, we run a TPCW application and continuously precopy its data to the cloud during a thirty minute measurement interval – each precopying only takes 1-2 minutes. While the replication process is

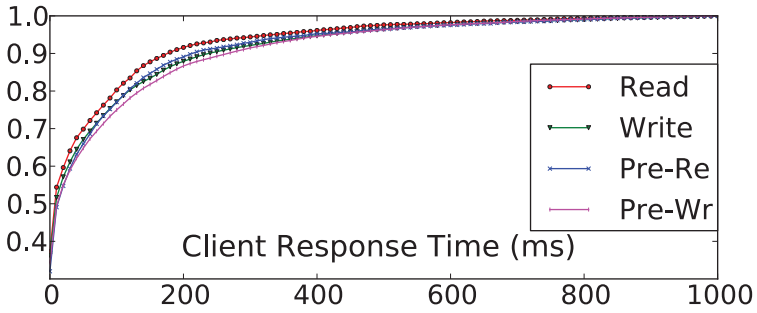


Figure 5.5: CDF of client response time with heavy workload.

running, we measure the response time observed by the TPCW clients running a "shopping" workload. We repeat this experiment ten times and report average statistics across these runs.

The response time performance for light (100 clients) and heavy workloads (600 clients) is shown in Table 5.1. When there is only a light workload, the average response time of all request types only increases by 2ms. When the workload is at peak capacity, the average response time of all requests increased by 19%, but write-requests observed a 37% increase. However, we observe in Figure 5.5 that 90% of write-requests see only a small performance change, i.e. response times rose from 300 ms to 350 ms with precopying (less than 17%).

5.6.2.3 Migration Downtime

We next study the application downtime when migrating applications over the WAN with Seagull. Our current implementation relies on non-live migration of VMs because existing clouds do not support live migration into their platforms. However, we can test what the performance of live migration might be by running a process on the cloud platform that receives a stream of Xen migration data from a private data center.

Our experiment migrates a VM running the TPC-W benchmark application, which is being accessed by a set of 200 clients running an ordering workload. The VM is configured

with 1.7GB of RAM and a 5GB disk. When using non-live migration, the application is inaccessible during the shutdown process at the origin site (1.2 secs), for the final copy of data to the cloud (7.0 secs), and while the application is reinitialized in the cloud VM (1.2 secs). In total, Seagull's non-live migration incurs 9.422 seconds of downtime. Note that a naïve approach to VM migration could increase this to a minute or more if the cloud VM was not booted until after the origin VM was shutdown. In comparison, running a live migration to EC2 incurs only 0.978 seconds of downtime while copying the virtual machine's memory. Using live migration does cause a slight increase in cost since more data must be sent; in total, the memory migration added 1.84GB of data transfer and required 236 seconds to run.

Conclusion: Precopying has only a modest impact on response time, but can dramatically reduce the total time required to burst an application to the cloud and reduces the downtime to less than 10 seconds.

TPC-W workload	READ		WRITE		ALL	
	None	Precopy	None	Precopy	None	Precopy
Shopping Light	29	31	21	25	28	30
Shopping Heavy	117	131	120	190	118	140

Table 5.1: Average client response time (ms) comparison for TPC-W in Shopping Mode

5.6.3 Placement and Precopying Algorithms

In this section we examine the algorithms used by Seagull to decide which applications to burst to the cloud and which are selected for precopying.

5.6.3.1 Placement Decisions

We first analyze the placement efficiency of Seagull compared to a naïve algorithm in a small scenario that demonstrates the intuition behind Seagull's decision making. We show

that when a hotspot occurs, Seagull is able to make better use of local resources as well as pick cheaper applications to move to the cloud.

We use three hosts of 6 cores, each hosting two applications, and three types of applications: TPC-W(A, D) Wikibooks (B, E), and CloudStone (C, F). Each application is running inside a single VM and we treat all applications in a scale up style. The initial arrangement of applications and the number of cores dedicated to each is shown under t_0 in Figure 5.6. To simplify the scenario, we assume that all applications have identical storage requirements.

We change application A's workload every hour (marked by instants t_i, where $i = 1 \ldots 3$) such that its CPU requirement increases to 4 cores, then 6 cores, before falling back to four cores at t_3. To eliminate the impact of prediction errors in this experiment we assume a perfect forecaster.

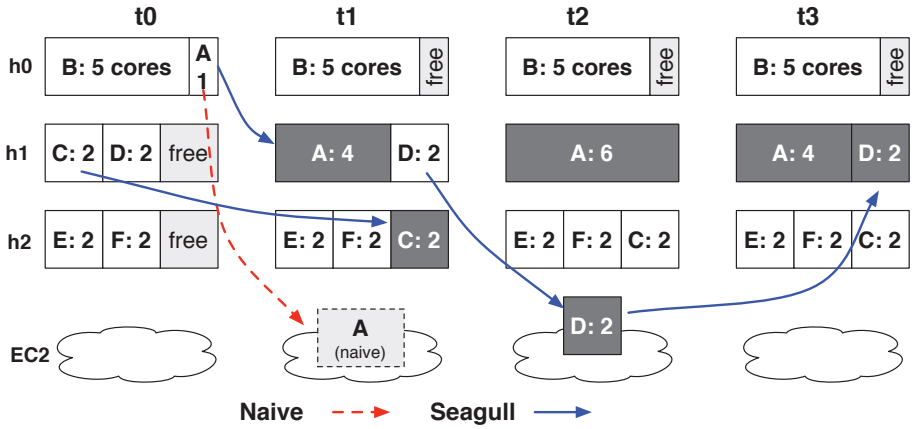


Figure 5.6: The naïve approach uses only one migration, immediately moving A from h_0 to the cloud. Seagull initially avoids any cloud costs by rebalancing locally, and is able to move back from the cloud sooner than the naïve approach.

When Seagull detects the first upcoming workload spike at t_1, it attempts to resolve the hotspot by repacking the local machines, shifting application C to h_2 and then moving A to h_1 at effectively no cost. In the naïve solution, application A is cloud burst to EC2 directly without considering local reshuffling.

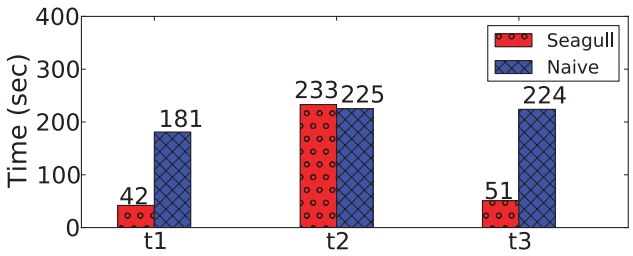


Figure 5.7: Seagull uses local, live migrations at t_1, and benefits from reverse pre-copying at t_3, substantially reducing the time spent at each stage compared to naïvely cloud bursting at t_1 and restarting instances at t_2 and t_3.

In the workload's second phase, Seagull migrates a cheaper application D to EC2 since the local data center could not provide enough capacity needed for A. On the other hand, the naïve algorithm had already moved A to the cloud, so it simply allocates extra resources to it making it more expensive.

Eventually, the workload spike for application A passes, Seagull migrates D back to the local data center while naïve algorithm, lacking the ability to perform local reshuffling, still needs to keep A in the cloud, consuming more money.

The use of local resources in Seagull allows it to respond to overload faster than the naïve approach. Figure 5.7 shows the amount of time spent by each approach to resolve the hotspots at each measurement interval; note that for both systems we precopy all applications once to the cloud before the experiment begins. Seagull is substantially faster because it uses only a local, live migration at t_1 whereas the naïve approach approach requires a full cloud burst. Subsequent actions performed by Naïve also incur substantial downtime since VMs must be rebooted in the cloud to adjust their instance type to obtain more cores. Seagull's migration back from the cloud at t_3 is also quite fast because it does not require the full image registration process needed for moving into the cloud. Most importantly, the fact that Seagull only requires a virtual machine in the cloud for the hour starting at t_2 means that it pays 30% less in cloud data transfer and instance running costs.

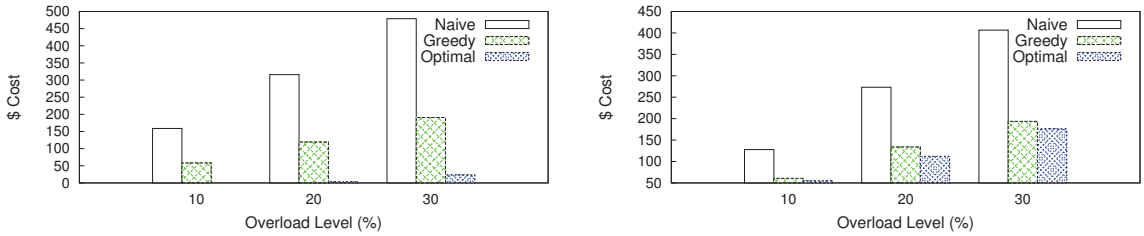
Conclusion: This experiment illustrates the intuition behind Seagull's placement algorithm. To find more capacity while minimizing the infrastructure cost and transition time, Seagull first tries to find free resources locally; if cloud resources are needed, then it moves the cheapest application possible to the public cloud.

5.6.3.2 Cost Efficiency

We study the difference in cost of our algorithm compared with the naïve approach and the optimal solution, described in section 5.2.3. We simulate a data center comprising of 100 quad-core hosts and test the strategies using three types of applications, namely small medium and large. We create a random set of applications using the categories provided in Table 5.2.

In order to evaluate the effect of local reshuffling, we fill the private data-center to approximately 70% of its compute capacity (in terms of compute cores). We then vary the data center workload by increasing the percentage of overloaded applications from 10% to 30%; each overloaded application observes an average compute capacity increase of 20 cores. For each overload situation, we compute the cost of cloud bursting decisions made by each of the three strategies, namely naïve, greedy and optimal (solution of ILP).

Figure 5.8a presents the performance of the three mentioned strategies averaged over 30 simulation trials. We compute the cost of cloud bursting by applying Amazon's resource pricing scheme in (5.7) and (5.8).



(a) With local reshuffling (b) Without local reshuffling

Figure 5.8: Comparison of average cost of cloudbursting with optimal

Figure 5.8 shows that the optimal solution offers a zero cost solution for 10% overload; our heuristic driven approach (greedy) offers an average of 100% improvement compared to naive approach, but is not very close to optimal to solution. The primary reason for a large disparity between optimal and greedy approach is because our heuristic performs a greedy search to obtain the local placement configuration; this leaves many possible local reshuffling options.To be sure of this, we conducted an experiment where we remove the possibility of local reshuffling. This will force a comparison of the other half of the algorithm, which is responsible for selection of applications for cloud bursting. We observe in figure 5.8b that the greedy approach provides nearly as good a solution as the optimal approach.

5.6.3.3 Precopying Efficiency

This experiment evaluates the effectiveness of Seagull's intelligent precopy strategy compared to the random (*SG-random*) and naïve precopy strategies (described in section 5.3.2) at a larger scale. We simulate a data center comprising of 200 quad-core hosts and test the strategies using three types of applications, namely small medium and large. We create a random set of applications using the categories provided in Table 5.2. To eliminate the impact of Seagull's local reshuffling on precopying efficiency, we assume that the data center runs only scale out applications, preventing the need for local reconsolidating. For computing the cost of each of these precopying strategies we use Seagull's placement algorithm.

App-Type	Reps	Active Disk Size	Image Size	Write Rate
Small	1-2	3-4GB	10G	1MB/min
Medium	3-6	6-8GB	15GB	2MB/min
Large	7-12	12-16GB	20GB	3MB/min

Table 5.2: Application Details

Again to eliminate the effect of forecasting errors we assume perfect workload forecasting with a 24 hour horizon, and perform precopying every hour. We study the decisions made when the level of overload in the data center increases from 10 to 30 percent. Figure 5.9 presents the average performance of these three strategies when the simulation is repeated 40 times for each level of overload.

In Figure 5.9(a), Seagull achieves the lowest precopying cost across all the overload levels. The benefits of Seagull increase with rising overload levels, and it is able to lower precopying costs by up to 75%. The naïve approach shows the highest cost because there are often applications which can be precopied more cheaply than those which are expected to become overloaded.

Figure 5.9(b) shows the total cost including both precopying and cloud bursting. Seagull reduces the cost by 45% compared to the naïve approach because running the overloaded applications in the public cloud is more costly. SG-Random and Seagull have similar total cost because they select the same applications to burst.

Figure 5.9(c) shows that our intelligent precopying strategy outperforms SG-Random because it has a poor chance of precopying the applications which will be selected by the placement algorithm. The naïve one behaves well because it selects the overloaded applications for precopying thereby substantially reducing the actual amount of data transferred during bursting.

Our evaluation demonstrates that precopying can significantly reduce burst time with minimal impact on application performance, however, we must also consider the monetary cost added by precopying. To study this, we consider the results of simulation similar to that described in the previous section. We increase the data center workload by overloading 30% of the applications in datacenter and compare Seagull with and without precopying.

