Dynamic Resource Management of Cloud-Hosted Internet Applications
by

HANGWEI QIAN

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Dissertation Advisor: Dr. Michael Rabinovich

Department of Electrical Engineering and Computer Science
Case Western Reserve University
August, 2012
CASE WESTERN RESERVE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

________________________________________________________________________

Hangwei Qian

candidate for the ________Ph.D._______ degree∗.

(signed)    Michael Rabinovich
________________________________________________________________________

(Chair of Committee)

________________________________________________________________________

Vincenzo Liberatore

Guo-Qiang Zhang

Christos Papachristou

________________________________________________________________________

________________________________________________________________________

(date)  May 21th, 2012

∗We also certify that written approval has been obtained for any proprietary material
   contained therein.
To my parents, grandparents, wife and sisters with love.
## Contents

### 1 Introduction

1.1 Target Environment .................................................. 1

1.2 Dynamic Resource Management for Global Cloud Computing Platforms .. 4
   1.2.1 Local Resource Management .................................. 5
   1.2.2 Global Resource Management .................................. 7

1.3 Summary ............................................................... 9

### I Local Resource Management ........................................... 10

#### 2 Agile Resource Management in Virtualized Data Centers ................. 12

2.1 Agility in Virtualized Data Centers .................................. 14
   2.1.1 VM-based Alternatives for Resource Allocation ............... 14
   2.1.2 Agility Results .................................................. 17

2.2 Dynamic Resource Allocation ........................................ 23
   2.2.1 Resource Reassignment Mechanism .............................. 23
   2.2.2 System Architecture ............................................ 24
   2.2.3 Algorithms ....................................................... 25

2.3 Evaluation ............................................................ 33
   2.3.1 Setup ............................................................. 33
### 2.3.2 Ghost Promotion and Demotion ................................. 35
### 2.3.3 Platform Agility ................................................. 37
### 2.3.4 Dynamic Resource Allocation ................................. 40
### 2.4 Related Work ..................................................... 43
### 2.5 Summary ........................................................... 45

### 3 Mega Data Center for Internet Applications 47

#### 3.1 Challenges ....................................................... 48
#### 3.2 Approach ......................................................... 50
    - 3.2.1 Hierarchical Resource Management .................... 50
    - 3.2.2 Access Link Load Balance ............................... 51
    - 3.2.3 Load-balancing Switch Load Balance .................. 53
    - 3.2.4 VIP/RIP Management ................................... 53
#### 3.3 System Overview ............................................... 54
    - 3.3.1 Basic Architecture ..................................... 55
    - 3.3.2 Dedicated LBG Architecture ............................ 58
    - 3.3.3 Two-layer Architecture ................................ 59
    - 3.3.4 Summary .................................................. 61
#### 3.4 Datacenter-Scale Resource Management .................... 61
    - 3.4.1 Server Transfer ......................................... 62
    - 3.4.2 Dynamic Application Deployment ..................... 63
    - 3.4.3 Weight Adjustment ..................................... 63
    - 3.4.4 Selective VIP Exposing ................................. 64
    - 3.4.5 Summary .................................................. 65
#### 3.5 Related Work ................................................... 65
#### 3.6 Conclusion ....................................................... 66
II  Global Resource Management

4  Unified Approach for Application Placement and Server Selection Across Data Centers

4.1  System Overview ........................................... 70
  4.1.1  Problem Statement ..................................... 72
  4.1.2  Framework ............................................. 74

4.2  Demand Distribution with Full Deployment .................. 75
  4.2.1  Problem Modeling .................................... 75
  4.2.2  Permutation Prefix Clustering ......................... 77
  4.2.3  Basic Idea ........................................... 77
  4.2.4  Application to Min-Cost Model ....................... 78

4.3  Application Placement .................................... 80
  4.3.1  Heuristics .......................................... 80
  4.3.2  Tiny Flow Removal .................................. 81
  4.3.3  Hysteresis Placement ................................ 83

4.4  Further Optimizations: Flow Splits ......................... 83
  4.4.1  Region Flow Splits .................................. 84
  4.4.2  Tiny Flow Splits .................................... 86

4.5  Evaluation ................................................. 88
  4.5.1  Cloud Model ......................................... 88
  4.5.2  Workload ............................................. 90
  4.5.3  Clustering Performance ............................. 92
  4.5.4  Deletion Threshold .................................. 93
  4.5.5  Hysteresis Placement Effects ....................... 95
  4.5.6  Policy Evaluation .................................... 96
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.7</td>
<td>Scalability</td>
<td>98</td>
</tr>
<tr>
<td>4.6</td>
<td>Prototype</td>
<td>100</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Demand Shift</td>
<td>101</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Flash Crowd</td>
<td>102</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work</td>
<td>103</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>Improving DNS-based Server Selection</td>
<td>106</td>
</tr>
<tr>
<td>5.1</td>
<td>Motivation</td>
<td>108</td>
</tr>
<tr>
<td>5.2</td>
<td>Architecture</td>
<td>112</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Overview</td>
<td>112</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Traversing Firewalls and NAT Boxes</td>
<td>113</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Degenerate Clusters</td>
<td>114</td>
</tr>
<tr>
<td>5.3</td>
<td>Prototype Implementation</td>
<td>115</td>
</tr>
<tr>
<td>5.4</td>
<td>Evaluation</td>
<td>116</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Hit Ratio</td>
<td>116</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Resource Consumption</td>
<td>117</td>
</tr>
<tr>
<td>5.4.3</td>
<td>P2P Lookup Delay</td>
<td>118</td>
</tr>
<tr>
<td>5.4.4</td>
<td>End-to-end Improvement</td>
<td>120</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion</td>
<td>123</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>124</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of Contributions</td>
<td>124</td>
</tr>
<tr>
<td>6.2</td>
<td>Future Work</td>
<td>126</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Target environment ...................................................... 3

2.1 Overhead of Needless Active VMs ................................... 17
2.2 VM state hierarchy ....................................................... 24
2.3 Architecture of Local Resource Manager ........................... 25
2.4 Application State Transition Diagram ............................... 28
2.5 Detection and Promotion Delay (VMware Server Platform) .... 36
2.6 Load Growth Patterns used for Platform Agility Tests .......... 37
2.7 Platform Performance and Response for Fast Growing Load .... 38
2.8 Load growth rate patterns. .............................................. 41
2.9 Slow Response Rate Performance for +10 EBs/min ............. 41
2.10 Performance for Multiple Applications Workload (VMware Server Cluster) I 43
2.11 Performance for Multiple Applications Workload (VMware Server Cluster) II 44

3.1 Basic architecture ...................................................... 55
3.2 Dedicated LBG architecture .......................................... 58
3.3 Two-layer architecture ................................................ 60

4.1 Overview ............................................................. 70
4.2 Min-cost network model ............................................... 76
4.3 Clustered network model .............................................. 79
4.4 Residual demand distribution network ........................................ 82
4.5 Region flow split ............................................................ 86
4.6 Tiny flow split ................................................................... 87
4.7 Performance of prefix clustering ............................................. 89
4.8 The effects of the deletion threshold ........................................ 93
4.9 The effects of the hysteresis ratio ............................................. 94
4.10 Policy performance (Vary-All-App workload) ......................... 97
4.11 Policy performance (Exchange-Rank workload) .................... 97
4.12 Policy performance (Reshuffle-All workload) ....................... 98
4.13 Scalability ...................................................................... 99
4.14 DC utilization of the prototype ............................................. 102

5.1 Extra air miles between clients and CDN servers due to LDNS-based server selection ...................................................... 109
5.2 System architecture ............................................................. 111
5.3 DNS hit rate ...................................................................... 116
5.4 Routing delay .................................................................... 120
5.5 End-to-end improvement (object size 6.8k) ............................. 122
5.6 End-to-end improvement (object size 54k) ............................. 122
5.7 End-to-end improvement (object size 2M) .............................. 123
List of Tables

2.1 Agility summary .......................................................... 23
2.2 Performance Comparison of Ghost VM vs. Legacy Approach on ESX Platform ........................................... 36
2.3 Performance Comparison of Ghost VM vs. Legacy Approach on VMware Server Platform ................................. 37
5.1 Ranking correlation ....................................................... 123
Acknowledgements

During the years of my study at Case Western Reserve University, my advisor, Professor Michael Rabinovich, has given me infinite support and encouragement, great patience and continuous guidance. I want to express my deep gratitude to him. It is my great honor to work with him, and I hope for continued collaboration with him in the future.

My thanks are also given to the other dissertation committee members: Professor Vincenzo Liberatore, Professor Guo-Qiang Zhang, and Professor Chris Papachristou. I really appreciate their invaluable comments and suggestions. Meanwhile, I want to thank all my fellow lab mates in the Networking and Distributed Group of Case Western Reserve University: Zhihua Wen, Sipat Triukose, Hongbo Jiang, Tu Ouyang, Zakaria Al-Qudah, Hussein Alzoubi, Zheng Liu, Tom Callahan, Kyle Schomp, Matthew Sargent, Song Zhao. They gave me a lot of help and made the lab feel like home.

Finally, I want to give my special thanks to my wife - Hailin Ouyang. Without her great patience and support, nothing is possible.

Thanks everyone, I wish you great happiness forever.
Dynamic Resource Management of Cloud-Hosted Internet Applications

Abstract

by

HANGWEI QIAN

Internet is evolving toward service-oriented computing platforms (e.g., cloud computing platforms, such as Amazon EC2 and Microsoft Azure). In these platforms, service providers (owners of the platforms) offer resource pools by building multiple geo-distributed data centers; application providers (owners of the applications) outsource the hosting of their applications to these platforms, and pay by the amount of resources used as utility. These multi-tenant platforms need to dynamically allocate resources to applications so as to meet their demand variation.

In this thesis, we address several issues of the dynamic resource management in these platforms. On the one hand, we consider the resource provisioning problems within data centers. In order to allocate resources to applications quickly, we propose deploying ghost virtual machines (VMs) which host spare application instances across the physical machines. When an application needs more instances, we can configure the request distributer to forward requests to ghost VMs, which takes only 5-7 seconds. Also, to deal with the scalability issues in mega data center (with hundreds of thousands of servers), we introduce hierarchical resource management scheme in which servers are divided into groups (pods), each with about 5k servers, and existing techniques are employed to manage resources in each pod efficiently. Meanwhile, multiple strategies are explored to balance the load among the pods. In addition, we also propose a new data center architecture in which
we can apply DNS-based mechanism to balance the load among the access links which connect data center to Internet.

On the other hand, we address the resource management problems among multiple data centers. We proposed a unified approach to decide in how many/which data centers each application should be deployed, and how client requests are forwarded to the geo-distributed service replicas. We make these decisions based on a min-cost network flow model, and apply a novel demand clustering technique to overcome the scalability issue when solving the min-cost problem. Furthermore, we also introduce a new client-side DNS architecture which brings local DNS server close to clients so that DNS-based server selection can precisely choose close service replicas for clients.
Chapter 1

Introduction

Internet has revolutionized the way how people access information and communicate with each other ever since its invention about 30 years ago. In recent years, it is evolving from information access systems to service-oriented computing platforms. In this chapter, we first explain the environment of the service-oriented computing platforms. Then we discuss the dynamic resource management problems in these platforms and introduce our proposed solutions to these problems. Finally, we give a brief summary of this chapter.

1.1 Target Environment

Traditionally, applications are hosted directly at the dedicated servers with enough capacity to meet their demand peaks. Due to the variation of the demand, servers are significantly underutilized. For example, according to [20], the studied data center is only 10% or less utilized half of the time. Meanwhile, the dramatic increase of the information and users of the Internet requires the matching up of computing capacity. In order to reduce the IT cost while still meet the capacity requirement, it is critical to improve the utilization of the servers. Grid computing [36] has been developed to utilize the idle resources across multiple organizations, but mostly focuses on scientific computing field.
A more general solution is provided by the service-oriented computing platforms [27, 42, 47], in which servers are aggregated into resource pools and business applications are consolidated to share the resources on demand from these pools. Since the peak demands of applications are not synchronized, these shared computing platforms can meet the capacity requirements of all the applications with much less computing resources. [20] found that by sharing resources among the applications on demand, a hosting provider can cut its resource requirements by half. These platforms usually consist of multiple data centers geographically distributed around the world, mainly for: i) high availability - failure of one location would not eliminate the accessibility of the services; ii) good user-perceived experience - client requests are forwarded to close replicas of services to reduce the response time.

A typical embodiment of this paradigm is cloud computing. Cloud computing platforms, like Amazon EC2, Microsoft Azure and Google AppEngine, etc, span multiple geographically distributed data centers around the world and are offering services to application providers. Application providers can outsource the hosting of their applications to these systems and pay them based on the amount of resources used as utility [21, 59, 79], which not only reduces their investment in the IT infrastructure, but also achieves great flexibility in their usage of resources depending on their business conditions. According to Gartner, the cloud service market was $68.3 billion in 2010, and expects to reach $150 billion by 2014 [1].

We assume the high-level architecture as shown in Figure 1.1 in our environment. In this architecture, applications are deployed at multiple data centers, and the global load balancing controller (e.g., DNS system) is responsible for forwarding client requests to appropriate data centers hosting the requested applications. Within each data center, all the instances of each application sit behind a local load balancing controller (e.g., load-balancing switches [45]). For each application, an external virtual IP (VIP) is configured
at the load-balancing switch. Also, for each instance of the application, an real IP (RIP) is allocated and an VIP-to-RIP mapping is configured on the load-balancing switch. The load-balancing switch distributes the requests for each application among the set of RIPv to balance the load among the multiple instances of the application, with policies like, round-robin or least-connections, etc. Meanwhile, instances of each application form an application-level cluster to be able to support session failover. All the cluster members interact with each other to exchange the session information. If the original application instance serving the requests of the same session fails, then other instance members of the cluster can still have the session information and serve the client requests correctly. When a client sends out a request, it would reach the global load balancing controller first. Then it would be forwarded to the local load balancing controller of the selected data center. Finally, it arrives at the application instance chosen by the local load balancing controller.
1.2 Dynamic Resource Management for Global Cloud Computing Platforms

In order for these platforms to be successful, it is critical to allocate resources for the Internet applications appropriately. A typical feature of Internet applications is that the demand from Internet clients is unpredictable - not only the volume of demand is unforeseeable, but the locations where the demand come from also vary dynamically. In the face of the dynamic demand, if we provision too much resources to the applications, we would waste resources; too little, miss business. Also, client demand should be served at close service replicas meanwhile without causing overloading of data centers. Therefore, these platforms need to dynamically decide: i) how many/which data centers each application needs to be deployed; ii) how to forward client requests to the geo-distributed service replicas; iii) how much resources are needed for each application within each data center.

One option is to make all of these decisions in a centralized manner. However, due to the extremely large volume of Internet clients, large number of servers as well as applications hosted in the data centers, it would be an extremely complex if not infeasible job to do that. Instead, in this thesis, a two-level resource management scheme is applied: local resource management and global resource management. The local resource manager in each data center deals with how much resources are needed for each application locally, while the global resource manager decides how many/which data centers each application needs to be deployed in and how to forward client requests to the geo-distributed service replicas. By separating these tasks, we can reduce the complexity of the problem significantly - although, since the problem is so complex, it still suffers from scalability issue at each resource management level, which we try to address as shown later on.
1.2.1 Local Resource Management

In each data center, the local resource manager is responsible for resource provisioning for each application. In order to do that, generally three steps are needed: i) detect the demand of each application and calculate how much resources are needed according to the demand to satisfy the service level agreement (SLA); ii) decide how to provide the needed resources for each application; iii) enact the resource provisioning decisions. Step i) often involves effort monitoring the utilization of servers, getting the client request statistics through logs and transforming the demand into the resource requirements for the applications. Step ii) needs to decide on which servers to allocate the needed resources for the applications, usually with the goal to balance the load among the servers in the data center. In step iii), the local resource manager carries out the resource provisioning decisions generated at step i) and ii).

The first problem we try to address for local resource management in this thesis is how we can allocate resources to applications as quickly as possible when needed, which we refer to as agility. The flash crowds surge very fast [31, 48] and resources should be allocated quickly to avoid service degradation or even melt-down. While existing works try to investigate efficient methods and algorithms for step i) and step ii), we also explore the solutions for the step iii) - how we can carry out the provisioning decisions in an agile manner. This is important because, even if one can invent very efficient algorithms to decide optimally how much resources each application needs and where to allocate these resources, if it takes long time to allocate that amount of resources, then still the applications would suffer from shortage of resources for long time resulting in poor user-perceived quality of service.

While our research work investigate an entire solution for the local resource management, the main contribution of this thesis is the solution for step iii). Our techniques for step i) and ii) are mainly designed by our collaborators at Worcester Polytechnic Institute, and are
also included in the thesis for completeness.

In our agile resource management, we assume that virtual machine (VM) technology is employed to host software stacks for the applications. In particular, we assume each VM hosts only one application instance. By running multiple VMs on a physical machine (PM), the resources of the physical machine can be shared by multiple applications. Also, since virtualization technique [25, 40] ensures that VMs running on the same physical machine are isolated against each other very well, the misbehavior of one application hosted in one VM would not affect applications running at other VMs on the same physical machine. In this way, the platform can achieve better security. We propose the concept of **ghost VM** to improve the agility in the resource management in virtualized data centers. The ghost VMs are pre-deployed VMs that are alive but do not receive any client requests since they are eliminated from the request routers. When an application needs more resources, we can configure the request router to forward requests to ghost VMs within some seconds. In this way, we can allocate resources to the application very quickly. On the other hand, due to the fact that ghost VMs still occupy the memory, their number is limited. In order to provide enough instances for applications receiving large volume of demand, we also pre-del部署 VMs, suspend and store them on disk. When ghost VMs are not enough to deal with the demand, we can resume the suspended VMs from disk to process the requests. The agility of ghost VMs and suspended VMs and their cost are different. We studied the trade-offs and proposed a resource provisioning mechanism based on them. Through evaluation on real testbeds, our proposed agile resource management exhibits much better performance across a set of metrics than legacy systems as shown in Section 2.3.

The second problem we deal with is the scalability issue for resource provisioning and load balance of access links in very large data centers. Modern cloud providers are building data centers with hundreds of thousands of servers, known as **mega data centers**. For example, Microsoft’s data center in Chicago can potentially pack up to 300,000 servers [8].
For the dynamic resource management in these mega data centers, existing methods and algorithms suffer from scalability problem. For example, the technique in [75] takes 30 seconds to manage resources for only 5000 servers. Meanwhile, mega data centers would have multiple access links connecting them to Internet. How to balance the load among the access links becomes important because the overloading of any access link would affect the traffic communication and severely degrade the user-perceived quality of service.

In our work, we proposed hierarchical resource management to deal with the scalability problem. We divided the servers into groups, each with about 5k servers, which we call *pods*. Within each pod, existing resource management techniques are applied to manage the resource efficiently. Meanwhile, we develop strategies, like server transfer and dynamic application deployment, to balance the load among the pods.

Also, in order to balance the load among the access links, we add load balance switch layers at the access network of the mega data centers. By configuring the applications in a special way and intelligently advertising the external IPs of the applications among the access routers, we can apply the DNS-based mechanisms to balance the load among the access links. For example, if we want client requests to come through lightly-loaded access links, we can ask DNS system to selectively expose external IPs configured on the corresponding access routers to Internet clients.

Note that, this thesis only outlines a methodology to solve foregoing problems. Research work is still going on for further investigation and quantitative evaluation.

### 1.2.2 Global Resource Management

The global resource manager decides in how many/in which data centers each application is deployed, and how to forward client requests to the geo-distributed service replicas. We refer the first problem as global application placement problem, and the second problem as global server selection problem. For global application placement, we only concern
about whether or not an application is deployed in a data center. As for how many application instances each data center would have for each application, it is the topic of the local resource management within each data center.

In this thesis, we proposed a unified approach based on the min-cost network model to address these two problems simultaneously, in contrast to most traditional ways in which these two problems are solved in separation. Also, a novel request clustering technique is proposed to address the scalability issue in solving the min-cost problem. In addition, we also consider the location of backend database which oftentimes do not co-locate with the application servers when making the decisions. Our method shows significant improvement over an existing approach as shown in Section 5.4.

Another issue this thesis deals with is to improve DNS-based server selection. DNS infrastructure is mainly used to translate user-readable domain name, like www.cnn.com, to machine-readable IP address, and plays an important role in today’s Internet. In particular, by selectively exposing IP addresses of servers hosting the services to Internet clients, DNS system is commonly adopted to transparently implement the server selection policies. For example, it is employed by Content Distribution Networks (CDNs), like Akamai and Limelight, select edge servers for their clients. Cloud Computing platforms like Google AppEngine, also use it to forward client requests to appropriate service replicas.

One problem with the DNS-based server selection is that only the location of local DNS instead of the location of client is known when making the server selection decision at the DNS system. If the clients are far away from their local DNS, as shown by the existing research work [57, 73], then the servers selected may not be close to the clients. In order to address this issue, we proposed a novel client-side DNS infrastructure based on P2P technology. In our architecture, each host runs an DNS software capable of doing DNS queries in the DNS infrastructure. Meanwhile, hosts that are of the same gateway form an P2P system. When a client want to access an object, it send DNS query to the P2P
system, and the target peer whose hash ID is the numerically closest to the one of the domain name pursues the real DNS query into the DNS infrastructure and send back the response. Since the peers in the P2P system are of the same gateway and are very close to each other, the server selected for the target peer is also close to the source peer. Also, the P2P infrastructure would serve the DNS cache functionality as the legacy system.

1.3 Summary

In short, this thesis addresses critical issues in the dynamic resource management for the Internet applications in the service-oriented computing platforms. It deals with the agile resource provisioning problem and scalable load balance problem within each data center of the platforms. At the same time, it also handles the global application placement and server selection problem across the data centers. In addition, it proposed a new client-side DNS architecture to improve the DNS-based server selection.
Part I

Local Resource Management
In this part, we introduce the dynamic resource management within each data center. We first introduce our local resource management aiming at reassigning resources to applications as quickly as possible based on the virtualized technology. Then we introduce the scalability issue for local resource management and load balance problem of access links in mega data centers, and explain our proposed solutions to these problems.
Chapter 2

Agile Resource Management in Virtualized Data Centers

An important property of a utility computing platform is how quickly it can react to changing demand. Flash crowds have been shown to produce rapid surges in demand [30, 49], and the quicker the system can reassign resources the less slack it needs to provision for each application and the more efficient its operation will be. We refer to the capability of a utility computing platform to quickly reassign resources as the agility of the platform. In this section, we examine resource reassignment mechanisms from the agility perspective and outlines a new mechanism that promises to significantly improve agility at a negligible cost.

Our approach is based on using virtual machines, the technology increasingly employed in hosting platforms already, and was inspired by the observation that a virtual machine can be resumed from a suspended state quickly provided the suspended VM remains in memory. Unfortunately, application servers that implement a particular application usually operate in an application cluster - see Section 1.1. The cluster members need to communicate with each other to exchange the session information so that the entire cluster can
process all the requests correctly in the face of failure of some cluster members. However, the application server on a suspended VM is not able to interact with other cluster members, thus is considered inoperable by the cluster. As we show later in Section 2.1.2, it will take a long time to reintegrate the suspended application servers into the cluster upon reactivation.

Consequently, we propose to have a small pool of spare VMs on every physical machine, which remain active but detached from the Internet. Since they cannot be reached by the clients, these VMs will not service requests yet their application servers will participate in their respective clusters, which are managed over the private network. We call these VMs ghost virtual machines because their existence is invisible to the load-balancing switch receiving and redirecting client requests. Ghost VMs only consume the CPU for application cluster management, and we show that this CPU consumption is negligible. We retain only a small enough number of ghost VMs on each physical machine to ensure that most of their memory footprint does not page out. Resource reassignment amounts only to the reconfiguration of the load-balancing switch, which takes on the order of single seconds as opposed to minutes in existing approaches.

The concept of ghost VMs introduces a hierarchy of states for a VM in a utility computing platform: active, ghost, suspended, and not booted. The resource management algorithm can now move VMs between these states on each machine. Because of the state hierarchy, the resource management problem becomes loosely analogous to cache management. We implement the ghost VM concept into a fully functioning utility computing prototype and evaluate the resulting platform. We further show that the ghost concept is general enough to apply across different virtualization technologies. To demonstrate this aspect, we implemented and deployed our architecture with two virtualization technologies at the opposite ends of the degree of resource isolation they provide. Specifically, we used the VMware ESX technology, which allows VMs to be allocated hard slices of physical
resources, and VMware Server, which only guarantees memory allocation.

2.1 Agility in Virtualized Data Centers

In this section, we first considered a number of VM-based alternatives for adding/removing resources to/from applications, including our new approach - the deployment of ghost VMs. To better understand how well these alternatives work for resource reallocation, we then constructed a test environment to evaluate the agility of these alternatives.

2.1.1 VM-based Alternatives for Resource Allocation

Migration of VMs. By migrating a VM from an overloaded host to a relatively idle host, we can achieve better load balancing and utilize our resources more efficiently. There are several ways in which migration can be accomplished.

1. Migration of a VM can be achieved by stopping a VM on one host and booting a pre-stored mirror VM on another host. In this case, the destination VM does not reflect the current state of the source VM.

2. A VM from one host can be cloned to another host. This method requires that we first suspend the VM on the source host, then copy it across hosts, and finally resume it on the target host. The state of the source VM will be preserved but this method requires moving the VM across machines in the critical path.

3. Recently, techniques have been developed for live migration of a VM, where the migration occurs without stopping the source VM, and the updates to the source VM state are forwarded to the target host during migration [33, 85]. This method dramatically reduces the downtime during migration but does not by itself improve the agility of resource reassignment. First, the target host will only be able to start
performing its work after a delay that is similar to the cloning technique. Second, the source machine will not be relieved of the overload due to the operation of the source VM until the migration is complete.

**Redeployment of Application Servers On Demand.** In this method, instead of migrating VMs across hosts, we pre-deploy VMs across the set of hosts and use them for different applications as dictated by the demand. In cases of overload, we determine which applications are targeted by large number of requests. We then search to find relatively idle hosts running VMs with applications that are not popular. At this point we stop application servers on those VMs with unpopular applications and remove them from their application clusters. Finally, we add new application servers on those VMs and join them to the corresponding application cluster.

**Suspend/Resume VMs.** This method is similar to the previous method in that we pre-deploy a number of VMs across all hosts, but each VM now always runs its assigned application. With some number of VMs left alive for serving client requests in the initial stage, all other VMs are suspended. We then suspend or resume various VMs on different hosts depending on the observed demand for their corresponding applications.

**Active/Ghost VMs.** All previous approaches share the property that newly active instances of application servers must (re)establish their cluster membership, which can take a long time. As a new approach to improve agility of resource allocation, we attempt to hide the delay in resource reassignment. We pre-deploy VMs, each with its own application(s), on each host. In contrast to previous methods, all VMs are alive using this approach. However, the switch at each data center is configured so that it is not aware of all of these VMs and consequently not all are used to service client requests. We refer to VMs that are known by the switch and can handle client requests as active VMs and the others as ghost VMs. While hidden from the public Internet, the ghost VMs can still communicate with
each other using either the host's second network card, available on most modern servers, or through the L2 functionality of the switch. All VMs (both active and ghosts) use this private network for managing application clusters. Thus, the application servers on ghost VMs fully participate in their application clusters and maintain both their membership in these clusters and the current application state.

Promoting a VM from the ghost to active state amounts only to the reconfiguration of a network switch, which takes on the order of single seconds as opposed to minutes in existing approaches. This approach forms the basis for agile resource reassignment. In the initial stage, there is an active list of VMs that are enough to serve the client requests. All the other VMs lie in the ghost list. After detecting an overloaded application, additional resources can be allocated to it by identifying a ghost VM on a host with this application and promoting it to the active state. Depending on the overall host utilization, an existing active VM with another application might be demoted to the ghost status.

Ghost VMs only consume CPU for application cluster management, and we show that this CPU consumption is negligible. At the same time, to ensure quick state promotion, we retain a small enough number of ghost VMs on each physical machine so that most of their memory footprint does not page out. In other words, while not consuming other resources, ghost VMs do consume memory. We believe this is an economical approach to utility computing when agility is a concern. Indeed, memory constitutes a relatively small portion of the server costs. A quick visit to dell.com shows that a PowerEdge 1950 server costs $4143, while doubling its memory from 4G to 8G costs only $500, or 12% of the cost of another server.

The alternate is to dedicate all capacity of a data center to active VMs, but Figure 2.1 shows that needlessly activating VMs, versus leaving them in the ghost state, does not improve reply rate and response time for incoming requests. The results were obtained by running an experiment using httperf to send 1700 requests/sec to a sample WebSphere
application with seven VMs where each VM is either in active or ghost state. As shown in the figure, with the number of active VMs increasing, the overall performance of the system actually degrades – the reply rate drops and the average response time increases. We attribute the overhead to the scheduling and context switching among the multiple active VMs.

2.1.2 Agility Results

We constructed a test environment at Case and performed experiments to evaluate each of the methods described in Section 2.1.1. The testbed data center consists of three identical machines as servers, each having Intel (R) 4-core 2.33 GHz CPU, 4G memory, 146G disk with 15K RPM and running Linux 2.6.17-10-generic SMP as host OS. The testbed uses a Nortel Alteon 2208 Application Switch as the local request router, VMware Server 1.0.1 and 1.0.2 as the virtualization technology and WebSphere 6 Network Deployment as the application server. Each VM is configured to have 2 virtual CPUs and 1G memory. All the reported results are the average of at least three measurements. Agility has two consid-
erations – how quickly more resources can be assigned to an application and how quickly resources can be de-allocated. In most cases, much of the delay is due to the former, and we concentrate on the delay in assigning a new VM to an application in this section.

Migration of VMs. If we perform VM migration by stopping a VM on one host and starting its mirror VM on another host, three factors contribute to the total delay before the mirror VM can serve the client requests: time for starting the mirror VM, time for starting a cluster agent on this VM (the clustering mechanism of WebSphere, as well as WebLogic, requires a host to run a clustering agent before any of the application servers can participate in a cluster, as discussed in Section 1.2.2), and time for starting the application server on the mirror VM.

We found that the average startup time of a VM varies from 46s on an otherwise idle machine that may have some of the VMs memory pages in RAM already, to 55s on an idle machine with VM memory pages flushed from RAM prior to the startup (which we achieve by executing a memory intensive application before the startup), to 64s on a machine with flushed VM memory and a moderately active (using 8% CPU) concurrent VM. We believe the last case represents a typical scenario in utility computing since physical machines are likely to be shared between VMs and the new VM is unlikely to retain its pages in memory from previous execution. We further found that starting the cluster agent can take around 20s. Note that, the agent has already joins the cluster after being started (it takes less time to start and proactively join the cluster than the integration of a newly resumed VM into the cluster, as shown for the “Suspend/Resume VMs” method shortly.) . We measured the time to start an application server to be about 97s on average when done for the first time after a VM reboot. Thus, the total time it takes before the mirror VM can serve client requests to be on the order of 180s – not a good result for agility.

If we achieve VM migration by cloning, the VM must be resumed on the target host, which as we will see in the ”Suspend/Resume VMs” method shortly, takes somewhat less
time. However, before resuming the VM, cloning involves copying the state of the VM and its disk files to another host. For example, to clone a VM with 1G memory and 10G disk, we need to transfer 11G data across the hosts. The bandwidth between the hosts is 100Mbps in our environment. So the transfer time can be 15 minutes, making this method untenable for agility. To make the transfer time acceptable, the platform would have to use a 10Gbps network, which is economically feasible only on a LAN within one data center. Even then, the resume delays are significant.

As for the live VM migration, similar with cloning, the time needed for the migration also depends on the size of VM and the bandwidth of the network. Also, some live migration techniques, like [33], apply iterative copy of dirty pages, which further increases the whole migration time. Thus, live migration would not help to improve the agility for resource reassignment.