Figure 5.10 shows a modest 22% increase in cost and a substantial 95% saving in data transfer by using precopying. Since the migration time largely depends on the amount of

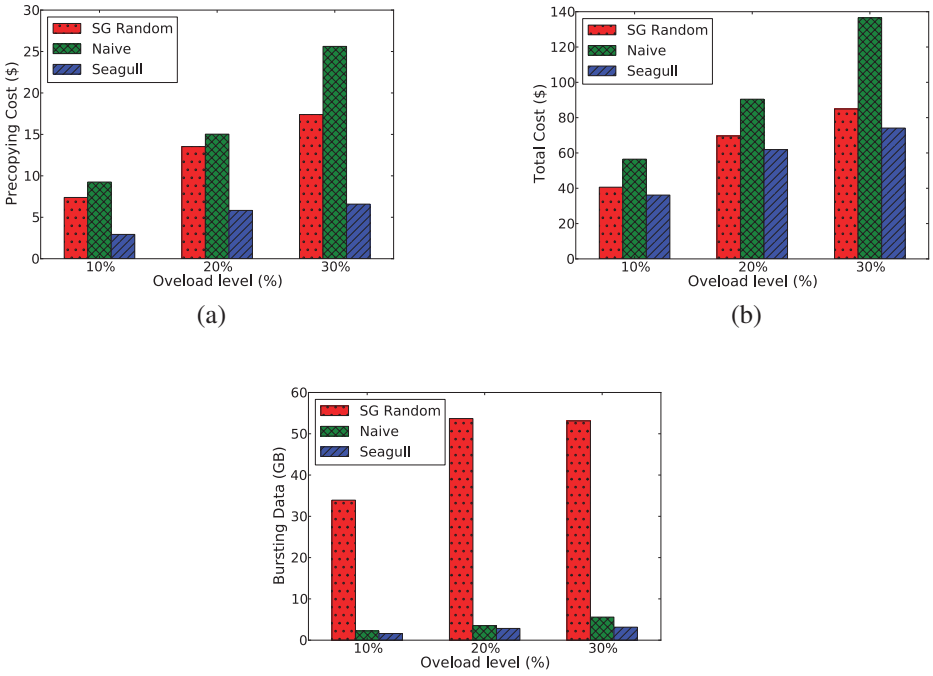


Figure 5.9: Intelligent precopying reduces total cost and data transferred by over 45% compared to the naïve algorithm.

data needed to transfer given the bandwidth, we conclude that our intelligent precopying strategy provides a reasonable tradeoff between cost and migration time.

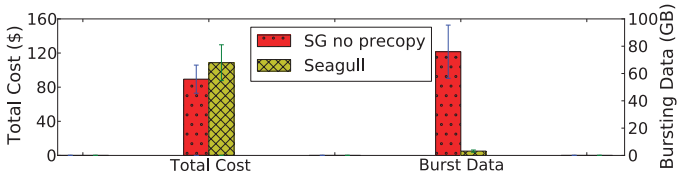


Figure 5.10: Precopying causes a marginal increase in cost, but a dramatic reduction in burst time.

Conclusion: Seagull reduces total cost by up to 45% compared to other precopying strategies while reducing data transfer cost by up to 95% compared to cloud bursting without precopying

5.6.3.4 System Scalability

We executed Seagull's heuristic driven application selection algorithm as well as the ILP solver to obtain the optimal solution on simulated data centers with variable numbers of hosts and applications (assuming that each application is completely packing in a single virtual machine). We again considered three types of applications as outlined in Table 5.2. We increased the complexity of application selection problem by increasing the number of hosts and proportionally the number of applications. We then measure the running time of the algorithms before they report the final solution. The results are plotted in Figure 5.11. It is apparaent that as the size of the problem increases the optimal algorithm becomes unusable. Seagull uses a greedy heuristic to make its placement decisions. While this leads to a non-optimal solution, it makes this multi-resource bin-packing problem more tractable.

Seagull is able to process data centers of 800 virtual machines within thirty seconds as compared to 7678 sec taken by the optimal. While large data centers may have many more hosts and virtual machines than this, we believe that Seagull can report a solution within a few minutes.

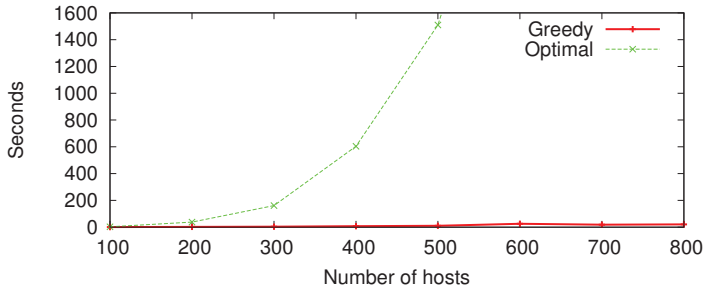


Figure 5.11: Scalability of the algorithm.

5.6.4 Multiple Overload Scaling

In this section, we show how Seagull deals with simultaneous overload from multiple applications that require both scale up and scale out strategies. We deploy seven web applications across the five hosts in our local cluster as shown in Figure 5.12(a). Two of

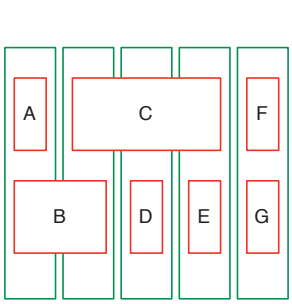
the applications, B and C, are VMs running the CloudStone application, which supports replication of its memcached layer. The other applications are scale up, and run different instances of TPC-W. The full details of each application's characteristics are shown in Figure 5.12(c). At time $t = 12$ minutes, both applications A and B become overloaded for a period of ten minutes; later at $t = 70$ minutes, application B sees a second spike that lasts until the end of the experiment.

Seagull uses workload forecast information to guide its precopying strategy, and immediately begins precopying Application A and F once the experiment starts. Throughout the experiment, we use xentop to gather the CPU utilization of Application A and B, which is illustrated in Figure 5.12(b), along with annotations indicating where each is located. The periodic spikes at ten minute intervals indicate the CPU overhead of precopying.

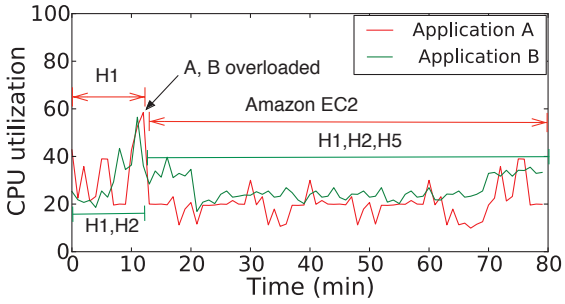
Seagull detects the overload condition expected to begin at $t = 12$ from A's and B's rising demands. It decides that bursting Applications A and F is the most efficient use of resources; since these applications have already been precopied, Seagull is able to burst both applications to the cloud within one minute. Moving A to the cloud allows it to start with a larger VM (a 4-core instance), causing its relative CPU consumption to decrease since it now has more resources available. Application B, running the replicable Cloudstone application, is able to expand its resource consumption in the local data center by spawning a new two core VM on host 5 using the resources freed up by Application F.

The applications continue running in this manner and the workload spike subsides. While it would be possible to immediately move the applications back to the local data center, EC2 charges in hourly increments, so there is no economic reason to do so. However, Seagull does continue to perform "backwards precopying" to replicate the state changes occurring to applications A and F in the cloud to the local data center. At $t = 70$ minutes, Seagull computes new placement decisions before the next hour of EC2 charges will begin. Seagull must plan for B's second workload spike, so it can only move either A or F back from the cloud. Since both are using the same cloud instance type, Seagull selects appli-

cation F to move back to the private data center as it has a slightly higher cloud cost than A. At the end of the 80 minutes experiment, Application A remains in the cloud to make space for overloaded Application B.



(a)



(b)

	A	B	C	D	E	F	G
Num of VMs	1	2	3	1	1	1	1
Cores	(2)	(2,2)	(2,2,2)	(2)	(2)	(2)	(2)
Active Disk Size (GB)	(1.2)	(6.8,4.6)	(4.6,3.2,4.4)	(6.2)	(3.2)	(2.8)	(5.4)
Image Size (GB)	(10)	(12,12)	(10,10,10)	(10)	(10)	(10)	(10)
Write Rate (MB/min)	(1)	(2,2)	(1,1,1)	(1)	(1)	(1)	(1)

(c)

Figure 5.12: **(a)** The initial set up of local data center. **(b)** The average CPU utilization of Application A and B over 80 minutes experiment. **(c)** Detail information of each application in the local data center.

5.7 Related Work

Cloud Bursting was first proposed by Amazon's Jeff Barr as a way to allow enterprises who already own significant amounts of IT infrastructure to still make use of the cloud during periods of high demand [27]. Researchers have been investigating the potential economic savings by using cloud bursting in specific domains such as medical image processing [57] and publishing [55]. Cloud Bursting generally assumes that a private data center is connected to a public cloud, producing what is known as a hybrid-cloud. Hybrid

clouds have become a popular service offering for hosting and data center companies, and also have been the subject of research [62, 96].

WAN Migration tools seek to move applications between data center sites with minimal downtime. Full VM WAN migration techniques such as [110, 69, 116, 12] attempt to seamlessly move the memory and storage of a virtual machine, usually by building upon the existing LAN migration tools included in Xen [26] and VMWare [71]. Alternatively, storage migration tools such as [61, 124] only focus on moving the disk state of applications. Since current cloud platforms typically do not support live VM migration into the cloud, our work focuses only on storage migration. We use a simple rsync-based replication scheme, but we note that Seagull could easily be enhanced to use more advanced migration tools, including support for full VM live migration.

5.8 Conclusions

Cloud bursting is a technique to dynamically move applications running in a private data center to the public cloud to take advantage of additional resources there. In this work we propose Seagull, a cloud bursting system that automates the decision processes about which applications can be run in the cloud most efficiently. Seagull uses selective precopying to proactively replicate some applications from the private data center to the cloud, reducing the migration time of large applications by orders of magnitude. This allows Seagull to perform agile provisioning of resources across a local data center and the cloud, resulting in more efficient utilization of local resources while incurring only minimal expense in the cloud. Our evaluation demonstrates how Seagull can burst applications to the cloud in under three minutes, while incurring only minimal performance overhead due to precopying. Seagull's placement algorithm considers both local reconsolidation opportunities and application cost characteristics, lowering the total cost of cloud bursting in response to data center overload by 45%.

CHAPTER 6

FLEXIBLE ADAPTIVE CONTROL PLANE FOR PRIVATE CLOUDS

Enterprises with existing IT resources are beginning to employ private clouds to manage them. But IT requirements grow with business needs and the cloud management system has to undergo a transformation to include this growth. This is a difficult challenge for private cloud administrators from a design as well as an execution point of view. It often demands reconfiguration of individual control plane services of the cloud management system so that each of them can sustain the increase in workload without violating their *Service Level Objectives* (SLOs). The control plane services support multiple configurations and manually deploying and managing the cloud management system is a challenging task; this is mainly because of the fact that it will involve configuring a large number of interdependent control plane services. In this work we design and implement a system to automate the process of deployment and reconfiguration of the cloud management system of a private cloud.

In the earlier chapters we presented methods of elastic provisioning of applications but all of them supported a single configuration, i.e. a three tier configuration. In this chapter we address the problem of automatic deployment and dynamic reconfiguration of control plane subsystems, which support multiple configurations, so that they conform to their *Service Level Objectives* (SLOs).

6.1 Background and Problem Description

In this section we provide the background information on private clouds and formulate the problem addressed in the chapter.

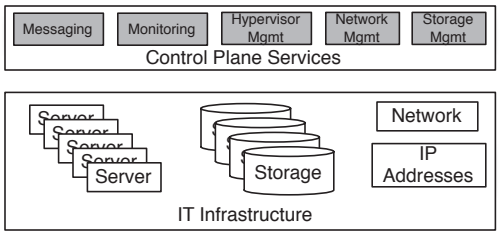


Figure 6.1: An example private cloud

6.1.1 Background: Private Cloud

A private cloud consists of infrastructure resources like compute storage and network and allows users to create virtual resources remotely on demand. Private clouds provide similar functionality like public clouds like Amazon Elastic Compute Cloud except that they are created using the infrastructure owned by the organization and are for its dedicated use unlike a public cloud, which leases resources.

A number of open source cloud management platforms are available to establish and operate a private cloud, namely OpenStack [79], CloudStack [77], Eucalyptus [73], OpenNebula [74] etc. These assume a cluster of linux machines and provide a control plane to manage the cloud infrastructure and perform management tasks, like hypervisor management, user management, messaging, monitoring, image management, etc. as depicted in Figure 6.1. Each such management task is performed by a control plane service that runs in one or more virtual machine. In this work we have chosen OpenStack as our cloud management system of study; this is primarily because it offers a rich set of control plane services and has become the most popular choice amongst the open source community [52].

6.1.2 Problem Formulation

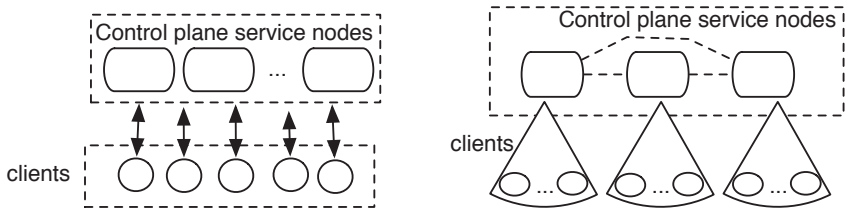
Consider an organization that wishes to deploy a private cloud on a cluster of size N. Most of the open source cloud management systems are designed to work with as few as tens of hosts/machines to very large clusters consisting of thousand machines but

for successful and efficient operation, the cloud management system has to be configured according to the size of the cluster. For this to happen, the administrator must configure each control plane service with sufficient capacity so that it can service the control plane workload generated by the management tasks in the cluster.