**Redeployment of Application Servers On Demand.** Unlike the migration method, this method does not include starting the cluster agent because there is only one agent per VM and the VM is not stopped here. The delay of this method mainly consists of three parts: time for stopping an old cluster member (application server) to free up a VM, time for creating a new cluster member (note that we did not need to create a cluster member in "Migration of VMs” method because in that case the same application is associated with a VM and hence the cluster membership persists across reboots), and time for starting the new cluster member. In this method, since the cluster membership is already assigned when creating and starting the new cluster member, no explicit rejoining delay is needed, unlike the “Suspended/Resume VMs” method mentioned below. Using our experimental setup, we find that time for stopping a cluster member is about 95s on average, the time for creating a new cluster member is about 19s, and the time for starting an application as part of the cluster member is, as given above, about 97s on average. Note that these operations can take significantly less time when done repeatedly, due to caching effects. Because in
a real platform these operations occur infrequently, only in the course of resource reas-
signment, we obtain multiple measurements by executing these operations each time for
different clusters. In total, the delay in this method is on the order of 210s on average.
Thus, again, this method is not good for agility.

**Suspend/Resume VMs.** Instead of migrating VMs or dynamically creating new applica-
tion servers, this method maintains the frozen state of VMs and does not need to explicitly
rejoin clusters or startup application servers: the application servers resume along with
their VMs, and the clusters health check functionality will detect the recovered members.
In addition, a suspended VM consumes no CPU or memory resources. Measuring the time
to suspend or resume a VM using our experimental setup we find that it takes about 6s to
suspend a VM and 5s to resume one. The agility of this method seems good.

However, through further investigation, we find that if the memory of a suspended VM
is swapped out, which we force by running a memory-intensive application after the VM is
suspended, it can take much longer time to resume it. We measure the average resume delay
of 10s (range 5-21s) on an otherwise idle machine and 14s (range 7-31s) on a machine with
one other moderately active competing VM (consuming about 8% CPU). We attribute the
ranges to the dependence on the state of the VM when it is suspended. Furthermore, this
represents a lower bound on the actual delay because we only measure the time until the
call to a resume API completes. However, this completion time does not mean that the
memory required for the application to process a request has been swapped in. In fact, we
observed that, the resumed VM had on average only 20-25% of its memory swapped in.
Thus, the actual delay can be longer.

Another significant delay component in the VM resume case is due to the need for other
cluster members to detect the return of the cluster member running on the resumed VM.
Although a resumed VM can serve the client requests after the switch has detected it, the
application server can not fulfill the responsibilities as a cluster member, such as client
session replication and fail-over. In WebSphere, a new cluster member is admitted by the distributed voting of the existing members, which involves a 30s wait to amortize the voting for potential additional new members. We conclude that the delay of the suspend/resume resource reassignment can be conservatively put at 44 seconds and in reality is likely to be much higher.

**Active/Ghost VMs.** In this method, not only are VM migration and dynamic new application server creation not needed, but there is also no cluster reconfiguration delay, since the ghost VMs are alive all of the time. However, two issues must be addressed here: overhead of the ghost VMs and the swap-in problem.

For the overhead of the ghost VM, we have focused on use of the CPU. According to our experiments, the CPU overhead of a ghost VM, both clustered and stand-alone, is 1.4%. Note that CPU utilization depends on the applications running on the VM. For example, applications that need to run on the background frequently may cost more CPU. In fact, a cache monitoring application, which is a sample application of WebSphere, has an CPU overhead of 3.3%. But even in this case, it is still acceptable.

The swap-in problem would arise if most of the memory of the ghost VM is swapped out. In this case, it can take a long time to swap in enough memory of the VM to work smoothly. Our experiments show that if active VMs run memory-consuming applications, as much as 98.5% of the memory of ghost VMs on the same host can be swapped out. We also estimated the upper bound on the swap-in time as follows. We first exhausted the memory of the host by running a competing memory-intensive application, so that the memory of the ghost VM is swapped out up to about 98%. We then ran a program allocating memory repeatedly in the ghost VM. Our result shows that to swap in all memory (1G) of the VM may take about 652s. For an application requiring half the memory (500M), swapping in still takes about 267s. We consider these results as upper bound for swap-in delay because memory requirements are application specific. Still it is clear that VMs with
busy application servers may take a long time to swap in.

To address the swap-in problem we hypothesize that leaving some amount of memory, such as 1G, unused for the host OS, will avoid the problem of ghost VMs being swapped out and taking too long to swap in. For example, our host has 4G memory and we only deploy 3 VMs with 1G of memory each. If the host is devoted entirely to only VMs, the additional 1G can satisfy the memory demands well for the daemons on the host OS. On the other hand, the VM Monitor (VMM) makes sure that the memory usage of each VM is limited to 1G. Thus, regardless of the level of activity of these VMs, there is no need for the memory of any of them to be swapped out. Our experiments verify this expectation. Our experiment lasts about 5 hours and the memory of ghost VMs stays loaded in RAM. In the future, we expect to use VMware ESX, which can guarantee that each VM is provided with a minimum amount of RAM.

Based on these results, we extended our experiments. In this experiment, there are three identical physical machines. Each physical machine is installed with one VM belonging to the same cluster. VMs on physical machine 1 and 2 are active, while we have a ghost on physical machine 3. We send a large number of requests to the switch, which initially knows about only VMs on physical machine 1 and 2, and just forwards requests to them. When detecting overload on physical machines 1 and 2, we move the VM on physical machine 3 from the ghost list to the active list by configuring the switch. Our measurements show that it takes 7s for the switch to start forward requests to the VM on physical machine 3 for load balancing. This time includes 1s to apply and 6s to save the new configuration. As an alternative approach, we found we can reduce the reconfiguration time to 2s if we make the switch aware of the ghost VM, but limit the number of connections it can handle to one. Activating this ghost VM then simply requires increasing the number of connections the switch is allowed to make with the VM.

**Summary.** A short summary of agility of the alternatives mentioned above is in Table
Table 2.1. Agility summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Agility (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration of VMs</td>
<td>~180</td>
</tr>
<tr>
<td>Redeployment of App Servers</td>
<td>~210</td>
</tr>
<tr>
<td>Suspend/Resume VMs</td>
<td>&gt;44</td>
</tr>
<tr>
<td>Active/Ghost VMs</td>
<td>2-7</td>
</tr>
</tbody>
</table>

2.1.

2.2 Dynamic Resource Allocation

In this section, we explain in details how the local resource manager dynamically reassign resources among the applications according to the demand change. We monitor the demand of the applications by simply measuring the utilization of VMs on which the instances of the applications run. In order to calculate the amount of resources needed by each application, we assume that the utilization of the VMs would decrease linearly when more capacity is added, which is explained in more details in Section 2.2.3. In addition, we use our proposed resource reassignment mechanism to enact the provisioning decisions.

2.2.1 Resource Reassignment Mechanism

Because of memory limitation, each host can only have a small number of ghosts, a larger pool of suspended VMs still must be kept on disk. If no appropriate ghost VM is found for an application in need of extra resources, the system will fall back to resuming a suspended VM from disk. Thus, in our system, we would have three kind of VMs: active, ghost and suspended, as shown in Figure 2.2.

By promoting/demoting VMs across these states among the applications, we can achieve dynamic resource provisioning according to the application demands. When detecting the overloading of an application, we can promote ghost VMs to be active on relatively lightly-loaded machines. Also, we can promote suspended VMs to be ghost if the demand of
an application tends to increase significantly so that more ghost VMs are available when needed. On the contrary, if the demand of an application is small, we can demote its active VMs to be ghost, and can further demote some of its ghost VMs to be suspended so that memory can be freed up to accommodate more ghost VMs for other applications.

### 2.2.2 System Architecture

The architecture of the loca resource management is shown in Figure 2.3. The local resource manager has three components: Resource Collector, Resource Reallocator and the Ghost Manager. The Resource Collector communicates with agents on each of the PMs to gather the CPU utilization data of VMs at periodic intervals (currently every 10 seconds). The average of the last \( n = 3 \) VM utilization measurements is used as for calculations in the algorithm. When detecting that an application receives very small amount of demand, but occupies many application instances, the resource collector would try to remove resources allocated for this application, e.g., reduce the the number of active VMs running instance of that application.

The Resource Reallocator performs the resource reallocation in reevaluating whether
overloaded applications are detected based on the collected data. It is currently configured to execute every 10 seconds. It modifies the capacity of existing VMs and promotes ghost VMs. The Ghost Manager monitors the set of ghost VMs in the system and resumes new ghost VMs if an application is more likely to need additional capacity, and suspends existing ghost VMs if an application is less likely to need additional capacity.

### 2.2.3 Algorithms

The algorithm of calculating the amount of resources each application needs and deciding where to provide the needed resources for each application is designed by our collaborators at Worcester Polytechnic Institute. We include it in this thesis for completeness of the discussion and evaluation of our local resource management system.

**Capacity and Utilization.** An important aspect of our approach is that it formulates a general framework that applies to different virtualization technologies. This approach simplifies the construction of large utility computing platforms from heterogeneous server clusters. The basic notions behind our general framework are observed and projected capacity and utilization. These notions are defined based on one key notion—the capacity
of a virtual machine. We retarget the framework by simply redefining this one notion for a particular virtualization technology. For virtualization technologies that provide strong resource isolation by allowing one to configure virtual machines with hard slices of physical resources, such as ESX, the VM capacity is determined by the actual slice. For the virtualization technologies that do not provide this capability, such as VMware Server, our framework treats the VM capacity as a desired capacity. The desired capacity is a soft limit that is not enforced by VMware Server, but is adhered to in our algorithm. The algorithm adjusts the “soft slice” capacity for each VM over time and because it runs periodically, the duration a VM will be overloaded is limited. Note that with soft slices, the observed VM utilization (since it is computed relative to the desired capacity) can exceed 100%.

Let \( c(P_i) \) be the capacity of physical machine (PM) \( P_i \) and \( c(V_{i,j}) \) be the capacity of the \( j \)th VM on \( P_i \). Intuitively, the capacity of a PM is greater than or equals to the total capacity of its VMs. Thus, we assume that VM capacities are assigned to satisfy the following constraint where \( m_i \) is the number of VMs on PM \( P_i \):

\[
c(P_i) \geq \sum_{j=1}^{m_i} c(V_{i,j})
\]  

(2.1)

Let \( u(V_{i,j}) \) be the resource utilization on PM \( P_i \) due to running VM \( V_{i,j} \). Let \( u_r(V_{i,j}) \) be the relative utilization of VM \( V_{i,j} \), then

\[
u_r(V_{i,j}) = u(V_{i,j}) \times \frac{c(P_i)}{c(V_{i,j})}
\]  

(2.2)

For example, suppose the capacity of \( V_{i,j} \) is 30% of \( P_i \). If \( V_{i,j} \) consumes 24% of the resources on \( P_i \), i.e., \( u(V_{i,j}) = 24\% \), then the relative utilization \( u_r(V_{i,j}) = 80\% \). That is to say, \( V_{i,j} \) consumes 80% of its assigned resources.

Let \( u(P_i) \) be the overall resource utilization of PM \( P_i \). Intuitively, the overall utilization equals to the sum utilization of its VMs (we ignore the small overhead for the host OS or
hypervisor), i.e.,

\[ u(P_i) = \sum_{j=1}^{m_i} u(V_{i,j}) = \frac{\sum_{j=1}^{m_i}(u_r(V_{i,j}) \times c(V_{i,j}))}{c(P_i)} \quad (2.3) \]

Let \( c(A_k) \) be the total capacity of application \( A_k \). It is the sum capacity of VMs running application \( A_k \), i.e.,

\[ c(A_k) = \sum c(V_{i,j}), \quad \forall \ app(V_{i,j}) = A_k \quad (2.4) \]

Let \( u(A_k) \) be the average server utilization of application \( A_k \), then

\[ u(A_k) = \frac{\sum(u_r(V_{i,j}) \times c(V_{i,j}))}{c(A_k)}, \quad \forall \ app(V_{i,j}) = A_k \quad (2.5) \]

We introduce two load watermarks into our platform, *high-watermark (HW)* and *low-watermark (LW)*. We define the overloaded/underloaded situation for the servers and applications as follows:

If \( u_r(V_{i,j}) > HW \), \( V_{i,j} \) is overloaded. If \( u_r(V_{i,j}) < LW \), \( V_{i,j} \) is underloaded. If \( u(P_i) > HW \), \( P_i \) is overloaded. If \( u(P_i) < LW \), \( P_i \) is underloaded. If \( u(A_k) > HW \), \( A_k \) is overloaded. If \( u(A_k) < LW \), \( A_k \) is underloaded.

For example, suppose \( HW = 90\% \), then any VM which consumes more than 90% of its assigned resource, or any application which has an average utilization over 90%, is considered overloaded.

The goal of our resource reallocation algorithm is reassigning resources among VMs, PMs, and applications, so that none of them is overloaded, i.e., \( \forall i, j, k \), none of \( u_r(V_{i,j}) \), \( u(P_i) \), or \( u(A_k) \) is greater than \( HW \). In doing so we focus on keeping the VMs from being overloaded.

If a VM \( V_{i,j} \) running \( A_k \) is overloaded, then the maximum CPU usage for \( V_{i,j} \) could be
increased if such spare capacity exists on $P_j$. Alternately we could try to bring up one or more VMs for $A_k$ elsewhere to share the load, i.e., increase the capacity for the application rather than for the VM through the promotion of a ghost VM. Since the total capacity of the platform will not change, increasing the capacity for one application may require decreasing the capacity of others, so that the constraint (2.1) remains satisfied.

Let $c'(A_k)$ be the new capacity of application $A_k$ after resource reallocation. The goal of our algorithm is to find capacities for all active VMs such that none will be overloaded.

Figure 2.4. Application State Transition Diagram

In the following we express how the change of capacity affects the load distribution, and in turn affects the utilization of VMs. Let $u'(A_k)$ be the projected average server utilization of application $A_k$ after reallocation, then

$$u'(A_k) = \frac{u(A_k) \times c(A_k)}{c'(A_k)} \quad (2.6)$$

Intuitively, if we choose a proper $c'(A_k)$, we are able to keep the projected average utilization $u'(A_k)$ under the *high-watermark*. However, switch load balancing may not be perfect and individual VMs for the application may still be overloaded.

Let $u'(V_{i,j})$ be the projected relative utilization of $V_{i,j}$ after reallocation. Our goal is
to keep the utilization of every $V_{i,j}$ under $HW$. We assume that the change of utilization is in proportion to the change of capacity. For example, suppose we have two VMs $V_{1,2}$ and $V_{3,4}$ with the same capacity, but for some reasons, their utilization is not the same ($u_r(V_{1,2}) = 80\%$ and $u_r(V_{3,4}) = 60\%$). If we bring up two more VMs (double the total capacity), we assume that the projected utilization will drop in proportion to the current utilization ($u'_r(V_{1,2}) = 40\%$ and $u'_r(V_{3,4}) = 30\%)$. That is,

$$u'_r(V_{i,j}) = \frac{u_r(V_{i,j}) \times c(A_k)}{c'(A_k)},\quad \forall\ app(V_{i,j}) = A_k$$  \hspace{1cm} (2.7)

To bring utilization below $HW$, we need $u'_r(V_{i,j}) < HW$ i.e., from Equation (2.7),

$$c'(A_k) > \frac{u_r(V_{i,j}) \times c(A_k)}{HW},\quad \forall\ app(V_{i,j}) = A_k$$  \hspace{1cm} (2.8)

So, we need at least this amount of capacity to guarantee that $V_{i,j}$ will not be overloaded. If we calculate $c'(A_k)$ for all $V_{i,j}$ running application $A_k$ then get the maximum value. It is the value that keeps the utilization of all its VMs in bound.

We define $\Delta c(A_k) = c'(A_k) - c(A_k)$ as the change of capacity. Note that for some applications, $\Delta c$ might be negative which indicates a decreasing demand. This indicates that the current capacity is such that the maximum utilization of any VM running this application is below $HW$. We use LW to decide we can reduce the capacity of an application as follows.