In the simplest case each control plane service runs as a single process on a single virtualized node. A single node per service setup is adequate for a small to medium size cluster. However, as the managed cluster's size grows, a single node setup will become a bottleneck. For instance, consider a monitoring service, which performs two major tasks, recording the monitored metrics for all resource as well serving queries regarding the same. The monitoring service, supported by a single node can handle the monitoring data for a cluster of size hundred, but if the cluster grows to a thousand machines, the amount of monitoring data that is generated by the clients will overwhelm a single node setup.

To scale the monitoring service we need to distribute the the workload across all the instances of the control plane service. This can be done in two ways: i) Employ clustering, where a group of replicas of the control plane service collectively serve the requests made to it. ii) Employ federation, which partitions the workload across multiple instances of the control plane service. In the clustering approach, all replicas collectively serve all the requests as a single logical entity – as shown in Figure 6.2a. In federation, each service instance services a subset of clients and forwards only the necessary requests to the other – as shown in Figure 6.2b.

Given such a cloud management system and the control plane services, an IT administrator is faced with the task of appropriately configuring each control plane service and ensuring that there is sufficient capacity to service the requests. Manual configuration and capacity allocation is a challenging task as a large number of interdependent services are involved. We, thus, have the problem of configuring and provisioning each control plane service so that the task of deploying the control plane service can be automated.



(a) Clustering (b) Federation

Figure 6.2: Clustering and Federated approaches

While there are rules of thumb on how to configure these control plane services, it is not apparent which approach to use to scale up and in which situation. In addition to this it is not obvious how many instance to provision for a particular size of managed cluster. Our approach is to design a flexible control plane service that automates this task by solving two sub problems: i) Given the size of managed cluster, say N, choose which approach is suitable, i.e. clustering or federation. ii) Provision sufficient capacity according the requirements of the adopted approach.

6.1.2.1 Dynamic Provisioning

Initial provisioning is based on an estimate of the workload likely to be seen by the control plane service. However, the workload observed by services may change over time either due to imperfect initial estimates of client workloads or due to incremental growth of the managed infrastructure or even a sudden change in managed workload; for instance the administrator may increase the monitoring resolution from 15 min to 1 min, causing an order of magnitude increase in the monitoring data. In such situations, some services required to be reconfigured by dynamically increasing (or decreasing) the capacity of the control plane service. The problem of dynamic reconfiguration is one where we automate the task of dynamically re-provision each control plane service to desired capacity.

6.1.2.2 System Model

The control plane services are assumed to be composed of multiple software components; these components can be deployed in dedicated virtual machines – we call them *component nodes* of a control plane service. In this work we assume that all the component nodes of a control plane service are identical (thus we also address a component node as a replica). This is not a limitation of our approach but a simplification, which we have adopted for ease of exposition of our solution. A fully functional control plane service is assumed to be created by arranging these component nodes in a single node, clustered or federated configuration – as shown in Figure 6.2. We assume that each component node has an associated SLO it and the administrator must pick a configuration and number of nodes such that there is enough capacity to serve the request seen by the service.

We assume that the private cloud management system has a monitoring service that keeps time-series of each monitored metrics and has a capability to report events like SLO violations.

6.2 Capacity Model and Empirical Profiling

Since each control plane service can be clustered, federated or a single node, we model each control plane service as a set of one or more identical components (referred as component nodes). A component node is assumed to service two types of requests, namely requests from infrastructure nodes or from other clients of the service and requests from the other component nodes of the same service – intra service messages. Let λ_c and λ_n denote the average workload due to requests of infrastructure nodes and other component nodes respectively. We also assume that each control plane service needs to meet a performance threshold to meet an Service Level Objective (SLO). SLO of a control plane can be specified using a threshold on application performance metric (e.g. latency) or on a resource utilization metric, for instance 80% of CPU utilization. Administrators try to provision sufficient resource capacity to a control plane service for keeping it from violating

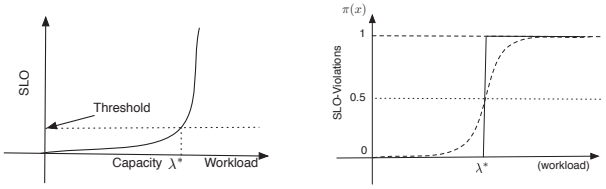
the SLO. Capacity estimation of a distributed system is a well known hard challenge and the challenge is intensified by the fact that software components behave differently in different configurations this is because different plugins are used to enable different configurations.

Our approach is to automate the task of configuring the control plane service by determining whether a single node or clustered or federated configuration is best suited for the control plane service and how many nodes are necessary to provide the desired capacity.

Our approach comprises of deriving an analytical model to determine the capacity needed and an algorithm to dynamically re-provision when the workload increase beyond the capacity.

6.2.1 Analytical model

Control plane service uses different resources, namely memory, CPU, network etc. The performance of a control plane service can be affected by many factors, including its configuration, workload variations, resource utilization, and also artifacts of the involved software components as well as those of the system. Our approach is to employ a probabilistic model to estimate the capacity needed by a control plane service to service a particular workload.



(a) (b)

Figure 6.3: Intuition of SLO violation curve

Let λ_T be the be the total estimated workload and let k be the number of replicas ($k \geq 1$) needed to service this workload. That means we must estimate the number of replicas k required by a control plane to service a workload of requests arriving at rate λ_T for a

given SLO. Our approach consists of gathering empirical data of SLO violations of each node/replica of the control plane service and use these observations to build a probabilistic model/function of SLO violations given the observed workload at the node. We then use this model/function to determine the max load λ_c^* that can be serviced by a single node; given this capacity of a single node, we can estimate the number of nodes, i.e. k, for a specific configuration (i.e. clustered, federated).

6.2.1.1 Formalizing the problem

The objective is to determine a function that relates λ to the SLO. More formally, let Y be a binary random variable, which represents presence/absence of an SLO violation and λ be the total workload observed by a node (i.e. $\lambda = \lambda_c + \lambda_n$). We, then, wish to estimate the conditional expectation of SLO violation, i.e. $E(Y|\lambda)$. There are a number of sophisticated non parametric techniques which can estimate conditional probabilities but these techniques often require a large amount of training data to create reliable models.

Logistic regression [50] is an alternative that does not require a large number of training samples to determine the conditional expectation.

Let $\pi(\lambda)$ denote the conditional expectation $E(Y|\lambda)$, when assuming a logistic distribution. The specific form of logistic distribution we use is:

$$\pi(\lambda) = \frac{e^{(\beta_0+\beta_1\lambda)}}{1 + e^{(\beta_0+\beta_1\lambda)}}, \tag{6.1}$$

where, β_0 is the intercept parameter and β_1s is the slope parameter. We re-write (6.1) to obtain a linear equation in λ:

$$g(\lambda) = \ln\left(\frac{\pi(\lambda)}{1-\pi(\lambda)}\right) = \beta_0 + \beta_1\lambda. \tag{6.2}$$

The parameters β_0 and β_1 can be estimated using logistic regression; they are maximum likelihood estimates of $\pi(\lambda)$ – expectation of SLO violation for a given λ.

Using (6.2), we can compute the value of λ for a given probability of SLO violation, say $=\lambda^*$. For instance, let us suppose we want to compute the capacity λ^* for a conservative threshold on probability of SLO violation, say 0.5; this essentially means that whenever $\lambda \geq \lambda^*$ there is more than 50% chance of SLO violation (as shown in Figure 6.3b). Thus equating $\pi(\lambda) = 0.5$ in equation 6.2 yields

$$\beta_0 + \beta_1 \lambda^* = 0 \tag{6.3}$$

Estimating β_0 and β_1 requires some real observations of workloads and SLO of a control plane node in a real setting. For that we perform offline empirical profiling of control plane services in different topologies as outlined in the next section.

6.2.2 Workload Estimation

The workload λ seen by a node of each control plane service has two components, namely requests from the clients (λ_c) and requests from the other replicas/nodes of the same service (λ_n), i.e. $\lambda = \lambda_c + \lambda_n$. Intra service workload (λ_n) is a function of client workload (i.e. $\lambda_n = f(\lambda_c)$) and the exact form of the function depends on the configuration.

We can use knowledge of the control plane to provide the function f. For instance, in a federated setup the clients are partitioned into smaller groups and each partition is serviced by one node/replica. Thus λ_n is a fraction of λ_c, i.e. $\lambda_n = \delta\lambda_c$. Similarly, in the case of clustering, the intra service workload depends on the size of the cluster and also on the way it has been implemented. This means that if a clustered configuration implements information exchange via broadcast then the messages received by each node will equal the size of the cluster; now, if the service uses multicast transmission to implement the same then only one message need to be sent, however,if the implementation adopts unicast transmission then the number of outgoing messages will be equal to the size of the cluster. Thus for a cluster of size n we will have $\lambda_n = 2(n-1)\lambda_c$ if the unicast is adopted, while in the case of multicast based implementation it will be $\lambda_n = n\lambda_c$.

On the other hand if nothing is known about the control plane service then we can treat the control plane service as a black box and estimate λ_n as function of λ_c by regressing over the empirical profiling data, i.e.

$$\lambda_n = \alpha_0 + \alpha_1 \lambda_c. \tag{6.4}$$

For the purpose of computing initial estimate of control plane's capacity, and also for performing empirical profiling, we require an estimate the client workload, i.e. λ_c, and the workload generated by a single client, say λ'_c. We make use of rules of thumb or prior experience for the same; for instance, if it is known that for each monitored machine a monitoring service records an average of 25 metrics at a granularity of 1-sec, then $\lambda'_c = 25$. Now, if the monitoring node services n clients then the average client workload observed by a single monitoring node will be $\lambda_c = n \times 25$ and the total client workload observed by the whole monitoring service for a cluster of size N will be $\lambda_T = N \times \lambda'_c$.

6.2.3 Provisioning Algorithm

Having modeled the control plane service and estimated the workload parameters, we compute the number of nodes required for service as follows:

Step 1: We compute a conservative capacity of a control plane node, i.e. λ^*, using logistic regression, i.e. (6.3).

Step 2: Next we estimate the maximum client workload a control plane node can service, say λ^*_c. Since λ_n is a known function of λ_c, we obtain the estimate by solving the following equation for λ^*_c:

$$\lambda^*_c + f(\lambda^*_c) = \lambda^*$$

Step 3: We estimate the capacity of a control plane service, i.e. total number of control plane nodes (say k), required to service a cluster of size N, i.e. $k = \lceil \lambda_T / \lambda^*_c \rceil$.

Step 4: We provision the estimated capacity of control plane service in a particular configuration.

6.3 Dynamic Reconfiguration

The initial configuration was based on an estimate of the workload before the cluster was even deployed. These capacity estimates need to be redefined after the actual observation of changes in workload or that in the control plane operations/settings. For instance, if the administrator changes the monitoring resolution from 1-min to 5-sec, the monitoring service needs to re-compute the new capacity to faithfully monitor and record the data.

We call this as the dynamic reconfiguration of the control sub-system; this involves two steps: i) *When* to trigger dynamic reconfiguration? ii) *How* to migrate from current configuration to new one?

When to trigger reconfiguration? We trigger re-provisioning in one of two ways, namely i) reactively, i.e. when we observe SLO violations, and ii) proactively, i.e. we forecast the workload and if the forecasted workload changes more than a threshold then we trigger re-provisioning.

- *Reactive*: In the situations when we observe SLO violations, we perform two type of reactive provisioning: i) We increment the capacity of the control plane service by a fixed amount. For instance, when we observe sustained SLO violations, we can increase the control plane's capacity by single node. ii) We learn new model for the configuration; for this we clean the monitored time series data of SLO and λ obtained from each node and append it to the empirical profiling data and then re-estimate new βs of (6.3) using logistic regression.

- *Proactive*: We use an ARIMA based time-series forecaster to predict the future workload at a node; the approach is similar to what is adopted in [105]. Specifically, we obtain a time-series of workload observations and model it as an ARIMA time-series. The forecasted workload allows us to plan a transition to a new configuration when it detects that the capacity of the current configuration may get exceeded.

- *Computing new configuration*: We re-use the *provisioning algorithm* outlined in section 6.2.3 for computing a new configuration but with an updated workload estimate.

How to migrate to new configuration? There are two main steps in migrating the control plane service from old configuration to new configuration, namely i) redeployment and ii) redistribution of the clients across the new configuration.

- *Redeployment*: There are two approaches to deploy the newly computed control plane service configurations: i) Full redeployment and ii) incremental. Full redeployment is useful when the control plane service will need to change its configuration from clustered to federated or vice-versa. In most of the other cases we perform incremental, where-in if the number of control plane nodes/replicas has change from k to k', we just provision the difference and set them up to communicate with the rest of the replicas. This might require stopping the service for a short period of time.

- *Redistribution*: If clients are connected to the control plane service nodes via a loadbalancer, we simply change the loadbalancer configuration to make it aware of the new replicas. However in certain cases, clients are directly connected to the control-plane service replicas and in that situation, first, we compute the increase in capacity in terms of number of client nodes, say δn, next we pick the top p maximally loaded replicas and take $\delta n/p$ clients from each of them and evenly distribute them across the new replicas. We assume that services nodes hold their state in a common shared repository which is over provisioned.

6.4 Prototype Design and Implementation

This section provide details of the prototype implementation carried out for enabling flexibility and adaptiveness to control plane services of a private cloud management system. Functional block diagram of our prototype is shown in Figure 6.4. We have used OpenStack for creation of private cloud management system.

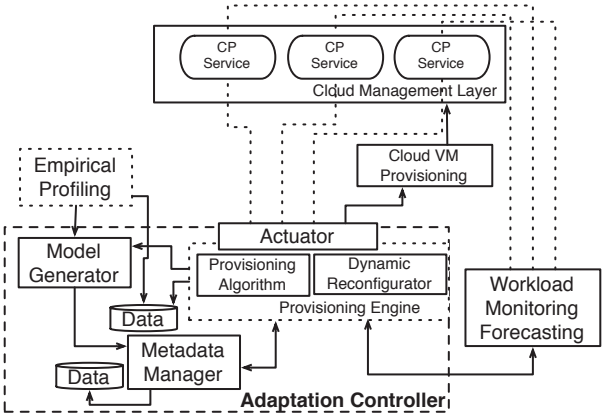


Figure 6.4: Logical architecture of the prototype

6.4.1 System Model

: Following the brief description of the various functional blocks of our prototype and also some details of their implementation. We have implemented all the modules in Python.

- *Model Generator*: It takes the empirical profiling data for each control plane service and generates a model (set of βs) and hands it over to the *Metadata manager* for storing it. We have used the STATA 10's implementation of logistic regression [13] to obtain our models.

- *Metadata manager*: It essentially stores and retrieves all necessary metadata details. It stores models for each control plane service, their current configuration and capacities. We have implemented this as a python class, which stores all the information in in-memory data structures.

- *Workload Monitoring and Forecasting*: It collects time-series monitoring data of all the virtual machines as well as of those of the control plane services. It stores all the results in a database, which can be queried. We have implemented this as a part of monitoring service of OpenStack using Ganglia. We have used STATA 10 for implementing the ARIMA forecaster [119].

- *Configuration and Provisioning Engine*: It implements the provisioning algorithm. It takes the generated model from *Metadata manager* and computes the number of replicas needed for a configuration. In case of change in configuration, it provisions new replicas using the *Actuator* module and updates the details of new configuration to Metadata manager. It also performs *dynamic reconfiguration* by constantly evaluating the SLO metric and by computing the change in average client workload. As a solution to the less frequent situation where the model requires re-learning, the *provisioning engine* queries and collects the cases of SLO violations and updates the learning data. It then re-estimates the model parameters and updates the records in *metadata manager*.

- *Actuator*: This is module is a part of configuration and provisioning engine. It performs the task of deploying new virtual machines of each control plane service. After deployment it executes the necessary scripts in each replica of the control plane service for creating the correct configuration. The actuator also looks up the dependent clients and alter's their configuration so that the client workload is evenly distributed across all the replicas. It essentially keeps a fixed number of clients for each replica.

6.4.2 Private cloud management system

We have used OpenStack as our cloud management system. It has four main components, namely, compute (Nova), image repository (Glance), authentication (Keystone), and swift (Storage). The nova, glance, and keystone provide the service of hypervisor management, image management and authentication respectively, while swift provides and object-store service [76]. We use *Nova* as an example to expose some of the design details of scalable cloud service components. Nova has multiple control plane services which together provide the the functionality of compute and volume storage management. Nova's various services communicate with each other via AMQP [108]. In our setup we have used the open source implementation of AMQP, namely RabbitMQ [88].

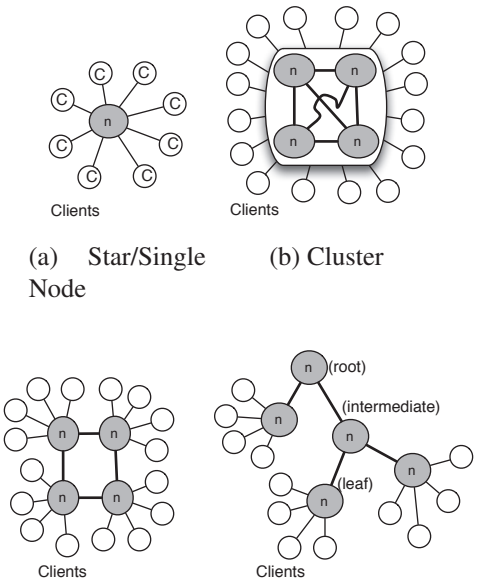
But OpenStack does not have a full fledged monitoring service; we thus have built a full fledged monitoring service using Ganglia and Nagios [54]. Ganglia helps in monitoring and storing the data in the database and Nagios, helps in putting simple triggers in the DB; for instance, we use Nagios to report if the average memory utilization of a bunch of nodes is over 300 MB.

6.4.3 Empirical profiling

As the first step towards learning the parameters we perform empirical profiling of each control plane service in both clustered and federated pattern types. In general, a configuration topology is a graph that is created using replicas of component nodes. In order to empirically profile a component nodes of a service in a particular configuration, we start with the smallest possible graph of that configuration and systematically increase the client workload on the service (i.e. λ_c) until we observe SLO violations; at each step we record the average intra service workload as well (i.e. λ_n). We, then, repeat the same procedure with the next bigger graph of the same topology type and so on.

- *Single node*: this is the smallest configuration for all types of topology – Figure 6.5a. We assume the average workload generated by a single client, i.e. λ_c and for the purpose of increasing λ_c we simply increase the number of clients until we observe SLO violations.

- *Cluster configuration*: We start with a cluster of size 2 and distribute evenly clients, say $2m$, between the two nodes. We, then gradually increase the workload on both the nodes, distributed evenly, until we observe SLO violations on any of the nodes. We measure both the average number of client requests per sec, λ_c, and the average number of intra service requests per sec, λ_n. We repeat the same experiment with larger cluster size. This helps us gather data necessary for capturing the impact of cluster size as well.



(a) Star/Single Node (b) Cluster

(c) Federated - Ring (d) Federated - Heirarchical

Figure 6.5: Example initial configurations; grey nodes represent control plane service nodes and the white nodes are its clients

- *Federated configuration*: In this configuration type, the clients are partitioned between different component nodes of the service, but the service nodes themselves could be arranged either in a tree based hierarchical manner or in a non-hierarchical fashion (for instance a ring). For nonhierarchical kind of federated configuration, the method of profiling is like that of clustered configuration, i.e. we start with smallest possible configuration and profile till it is saturation and then increase the number of component nodes by one and repeat the same procedure.

In the case of hierarchical configuration, we have to differentiate between nodes, i.e. leaf nodes, intermediate nodes and root node. This makes the profiling a little more involved. We start with a tree of depth=1, i.e. with a root node and single leaf node; thus to make life easier we have profiled assuming that the in a hierarchical topology only leaf nodes and root node will have client workload and not the intermediate nodes. Similar to cluster topology we increase the client workload on the leaf node while keeping the client

workload of root node to zero until leaf node's saturation. We keep the root node's client workload to zero and gradually increase the λ_n by adding more leaf nodes. We repeat the same experiment with a tree of depth two, where there is a leaf node and intermediate node and then a leaf node. We empirically profile the intermediate node.

6.5 Case Study: Monitoring Subsystem

The monitoring subsystem offers one of the core services for cloud management. Almost all system management analytics, namely elasticity, problem determination performance monitoring etc. are dependent on it. Essentially, a monitoring system collects utilization and performance metrics of various system resources like compute nodes, storage devices, network resources etc and persists them. It also offers a system to query and retrieve the recorded information.

At a conceptual level a monitoring system needs minimally the following three systems i.) *storage system*: which stores the monitored data for posterity ii) *querying system*: which provides methods to query the stored data and iii) *collection system*: a method of collecting monitoring information. For the monitoring system to be able to scale, each of these sub-systems should scale individually. There are a number of commercial and open-source software systems that support scalable monitoring systems; for instance IBM Tivoli monitoring system, Supermon [98], CARD [4] etc.

We have used Ganglia as our choice of monitoring subsystem. Ganglia also has a monitoring agent, *gmond*, which listens and broadcasts monitored data using UDP multicast/unicast. The monitored data is pushed to a metadata server, *gmetad*, for persisting; *gmetad* stores data in a Round Robin Database (RRD) [78] and leverages *rrdtool* to extract and graph the monitored data using Apache web-server and php technology. Unlike many of the open source solutions, ganglia uses listen/announce protocols and automatic discovery

of cluster membership, which affords no manual configuration for addition of new member or metric. Ganglia supports a federation in a hierarchical manner[1].

Ganglia uses UDP to transfer data from client to server. Consequently, message delivery is very unreliable. This lack of reliability is particularly prominent in a reasonably loaded datacenter. This makes ganglia a very relevant system to stress test our approach, particularly for testing our approach to dynamic provisioning. We noticed that the Ganglia monitoring suffers data loss either due to unreliable message delivery or because of software artifacts. We, thus, have defined SLO of our monitoring subsystem as a threshold on the percentage of data loss, i.e. data loss is less than 10%. To test the efficacy of our approach in a real life setting we conducted our experiments on Amazon EC2.

In the following sections we describe the experimental setup for conducting experiments on various configuration patterns.

6.5.1 Experimental Setup

We have used Ganglia 3.3.6 over Ubuntu 12.04 machines for creating a *monitoring node* by deploying both a *gmetad* and a *gmond* daemon on it. This server is responsible for gathering all data from the monitored nodes. Client workload generated by a single client (i.e. λ'_c) is dependent on number of metrics being monitored and the frequency at which they are monitored. We have considered three types of monitoring workloads for our experiments. Each of these workloads involve monitoring 25 metrics but at different monitoring frequencies, i.e. 1-sec, 5-sec and, 15-sec.

We generated client workloads by simulating monitored clients via a data generation daemon in python. It sends gmond 2.x data packets to gmond. For each monitored node (i.e. client machine, labelled as *gmond* in Figure 6.6a) we executed our simulator 25 times to send 25 separate metrics at the pre-configured monitoring frequency. On the *monitoring*

[1]Ganglia creates one rrd file for each metric of each node; which means that if there n nodes and m metrics then Ganglia will generate/create $n \times m$ rrd files on the metadata server.

node The metrics are saved in separate files and folders, where files are named using the metric's name and the folder is named using the host name.

6.5.1.1 SLO metric

Ganglia is designed to use UDP to transfer monitoring metrics to *gmond*. These metrics are collected by *gmetad* using a TCP connection with *gmond*. We have chosen *data-error percentage*, i.e. percentage of data lost per unit of monitoring time, as our SLO metric. For measuring the capacity of Ganglia monitoring subsystem we have used 10% as the threshold for our SLO-metric, which means that we consider more than 10% data-loss as an SLO violation.

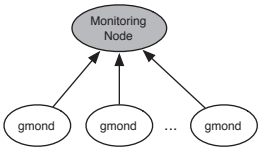
6.5.2 Empirical Profiling and Capacity Estimation

In this section we empirically profile Ganglia based monitoring subsystem in two different configurations, namely, single-node and federated (nodes deployed in a hierarchy). We then use the profiling data and the results developed in Section 6.2 to compute the maximum capacity of a configuration. Finally, we test our dynamic reconfiguration approach on monitoring subsystem deployed in a federated topology.

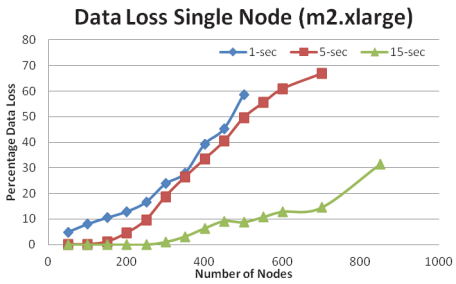
6.5.2.1 Single Node Configuration

We created a single node configuration, as shown in Figure 6.6a, by using the *m2.xlarge* instance of EC2 as our *monitoring node*. Ganglia metadata node has a software setting to alter the monitoring granularity of the metadata server. To show the efficacy of our approach we conducted empirical profiling and capacity estimation of single-node Ganglia setup in three different monitoring granularities, i.e. 1-sec, 5-sec and 15-sec. We conducted three different profiling experiments and trained three different models (one for each monitoring granularity).

We have simulated n_c client nodes by sending the monitored data of $n_c \times 25$ metrics[2] to the *monitoring node* from 5 client machines running the client workload simulator. The λ_c observed by the monitoring node in the three respective monitoring granularities is $n_c \times 25$, $5 \times n_c$, and $5 \times n_c/3$. The observed data-loss at the *monitoring node* for each of the three different monitoring granularities is recorded for training the model. The SLO plots for each of the experiment are shown in Figure 6.6b, where each point on the graph is an average of more than 50 samples.



(a) Configuration

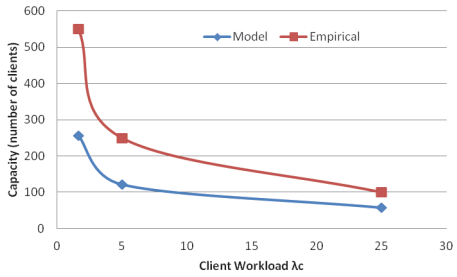


(b) Data loss

Figure 6.6: Data loss in a single node configuration

We used the complete profiling data to compute the capacity model of a single node configuration in three different settings. We estimate the parameters of the model using logistic regression and compute capacity using (6.3). Figure 6.7b summarizes the results of profiling of a single node configuration with the three different monitoring workloads as a table.