By Equation (2.7) and (2.8), we define $c'(A_k)$, the minimum capacity of application $A_k$, which guarantee $A_k$ will not be overloaded. Similarly, we define $c''(A_k)$ as the maximum capacity of application $A_k$, guaranteeing that it will not be underloaded.

$$u''_r(V_{i,j}) = \frac{u_r(V_{i,j}) \times c(A_k)}{c''(A_k)} > LW,\quad \forall\ app(V_{i,j}) = A_k$$  \hspace{1cm} (2.9)

and
Again, because load balancing may not be perfect, we choose the minimum value as the final $c''(A_k)$ for application $A_k$ which keeps all of its VMs from being underloaded. For any application $A_k$, if $c(A_k) > c''(A_k)$, we can safely decrease its capacity.

Thus, our resource reallocation algorithm is transformed to a problem of satisfying all positive $\Delta c(A_k) = c'(A_k) - c(A_k)$ by using negative $\Delta c(A_k) = c''(A_k) - c(A_k)$ or residual PM capacities.

**Multi-stage algorithm.** Given these specifications, the resource reallocation algorithm is periodically executed within a data center using data gathered about the current status of PMs, VMs and applications. The multi-stage algorithm iterates over each of these entities. A diagram showing the transition between states for an application within a data center is given in Figure 2.4.

The diagram illustrates a number of distinguishing aspects of our approach. First, each active VM executing the application is represented as a different state. Once at least one instance of an application has been activated then one or more ghost VMs may be in existence between each of the active VM states.

Second, the activation of the first active VM within a data center is under the control of a global decision manager, which determines the placement of applications amongst a set of geographically dispersed data centers along with the assignment of user requests to specific data centers. While not all applications are active at each data center, we do assume that suspended instances of an application exist in all data centers. If not, then the initial activation of an application requires that a VM be created and the application to be started, which takes much longer than resumption. Once one instance of an application has been activated at a data center then management of that application is controlled by a
local decision manager, which performs the algorithm described in this work. Details of the
global resource manager and how it interacts with the local decision manager are current
work, but not discussed in this paper. The focus of this paper is how the local decision
maker manages available resources within a single data center.

Third, once an VM is active for an application within a data center, the operations of the
local decision manager can be divided into three stages: decreasing the capacity for those
applications with too much, increasing the capacity for those applications with too little,
and managing the availability of ghost VMs so they will be available to the applications
that are most likely to see an increase in demand. More details on each of these stages is
described below.

The first stage of the algorithm seeks to remove extra capacity from underloaded appli-
cations with operations D1, D2 and D3 shown in Figure 2.4. If the utilized capacity of
any application drops below the LW threshold then a VM with a relatively low capacity
compared to the application is demoted to a ghost VM as shown with operation D1. If its
capacity is relatively higher then its capacity simply reduced as shown with operation D2.
The algorithm will not demote the last VM for an application in order to ensure there is
some capacity in the data center for the application. Only in case that the global decision
manager decides to de-activate the application at this data center is operation D3 invoked,
which causes the last active VM to be suspended.

Now that any extra capacity for underloaded applications has been reclaimed, the next
stage of the algorithm seeks to reallocate residual capacity if applications are overloaded.
This increase in application capacity is accomplished via one of three operations. If avail-
able capacity exists on one of the physical machines currently running an active VM of the
application then the capacity of that VM is increased as shown with operation I1. If in-
creasing the capacity of an active VM does not meet the application’s demands and a ghost
VM is available then the ghost VM is promoted as indicated with operation I2. Finally,
if a ghost VM is not available then a suspended VM for the application must be resumed
directly to the active state via operation I3. This is an undesirable situation from the stand-
point of agility as resumption of an active VM takes much longer than promotion from a
ghost VM. This stage of the algorithm continues to iterate until all overloaded applications
are allocated sufficient capacity or there exists no more available capacity.

The final stage of the algorithm detects and handles management of ghost VMs. This
function is implemented through a ghost manager, which runs periodically to reallocate
ghosts among applications, and is also invoked explicitly any time the ghost allocation
changes through ghost promotion or suspension. We assume that the amount of additional
capacity that an application might need quickly is proportional to its current load, thus the
ghost manager allocates the available number of ghosts among applications in proportion
to their observed load. However, this correlation may not always exist as a heavily loaded
application may exhibit relatively constant demand while a lightly loaded application may
have much potential for a sharp increase in demand. We leave more elaborate ghost man-
agement algorithms for future work.

When a ghost VM for an application is allocated, the ghost manager is likely to create a
new ghost instance by resuming a suspended VM as shown with operation M1. Similarly if
an application currently has a ghost VM determined to be no longer needed, operation M2
suspends the ghost VM. Finally, operation M3 in Figure 2.4 shows that once at least one
ghost VM exists for an application, the number of ghost VMs can be updated as application
conditions and available capacity exist. Because ghost VMs are specific to an application,
it is possible for a ghost VM to be switched from one application to another, but only after
stopping the current application on the VM then starting the new application. As we found
in [67] the time to create/resume a VM, start an application and join an application cluster
can be on the order of minutes so it is important to keep ghost VMs available.
2.3 Evaluation

In this section, we discuss the evaluation of our local resource management system.

2.3.1 Setup

Our approach has been successfully deployed in two data centers—one using VMware Server v1.0.1 for virtualization and the other using VMware ESX Server v3.5.0. Each data center uses WebSphere V6 as the application server software with a back-end database server running Oracle. Each application server runs within its own VM.

The VMware Server data center consists of three physical machines each hosting up to three virtual machines. Each physical machine has two CPU cores and 2GB of RAM. Each virtual machine is configured with one virtual core and 512MB of memory. A Cisco Content Switch 11501 is used on the front end.

The ESX Server data center consists of three physical machines each hosting up to four virtual machines. Each physical machine has four cores and 4GB of RAM. Each virtual machine is configured with one virtual core and 1GB of memory. A Nortel Alteon 2208 application switch is used on the front end.

Within each data center, each VM is assigned one virtual CPU core. According to previous work [43], this option has good performance and utilization of the system. Each application within each data center is deployed with at least one active and one ghost VM.

The primary testing tool we use is TPC-W [76], a well-known “bookstore” workload for multi-tiered applications. We use the “browsing” request mix of TPC-W for testing. Requests are generated by Emulated Browsers (EBs). To increase the generated workload, two changes were made to the standard TPC-W configuration: the mean think time was reduced tenfold from the default of 7s to 700ms; and the TPC-W client software was modified to confine a client to a specific range of users for a test. This change allows testing
from multiple client machines under the control of a testing harness that we created.

We first measured the time for the system to detect that an increased workload requires additional active VMs and the time for a promoted VM to begin handling requests after switch reconfiguration.

We next examined how the faster resource reallocation of our approach translates into an end-to-end platform agility. To this end multiple applications with different load scenarios were run causing all aspects of the algorithm to be employed in not only promoting ghost VMs to active, but also for the ghost manager to recognize and resume suspended VMs to the ghost state. These tests confirmed the correct performance of our approach, and showed significant performance benefits over a legacy system that can respond to load changes only by resuming or suspending VMs.

We compared performance of our platform using ghost VMs with the legacy approach using four performance metrics available from the TPC-W clients. We determine the request error rate and the number of successful requests during the 10-minute interval in which each TPC-W client executes. One result of overloaded application servers is an increase in the number of request errors and fewer client requests that are handled. We also used TPC-W to determine the slow response rate, which is when the response time for successful client requests is greater than a threshold. Finally, we used TPC-W to measure the median response time for all successful requests.

Finally, we examined the ability of our approach to dynamically reassign resources among competing applications within a data center in response to changing demand. Specifically, we contrasted resource allocation and the resulting performance of three approaches:

1. Variants of the algorithm—we use our algorithm to manage resources with different parameter settings for the HW and LW thresholds.

2. Fixed—a fixed number of VMs were allocated to each application at the beginning
of a test and this number was not changed during the test. Two fixed cases were used, one in which the number of VMs was intentionally under-provisioned for at least some portion of the test and one in which the number of VMs was intentionally over-provisioned.

3. Manual—predetermined promotion and demotion changes are manually scripted for the test based on a priori knowledge of workload changes. This test is introduced to understand expected best case behavior, although there is some tradeoff as to whether maximize performance or minimize the number of VMs in use.

We compared these test approaches using the TPC-W client summary performance metrics described above as well as measuring the overload rate at each VM. This rate is the percentage of time that the CPU utilization of the VMs for an application are greater than 80%, which is used for consistency across all approaches.

2.3.2 Ghost Promotion and Demotion

Two critical measures in resource reallocation are the time needed to detect when a load increase has occurred and then time to activate additional resources. Figure 2.5 shows a scenario where a TPC-W application is running against our VMware Server platform causing a 40% CPU utilization for the VM serving this application. At time $t = 180$ seconds in the scenario, the client load is sharply increased causing the CPU utilization to similarly rise. Using a HW threshold of 70%, the algorithm soon detects that additional capacity is needed for this application and at time $t = 193$ seconds makes a call causing the switch to promote a ghost VM for the application. At time $t = 198$ seconds the newly activated VM begins handling requests as evidenced by the drop in CPU utilization for the first VM shown in Figure 2.5.

We performed four tests and observed a median detection time of 13 (range 11-15) sec-
seconds and median switch reconfiguration time of 5 (3-5) seconds for a total of 18 (14-20) seconds to detect and act upon a sharp increase in load. This result demonstrates good agility by our approach in rapidly responding to significantly increased resource demands of an application. Detection time could be improved by reducing the interval between load monitoring, which is currently done every 10 seconds with the average computed over the last three measurements. In contrast, creating additional capacity by migrating a VM, which was smaller than our VM, to an available machine took almost 20 seconds after detection [84], thus our approach reduced this delay by a factor of four.

While the response time is not as critical, we ran similar tests for a sharp reduction in the

<table>
<thead>
<tr>
<th>Approach/Growth Rate</th>
<th># Errors (%)</th>
<th>% Slow Requests</th>
<th>Median Req. Time (ms)</th>
<th># Successful Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghost/Fast</td>
<td>23 (0.0)</td>
<td>18.9 2.6 1.6</td>
<td>46</td>
<td>84089</td>
</tr>
<tr>
<td>Legacy/Fast</td>
<td>128 (0.2)</td>
<td>36.6 5.7 2.1</td>
<td>69</td>
<td>61556</td>
</tr>
<tr>
<td>Ghost/Slow</td>
<td>14 (0.0)</td>
<td>15.8 3.4 2.1</td>
<td>38</td>
<td>43819</td>
</tr>
<tr>
<td>Legacy/Slow</td>
<td>24 (0.1)</td>
<td>14.2 3.4 2.2</td>
<td>37</td>
<td>35608</td>
</tr>
</tbody>
</table>
load for an application. We started with two active VMs for an application and dropped the load so only one active VM is needed. In these tests we found the average detection time for demotion of a VM is 31 seconds with a total time of 35 seconds. These numbers make sense as we observe that the algorithm first tries to reduce the capacity of a VM before it eventually decides to demote the VM to a ghost.

2.3.3 Platform Agility

In our next set of tests we deployed three applications (all separate copies of TPC-W) on each of our platforms. Two of the applications, App1 and App2, are initially configured to

Table 2.3. Performance Comparison of Ghost VM vs. Legacy Approach on VMware Server Platform

<table>
<thead>
<tr>
<th>Approach/Growth Rate</th>
<th># Errors (%)</th>
<th>% Slow Requests</th>
<th>Median Req. Time (ms)</th>
<th># Successful Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghost/Fast</td>
<td>386 (0.6)</td>
<td>67.9 12.9 2.7</td>
<td>165</td>
<td>65692</td>
</tr>
<tr>
<td>Legacy/Fast</td>
<td>2235 (3.9)</td>
<td>68.0 18.2 5.0</td>
<td>182</td>
<td>56747</td>
</tr>
<tr>
<td>Ghost/Slow</td>
<td>0 (0.0)</td>
<td>44.8 4.1 1.4</td>
<td>88</td>
<td>40281</td>
</tr>
<tr>
<td>Legacy/Slow</td>
<td>0 (0.0)</td>
<td>42.0 5.2 1.4</td>
<td>78</td>
<td>39694</td>
</tr>
</tbody>
</table>
have one active, one ghost and one suspended VM. The other application, App3, initially has one active and two ghost VMs. The initial load for each application is 10 emulated browsers on our ESX platform where each EB corresponds to the activity of one user in TPC-W.

During all tests, the load of two applications App1 and App3 is kept constant. However two separate tests are run where the load pattern of App2 is varied as shown in Figure 2.6. In the first test, a fast load growth to 85 EBs occurs for App2. In a separate test, a slow-growing load pattern is used that gradually increases the client load to 60 EBs.

Figure 2.7(b) shows a time-series graph of the CPU load as well as the capacity for each of the three VMs for App2 during the lifetime of the test for the fast-growing load pattern. Note that VM1 is initially active and has a 100% utilization that is allowed while VM2 is a ghost VM and as a nominal slice of 12%\(^1\) As shown at time 70s, the resource reallocator detects and promotes ghost VM2 for App2 to be active. In addition, a message is sent to the ghost manager indicating that no more ghost VMs exist for this application. Normally the ghost manager executes every 120s, but in this case the ghost manager causes the suspended

\(^1\)All VMs are configured to run on one CPU core, and our utilization numbers all refer to the utilization of a single core, e.g., the ghost’s 12% slice means it consumes at most 3% of our 4-core physical server.
VM3 for App2 to be resumed to the ghost VM state at time 116s. Finally, the resource reallocator determines that additional capacity is needed and promotes VM3 to active at time 132s. Due to the sharp increase in load for App2, the newly activated VM3 will not be fully resumed when it is transitioned to the active VM state, but at least it has already begun resumption. Alternately, we could wait until it is fully resumed before activation, but our approach allows the new VM to be used as soon as it is ready. Similar functioning for the fast load growth is shown in Figure 2.7(a) for the VMware Server cluster, although the timing of actions is a bit different from Figure 2.7(b) as the physical servers have different capabilities in the two clusters.

We performed separate tests using the slow-growing load pattern, although a time-series graph of actions is not shown. In that test the ghost manager ran after a 120s interval and resumed another ghost VM for App2 because it had much more demand than the other applications. Thus App2 had two ghost VMs and when the load kept increasing, both of them were promoted to active.

To compare with a legacy system that does not utilize ghosts, we disabled the ghost manager portion of our platform so that the promotion and demotion of ghost VMs was not possible. Instead the platform could only resume or suspend VMs in and out of the active VM state. This approach mimics the behavior of a legacy system where resource reallocation can only be done with resumption and suspension of active VMs. We re-ran our fast- and slow-growing load test where initial ghost VMs in our platform are replaced with suspended VMs. Thus the initial state of VMs for App2 consists of one active VM and two suspended VMs. We re-tested each of the load growth patterns for App2 with this legacy platform and resource reallocation causing resumption of suspended VMs in the face of increased load. Table 2.2 shows a summary comparison of the TPC-W performance metrics for each of the four testing scenarios, for the ESX cluster, and Table 2.3 shows the same results for the VMware Server cluster.
Both tables show similar performance results across the virtualization technologies. In the fast growth case, our approach significantly outperforms the legacy system in all metrics we considered: the error rate, response time, and the number of completed requests. The higher agility of our approach allows the system to reassign resources among applications quickly to keep pace with the demand patterns. The legacy system experienced periods of degradation due to slow resource reassignment.

In the slow growth scenario, both ghosts and legacy platform were agile enough to cope with the growing demand. In fact, the legacy systems had somewhat better median response time, which we explain by the ghost overhead. Still, in the ESX case, the legacy system processed fewer total requests than our system. Because TPC-W clients wait for a current response before submitting the next request, this indicates that the legacy system experienced occasional slow-down periods.

### 2.3.4 Dynamic Resource Allocation

We compare the resource allocation and resulting performance of our approach with that of the fixed and manual resource allocation as described earlier. We first examine the scenario where the load for an application exhibits linear growth followed by linear decline back to the original level. This workload was first proposed in [29], to study the behavior of a system under a flash crowd of varying severity. We then consider a more dynamic scenario where multiple applications experience varying levels of demand.