[2]Amazon cloudwatch monitors 25 metrics for an instance and its volume



(a) Estimated and observed capacities as a function of λ_c

	1-sec ($\lambda_c = 25$)	5-sec ($\lambda_c = 5$)	15-sec ($\lambda_c = 5/3$)
Capacity	58	122	256

(b) Estimated Capacities

Figure 6.7: Empirical and estimated capacities of single node monitoring configuration with monitoring node on an *m2.xlarge* instance type.

It can be seen from Figure 6.7a that capacity does not vary linearly in λ_c and that the model provides a conservative estimate of capacity with the data generated by empirical profiling.

Conclusion: Empirical profiling effectively captures the software artifacts. In addition the model allows us to capture that knowledge and generate conservative estimates.

6.5.2.2 Federated configuration

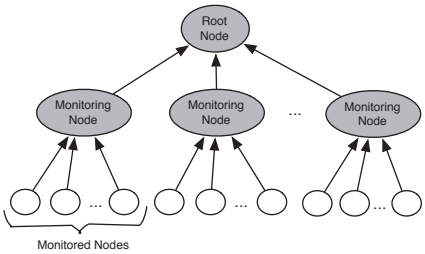
For a federated configuration we conducted an experiment with a tree of depth two, as shown in figure 6.8a (which means a tree of depth one for control plane nodes). In this configuration there are multiple monitoring nodes each of which gathers the data from their individual group of monitored nodes, called clusters. It is often useful for administrators to have a summary of monitored metrics at cluster level. Ganglia collects summary statistics at the monitoring node, which monitor another monitoring nodes; in our case it will be the root monitoring node. The root node pulls the summary statistics data periodically after t-seconds which can be set using a config-file. This places additional load on the leaf monitoring nodes and thus impacts data loss.

In a hierarchical configuration there are three types of nodes, namely a leaf metadata node (subjected to both client and intra-service workload) and the other is a root node (with only intra service workload); we profiled each of these nodes. For leaf metadata nodes, we generated the client workload in the same manner as for single-node configuration profiling. However, for generating intra service workload (λ_n), we setup the root node to pull data from the leaf metadata nodes at three different granularities, i.e. 15-sec, 30-sec and, 1-min. This is because the higher level nodes in the tree collect only summary statistics of the lower level nodes and thus the resolution is often quite low. For each resolution, we measure SLO while gradually increasing λ_n. The variation in the SLO metric with increase in workload, for both leaf as well as root metadata node, is shown in Figure 6.8b and 6.8c respectively. We have used an average workload of 25 metrics per monitored client, thus $\lambda_c = 25/t_l$, where t_l is the monitoring granularity of the leaf metadata node. Similarly average workload generated by a leaf metadata node for its parent node is $\lambda_n = 150/t_r$, where t_r is the monitoring granularity of the leaf metadata node. We conducted nine profiling experiments and developed capacity models for each of them.

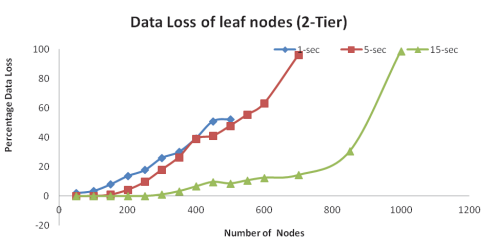
Like the single node setup, we compute the capacity of a federated pattern by assuming 10% as the maximum allowed data loss. As expected, we find that the data loss characteristics of the leaf monitoring nodes are very similar to those of a single node configuration except only slightly less (shown in Table 6.1a). However, as the root metadata node's monitoring granularity increases to 30-sec and 60-sec the impact becomes nearly negligible.

For estimating the capacity of a tree of deputy on, we estimated capacity for both leaf and root metadata node for each of the nine empirical profiling experiments (shown in Tables 6.1a and 6.1b respectively). We then evaluated the capacity of a tree topology type (with depth one) by assigning maximum number of child nodes which each service node can handle.

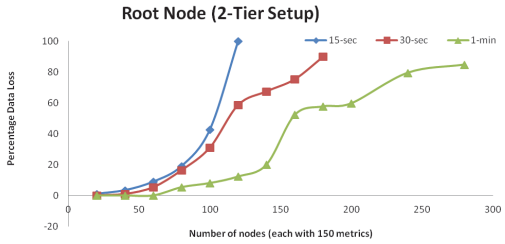
Table 6.1c summarizes the maximum capacities of a tree topology of depth one with nine different settings of monitoring granularities. The total capacity of each of the nine



(a) A federated configuration



(b) Data loss on leaf monitoring node; root node monitoring at 15-sec granularity



(c) Data loss on root monitoring node

Figure 6.8: Data loss in a federated configuration

configurations is computed by multiplying capacities of leaf and root nodes. This is because of the fact that we assume $\lambda_n = 150/t_r$ (a constant). Note that an approximate functional form of $\lambda_n = f(\lambda_c)$ is estimated using the knowledge of the monitoring service. Since the root node collects only the averaged values from each child metadata nodes, it computes to $\lambda_n = \lambda'_c \times 6/t_r$.

Conclusion: Empirical profiling assists in capturing the application artifacts. This coupled with our modeling approach helps in estimating maximum capacity of any configuration.

6.5.3 Adaptation of Monitoring Subsystem Model

In certain situations the initial estimate of node capacities and hence the capacity of a configuration could be quite off from the the actual values needed in real deployment. This, either, could be because of change in hardware configurations or new version of softwares. The actual values needed might require re-computation of the model. To showcase

RootNode/LeafNode	15-sec $\lambda_n = 10$	30-sec $\lambda_n = 5$	1-min $\lambda_n = 2.5$
1-sec ($\lambda_c = 25$)	56	58	58
5-sec ($\lambda_c = 5$)	118	133	133
15-sec ($\lambda_c = 1.67$)	272	284	285

(a) Leaf metadata node capacity

	15 -sec	30-sec	1-min
Capacity	28	32	55

(b) Root metadata node capacity

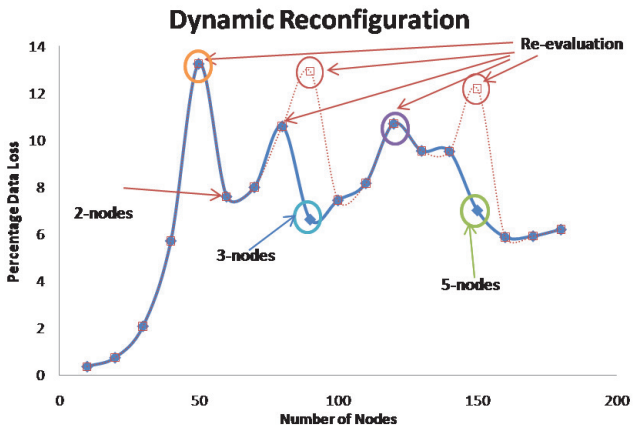
RootNode/LeafNode	15-sec	30-sec	1-min
1-sec	1568	1856	3190
5-sec	3304	4256	7315
15-sec	7616	9088	15675

(c) Capacity of a hierarchical configuration pattern

Table 6.1: Empirical capacity of federated monitoring configuration deployed as a tree of depth of two; monitoring node on an *m2.xlarge* instance type.

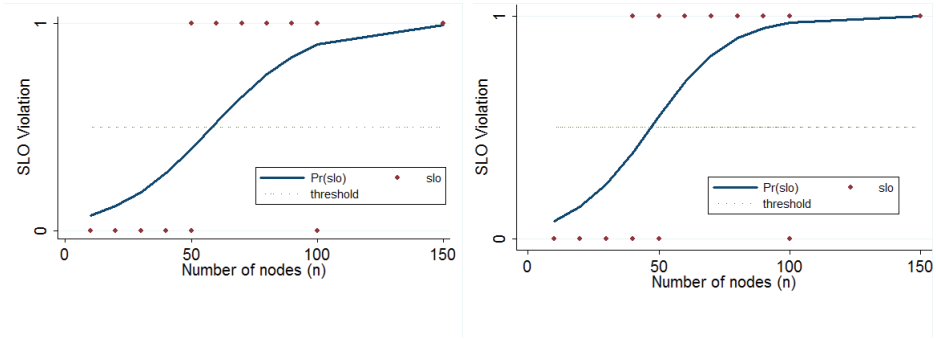
a scenario of dynamic reconfiguration of the model we conducted an experiment with a federated topology type, more precisely, we used a tree with one root metadata node (n_r) and one leaf metadata node (n_1). We started with the capacity models of both root and leaf metadata nodes, obtained in the previous experiment. The initial estimate results in maximum client capacity of 56. We conducted the following experiment: i) We started with the two node tree with 10 clients attached to the leaf metadata node. The leaf metadata node was configured to monitor at 1-sec monitoring granularity, while the root metadata node at a 15-sec granularity. ii) We gradually increased the workload in units of 10 clients (i.e. $n_c+ = 10$). We then learn a new model with the new data of SLO violation. For increasing the capacity of the system we adopted the mechanism outlined in Section 6.3.

Our reconfiguration process triggered when the n-c connections reached 50 nodes because the SLO got violated (shown in Figure 6.9a). The new capacity turned out to be 55 client nodes, the reconfiguration algorithm returned the same topology back (shown in Figure 6.9a. When the workload was made 60 it automatically triggers re-computation as



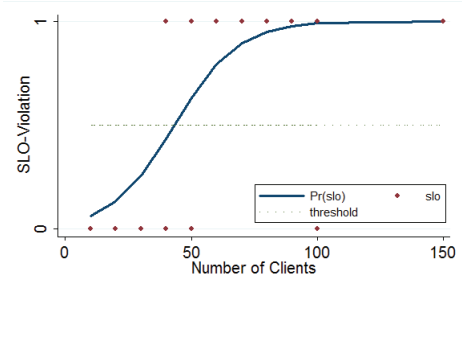
(a) Dynamic reconfiguration

the workload has exceeded the capacity of the configuration. The new configuration at this point contains a second leaf node (n_2). We keep 55 clients assigned to n_1 for monitoring and the remaining are assigned to n_2. As we gradually increased the workload to 80 client nodes, the SLO violation again exceeded 10% and on re-evaluation lead to a new maximum capacity of 47 client nodes for each leaf node (shown in figure 6.9b). With this evaluation the algorithm believes that the system is within capacity and we increased the capacity to 90 clients. Due to SLO violation yet another re-evaluation happened and the new capacity computed was 44 (shown in figure 6.9c) and this caused a new topology with an addition of third node n_3. At this point each of the first two nodes account for 44 clients while n_3 gets two clients and SLO violations dropped below threshold. Next re-evaluation happened at 120-client nodes again because of SLO violations and the new capacity for leaf metadata nodes that was computed to be 38 client nodes (shown in figure 6.9d). Due to rebalancing the average SLO violations go down but at 150 requests the 5th revaluation leads to the final capacity of 30 client nodes for each leaf node and the SLO violations reduce to around 6%. The initial model which we started with was off by 45%. Our approach is able to learn the correct model quickly without interfering with the model for other granularities.

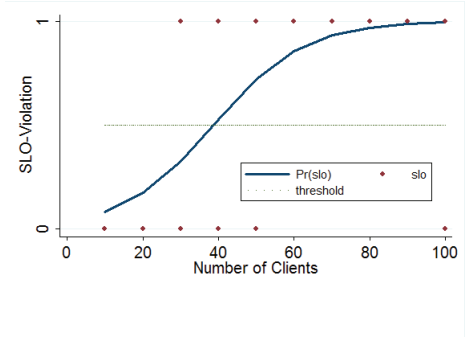


(a) First violation (cap=55)

(b) Second violation (cap=47)



(c) Third violation (cap=42)



(d) Fourth violation (cap=30)

Figure 6.9: Dynamic scaling and adaptation of capacity rule

Conclusion: i) Dynamic adaptation is effective. ii) Logistic regression based model is able to quickly learn new parameters and configures the system with correct topology. iii) Models for specific workloads can be simple but effective.

6.6 Case Study: Messaging Subsystem

Distributed systems communicate with different components synchronously or asynchronously by invoking remote objects transparently. A messaging system offers a scalable backbone service to a distributed system for synchronous as well as asynchronous communication.

In a message queuing based architecture distributed components communicate by publishing messages in specific source queues. The messages hop across multiple communication servers before eventually getting delivered to its destination queue(s).

There are a large variety of commercial and open source message queuing systems, e.g. IBM Webshpere MQ, Microsoft Message Queuing, RabbitMQ, Apache Active MQ, Sun Opend Message Queue, Apache Qpid etc. A messaging system, essentially, has three main components: i) *Clients* which post and receive messages by registering specific queues ii) *Local queue managers*, which manage the queues of clients that are directly connected to them and iii) *message routers or exchanges*, which forward the incoming messages to other exchanges or to destination queues. Exchanges help in building a scalable message-queuing system. However, there are multiple configurations in which these exchanges can be arranged to scale the queuing network.

This case study is on OpenStack's message queuing subsystem, that forms the backbone of this scalable private cloud management system. All the components of the compute cloud of OpenStack (i.e. Nova) communicate with each other via blocking and non-blocking RPC calls using AMQP [109].

In the following sub-sections we describe the architecture of OpenStack and then our experimental setup and finally experiments and results on the messaging subsystem of OpenStack.