Similar to [29], we performed an experiment to show how our algorithm performs with different load growth rates. The four workload patterns that we tested are shown in Figure 2.8 where each pattern exhibits a four-fold increase in load with different rates of increase.

We tested these workloads with different HW and LW thresholds of our algorithm as well as for fixed over- and under-provisioning. We also tested with a ”manual” approach
where reallocation decisions were scripted based on knowledge of the workload.

Figure 2.9(a) shows representative results that we found for our VMware Server platform. The graph focuses on the slow response rate metric measured by the TPC-W client with the most shallow growth rate for five test approaches. The two algorithm parameter sets are based on experience for what parameters are a good fit for the data center.

The figure shows results for the percentage of responses to the TPC-W that take more
than one second. For each approach this result is plotted against the mean number of VMs that were active over the course of the test. Thus the "Fixed 2" approach is under-provisioned at two active VMs for the application and yields the worst performance. At the other extreme, the "Fixed 3" approach is over-provisioned and provides the best performance. In between these extremes are the two instances of our algorithm along with the manual approach. The results for these three approaches are comparable, although the manual approach is not as aggressive in activating a new VM as the algorithm. The result is more slow responses compared with using fewer VM resources.

This comparison shows our system demonstrates competitive performance with the result if manual intervention was used for the known workload. The other performance measures show similar orderings and tradeoffs between the five approaches tested over the set of workloads.

We also performed tests on the ESX Server platform using the same load growth patterns shown in Figure 2.8. Again focusing on the slow response rate performance, results are shown in Figure 2.9(b) for the +10EBs/min growth rate. These ESX Server results contain two important distinctions from the previous VMware Server results to account for the platform and its more powerful servers. First, the slow response threshold is 100ms rather than 1 sec so it still represents on the order of 10% of responses. Second, because a hard CPU slice is used to enforce the desired capacity in the algorithm, the mean number of active VMs is weighted by their capacity. In looking at the results for this platform their tone is the same as what we previously observed with the two versions of our algorithm providing competitive performance with a manual intervention.

We finally tested the prototype with a workload of multiple applications. For this test we used two separate TPC-W applications (with separate databases) and a third application running as a simple WebSphere test with no database access. The initial configuration of this system is one active VM for each application as well as additional ghost VMs for the
first two applications. The regular and periodic workload for each application over the
course of our test is shown in Figure 2.8.

The summary performance metric results for the first application are shown in Fig-
ure 2.11 for our VMware Server platform. Results similar in nature were found for the
ESX platform. The overall results are consistent with expectations with the under- and
over-provisioned fixed cases defining the range of performance. The Fixed 2 approach is
clearly under-provisioned with a relatively high overload and error rate. The better slow
response rate for successful responses is deceptive because of the relatively high error rate.
The Manual approach provides the best performance across the set of metrics with the
HW=70,LW=35 algorithm providing competitive performance albeit with more resource
usage.

2.4 Related Work

There have been a lot of research works on the dynamic resource provisioning for appli-
cations. We categorize them by the three steps for local resource management as mentioned
at Section 1.2.1. Among the works that addressed the demand estimation and resource calculation needed for the applications, [28, 77, 77, 80] used queuing theory to model the behavior of the applications and determine how much capacity is needed to satisfy the SLA. Also, authors in [35, 74, 88] used models based on regression techniques to approximate the CPU cost for the application requests and plan the capacity for applications. In [78], authors proposed to profile of applications with low level monitoring, such as the number of system calls, to estimate the capacity requirement of the applications. Finally, [62, 63] used models based on control theory to adaptively plan the capacity of applications based on the measured utilization of servers and the required SLA. The method we used in this work falls into last category, except that we also introduced HW and LW, and only adjust the resource capacity of applications when the utilization goes below the LW or above the HW. In this case, we would significantly reduce the number of resource reconfigurations for the applications, with the drawback that some resources are wasted because we can remove certain amount of resource from the applications without causing the overloading of them.

Some works have addressed the problems of how to allocate the amount of the calculated
resources to the applications. [75] proposed an application placement algorithm with load shifting based on a min-cost max-flow model. The key observation is to co-locate residual memory and residual CPU on the same machines so that these resources can be used to start new application instances. [78, 86] used heuristic algorithms in which machines are searched through by the descending order of their load (also need to satisfy the requirement of other resource types, like memory) to be selected to host the application instances for overloaded applications. Our work also adopted this approach.

Finally, several works addressed the enaction of resource provisioning decisions. Authors in [33, 86] discussed the migration of VMs among the hosts to avoid the hot spots in the data centers. Different from their work, we focus on how to add new instances for applications in an agile manner. [17] proposed to suspend and resume the applications instances that are pre-deployed directly on physical machines to enact the resource provisioning decisions efficiently. This work is closely related with our work. The difference is that we addressed the issue in virtualized servers while they assume traditional environment. Also, our resource reassignment mechanism consists of multiple VM states allowing much larger flexibility in enacting the resource provisioning decisions.

### 2.5 Summary

In this work we have developed, implemented and tested our agile resource management mechanism in a virtualized data center. This mechanism makes use of ghost VMs, which participate in application clusters, but are not activated until needed by an application experiencing a significant load increase. Using this approach, our system exhibits good agility—being able to detect the need for and promote a ghost VM to be handling requests in 18 seconds. In comparison with legacy systems needing to resume VMs in the face of sharply increased demand, our approach exhibits much better performance across a set of
metrics. The mechanism has been deployed on multiple virtualization platforms providing different features. We found that the system demonstrates competitive performance when compared with scripted resource changes based on a known workload. It also works well when tested with multiple applications exhibiting periodic workload changes.
Chapter 3

Mega Data Center for Internet Applications

Modern cloud providers are building data centers with hundreds of thousands of servers, known as mega data centers. For example, Microsoft’s data center in Chicago can potentially pack up to 300,000 servers [8]. Mega data centers are especially attractive to host elastic clouds, which dynamically add or reduce resources allocated to applications based on the demand and which are particularly valuable for Internet applications where the demand is often hard to predict in advance. In this environment, mega data centers not only reduce the overall operating costs because of the economy of scale, but also promise better resource utilization through the statistical multiplexing of resource usage among the hosted applications.

However, to fully utilize this promise requires effective resource management across the entire data center. While efficient dynamic resource management is relatively well understood for traditional data centers, the scale of mega data centers brings new challenges. In this thesis, we pursue the conceptual design for scalable architectures of mega data centers. We first describe the challenges when managing the resources (servers, load-balancing el-
elements as well as access links which connect data centers and ISPs) in mega data centers. Then we propose the methodologies to efficiently manage the large number of servers and balance the load among the load-balancing elements and among the access links. Finally, we introduce several new architectures that can meet the requirements for efficient resource management in mega data centers and discuss their advantages and disadvantages. In our ongoing work, we are investigating concrete algorithms and evaluation strategies to study our proposed solutions. In particular, we concentrate on supporting elastic Internet applications where demand is driven by external client requests.

3.1 Challenges

With quantitative increase in scale of mega data centers, many traditional solutions in data center management and architectures no longer apply. We formulate some of these new challenges below.

1. Scalability of resource provisioning algorithms. The resource management in the data center aims to balance load among servers and other elements, minimize application placement changes, and increase utilization of the servers (e.g., to conserve energy). As shown in [75], this is an NP hard problem. A number of heuristic algorithms have been proposed but they do not work at the mega data center scale. For example, in the scheme in [75], the algorithm execution time increases exponentially with the increase of the number of managed machines and needs about half minute to create provisioning decisions for only about 7,000 servers and 17,500 applications. Also, in [86], it takes about 30s to manage 1500 virtual machines.

One way to address this problem is to use a distributed approach. Instead of having a central controller monitoring the demand and computing resource reallocation decisions, these tasks are distributed among agents, either at the host level or application
cluster level, as proposed in [14, 87]. However, distributed solutions cannot achieve
global optimality in the general case (although it is possible in some restricted set-
tings, e.g., in the case of content delivery networks that cache static files [83]).

2. Limitations of load-balancing elements. Traditional data centers commonly deploy
load-balancing (LB) switches to achieve fine-grained load balancing and fail-over
among replicated servers [45]. According to [37], 10% of devices in the studied
data centers were load-balancing switches. However, these switches do not scale to
the demands of a mega data center. For example, the maximum number of external
IP addresses for applications (known as virtual IPs, or VIPs) and real internal IP ad-
dresses for replicated servers (known as real IPs or RIPs) supported by Cisco Catalyst
switches are 4,000 and 16,000 respectively [44]. A naive way to support a greater
number of applications is to partition them among multiple switches; this however,
compartmentalizes the data center resources and diminishes the benefits of statis-
tical multiplexing. Furthermore, the maximum throughput of these load-balancing
switches is 4 Gbps [44]. Due to the extremely huge volume of traffic in the mega
data centers, this throughput limitation could potentially make these switches to be
the communication bottleneck.

3. Variety of load-balanced resources. A traditional data center typically has a small
number of links to the Internet, often a single default link and a backup, and a fairly
streamlined internal network organization. Thus, resource provisioning traditionally
has concentrated on server load. A mega data center can have multiple Internet links
and border routers as well as complex internal network topology, necessitating load
balancing among these components. In particular, in this work, given recent advances
in intra-data center network organization [16, 38, 60], we address the issue of load
balancing among Internet links that received less attention.
3.2 Approach

In this section, we outline our methodologies to address the challenges mentioned above. We first introduce a two-level hierarchical resource management architecture which is able to efficiently manage the resources of the entire data center. Then we propose to allocate multiple VIPs for each application and selectively expose VIPs to Internet clients to balance the load among the access links as well as among the LB switches. Finally, we discuss the management of VIPs/RIPs of LB switches, which also plays an important role in the dynamic resource management of the data centers. For specificity, we assume the load-balancing switch parameters of Cisco switches: 4,000 VIPs, 16,000 RIPs and 4 Gbps throughput throughout the discussion. Also, we target the mega data centers with about 300,000 servers and applications.

3.2.1 Hierarchical Resource Management

In the two-level hierarchical resource management architecture, servers are divided logically into groups, which we call pods. A local resource manager provisions resources to applications within each pod dynamically using an existing technique, e.g., as proposed in [75, 89]. The size of a pod (the number of hosts and deployed applications) should be small enough so that existing techniques can cope with the pod’s scale. In our architecture, given the reported scalability of existing resource managers [75, 89], we target each pod to have about 5,000 servers and 10,000 applications.

The local resource manager only knows the servers and applications of the local pod. When the local resource manager detects that the load of the some application is increasing, it will allocate more resources from the local pod to the application, for example, by creating more active instances on lightly loaded servers in the same pod; when the load is decreasing, it will reclaim resources back into the local server pool, e.g., by remov-
ing some application instances. For more information on local resource managers, refer to [75, 86, 89].

While local resource managers allocate resources within their pods, a datacenter-scale resource manager will monitor the resource utilization of all the pods and balance the load among them. We discuss datacenter-scale resource management further in Section 3.4.

3.2.2 Access Link Load Balance

Data centers are connected to Internet through border routers. Even traditional data centers are often connected to multiple ISPs for the reliability and economic concerns, and one can certainly expect this to be the case with mega data centers – not just for the above reasons but also to procure appropriate bandwidth. In order to reduce the cost, cloud providers may want the traffic to go through ISPs with lower price as much as possible yet avoid overloading of any access links. Thus, it is necessary to: (i) control the traffic among the different access ISPs according to the economic model and (ii) avoid overloading of any given access link.

One naive way for this is VIP transfer – we can remove the VIPs of some applications from the access routers associated with the overloaded access links and re-advertise them at other access routers with lightly-loaded access links. In this way, the traffic of these applications would go through the lightly-loaded access links, thus releasing the burden on the overloaded ones. However, it would lead to service disruption (these application instances would become unaccessible) during the transfer of VIPs. Another way is still keeping the VIPs at the access routers with overloaded access links when re-advertizing them. But since we have no control of the routing, large portion of the traffic of these applications may still come through the overloaded access links.

In this thesis, we propose to assign multiple VIPs to each application in a data center, and advertize them at different access routers. Then we apply an DNS-based mechanism to
selectively expose VIPs to Internet clients to balance the load among the access links. We refer it to be the selective VIP exposing. For example, when an access link becomes over-loaded, we can ask the DNS system to stop exposing the VIPs advertized on the associated access routers, and instead to choose VIPs advertized at access routers with lightly-loaded access links for Internet clients. In this way, new connections would go through lightly-loaded access links. Meanwhile, when detecting that the VIPs on the access routers with overloaded access links are not needed any more (existing sessions finish and new connections go through other access links), we can remove these VIPs and re-advertize them on other access routers. The motivation behind is that because of the limitation of the number of VIPs each load-balancing switch can support, the number of VIPs each application can have is also limited. If we do not use these VIPs any more, they would be wasted. Moreover, by re-advertizing these VIPs at access routers with lightly-loaded access links, we can use the same approach to handle the overloading which occurs at different access links (in that case, these re-advertized VIPs become the candidates to be exposed to Internet clients). Note that, if we choose to keep these VIPs at the access routers with overloaded access links, we can not use them until these access links recover from the overloading.

The more VIPs each application is allocated, the more flexibility we would have for the load balance. However, the number of VIPs for each application should not be too big, otherwise, we would need too many load-balancing switches to host large number of applications in a mega data center due to the VIP number limitation of the load-balancing switches, thus the cost is larger. In our current work, we assign 2 VIPs per application on average (popular applications are assigned more than unpopular applications) by default. The tradeoff between the flexibility for load balance and the overall cost of LB switches would be further evaluated quantitatively in our ongoing work.
3.2.3 Load-balancing Switch Load Balance

When the load of a load-balancing switch is approaching the throughput limitation (4G), we need to relieve the load from it. Selective VIP exposing mentioned above can also be applied here – we can choose to expose only the VIPs of the applications configured at other lightly-loaded load-balancing switches until the overloading disappears. Note that, similar with access link load balance, we can not simply transfer some VIPs from overloaded load-balancing switches to other lightly-loaded switches when these VIPs are still used by Internet clients because of the session affinity issue – load-balancing switches need to distribute the requests of the same session to the same real server and only the original load-balancing switch knows how to do that. We can transfer the VIPs to other load-balancing switches only when detecting that no Internet clients are using them.

3.2.4 VIP/RIP Management

As mentioned before, given the VIP/RIP limitation of LB switches, if we just simply partition the applications among the LB switches, it would result in the resource compartmentalization in data centers, since the applications configured at each LB switch are only able to access the resources reachable by the associated LB switch respectively. By assigning each application multiple VIPs and configuring the VIPs as well as the VIP-to-RIP mappings across multiple LB switches, we can significantly reduce the resource compartmentalization problem (note that, to completely get rid of the problem, we also need the datacenter-level resource management strategies, e.g., dynamic application deployment and server transfer, as discussed in Section 3.4).

Meanwhile, the VIP/RIP management is also necessary for the local resource management in each pod. As mentioned at Section 1.1, instances of an application form an application level clusters. For each application, an VIP that is externally visible is configured at
the LB switch and is mapped to the a set of RIPs which are the IP addresses of the VMs hosting the application instances. When a new instance of an application is created, an RIP needs to be allocated at one of LB switches configured with the VIPs of that application and an VIP-to-RIP mapping should be generated accordingly.

One way for managing the VIPs/RIPs is to put all the LB switches as the shared resources for all the control elements that need to update the configurations of LB switches (like the local resource managers of the pods, the control knobs for access link load balancing and LB switch load balancing, etc.). However, this distributed method has many drawbacks. Firstly, it needs a complex synchronization scheme for the system to work properly. More importantly, it is very hard to support the prioritized VIP/RIP management. For example, allocation of RIPs when new instances are created to absorb more demand should have higher priority than reclamation of RIPs when application instances are removed due to the demand drop. In addition, distributed method is hard to achieve optimal solutions in general sense.

In this thesis, we have a dedicated control knob – VIP/RIP manager, to manage the VIPs/RIPs. The VIP/RIP manager maintains several queues of different priorities, each storing the requests with the same priority. When a control element wants to update the VIP/RIP, it sends a request to the VIP/RIP manager, which is put at one of the queues with the right priority. The VIP/RIP manager processes the requests among the queues in the decreasing order of priorities; within each queue, first-come-first-serve policy is applied.

### 3.3 System Overview

In this section, we discuss the high-level view of the system that meets the requirements mentioned above.
3.3.1 Basic Architecture

The basic architecture proposed for mega data centers is as shown in Figure 3.1. As mentioned above, servers are grouped into pods. Applications, e.g., www.cnn.com, are allocated with multiple VIPs (VIP1 and VIP2 in this example) configured at different load-balancing switches to facilitate the load balancing among the access links and among the load-balancing switches as discussed above.

An important feature of this architecture is that all the load-balancing switches are positioned at the access network, forming a load-balancing layer, which is connected to the access routers with powerful border routers on the one side, and is connected to server pods with l2/l3 switches on the other side. In traditional data centers, LB switches are placed close to, or directly connected to servers, which limits the number of servers they are able to reach, and thus limits the amount of resources the applications configured at each
load-balancing switch can obtain. This greatly diminishes the elastic resource provisioning for applications – cloud computing promises to be able to allocate as much resources as possible to applications when needed. One can configure the applications at a lot of load-balancing switches to achieve the elastic resource provisioning, but it would significantly increase the number of LB switches. For example, suppose that there are 300,000 applications in the data center, each application requiring one server to process its demand. Also suppose that an LB switch can only reach 100 servers. Then, 3000 LB switches are needed to configure these applications in order to be able to access 300,000 servers. By putting the LB switches at the access network, each load-balancing switch is able to reach all the servers in the data center, thus only about 80 LB switches are needed if each application is configured with one VIP (since we propose to have multiple VIPs for each application as discussed in Section 3.2.2, more LB switches are needed actually, but still much less than the way in traditional data centers), which greatly reduces the cost.

Moreover, positioning the load-balancing switches at the access network allows us to be compatible with existing interconnection architectures. Recently, several data center network architectures have been proposed [16, 38, 60] to improve the bandwidth between any pair of hosts, provide flat address space to servers, etc., – issues that are of critical importance to next generation data centers. Our new architecture must not negate these gains. In order to do that, we just use L2/L3 switches to connect the load-balancing layer to the existing interconnection architectures – without any change of them.

**Issues.** In the basic architecture, the following issues need to be addressed in order to effectively balance the load among the access links, pods and the load-balancing switches.

1. **Scalability.** Let $A$ be the number of applications hosted in the data center, and $L$ be the number of LB switches at the demand distribution layer. Also, assume that there are $k$ VIPs assigned for each application on average. Then the total number of
possible ways to pack the applications among the LB switches is about $A \times L^k$, subject to the limitation of LB switches (4000 VIPs, 16000 RIPs and 4 Gbps throughput). Given the large number of applications and the exponential increase property of the problem size, the management of the LB switches at the load-balancing layer would potentially suffer from scalability problem.

2. Policy conflicts. As discussed above, the load balance among the access links, among the servers and among the LB switches themselves all involves the management of the LB switches at the load-balancing layer. When balancing the load among the access links, we would expose the VIPs advertised at access routers with lightly-loaded access links to relieve the burden on the overloaded ones. At the same time, however, if these VIPs may be mapped to real servers at pods with high utilization, we would try to reduce the amount of traffic going through these VIPs to avoid the overload of these pods. Thus the policies for balancing the load among the access links may conflict with the policies for balancing the load among the pods. For the same reason, there could also be conflicts between policies to balance the load among access links and the policies to balance load among the load-balancing switches when the VIPs are all configured at highly-loaded load-balancing switches (approaching 4 Gbps throughput). In order to avoid the first conflict, we need to reduce the load of the corresponding pods or dynamically remap the VIPs to RIPs of application instances running at lightly-loaded pods before applying the selective VIP exposing. For the second conflict, we should also release the highly-loaded LB switches before selectively exposing VIPs. These conflicts can potentially delay the effectiveness of the load balance behaviors. We would conduct quantitative evaluation for this in our ongoing work.
3.3.2 Dedicated LBG Architecture

To address the potential scalability problem for the basic architecture, we can divide the load-balancing switches into groups (LBGs), each is dedicated to a pod, as shown in Figure 3.2. The local resource manager in each pod only knows the LB switches of the LBG in front of the local pod and takes care of management of LB switches locally. Since each pod has about 10,000 applications, as discussed in Section 3.2.1, only about 3 LB switches are needed to support them in each LBG in terms of VIPs. In this way, the management of LB switches is distributed among all the local resource managers – each one is able to handle the tasks efficiently at a much smaller scale. Note that, the datacenter-scale management of LB switches are still needed, but with much lighter weight, as discussed in Section 3.4.

One drawback of this architecture is that it needs more load-balancing switches. For example, assuming that there are 300,000 servers and 300,000 applications in the data
center, and 2 VIPs are assigned to each application, there would be 60 pods and each LBG has 3 load-balancing switches. So totally 60*3=180 switches are needed. In the basic architecture, 300,000*2/4000=150 switches are needed.

3.3.3 Two-layer Architecture

In order to relieve the policy conflicts and make the management of LB switches simpler, we can decouple the different load balance tasks by adding a new LB switch layer, which we call demand distribution layer, as shown in Figure 3.3. In this architecture, the external VIPs of each application is configured at the LB switches at the demand distribution layer. Meanwhile, each external VIP is mapped to several middle-layer VIPs configured on LB switches of the load-balancing layer; we call these middle-layer VIPs m-VIPs. To conserve m-VIPs (recall that each switch can handle only a limited number of VIPs), external VIPs of each application can map to the same set of m-VIPs. Finally, each LB switch in the load-balancing layer with an m-VIP maps this address to a group of RIPs configured on the servers. Note that, m-VIPs and RIPs are both private. We call the pods that host active application instances covered pods for that application.

As an example, suppose we want to deploy application ”www.cnn.com”. The domain name is resolved to 2 VIPs: VIP1 and VIP2, which are configured on the LB switches of the demand distribution layer. Then, each VIP1 and VIP2 is mapped to m-VIP1, m-VIP2,...,m-VIPN. Finally, each m-VIP is mapped to RIPs of servers in the corresponding pod that has active ”www.cnn.com” instances (typically a virtual machine). When a request arrives at the data center for application ”www.cnn.com”, it first reaches one of the LB switches of the demand distribution layer depending on the VIP returned from the DNS system. LB switches forward requests belonging to an existing connection, or (optionally) arriving from a recently seen client, to the previously selected m-VIP for this connection (client) for session affinity. Otherwise, depending on the load-balancing policy at the LB switch, the
request will be delivered to a newly selected m-VIP. Finally, the LB switch with the m-VIP would forward the requests to the appropriate server in its pod.

In the two-layer architecture, the selective VIP exposing for balancing the load among the access links only involves LB switches at the demand distribution layer. Also, we only need to consider the LB switches at the load-balancing layer when balancing the load among the pods (we might also need to update the weight of the m-VIPs of LB switches at the demand distribution layer as discussed at Section 3.4.4, but it gives rise to no conflicts with load-balancing tasks pursued at the demand distribution layer). In this way, the access link load balance and the pod load balance are decoupled, each limited to a separate set of LB switches respectively.

While the addition of the demand distribution layer simplifies the management of LB switches, it increases the number of LB switches needed in the data center - we need to have
enough LB switches at the demand distribution layer to host the VIPs of all the applications as well as providing enough aggregate throughput.

3.3.4 Summary

As we can see from the discussion above, different architectures have different advantages and disadvantages. Extensive study of data center traces and quantitative evaluation are necessary in order to make a proper selection among these architectures, which is our ongoing work.

3.4 Datacenter-Scale Resource Management

Datacenter-scale resource management plays a critical role in our hierarchical resource management. In particular, it must avoid overload of pods, load-balancing switches and access links.

A pod can be overloaded for several reasons. Its servers may be overloaded with no spare capacity within the pod for the local resource manager to resolve the issue. A more subtle issue is that the local resource manager itself may become overloaded with too many servers and applications in the pod, slowing down its resource allocation algorithms beyond acceptable levels.

With regard to the management of LB switches, the datacenter-scale resource management has different tasks in different data center architectures. For the basic architecture, the datacenter-scale resource management needs to manage the VIPs/RIPs of all the LB switches as well as balancing the load among them.

For the dedicated LBG architecture, with the local resource manager of each pod managing the LB switches at the local LBG, the datacenter-scale resource manager is responsible for managing LB switches across the LBGs. For example, if the local resource manager...
of some pod detects that the local pod has no spare capacity to process the received requests, then the dynamic application deployment strategy (discussed in Section 3.4.2) can be applied to deploy the popular applications at other lightly-loaded pods, in which case the datacenter-scale resource manager will need to update the configuration of the LBGs in front of those pods accordingly.

Additional management tasks are needed at the demand distribution layer for the two-layer architecture, e.g., updates of LB switches when balancing the load among the access links, etc. As mentioned above, however, it simplifies the management of LB switches at the load-balancing layer.

Finally, it must be able to shift traffic among access links to avoid access link overload.

In the rest of this section, we describe the knobs available to control load at these components.

3.4.1 Server Transfer

As mentioned above, pods are formed logically by the configuration of RIP addresses in the pod’s LB switches. (In fact, a physical server can in principle be shared by pods with each pod “owning” different virtual machines on the server; however, we do not anticipate practical need for this possibility.) Furthermore, switches allow programmatic reconfiguration of their VIP-to-RIP maps. The combination of these two features enables a flexible knob for resource reallocation among the pods. When the global resource manager detects an overloaded pod, it can allocate more resources to it by transferring servers from lightly loaded pods. To this end, it must ask the local resource managers in the donor pods to vacate some servers (remove any application instances running there and remove the corresponding RIPs from the switches’ VIP-to-RIP maps). The vacated servers can then be handed to the local resource manager of the recipient pod.

When applying this strategy, we must avoid ”elephant” pods. Some applications in a
given pod may become so popular that the global resource manager might add a large number of servers to the pod. As discussed earlier, this can hamper the operation of the local resource manager.

### 3.4.2 Dynamic Application Deployment

When the overloading is detected at some pods, instead of adding more resources to busy pods, we can shift the load to other pods by migrating or replicating applications to underloaded pods. This knob is enabled by recent advances in efficient virtual machine migration [55, 86] as well as the ability to configure programatically new VIPs on load-balancing switches. Similarly, if an underutilized application is deployed at many pods, the global resource manager can remove all instances of this application from the busier pods. This in particular releases the applications’s VIPs in these pods to be used for other, more popular applications. Note that any change in the set of pods covered by an application requires reconfiguration of the LB switches at the load-balancing layer.

Again, the global resource manager must avoid elephant pods here, since an underutilized pod can be easily selected to receive many applications, which can slow down the local resource manager beyond acceptable levels. Yet another consideration is that the number of application deployments and removals must be minimized as they are very resource-intensive and can create turbulences in the platform operation.

### 3.4.3 Weight Adjustment

Load-balancing switches allow programmatic change to the weights they use in their load-balancing algorithms. In the basic architecture, we can adjust the load among the pods for each application by updating the weights of the RIPS at the LB switches of the load-balancing layer; in the two-layer architecture, we update the weight of the m-VIPs at the LB switches of demand-distribution layer to balance the load among the pods. (In the
dedicated LBG architecture, the local resource managers can do the similar thing at the LB switches of LBGs fronting their pods to balance load among its servers.)

One advantage of this knob is that the resultant change can occur quickly, leading to highly agile resource management. Indeed, configuring the load balancing switches takes only several seconds. At the same time, this knob can only redistribute the load among the pods with application instances deployed already.

3.4.4 Selective VIP Exposing

As discussed above, we can selectively exposes VIPs to Internet clients to balance the load among the access links as well as LB switches. When detecting that overload occurs, this control knob would interact with the DNS system and suggest it to expose VIPs that are advertized at access routers with lightly-loaded access links or configured at lightly-loaded LB switches to Internet clients.

One limitation of the selective VIP exposing is that Internet clients usually cache the IP addresses returned from the DNS system for some time (known as time to live, or TTL), during which the Internet clients use the cached IPs instead of turning to the DNS system for IP addresses to access the applications. Thus the selective VIP exposing is effective to relieve the overloading of access links or LB switches only when the TTL expires. In order to address this limitation, we can assign small TTLs for the hosted applications (this approach is already adopted by some service delivers for more effective dynamic server selection, e.g., Akamai uses 20 seconds as the TTL for its delivered services). Also, we can use lower alerting watermark to invoke the selective VIP exposing earlier so that the load of the access links or LB switches would not be too high when the TTL expires. We would study the effect of this limitation in our ongoing work.
3.4.5 Summary

Our architecture makes a large number of control knobs available to manage resources across the data center. Their sheer number increases the decision space and makes computing proper policies especially challenging. Moreover, most of the knobs affect each other, further complicating the issue. For example, weight adjustment would be able to balance the load more effectively if more pods were covered by popular applications through dynamic application deployment. We are investigating algorithms involved in our ongoing work.

3.5 Related Work

Several new architectures for data centers have been proposed [16, 38, 60]. These approaches concentrate on intra-DC networks. Our work extends these efforts by addressing the problem of scalable architectures for supporting elastic Internet applications. The approach in [39] suggests to use common servers as load balancing elements. Our architecture assumes hardware switches.

Authors in [14, 87] proposed distributed approaches to address the scalability problem in data centers. In particular, [14] mentions a hierarchical scaling method, which envisions a layer on top of cluster-level resource managers, each managing up to 32 hosts and 3000 VMs. Our pods are decoupled from physical connectivity, facilitating logical pod formation; this allows much larger pods and dynamic transfer of servers among the pods.

Further, unlike the above works, we consider the scalability of load-balancing fabric itself as well as load balancing among access links of the data centers.
3.6 Conclusion

This thesis outlines a scalable architecture that supports datacenter-wide resource management for elastic Internet applications in a mega data center. Our architecture includes a scalable load-balancing fabric and provides effective knobs to balance load among the applications, servers, access links, as well as the load-balancing components themselves – the pod resource managers and switches in the load-balancing fabric. This work is still at the preliminary status. In our ongoing work, we are investigating algorithms for the resource management in this environment and pursuing extensive evaluation of our proposed solutions.
Part II

Global Resource Management
In global resource management, we address the global application placement and server selection problem. We proposed a unified approach to handle these two problems simultaneously according to the dynamic demand from Internet clients. In addition, we proposed a novel client-side DNS architecture to improve the DNS-based server selection. In the following sections, we discuss them in details.
Chapter 4

Unified Approach for Application Placement and Server Selection Across Data Centers

When addressing the global application placement and server selection problem, not only the load of data centers needs to be considered - so that no data center is overloaded, but the proximity between client requests and the data centers should also be taken into account - in order that client requests are served by close service replicas.

While these two problems are addressed traditionally in isolation, this thesis proposes a unified way to solve them systematically based on min-cost optimization model. Most existing works either only consider the application placement problem assuming client requests are forwarded to the closest replicas, or only deal with the server selection problem given an application placement strategy. These works either ignore the fact that data centers can be overloaded if too many requests are forwarded to them or are suboptimal due to the fact that the assumed application placement strategy may not be good. Because of the dynamics of the demand of Internet applications - both the amount of volume and the
location, global resource management needs to dynamically handle these two problems simultaneously.

4.1 System Overview

The high-level view of our targeted environment is shown in Figure 4.1. Each client connects to a request-routing component (e.g., DNS server or HTTP redirector), which directs it to a data center with instances of the requested application. Known mechanisms (such as one provided in WebLogic [3]) ensure continued session state availability even if a client is redirected to a new instance mid-session. We assume each application accesses a back-end database when processing a request. While some cloud platforms offer a database service to hosted applications, application providers often prefer to maintain their databases in their own control for security or legal reasons. We assume the latter arrangement. Thus, user-perceived performance is affected by the network distance from the client to the data center and from data center to back-end database at the customer premises. In this work, we consider the aggregate distance by adding these two distances together. It is straightfor-
ward to accommodate multiple back-end servers and more complex combinations of these
distances by simply replacing the cost function in our models.