6.6.1 OpenStack Messaging Subsystem

Openstack's compute cloud is called Nova. Its a distributed system whose sub-components communicate with each other using AMQP [109] protocol. A high level architecture of Nova[3] is shown in Figure 6.10

[3]In Essex release of OpenStack nova-console, nova-cert/objectstore and nova-consoleauth modules also leverage AMQP for communication. We have not considered their workload in our simulations but it should be noted that they introduce a nearly insignificant load on the message queuing subsystem

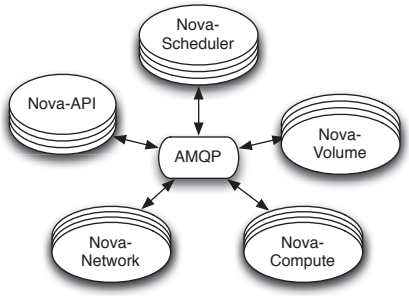


Figure 6.10: OpenStack Nova components

Each of the OpenStack components communicate over AMQP using an exchange named *nova*. Following is a brief description of these components, which is relevant to our work:

- *Nova-api*: This is the external facing component of OpenStack. It exposes interfaces for clients to be able to initiate most of the resource management/query activities. It, essentially, takes the client requests, marshals them and posts it on to the scheduler-queue.

- *Nova-scheduler*: It determines which instance executes where. There could be multiple schedulers in an OpenStack installations and each scheduler registers two queues on the nova-exchange, namely scheduler and scheduler.hostname. Scheduler, essentially, dequeues tasks posted on the shared scheduler queue or on its specific queue, i.e. scheduler.hostname.

- *Nova-compute*: this is a worker process which runs on host machines and handles the task of creation and termination of virtual machines. There are multiple nova-compute instances, one running on each compute node, and each of them registers two queues at the nova exchange, namely compute and compute.hostname. Nova-compute dequeues its tasks from the exclusive queue, compute.hostname.

- *Nova-network*: it accepts the tasks from the queue regarding allocating and setting the network of the provisioned virtual machine. OpenStack supports a single network-

node mode as well as a multi-network node mode. Each network node registers two queues at the exchange, namely, network (shared) and network.hostname (exclusive)

- *Nova-volume*: it handles the tasks of creation, attaching and detaching of persistent volumes to compute instances. Like compute, network and scheduler, it also registers two queues at the nova exchange, namely volume (shared) and volume.hostname (exclusive). It initiates its processes after receiving requests on the shared queue and exclusive queues.

6.6.2 Workload Simulator

OpenStack supports two open source implementations of AMQP, namely RabbitMQ and Apache Qpid. We have used RabbitMQ as the messaging service subsystems and instead of Kombu client library[4] we have used Pika version 0.9 client library to communicate with RabbitMQ. We simulate the message sequences and RPC calls of each of the OpenStack components.

- *Empirical Profiling*: For each configuration of RabbitMQ server configuration we increase the number of compute nodes and volume nodes gradually. We assume that each host (nova-compute) will receive a VM create and terminate request per hour per core; associated with it will be volume creation and termination request. We assume a host of 64 cores, which means 64 VM and volume creation and deletion requests per host per hour.

- *SLO metric*: OpenStack's components do not inject a lot of messages into the messaging system and neither are they very sensitive to the delay in their delivery. Openstack's components start observing trouble in their operations, when the messaging system stops publishing messages. This happens when RabbitMQ reaches its maximum allowed memory utilization limit. In this work we have kept SLO as when

[4]OpenStack uses Kombu but it does not support federation of exchanges in RabbitMQ.

average memory utilization shoots over 75% of the maximum allowed memory; for instance, if 400MB of memory is allotted to RabbitMQ, then we consider a SLO violation when memory utilization increase to a value more than 300MB.

- *Data collection*: We collect the memory utilization as well as message latency for our study. For memory utilization we use an admin tool, which reports current memory utilization of RabbitMQ server. Although we have not used message latency in this study, we collect it for our future work. For collecting message latency, we synchronized the clock on each server and each message is time-stamped by the publisher. The client receives the message and logs the performance metric, i.e. message latency. We have equipped each nova component with a TCP-socket based logger and created a logger-server for recording the logs at a single place for ease of processing. We compute the average latency by processing this central log.

6.6.3 Experimental Setup

We have used RabbitMQ 2.8 for our experiments. We experimented on a private cloud created over 12 Intel Xeon (X3430) machines each with 8 GB RAM and 500 GB SATA Disk. The machines were installed with Ubuntu 12.04 and we created the private cloud using OpenStack Essex release. Each RabbitMQ node possesses both queue-management capabilities as well as router capabilities. They just need to be configured in a particular manner to create different configuration topologies. We have installed them on single core VMs with 6GB of RAM. The clients were deployed on the hosts described above. On each VM as well on each host we set the limit to number of open files to 81920 (80 K). In order to synchronize the clock we used NTP 4.2.6.

6.6.4 Empirical Profiling and Capacity Estimation

In this section we perform the empirical profiling of RabbitMQ based messaging system in two different topologies, namely single-node and ring.We will use the offline profiling

data and the results developed in Section 6.2 to compute the maximum capacity of a topology. We will finally test our dynamic reconfiguration approach on monitoring subsystem deployed in a federated topology.

We assume that each client, i.e. Nova-compute and Nova-volume generate one VM and volume creation and deletion request every hour. We have assumed that each client node is of 64 cores and thus each node create one VM as well as volume creation request every second. Since a VM creation requires 5 messages and VM-deletion requires 6 messages and equal for volume creation and deletion, thus $\lambda'_c = 22$.

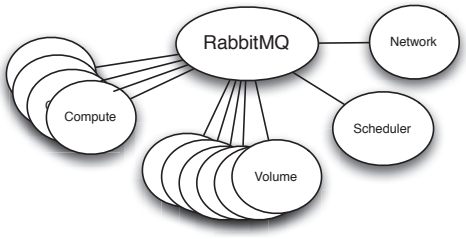
6.6.4.1 Single Node Configuration

We conducted the experiment with a single node configuration by using a single core VM but with amount RAM to RabbitMQ, i.e. starting from from 400MB to 2.4GB. We gradually increase the number of compute and volume nodes, which increases the message traffic via the message queue. We, then, measure memory utilization of the RabbitMQ node. The results are shown in Figure 6.11.

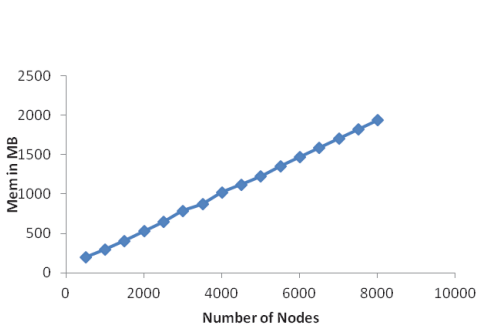
In each experiment we gradually increased the number of compute nodes, keeping a single scheduler and a single network node. Increasing scheduler and network is not a recommended configuration in openStack.

In each experiment we scaled up the number of clients in batches of 250 clients. We stop adding clients when the memory utilization reaches 75% of the total allowed memory to RabbitMQ server. We found that the memory utilization linearly increases with number of clients, as shown in figure 6.11b. To generate a model of single node configuration, we conducted experiments where we varied the RAM from 400MB to 2400MB and the results are shown in figure 6.11c. It can be observed that the capacity scales linearly with RAM for workload generated by OpenStack clients. The model also captures the same.

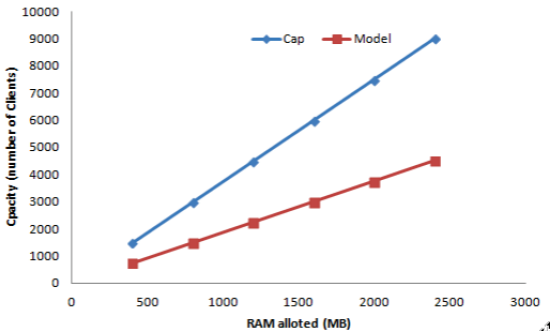
Thus we conclude that for a fixed size RAM the capacity in terms of number of client nodes is a constant, for all practical purposes, i.e. 750 clients for 400MB of RAM.



(a) Single node setup



(b) Memory utilization



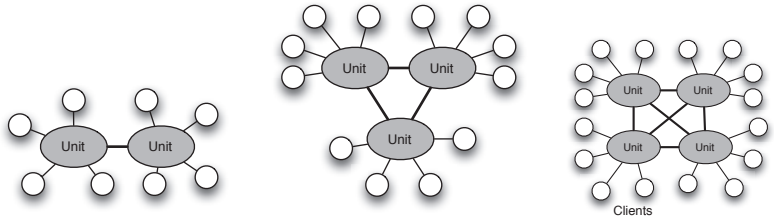
(c) Capacity

Figure 6.11: Memory utilization and average message latency observed in a single node configuration of RabbitMQ

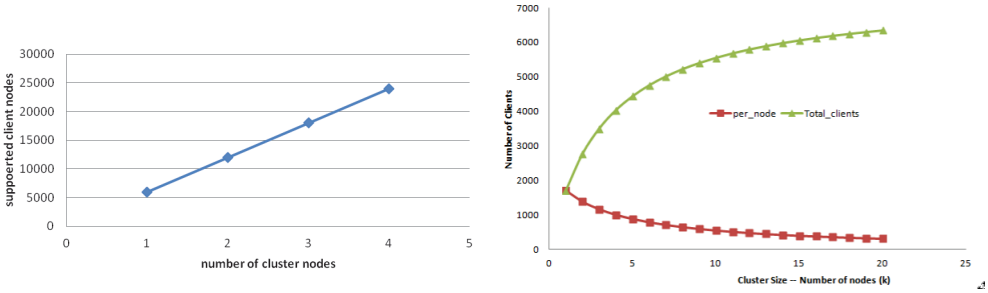
6.6.4.2 Cluster Configuration

RabbitMQ supports a clustering configuration, where each broker node in the cluster has all a replica of all the data necessary for operation. This means that any queue can be accessed from any broker node, however, the queues and its messages are not replicated and thus it saves unnecessary excess communication. We experimented with three types of cluster node configurations and the results are shown in Figure 6.12.

We conducted two set of experiments: First with by hosting RabbitMQ on a single core VM but with 2.4GB RAM, second with a RabbitMQ server with single core VM and with 0.4GB RAM. For both the experiments we created cluster of different sizes and for each such cluster, we increase the number of clients gradually till SLO violations were observed. The second experiment was conducted with a limited RAM to study the asymptotic behavior of the configuration and also to test if the model can capture this knowledge faithfully.



(a) Two Node (b) Three Node (c) Four Node



(d) Configuration Capacity (RAM=2.4GB) (e) Configuration Capacity (RAM=0.4GB)

Figure 6.12: Various cluster configurations and their empirically estimated capacity.

In the case of clustering configuration, the intra service workload (i.e. λ_n) scales linearly with λ_c. The linear function is such that it also depends on the size of cluster. So we estimated the following function from our empirical profiling data $\lambda_n = \alpha_0 + k\alpha_1\lambda_c$.

We estimate the capacity of the clustered setup using our logistic regression. Results of empirical observations for the cluster with 2.4GB RAM are shown in figure 6.12d. We observed that this data creates a model which depicts a linear growth. To study the impact of increasing cluster size on the capacity we conducted the same experiment but with much less amount of RAM to the RabbitMQ nodes, i.e. 0.4GB. Since λ_n is not linearly dependent on λ_c (because cluster size k is also a variable), we ran a multiple logistic regression with λ_c and λ_n as our independent variables and SLO as the dependent variable. Capacity in terms of number of clients which a node can handle reduces to the following form: $\lambda *_c = 22.25/(0.01 + k \times 0.003)$. We plot the capacity for each k using this result and the capacity curve is shown in figure 6.12e. The figure depicts that the capacity of a clustered

configuration starts to saturate as cluster size increases. Thus after some point in time it will not be useful to scale using clustered configuration.

Conclusion: Capacity of a clustered configuration starts to saturate as the size of the cluster increases. Also the model provides better estimates of SLO violation with more number of independent parameters, namely λ_c and λ_n.

6.7 Related Work

6.7.1 Cloud Benchmarking

Many researchers have conducted empirical evaluation of cloud platforms; Researchers in [9] and [33] benchmark Amazon EC2 to quantify CPU, disk and network performance of the provisioned virtual machines. Sharada et al. in [11] evaluate different virtualization technologies by running database workloads in a virtualized environment. Cooper et al. in [29] propose benchmark for the data storage subsystems popular in clouds, namely Hadoop, Cassandra, HBase and compare their bench marking results with shared MySQL implementation.

6.7.2 System Performance Modeling

Generic system performance model based on ensemble of tree augmented bayesian networks has been developed by Zhang et al in [123] to capture the performance behavior of a system application under changing workload conditions. Watson et al. in [114] develop a probabilistic performance model for virtual machines with the objective of capturing the effect of statistical multiplexing in clouds and impact of other measurable factors to provide performance guarantees expressed in percentiles. Our work in this chapter is closest to this approach, however, unlike them we use a simple model and also support dynamic update of model for adjusting to dynamic changes in workload.

6.8 Conclusion and Future Work

Scaling the control plane services of a cloud management system is a challenging task because of diversity involved in the type of systems. We present a system, which adopts a control-plane of control plane services and present an approach to perform empirical profiling of the control plane services. We also develop a performance model which leverages the data from empirical profiling to determine the initial configuration of the service. We present a four-step mechanism to compute for new configuration and provision it. We present a method of dynamically scaling the control plane in accordance to the dynamism of the workload using either a clustered or a federated approach. We have developed a prototype for monitoring subsystem and tested it on public cloud (i.e. Amazon EC2) with two different topologies, namely single-node and federated. We also developed a prototype of messaging subsystem of OpenStack and tested our methodology using three different configuration, namely star, cluster and ring. Our experiments indicate that initial configuration could be far from actual configuration but we converge to the desired configuration in a few iterations.

Logistic regression gives better models with more number of uncorrelated independent metrics. Thus as a future work, we plan to extend the model to include other system metrics, namely CPU and Disk and conduct a study again on some of the control plane services to test its efficacy.

CHAPTER 7

SUMMARY AND FUTURE WORK

7.1 Thesis Summary

This thesis has explored *how complex systems/applications can be made elastic on a cloud computing platform*. We have proposed models and algorithms to automate the process and have demonstrated their efficacy by building prototypes in both public as well as private cloud computing environments.

7.1.1 Cost Aware elasticity

As the first part of the thesis, I presented a system and methodology for performing *cost aware* elasticity of a cloud application on any IaaS cloud platform. The approach optimized for infrastructure cost or transition latency (or a weighted combination of both) while ensuring that application's SLA is not violated. The evaluation demonstrated that, even in a modest size setup, our system afforded as high as 24% higher cost savings as compared to the cost oblivious approaches. On the other hand our system demonstrated an ability to perform elasticity in times two orders of magnitude smaller than the transition cost oblivious approach.

7.1.2 Planning for the Tail

As the second piece of this thesis, I presented a model for computing capacity of a multi-tier cloud application when the SLA is expressed as a threshold percentile of end-to-end response time. The model was extended to account for the non-linear pricing in cloud platforms and also to account for the heterogeneity present in cloud platforms. Our system

is able to make the multi-tier cloud application elastic with as high as 80% savings in cost. The study showed that its better to use bigger servers on clouds than smaller servers for high percentile provisioning.

7.1.3 Hybrid Cloud

Next, I described a new dynamic provisioning system, *Seagull*, which supports dynamic provisioning of enterprise applications in a hybrid cloud environment. It automates the decision about which applications can be run in the cloud most efficiently. It leverages selective pre-copying as an optimization for reducing the migration time of large applications by orders of magnitude. Our evaluation demonstrated how Seagull can burst applications to the cloud in very short times, while incurring only minimal performance overhead due to precopying.

7.1.4 Flexible Adaptive Control Plane for Private Clouds

Finally, I presented a generic methodology to generate a topology for any control plane service of a cloud management system. I developed a logistic regression based generic model and used it to generate specific models for each kind of topologies. I empirically profiled various topologies for two core control plane services and used this data to train the model. I also presented a dynamic reconfiguration mechanism to assist the control plane service in adapting to wrong configuration and workload changes. Our experiments on both private and public cloud indicate that our model is able to suggest reasonable initial configurations for control plane services for a given SLO. Also in situation where the initial configuration was far from satisfying the SLO, the dynamic reconfiguration algorithm quickly converged to suggest the correct configuration.

7.2 Future Work

In this section we discuss some future research directions that have emerged from this dissertation.

- **Cost Aware elasticity:** *Kingfisher* , currently works in a single cloud environment, however it will be very useful to extend this work to incorporate hybrid cloud environments. This would require factoring in the application's limitations and also performance degradation caused by spreading the workload across different clouds platforms.

- **Hybrid Clouds:** Seagull has methods and mechanisms for supporting applications in a hybrid cloud environment. It can be extended to support cost efficient disaster recovery and resiliency to applications which guarantees a particular RTO and RPO.

- **Flexible Adaptive Control Plane for Private Clouds:** In our current solution the we have developed an SLO model of each node based on the number and type of requests processed by the node. However, the model will be more robust and effective if we could base it on more number of relevant metrics. But in any large distributed system, the number of metrics could easily become very large and thus the challenge would be automatically identify the relevant system metrics and train the model on those metrics.

BIBLIOGRAPHY

[1] Abdelzaher, Tarek F., Shin, Kang G., and Bhatti, Nina. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems 13*, 1 (2002), 80–96.

[2] Airoldi, Edoardo M., and Carley, Kathleen M. Sampling algorithms for pure network topologies: a study on the stability and the separability of metric embeddings. *SIGKDD Explor. Newsl. 7*, 2 (Dec. 2005), 13–22.

[3] Al-Fares, Mohammad, Loukissas, Alexander, and Vahdat, Amin. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74.

[4] Anderson, Eric. Extensible, scalable monitoring for clusters of computers. In *Proc. 1997 Large Installation System Administration Confere (LISA XI* (1997), pp. 9–16.

[5] Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D., Katz, Randy, Konwinski, Andy, Lee, Gunho, Patterson, David, Rabkin, Ariel, Stoica, Ion, and Zaharia, Matei. A view of cloud computing. *Commun. ACM 53*, 4 (Apr. 2010), 50–58.

[6] Armbrust, Michael, Fox, Armando, Griffith, Rean, Joseph, Anthony D, Katz, Randy H, Konwinski, Andrew, Lee, Gunho, Patterson, David, Rabkin, Ariel, Stoica, Ion, and Zaharia, Matei. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb. 2009.

[7] AWS Economics Center. http://aws.amazon.com/economics/.

[8] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebuer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtulization. In *Proceedings of the 19th SOSP* (2003).

[9] Barker, Sean Kenneth, and Shenoy, Prashant. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems* (New York, NY, USA, 2010), MMSys '10, ACM, pp. 35–46.

[10] Bennani, Mohamed N., and Menasce, Daniel A. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 229–240.

[11] Bose, Sharada, Mishra, Priti, Sethuraman, Priya, and Taheri, Reza. Performance evaluation and benchmarking. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess, Eds. Springer-Verlag, Berlin, Heidelberg, 2009, ch. Benchmarking Database Performance in a Virtual Environment, pp. 167–182.

[12] Bradford, Robert, Kotsovinos, Evangelos, Feldmann, Anja, and Schiöberg, Harald. Live wide-area migration of virtual machines including local persistent state. In *VEE* (San Diego, California, USA, 2007), ACM, pp. 169–179.

[13] Buis, M. L. predict and adjust with logistic regression. *Stata Journal 7*, 2 (2007), 221–226(6).

[14] Buyya, Rajkumar, Ranjan, Rajiv, and Calheiros, Rodrigo N. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *International Conference on Algorithms and Architectures for Parallel Processing* (2010).

[15] Cain, Harold W., and Rajwar, Ravi. An architectural evaluation of Java TPC-W. In *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture* (2001), pp. 229–240.

[16] Case Study: USA.gov Achieves Cloud Bursting Efficiency Using Terremarks Enterprise Cloud. http://bit.ly/ua5Qq2 .

[17] Cecchet, Emmanuel, Udayabhanu, Veena, Wood, Timothy, and Shenoy, Prashant. BenchLab: An Open Testbed for Realistic Benchmarking of Web Applications. In *Proceedings of 2nd USENIX Conference on Web Application Development (WebApps)* (June 2011).

[18] Cerbelaud, Damien, Garg, Shishir, and Huylebroeck, Jeremy. Opening the clouds: qualitative overview of the state-of-the-art open source vm-based cloud management platforms. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (New York, NY, USA, 2009), Middleware '09, Springer-Verlag New York, Inc., pp. 22:1–22:8.

[19] Chase, Jeffrey S., Anderson, Darrell C., Thakar, Prachi N., Vahdat, Amin M., and Doyle, Ronald P. Managing energy and server resources in hosting centers. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 103–116.

[20] Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., and Doyle, R.P. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 103–116.

[21] Chen, Jin, Soundararajan, Gokul, and Amza, Cristiana. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *ICAC* (June 2006), pp. 231–242.

[22] Chen, Shigang, and Nahrstedt, Klara. Hierarchical scheduling for multiple classes of applications in connection-oriented integrated-service networks. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2* (Washington, DC, USA, 1999), ICMCS '99, IEEE Computer Society, pp. 9153–.

[23] Chen, Y., Paxson, V., and Katz, R. What's New About Cloud Computing Security. *University of California, Berkeley Report No. UCB/EECS-2010-5 January 20*, 2010 (2010), 2010–5.

[24] Cherkasova, L., and Phaal, P. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers 51*, 6 (June 2002), 669–685.

[25] Chohan, Navraj, Bunch, Chris, Pang, Sydney, Krintz, Chandra, Mostafa, Nagy, Soman, Sunil, and Wolski, Rich. Appscale design and implementation, 2009.

[26] Clark, C., Fraser, K., Hand, S., Hansen, J. G, Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of NSDI* (May 2005).

[27] Cloudbursting - hybrid application hosting. http://aws.typepad.com/aws/2008/08/cloudbursting-.html, Aug. 2008.

[28] Coffmann, E. G., Gary, M. R., and Johnson, D. S. Approximation algorithms for bin-packing-an updated survey. *Algorithm Design for Computer System Design* (1984), 49–106.

[29] Cooper, Brian F., Silberstein, Adam, Tam, Erwin, Ramakrishnan, Raghu, and Sears, Russell. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[30] Crovella, Mark. Performance evaluation with heavy tailed distributions. In *Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools* (London, UK, 2000), TOOLS '00, Springer-Verlag, pp. 1–9.

[31] Csirik, J., Frenk, J. B. G., Labbé, M., and Zhang, S. Heuristics for the 0–1 min-knapsack problem. *Acta Cybern. 10*, 1-2 (1991), 15–20.

[32] DeCandia, Giuseppe, Hastorun, Deniz, Jampani, Madan, Kakulapati, Gunavardhan, Lakshman, Avinash, Pilchin, Alex, Sivasubramanian, Swaminathan, Vosshall, Peter, and Vogels, Werner. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev. 41* (October 2007), 205–220.

[33] Dejun, Jiang, Pierre, Guillaume, and Chi, Chi-Hung. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 2009 international conference on Service-oriented computing* (Berlin, Heidelberg, 2009), ICSOC/ServiceWave'09, Springer-Verlag, pp. 197–207.

[34] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev. 19*, 4 (Aug. 1989), 1–12.

[35] Dowsland, Kathryn A., and Dowsland, William B. Packing problems. *European Journal of Operational Research 56*, 1 (1992), 2 – 14.

[36] Doyle, R., Chase, J., Asad, O., Jin, W., and Vahdat, Amin. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USITS* (Mar. 2003).

[37] Duda, Kenneth J., and Cheriton, David R. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *SIGOPS Oper. Syst. Rev. 33*, 5 (Dec. 1999), 261–276.

[38] EC2, Amazon. Amazon ec2. Website. `http://aws.amazon.com/ec2`.

[39] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. Cluster-based scalable network services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP'97), Saint-Malo, France* (December 1997), pp. 78–91.

[40] Ganglia monitoring system. `http://ganglia.sourceforge.net/`.

[41] Goyal, Pawan, Vin, Harrick M., and Cheng, Haichen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw. 5* (October 1997), 690–704.

[42] Grit, Laura, Irwin, David, , Yumerefendi, Aydan, and Chase, Jeff. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *In the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)* (November 2006).

[43] Gujarati, D.N. *Essentials of Econometrics*. McGraw-Hill higher education. McGraw-Hill Education, 2009.

[44] Gulati, Ajay, Shanmuganathan, Ganesha, Ahmad, Irfan, Waldspurger, Carl, and Uysal, Mustafa. Pesto: online storage performance management in virtualized datacenters. In *SOCC* (New York, NY, USA, 2011), SOCC '11, ACM, pp. 19:1–19:14.

[45] Gupta, Diwaker, Cherkasova, Ludmila, Gardner, Rob, and Vahdat, Amin. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware* (New York, NY, USA, 2006), Middleware '06, Springer-Verlag New York, Inc., pp. 342–362.

[46] Hamilton, James R. Architecture for modular data centers. *CoRR abs/cs/0612110* (2006).

[47] Haproxy the reliable, high performance tcp/http load balancer. `http://haproxy.1wt.eu/`.

[48] Harrison, Peter G., and Knottenbelt, William J. Passage time distributions in large markov chains. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2002), ACM, pp. 77–85.

[49] Hellerstein, J., Zhang, F., and Shahabuddin, P. An Approach to Predictive Detection for Service Management. In *Proceedings of the IEEE Intl. Conf. on Systems and Network Management* (1999).

[50] Hosmer, David W., and Lemeshow, Stanley. *Applied logistic regression (Wiley Series in probability and statistics)*, 2 ed. Wiley-Interscience Publication, 2000.

[51] Iyer, Sitaram, and Druschel, Peter. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. *SIGOPS Oper. Syst. Rev. 35* (October 2001), 117–130.

[52] Jiang, Qingye. Open Source IaaS Community Analysis. `http://www.qyjohn.net/?p=2233`.

[53] Jones, Michael B., Roşu, Daniela, and Roşu, Marcel-Cătălin. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 198–211.

[54] Josephsen, David. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[55] Kailasam, Sriram, Gnanasambandam, Nathan, Dharanipragada, Janakiram, and Sharma, Naveen. Optimizing service level agreements for autonomic cloud bursting schedulers. In *ICPP Workshops* (2010), pp. 285–294.

[56] Katz, R. H. Tech titans building boom. *IEEE Spectr. 46*, 2 (Feb. 2009), 40–54.

[57] Kim, Hyunjoo, Parashar, Manish, Foran, David J., and Yang, Lin. Investigating the use of autonomic cloudbursts for high-throughput medical image registration. In *GRID* (2009), IEEE, pp. 34–41.