Due to the large number of Internet clients, the platform is unlikely to maintain the pro-
ximity information for all clients. Thus we assume a clustering technique such as in [51],
which groups the clients based on the intersections of IP prefix entries found in multiple
BGP tables. We refer to the clients that are in the same group as a client cluster (CC). A
key part of the present research is a method to aggregate demand further so that computing
the application placement and demand distribution policies becomes tractable.

Many cloud providers concentrate their platforms in a small number of strategically lo-
cated mega data centers, leading to the factors of 5 to 7 decrease in the operational cost [21].
For example, Amazon’s EC2 cloud is deployed in four locations (US-East, US-West, Ire-
land, and Singapore). Other shared infrastructure providers, such as Limelight and AT&T,
have a couple dozens data centers. In this work, we assume roughly this number of data
centers (around 20). There are examples of shared infrastructures (Akamai) with presence
in thousands of locations. Our approach does not target these environments.

Different business models exist regarding quality of service. In some models, the cloud
provider exports an API to customers to add or remove their application instances in num-
bers and locations as the customers see fit. In other models (e.g., auto-scaling option in
EC2 or Google’s AppEngine), the cloud itself makes these decisions for the hosted appli-
cations. In this work, we target the latter model, with a goal of maximizing the performance
of the hosted applications while keeping the number of data centers where these applica-
tions are deployed to a minimum. Removing underutilized application instances keeps the
customer costs down and frees up resources for other applications. Our additional objec-
tive is to reduce the number of placement changes in consecutive configurations. Despite
recent advances in reducing the overhead of starting a virtual machine [54] or an applica-
tion server [17], deploying an application instance is a heavy-weight operation in terms of
CPU costs and system reconfiguration; fewer such changes result in smoother operation and better caching behavior in the system [64].

In making application placement decisions, we only consider whether or not an application is deployed in a data center. Others have addressed the problem of resource allocation among applications within a data center [75, 89]. Thereby, an ”application instance” means that the application is deployed at the data center, regardless of the amount of resources it is assigned locally.

Making intelligent application placement and demand distribution decisions requires monitoring the current demand and utilization of data centers. We assume a central controller collects this information periodically from each data center. We further assume that our applications (i.e., web sites) are sufficiently popular so that even if different requests to a web site may have different service demands, for a reasonable request rate (e.g., higher than the deletion threshold - see Section 4.3), these requests will result in a representative request mix. Thus, a given request rate would translate to the corresponding proportional utilization. (The experience with our prototype provides evidence of the validity of this assumption – see Section 4.6.) We can therefore measure data center utilization by the request rate and distribute the demand by assigning appropriate probabilities of directing requests from a particular client cluster to a particular data center. In particular, in our prototype, this is done through resolving clients’ DNS queries for hosted Web sites to different data centers with appropriate probabilities.

4.1.1 Problem Statement

Let \( D \) be the number of data centers, \( A \) the number of applications and \( C \) the number of client clusters. The application placement configuration can be described as an \( A \times D \) matrix \( P \), with element \( P_{ij} = 1 \) if application \( i \) is deployed at data center \( j \); \( P_{ij} = 0 \), otherwise. The demand distribution policy is an \( A \times C \times D \) matrix \( R \), whose elements \( R_{mn} \) are the
fractions of requests from client cluster \( m \) for application \( a \) to be directed to data center \( n \). The system enacts the distribution policy by directing a request from client cluster \( m \) for application \( a \) to data center \( n \) with probability \( R_{amn} \). Let \( r_{am} \) be the request rate for application \( a \) from client cluster \( m \). Assume each request is associated with a cost \( C_{amn} \) if it is served at data center \( n \), and \( u_n \) is the utilization of this data center. We can formulate our problem as a *multi-objective optimization problem* [58]. Ideally, our goal is to compute application placement \( P \) and demand distribution \( R \) that fulfill the following competing objectives:

\[
\text{Minimize } \sum_{a=1}^{A} \sum_{m=1}^{C} \sum_{n=1}^{D} r_{am} R_{amn} C_{amn} 
\]

subject to

\[
\sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} R_{amn} \leq u_n, n = 1, 2, \ldots, D
\]

\[
0 \leq R_{amn} \leq 1; \sum_{n=1}^{D} R_{amn} = 1
\]

\[
P_{an} \in \{0, 1\}
\]

\[
R_{amn} > 0 \text{ implies } P_{an} = 1
\]

where \( P_{an}^{prev}, a = 1, \ldots, A, n = 1, \ldots, D \) is the previous placement configuration. Objective
minimizes the overall cost. While $C_{\text{ann}}$ is an abstract cost function, we use the aggregate distance as the cost function thus trying to minimize the overall user-perceived network latency. Objective (2) minimizes the number of data centers with deployed application replicas while objective (3) tries to minimize the number of placement changes.

While multi-objective optimization problems are commonly dealt with by combining all the objectives into a single one with some weights assigned to each objective, in our case, this would transform the problem to a mixed integer programming formulation (in fact, a variant of a multicommodity – also referred to as multiactivity – capacitated facility location problem [15]), which is NP-hard. Further, choosing appropriate weights for different objectives is difficult in our context as their effect on the final policies is indirect and non-intuitive. Instead, we deal with the multiple objectives heuristically as discussed below.

4.1.2 Framework

Our heuristic approach to arbitrate among the competing objectives involves the following steps. First, we compute optimal request distribution among data centers assuming every application is deployed at every data center (full deployment). Here any optimization technique can be applied. For example, [82] uses a decentralized approach. We explore a centralized approach. As discussed in Section 4.2, we propose a min-cost max-flow model to solve the problem.

Second, given the optimal demand distribution policy, we attempt to remove underutilized instances. We introduce a Deletion Threshold (DT) as the level of demand that justifies the cost of running an application at a data center (note that this threshold can be selected independently for each application and has an easily grasped intuitive meaning). We try to remove instances whose demand share computed in the first step is smaller than DT by reassigning their flows to remaining instances in an optimal manner (refer to Section 4.3).
We also attempt to reduce the number of placement changes (objective (3)) in this step by assigning lower deletion threshold to already-deployed instances (Section 4.3.3). This step results in a desired application placement.

Finally, we revisit demand distribution, now with the desired deployment, using a series of small min-cost max-flow models to fine-tune our demand distribution policy as discussed in Section 4.4.

4.2 Demand Distribution with Full Deployment

As mentioned before, we begin by obtaining optimal demand distribution policy under the assumption of full deployment. We use a min-cost max-flow optimization model for this purpose. This model represents the system as a directed network, with source nodes generating demand, sink nodes consuming this demand, and demand flowing from sources to sinks along edges labeled with \((\text{cost}, \text{capacity})\). An edge label indicates the maximum amount of demand that can traverse this edge and the unit cost of such traversal. There are efficient algorithms that solve the min-cost max-flow problem, e.g., find the assignment of demand to edges that maximizes the total satisfied demand while minimizing the total cost. Refer to [13] for more details about min-cost flow problem and to [4] for transforming it to a min-cost max-flow problem. We use both terms interchangeably below. In our implementation, we use the tool by Cherkassky and Goldberg [11].

4.2.1 Problem Modeling

We would like to forward client requests to closest data centers and at the same time avoid overloading any data centers. We assume the service does not degrade appreciably as long as data center utilization is below its capacity. (In reality, this means that utilization must stay below a certain \textit{watermark}, which for now we view as capacity but set as a
parameter in the simulation – see Section 5.4.) Under this notion, we model our problem as the following min-cost flow network.

Because of different back-end servers, requests for different applications from the same client may have different aggregate distances to the same data center. Thus our model can not simply consider all demand from the same client cluster as a whole. Therefore, as shown in Figure 4.2, we have a pair-node $Y_{am}, a = 1, 2, ... A, m = 1, 2, ... C$ for each application and client cluster pair $(a, m)$. Also, each data center $n$ has a node $DC_n$. Finally, we have a source node $S$ and sink node $T$. From node $S$ to each pair-node $Y_{am}$, we add an edge with cost 0 and capacity $r_{am}$, the latter being the request rate from client cluster $m$ for application $a$. Then we add an edge from each pair-node $Y_{am}$ to each data center node $DC_n$, with cost being the aggregate distance when client cluster $m$ accesses the application $a$ at data center $n$, and infinite capacity (since the actual data center capacity is enforced by the subsequent edge). Finally, there is an edge from every data center node $DC_n$ to the sink node $T$, with cost 0 and capacity equal to the capacity of data center $u_n$.  

Figure 4.2. Min-cost network model
In this model, we try to move the total amount of flow \( \sum_{a=1}^{A} \sum_{m=1}^{C} r_{am} \) from the source node \( S \) to the sink node \( T \) with minimum cost. After we obtain the solution, flow \( f_{amn} \) on the edge between pair-node \( Y_{am} \) and data center node \( DC_n \) represents the amount of requests that should be forwarded to the data center \( n \), among all the requests coming from client cluster \( m \) for application \( a \).

### 4.2.2 Permutation Prefix Clustering

The min-cost flow problem in Figure 4.2 is extremely large, with the number of edges \( A \cdot C + A \cdot C \cdot D + D \), dominated by \( A \cdot C \cdot D \). According to [51], there are about 400,000 client clusters. Then, for \( C = 400,000 \) client clusters, \( A = 100 \) applications, and \( D = 20 \) data centers, the number of edges is in the order of \( 8 \times 10^8 \), making this problem intractable. In this section, we discuss our clustering technique of Internet client requests, which dramatically reduces the number of edges in Figure 4.2.

### 4.2.3 Basic Idea

With aggregate distance, each pair-node \( Y_{am} \) has its own preference of data centers in terms of proximity, which produces a permutation of data centers. For example, permutation \{1,4,2,3,6,5\} means there are six data centers and the requests for a given application from a given client cluster are the closest to \( DC_1 \), the second closest to \( DC_4 \), and so on. We define each permutation as a *region*, and client requests with the same preference of data centers fall into the same region. There is a region for each pair-node in Figure 4.2. We propose a method we call *permutation prefix clustering* to reduce the number of regions and hence the number of edges in Figure 4.2.

In this method, we merge regions if their corresponding permutations share the same prefix of certain length. For example, suppose there are 10 data centers. Let *region* \(_1\) have permutation \{1,2,3,4,5,9,6,8,10,7\} and *region* \(_2\) have permutation \{1,2,3,4,5,10,9,6,8,7\}. If
we set the prefix length to be 5, we could merge the region_1 and region_2 into one region_{12} with a prefix permutation \{1,2,3,4,5\}.\footnote{Note that different applications coming from the same client cluster may end up in different regions because their data center preferences may be completely different due to different back-end data centers.} After merging, we compute the distance from the new region to each data center, including those beyond the prefix, as the weighted average of the distances from region_1 and region_2, with request rate from each region as the weight.

Our observation behind this method is that unless most of the data centers are highly loaded, requests for an application will only go to a small number of closest data centers. So for each client request, we actually only need to consider the data centers in the front part of its corresponding permutation. Admittedly, there would be proximity penalty when the flows do need to go to the data centers beyond the prefix. However, this happens when most data centers are highly loaded, in which case the proximity becomes less of a priority as we need to satisfy all the demand first. Moreover, since we use weighted average distance to measure the proximity to data centers, including those beyond the prefix, we can reduce this penalty to some extent. This last observation is confirmed in our simulation in Section 4.5.3.

4.2.4 Application to Min-Cost Model

To illustrate how permutation prefix clustering is applied in our min-cost flow model, suppose we want to merge the regions for pair-nodes Y_{1C} and Y_{am} in Figure 4.2 because their permutations share a prefix. We will remove nodes Y_{1C} and Y_{am} and all their adjacent edges. We replace them with a new node Y'. An edge is added from source node S to node Y', and from Y' to each node DC_n, n = 1, 2, ..., D. The cost of the edge from S to Y' is still zero and capacity is the sum of the capacities of the edges (S, Y_{1C}) and (S, Y_{am}), or r' = r_{1C} + r_{am}. The cost of the edge (Y', DC_n), n = 1, 2, ..., D is the weighted average of cost of edges (Y_{1C}, DC_n) and (Y_{am}, DC_n), or d'_n = \frac{d_{1C}*r_{1C}+d_{am}*r_{am}}{r_{1C}+r_{am}}, and capacity is still \infty. The
updated network is shown in Figure 4.3. This technique generalizes trivially to merging more than two pair-nodes.

Let \( L \) be the length of the permutation prefix. Then the total number of possible regions after merging is:

\[
\min\{A \cdot C, \prod_{i=0}^{L-1} (D - i)\}
\]

which means the same number of merged pair-nodes \( Y' \). So after the region merging, the total number of edges in Figure 4.2 is reduced from about \( A \cdot C \cdot D \) to about:

\[
D \cdot \min\{A \cdot C, \prod_{i=0}^{L-1} (D - i)\}
\]

Since \( A \cdot C \) is very large, the total number of edges in Figure 4.3 is actually about \( D \cdot \prod_{i=0}^{L-1} (D - i) \), depending on \( D \) and \( L \) only. Generally, the smaller the prefix length, the smaller the size of the problem but the larger the potential proximity penalty. We study these effects in Section 4.5.3.
4.3 Application Placement

A solution to the model of Figure 4.3 provides a demand distribution policy assuming full deployment. Our next step is to reduce the level of deployment by eliminating underutilized application instances.

Let $f_{an}$ be the amount of request flow of application $a$ assigned to data center $n$. If $f_{an} \geq DT$, we refer it as normal flow and keep the instance of application $a$ at data center $n$. We denote the set of data centers with these instances as $U_a$. We can immediately remove an instance with zero demand (e.g., if $f_{an} = 0$). The rest of this section deals with the other instances with the assigned demand $0 < f_{an} < DT$. We refer to these instances as tiny instances and their flows as tiny flows. We attempt to remove these instances to the extent possible, as long as their flows can be accommodated by the data centers with the remaining instances.

4.3.1 Heuristics

Let set $V_a = \{DC_n | 0 < f_{an} < DT\}$ contain data centers with a tiny instance of application $a$. Also let $h_n$ be the number of normal flows data center $DC_n$ has. We first assume that all tiny instances are removed (unless it is the only instance of the application) along with their flows and increase the residual capacities of the affected data centers accordingly. We then attempt to distribute these flows (referred to as residual demand) to data centers with residual capacities. Our procedure is guided by the following observations:

1. We should try to remove the instances with the smallest flows first because the reassignment of small flows will result in lower latency penalty. In particular, it means that (1a) demand for a tiny instance should not be reassigned to an even tinier instance, and (1b) we should try to accommodate smaller flows (across all applications) first.
2. If we have to retain some tiny instances (because data centers in $U_a$ reach their capacity), we should keep the tiny instances with the largest flows first, and use them to accommodate more demand from other tiny instances. This is again motivated by the desire to keep the largest amount of demand assigned to optimal data centers.

3. When selecting data centers in $U_a$ to assign residual demand, we should favor the ones with the smaller $h_n$ because their residual capacity is harder to utilize (since they can only accept additional demand for the applications they host).

While the above set of heuristics might suggest a simple greedy procedure, where we reassign flows in the increasing size order and distribute them to normal instances first and then to the largest tiny instance with residual capacity, such a procedure may result in highly suboptimal flow assignment. Instead, we again build a min-cost flow model for this problem, so that we reassign the residual demand optimally, and at the same time manipulate the costs in the model to follow the above heuristics.

4.3.2 Tiny Flow Removal

Our min-cost flow model for tiny flow removal is depicted in Figure 4.4. Each tiny flow $f_{an}$ has a corresponding node $RD_{an}$, referred to as demand node. From source node $S$, we add an edge to every demand node. Also, from each demand node $RD_{an}$, there is an edge to data center node $DC_k$ if the latter has an instance of application $a$ and $f_{ak} >= f_{an}$. By not including edges to data centers with smaller flows (note the absence of edges from $RD_{an}$ to $DC_1$ and $DC_{D-1}$ in Figure 4.4), we enforce heuristic 1a. Finally, each data center node is connected to the sink node $T$.