[58] Kundu, Sajib, Rangaswami, Raju, Gulati, Ajay, Zhao, Ming, and Dutta, Kaushik. Modeling virtualized applications using machine learning techniques. *SIGPLAN Not. 47*, 7 (Mar. 2012), 3–14.

[59] Kernel Based Virtual Machine. http://www.linux-kvm.org/page.

[60] Lim, Harold C., Babu, Shivnath, and Chase, Jeffrey S. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing* (New York, NY, USA, 2010), ICAC '10, ACM, pp. 1–10.

[61] Mashtizadeh, Ali, Celebi, Emré, Garfinkel, Tal, and Cai, Min. The design and evolution of live storage migration in vmware esx. In *USENIX ATC* (Berkeley, CA, USA, 2011), pp. 14–14.

[62] Mateescu, Gabriel, Gentzsch, Wolfgang, and Ribbens, Calvin J. Hybrid computing-where hpc meets grid and cloud computing. *Future Gener. Comput. Syst. 27* (May 2011), 440–453.

[63] Matthews, Jeanna Neefe, Hu, Wenjin, Hapuarachchi, Madhujith, Deshane, Todd, Dimatos, Demetrios, Hamilton, Gary, McCabe, Michael, and Owens, James. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[64] Maxemchuk, Nicholas F., Ouveysi, Iradj, and Zukerman, Moshe. A quantitative measure for telecommunications networks topology design. *IEEE/ACM Trans. Netw. 13*, 4 (Aug. 2005), 731–742.

[65] Mell, Peter, and Grance, Tim. Effectively and Securely Using the Cloud Computing Paradigm, May 2009.

[66] Menasce, D. Web Server Software Architectures. In *IEEE Internet Computing* (November/December 2003), vol. 7.

[67] Menascé, Daniel A., Almeida, Virgilio A. F., Fonseca, Rodrigo, and Mendes, Marco A. A methodology for workload characterization of e-commerce sites. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce* (New York, NY, USA, 1999), ACM, pp. 119–128.

[68] Muppala, Jogesh K., Trivedi, Kishor S., Mainkar, Varsha, and Kulkarni, Vidyadhar G. Numerical computation of response time distributions using stochastic reward nets. In *Annals of Operations Research* (1994), pp. 155–184.

[69] Nagin, Kenneth, Hadas, David, Dubitzky, Zvi, Glikson, Alex, Loy, Irit, Rochwerger, Benny, and Schour, Liran. Inter-cloud mobility of virtual machines. In *Annual International Conference on Systems and Storage* (New York, NY, USA, 2011), SYSTOR '11, ACM, pp. 3:1–3:12.

[70] Nanda, Susanta, and cker Chiueh, Tzi. A survey of virtualization technologies. Tech. rep., Department of Computer Science, SUNY at Stony Brook, 2005.

[71] Nelson, Michael, Lim, Beng-Hong, and Hutchins, Greg. Fast transparent migration for virtual machines. In *ATEC '05: USENIX ATC* (Berkeley, CA, USA, 2005), USENIX Association, p. 25.

[72] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on* (may 2009), pp. 124 –131.

[73] Nurmi, Daniel, Wolski, Rich, Grzegorczyk, Chris, Obertelli, Graziano, Soman, Sunil, Youseff, Lamia, and Zagorodnov, Dmitrii. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2009), CCGRID '09, IEEE Computer Society, pp. 124–131.

[74] Open Nebula: The Open Source Toolkit for Data Center Virtualization. http://www.opennebula.org.

[75] Opennebula. `http://www.opennebula.org`.

[76] Amazon Simple Storage Service. `http://www.amazon.com/s3`.

[77] Cloudstack, opensource cloud computing. `http://cloudstack.org`.

[78] Round Robin Database Tool. `http://oss.oetiker.ch/rrdtool/`.

[79] openstack: Cloud Software. http://www.openstack.org.

[80] Pacifici, Giovanni, Segmuller, Wolfgang, Spreitzer, Mike, Steinder, Malgorzata, Tantawi, Asser, and Youssef, Alaa. Managing the response time for multi-tiered web applications. In *IBM, Technical Report* (January 2005).

[81] Padala, Pradeep, Shin, Kang G., Zhu, Xiaoyun, Uysal, Mustafa, Wang, Zhikui, Singhal, Sharad, Merchant, Arif, and Salem, Kenneth. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 289–302.

[82] Perilli, Alessandro, Manieri, Andrea, Algom, Avner, Balding, Craig, and Various. Cloud Computing Risk Assessment ENISA. Tech. rep., ENISA, Greece, Nov. 2009.

[83] Pickavet, Mario, and Demeester, Piet. Multi-period planning of survivable wdm networks. *European Transactions on Telecommunications 11*, 1 (2000), 7–16.

[84] Raghavendra, Ramya, Ranganathan, Parthasarathy, Talwar, Vanish, Wang, Zhikui, and Zhu, Xiaoyun. No "power" struggles: coordinated multi-level power management for the data center. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 48–59.

[85] Rajagopalan, Sampath, Singh, Medini R., and Morton, Thomas E. Capacity expansion and replacement in growing markets with uncertain technological breakthroughs. *Manage. Sci. 44*, 1 (Jan. 1998), 12–30.

[86] Ranjan, S., Rolia, J., Fu, H., and Knightly, E. Qos-driven server migration for internet data centers. In *Proceedings of IWQoS 2002, Miami Beach, FL* (May 2002).

[87] Rochwerger, Benny, Breitgand, David, Epstein, Amir, Hadas, David, Loy, Irit, Nagin, Kenneth, Tordsson, Johan, Ragusa, Carmelo, Villari, Massimo, Clayman, Stuart, Levy, Eliezer, Maraschini, Alessandro, Massonet, Philippe, Munoz, Henar, and Toffetti, Giovanni. Reservoir - when one cloud is not enough. *Computer 44* (2011), 44–51.

[88] Russell, J., and Cohn, R. *Rabbitmq*. Book on Demand, 2012.

[89] Sengupta, Bikram, and Roychoudhury, Abhik. Engineering multi-tenant software-as-a-service systems. In *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems* (New York, NY, USA, 2011), PESOS '11, ACM, pp. 15–21.

[90] Sharma, U., Shenoy, P., and Towsley, D. F. Provisioning Multi-tier Cloud Applications Using Statistical Bounds on Sojourn Time. Tech. Rep. UM-CS-2012-009, Dept. of Computer Science, Univ. of Massachusetts, March 2012.

[91] Sharma, Upendra, Shenoy, P., Sahu, Sambit, and Anees, Shaikh. Kingfisher: A System for Elastic Cost-aware Provisioning in the Cloud. Tech. Rep. UM-CS-2010-005, Dept. of CS, UMASS, May 2010.

[92] Shen, Zhiming, Subbiah, Sethuraman, Gu, Xiaohui, and Wilkes, John. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *SOCC '11* (New York, NY, USA, 2011), ACM, pp. 5:1–5:14.

[93] Shivam, Piyush, Iamnitchi, Adriana, Yumerefendi, Aydan R., and Chase, Jeffrey S. Model-driven placement of compute tasks and data in a networked utility. *ICAC* (2005).

[94] Shivam, Piyush, Marupadi, Varun, Chase, Jeff, Subramaniam, Thileepan, and Babu, Shivnath. Cutting corners: workbench automation for server benchmarking. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Berkeley, CA, USA, 2008), ATC'08, USENIX Association, pp. 241–254.

[95] Sobel, W, Subramanyam, S, Sucharitakul, A, Nguyen, J, Wong, H, Patil, S, Fox, A, and Patterson, D. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of Cloud Computing and its Applications* (2008).

[96] Sotomayor, B., Montero, R.S., Llorente, I.M., and Foster, I. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE 13*, 5 (sept.-oct. 2009), 14 –22.

[97] Sotomayor, Borja, Montero, Rubén S., Llorente, Ignacio M., and Foster, Ian. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing 13*, 5 (Sept. 2009), 14–22.

[98] Sottile, Matthew J., and Minnich, Ronald G. Supermon: A high-speed cluster monitoring system. In *In Proc. of IEEE Intl. Conference on Cluster Computing* (2002), pp. 39–46.

[99] Stewart, Christopher, and Shen, Kai. Performance Modeling and System Management for Multi-component Online Services. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (May 2005).

[100] Subashini, S., and Kavitha, V. Review: A survey on security issues in service delivery models of cloud computing. *J. Netw. Comput. Appl. 34*, 1 (Jan. 2011), 1–11.

[101] TPCW. Java implementation. Website. `http://tpcw.deadpixel.de`.

[102] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM Sigmetrics Conference, Banff, Canada* (June 2005).

[103] Urgaonkar, Bhuvan, Pacifici, Giovanni, Shenoy, Prashant, Spreitzer, Mike, and Tantawi, Assar. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proc. of the ACM SIGMETRICS Conf.* (Banff, Canada, June 2005).

[104] Urgaonkar, Bhuvan, Shenoy, Prashant, Chandra, Abhishek, Goyal, Pawan, and Wood, Timothy. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Adaptive and Autonomous Systems (TAAS), Vol. 3, No. 1* (March 2008), 1–39.

[105] Urgaonkar, Bhuvan, Shenoy, Prashant, Chandra, Abhishek, Goyal, Pawan, and Wood, Timothy. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst. 3* (March 2008), 1:1–1:39.

[106] Vahdat, Amin. Future directions in distributed computing. Springer-Verlag, Berlin, Heidelberg, 2003, ch. Dynamically provisioning distributed systems to meet target levels of performance, availability, and data quality, pp. 127–131.

[107] Villela, D., Pradhan, P., and Rubenstein, D. Provisioning Servers in the Application Tier for E-commerce Systems. In *Proceedings of the 12th IWQoS* (June 2004).

[108] Vinoski, Steve. Advanced message queuing protocol. *IEEE Internet Computing 10*, 6 (Nov. 2006), 87–89.

[109] Vinoski, Steve. Advanced message queuing protocol. *IEEE Internet Computing 10*, 6 (Nov. 2006), 87–89.

[110] Virtual machine mobility with VMware VMotion and Cisco Data Center Interconnect Technologies. http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns836/white_paper_c11-557822.pdf, Sept. 2009.

[111] VMware: Public & Hybrid Cloud Computing. http://www.vmware.com/solutions/cloud-computing/public-cloud/products.html .

[112] Waldspurger, C. A. Lottery and stride scheduling: Flexible proportional-share resource management. Tech. rep., Cambridge, MA, USA, 1995.

[113] Waldspurger, Carl A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36* (Dec. 2002), 181–194.

[114] Watson, Brian J., Marwah, Manish, Gmach, Daniel, Chen, Yuan, Arlitt, Martin, and Wang, Zhikui. Probabilistic performance modeling of virtualized resource allocation. In *Proceedings of the 7th international conference on Autonomic computing* (New York, NY, USA, 2010), ICAC '10, ACM, pp. 99–108.

[115] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the International Conference on Virtual Execution Environments (VEE'09)* (April 2009), pp. 31–40.

[116] Wood, Timothy, Ramakrishnan, K. K., Shenoy, Prashant, and Van der Merwe, Jacobus. CloudNet : Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *VEE* (Mar. 2011), pp. 121–132.

[117] Wood, Timothy, Shenoy, Prashant, Venkataramani, Arun, and Yousif, Mazin. Sandpiper: Black-Box and Gray-Box Resource Management For Virtual Machines. *Computer Networks: The International Journal of Computer and Telecommunications Networking 53*, 17 (Dec. 2009).

[118] Xiong, Kaiqi, and Perros, H. Qrp01-6: Resource optimization subject to a percentile response time sla for enterprise computing. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE* (27 2006-dec. 1 2006), pp. 1 –6.

[119] Yaffee, Robert A. Forecast evaluation with stata. United kingdom stata users' group meetings 2010, Stata Users Group, 2010.

[120] Yashkov, S. F. Processor-sharing queues: some progress in analysis. *Queueing Syst. Theory Appl. 2*, 1 (1987), 1–17.

[121] Zhang, Qi, Cherkasova, Ludmila, Mi, Ningfang, and Smirni, Evgenia. A regression-based analytic model for capacity planning of multi-tier applications. *Cluster Computing 11*, 3 (2008), 197–211.

[122] Zhang, Qi, Cherkasova, Ludmila, and Smirni, Evgenia. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC '07* (2007).

[123] Zhang, Steve, Cohen, Ira, Symons, Julie, and Fox, Armando. Ensembles of models for automated diagnosis of system performance problems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2005), DSN '05, IEEE Computer Society, pp. 644–653.

[124] Zheng, Jie, Ng, Tze Sing Eugene, and Sripanidkulchai, Kunwadee. Workload-aware live storage migration for clouds. In *VEE* (New York, NY, USA, 2011), VEE '11, ACM, pp. 133–144.

[125] Zheng, Wei, Bianchini, Ricardo, Janakiraman, G. John, Santos, Jose Renato, and Turner, Yoshio. Justrunit: experiment-based management of virtualized data centers. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (Berkeley, CA, USA, 2009), USENIX'09, USENIX Association, pp. 18–18.

[126] Zhu, Xiaoyun, Young, Donald, Watson, Brian J., Wang, Zhikui, Rolia, Jerry, Singhal, Sharad, McKee, Bret, Hyser, Chris, Gmach, Daniel, Gardner, Rob, Christian, Tom, and Cherkasova, Ludmila. 1000 islands: Integrated capacity and workload management for the next generation data center. In *ICAC* (2008), pp. 172–181.