For edges from source node $S$ to demand node $RD_{an}$, the capacity is $f_{an}$, and the cost is 0 - this represents the demand to be satisfied. All edges from demand node $RD_{an}$ to data center nodes have capacity $f_{an}$ (this demand could potentially be satisfied by any of these
data centers), and the edges from data center nodes to the sink have capacities equal to the residual capacity $rc_n$ of each data center. For data center $DC_k \in U_a$, the cost of the edge from node $RD_{an}$ to $DC_k$ is 0 (since it already has an instance and we would like to assign as much demand as possible to these nodes – see the edge from $RD_{an}$ to $DC_2$ in the figure).

The cost of other edges is chosen in a way such that:

1. For any two tiny flows $f_{an}$ and $f_{a'n'}$, if $f_{an} < f_{a'n'}$ then $cost_{i,j}$ of edges going from demand node $RD_{an}$ to data center nodes in $V_a$ is larger than $cost_{i',j'}$ of edges going from $RD_{a'n'}$ to data center nodes in $V_{a'}$. In this way, flow $f_{an}$ would have an advantage over $f_{a'n'}$ when competing for residual capacity of data centers with instances of both applications, thus following heuristic 1b.

2. The cost of edges going from residual node $RD_{an}$ to $DC_k \in V_a$ is inversely proportional to $f_{ak}$. In this way, the min-cost flow algorithm will try to follow heuristic 2.

3. For the edge from data center node $DC_n$ to the sink node $T$, the cost is the number of
normal flows \( h_n \) data center \( DC_n \) has. In this way, the min-cost flow algorithm will try to follow heuristic 3.

4. Because heuristics 1 and 2 have higher priority than 3, we make sure that the cost of edges from demand nodes \( RD_{an} \) to data center nodes in \( V_a \) dominates the cost of edges from data center nodes in \( V_a \) to the sink node. In Figure 4.4, \( C_{a,n} >> h_k \) and \( C_{a,D} >> h_k \) for all \( k = 1, 2, \ldots, D \).

After solving this min-cost problem, we remove all the tiny instances that became idle (assigned no demand).

4.3.3 Hysteresis Placement

As described so far, our scheme generates a new placement policy only based on the current demand distribution, regardless of the previous placement. Thus, it can result in a large number of placement changes every time a new policy is computed.

We propose hysteresis placement to control the number of placement updates. We introduce a parameter hysteresis ratio (HR) when categorizing flows. If application \( a \) is deployed at data center \( n \) in the previous placement, we consider \( f_{an} \) as tiny instance only when \( f_{an} < \frac{DT}{HR} \), where \( HR \geq 1 \). In this way, if application \( a \) is deployed at data center \( n \) in the previous placement, then it is more likely to be kept in place in the new placement. This added ”stickiness” may result in some increase in the number of application instances as well as some delay penalty. We evaluate these effects in Section 4.5.5.

4.4 Further Optimizations: Flow Splits

As a result of our demand clustering, our demand distribution is specified at the granularity of regions. E.g., it can specify that, e.g., \( x_1 \) portion of the requests from a given region must go to data center \( D_1 \) and \( x_2 \) portion to \( D_2 \). But instead of blindly tossing coins
to decide which requests from the region go to which data center, as a second order optimization, we could skew our decisions: for example, if without clustering, certain requests in a region are closer to $D_1$ than to $D_2$, while other requests from the same region have reverse preference, we can try to send requests according to their preferences as long as this does not violate the overall demand distribution: $x_1$ for $D_1$ and $x_2$ for $D_2$.

Similarly, tiny flows represent all the demand for a given application instance and thus can aggregate demand from different regions, which may have different preference over data centers. So, when reassigning a tiny flow to multiple data centers, it is desirable to account for regional proximity rather than splitting the flow blindly.

We use a series of small-size min-cost flow models to perform these flow splits in as beneficial way as possible. Because these splits occur within the confines of the overall demand distribution policy, the size of these problems is small, and they can be solved quickly. In the next two subsections, we first discuss in details how the requests within a specific region should be forwarded across data centers, given that we know from previous sections the amount of requests each data center would receive from the region. Then we explain how tiny flows should be distributed across data centers where the requested applications are deployed according to the desired placement policies generated above.

### 4.4.1 Region Flow Splits

As mentioned above, each client-application pair has its individual preference of data centers. We might consider the preference of data centers for each client-application pair when doing flow split in a region. However, the number of client-application pairs falling into the region can be very large, which makes the problem intractable. Instead, we choose to consider the preferences of data centers on the granularity of applications, whose number is limited. In other words, we categorize the requests in a region by the applications they request. Admittedly, it would incur some delay penalty, thus we make the trade-off between
efficiency and performance here.

We first get the aggregate distance $d_{an}$ from the requests for each application $a$ to each data center $n$ in each region. Then we formulate a min-cost model to decide how the requests of each application should be distributed to each data center (limited by the generated demand distribution policy of each region). In the model, there is an application node $A_i$ for each application $i$, and a data center node $DC_j$ for each data center $j$, plus one source node and one sink node. There is an edge from the source node to each application node. And from each application node, there is an edge to each data center node. Finally, each data center node is connected to the sink node. The cost of the edges from the source node to the application node is 0, and capacity is the aggregate request rate of all the requests for the application. For the edges from the application node to the data center node, the cost is the aggregate distance $d_{an}$ and capacity is infinite (still limited by the capacity of the edges from data center nodes to the sink node). Finally, for the edges from data center node to the sink node, the cost is 0, and the capacity is the amount of requests that should be forwarded to the data center from the region.

An example is shown in Figure 4.5. Suppose we merge the regions of the corresponding pair nodes $Y_{11}, Y_{15}, Y_{21}$ and $Y_{24}$ into a new region which has request rates $r_{11} + r_{15} + r_{21} + r_{24}$. The request rate of application 1 is $r_1 = r_{11} + r_{15}$ and that of application 2 is $r_2 = r_{21} + r_{24}$. Also among the $r_{11} + r_{15} + r_{21} + r_{24}$ requests, we know from the optimization solution that $x_1$ are forwarded to data center $DC_1$, $x_n$ to data center $DC_n$, and $x_D$ to data center $DC_D$ (only $x_1$ and $x_D$ are shown in the figure). $d_{1n}$ and $d_{2n}$ are the aggregate distances from application 1 and 2 to the data centers separately (still only the distances to data center 1 and $D$ are shown in the figure). After we solve this min-cost model, we know how much requests should be forwarded to each data center for application 1 out of $r_{11} + r_{15}$, and the same for application 2.
4.4.2 Tiny Flow Splits

A tiny flow represents the demand from all the regions for an application instance in a specific data center, and thus its included requests may have different preferences over data centers. So, when a tiny flow is to be reassigned to multiple data centers, it is desirable to account for regional proximity rather than splitting the flow blindly. Similar with section 4.4.1, we propose min-cost flow model for this as follows.

For each tiny flow $f_{an}$, let $f_{an} = \sum_{t=1}^{R} e_t$, where $R$ is the number of regions after clustering and $e_t$ is the portion of flow $f_{an}$ that comes from region $t$. Also, for $f_{an}$, let $g_n$ be the portion of flow that would be forwarded to data center $DC_n$ (this comes from the Section 4.3.2). As shown in Figure 4.6, each region $t$ has a node $RG_t$. And each data center $DC_n$ that receives a portion of flow $g_n$ has a node. In addition, there are source node $S$ and sink node $T$. From source node $S$ to each region node $RG_t$, there is an edge with capacity to be $e_t$ and
Figure 4.6. Tiny flow split

cost to be 0. From region node $RG_t$ to data center node $DC_n$, an edge is added with capacity to be $e_t$, and cost to be aggregate distance for requests of application $a$ from region $t$ to data center $n$. And each data center node $DC_n$ has an edge to sink node $T$ with capacity $g_n$ and cost 0. After run the min cost flow of this network model, we know how each element $e_t$ from region $t$ would be forwarded to data centers with minimum cost.

Since we need to forward every tiny flow $f_{an}$ across data centers, and overall there could be up to $A \ast (D-1)$ tiny flows, we need to build and run this min cost flow $A \ast (D-1)$ times. However, we can reduce the complexity of the problem by merging the network models of tiny flows that are of the same application. In this way, we only need to run the minimum cost algorithm $A$ times. Since the min-cost models for flow split in this subsection (also in subsection 4.4.1) occur within the confine of the overall demand distribution policy, the size of these problems is small, and they can be solved quickly.
After this, we can combine the server selection for tiny flows with that of normal flows, and come up with a complete server selection policy.

4.5 Evaluation

We now study the performance of our approach using large-scale simulation. We built our simulator using CSIM, a discrete-event simulation package. Mimicking the actual system, our simulator has a decision component and a request routing component. During the simulation, the decision component periodically updates application placement and server selection policy (every 30s by default). There is also a workload component that generates requests according to load patterns discussed later. The routing component forwards each request to the appropriate data center according to the policy generated by decision component.

4.5.1 Cloud Model

We simulate a global cloud platform deployed across 20 data centers (except for the scalability experiments in Section 4.5.7). We parameterize our model with network distances as follows. We extracted all pingable IP addresses – 157803 total – from the Gnutella peer list compiled at the University of Oregon [5] and found their geographical locations using the GeoIP database (commercial version)[12]. We “deploy” our 20 data centers according to the distribution of our client population as follows. We rank countries according to their number of clients and distribute our data centers among the top 9 countries according to their relative client population (the rest of the countries had too few clients). For example, the US got nine data centers, China three, Japan two, Spain one, etc. For the US, we used a similar procedure to distribute the data centers among states.

We then selected 20 Planet Lab nodes in the same locales as our data centers (for non-
Figure 4.7. Performance of prefix clustering
US countries with multiple data centers we simply picked the first available nodes in these countries from the list on the PlanetLab website) and measured ping latencies from each such PlanetLab node to each client. We were able to obtain complete distances to 100546 clients. We then used these clients to represent the locations of client clusters that access our applications, the 20 PlanetLab nodes to mimic our data centers, and the measured network latencies between the PlanetLab nodes and clients as the network distances. Turning to the back-end databases, we assume these are maintained at corporate premises outside the cloud. Thus, we randomly select 100 PlanetLab nodes to mimic the location of the databases and use ping distances between the 20 PlanetLab nodes representing data centers and 100 PlanetLab nodes representing databases to parameterize our simulation.

We divide the world into 20 geographic regions, each with a data center. Client clusters that share a common closest data center (according to the measured ping distances) fall into the same geographic region with the data center. We also divide applications into two categories, regional and global. Regional applications are particularly popular within a specific geographic region (hot region), e.g., the website of a state government; global applications are universally popular. For a regional application, we define regional rate as the portion of requests it receives from its hot region. We use regional rate of 0.9 except when we vary it in the experiment of Figure 4.12.

4.5.2 Workload

We simulate a cloud of 20 data centers hosting 100 applications, with each data center able to serve 10,000 requests/s, resulting in the total capacity of all 20 data centers of 200,000 requests per second (req/s). These parameters are dictated by the scalability of the simulator itself, but are sufficient to evaluate our approach. (We study the scalability of our approach for larger platforms separately in Section 4.5.7). We define load_factor as the ratio of the total request rate across all data centers to the total capacity. Each
(normalized –see Section 4.1) request is assumed to have service time 0.03 second, so every data center in the simulator has 300 CSIM facilities that mimic servers. We set the queue length of each facility to be 150; requests are distributed among servers within a data center in a round-robin fashion and are dropped when arriving at a facility with full queue. In the optimization models, we assume the capacity watermark of 0.9, that is, the decision component tries to keep each data center utilization within 9,000 req/s.

**Demand Generation.** We assume applications’ popularity follows power law distribution with $\alpha = 1$ [75]: for the $j$th most popular application, the probability of a given request going to this application is proportional to $1/j$. Given the target total request rate, $r$, determined by the load factor, the workload generates requests sequentially with exponentially distributed inter-arrival time with mean $t = 1/r$. For each request, it first selects an application according to the power law probability distribution. Then if the selected application is regional, it assigns the request to a random client cluster from its hot region with probability of $\text{regional_rate}$ and to a randomly selected client cluster from outside its hot region otherwise. If the application is global, the request is assigned to a random client cluster.

**Dynamic Demand Patterns.** During simulations, the demand pattern changes every $T$ seconds. We use the following dynamic load patterns in our experiments:

1). **Vary-All-App:** starting from the initial distribution generated as described above, the amount of load of each application changes randomly within $\pm \Delta\%$, where $\Delta$ is a parameter controlling the extent of demand variability. This workload is an extended version of the vary-all-apps in [75].

2). **Rank-Exchange:** the popularity rankings of $k$ randomly picked pairs of applications are swapped, where $k$ controls the extent of demand variability. This workload mimics the change of popularity among applications.

3). **Reshuffle-All:** in each cycle, the rankings of the applications are reset to a random permutation and each regional application is remapped to a new random region. This workload
mimics extreme case of change, where the demand pattern in each cycle is completely independent of the pattern in previous cycle.

### 4.5.3 Clustering Performance

One of the key contributions of our work is our permutation clustering technique, which makes global optimization scalable enough to be applied in our target environment. Thus, we begin our study with the evaluation of this technique. In each experiment, we initially generate the requests that would occur in one second and re-send these requests repeatedly every logical second for ten logical seconds, at which point we recompute the demand to be used for the next ten seconds, and so on. Note that while we use the same demand pattern, the demand will be different because of new random coin tosses during generation. The simulation lasts 50 logical seconds. To concentrate on clustering effect on server selection, we assume each application is always deployed at all data centers, deletion threshold 0 req/s, and hysteresis ratio 1 throughout the experiments in this subsection.

We measure the number of dropped requests (although we did not observe any) and the average response time. Figure 4.7a shows the response time penalty from clustering, expressed as the relative difference between average response times with and without clustering. As seen from the figure, for a given level of clustering (i.e., the prefix length value), the penalty is smaller for lower load factors. (The line for for load factor 0.4 deviates slightly from this trend for initial values of the prefix length. Since the penalty variations involved are very small - within 1% - we view this as a statistical aberration.) This is understandable because with low load, most demand is satisfied by the closest server, and the discrimination among more distant servers becomes unimportant. Perhaps more surprisingly, even for high load factors, the clustering penalty is small, never exceeding 10%, and drops quickly with the prefix length. We attribute this to the effect of our distance aggregation for all members of the cluster: even when client-application pairs are clustered, the information
Figure 4.8. The effects of the deletion threshold

about their proximity to servers beyond the common prefix is still utilized through aggregated distances. Indeed, Figure 4.7b shows the delay penalty increases significantly when all distances to data centers beyond the prefix are assumed equal. Finally, Figure 4.7c shows that while delay penalty introduced by our demand clustering is small, it leads to a dramatic reduction in the execution time. We study the scalability of our approach further in Section 4.5.7.

In summary, our experiments give an indication that prefix clustering could be a promising general technique for aggregating demand. Given these results, we use prefix 3 for subsequent experiments, which allows us to solve the min-cost problem efficiently while keeping the delay penalty small - within 4% in the above experiments.

4.5.4 Deletion Threshold

We now turn to the the deletion threshold effects. If the deletion threshold was 0, application instances would be removed from data centers only if they were completely idle. A higher deletion threshold would reduce the number of instances but lead to performance penalty, as those requests that used to go to the underutilized instances would have to be routed to alternative data centers. We use prefix 3 (given results from Section 4.5.3) and
Figure 4.8 quantifies these effects by showing the total number of instances and performance penalty for different deletion threshold values. The workload is the same as in the previous subsection. Since each simulation run involves five recomputations of the demand, each data point represents the average total number of application instances across the whole run. The figure shows that as the deletion threshold increases, the number of total instances plunges dramatically in the beginning, but then decreases very slowly. The delay penalty behaves the opposite way, although at low load the growth is more gradual and does not flatten. In general, this result indicates that with an appropriate deletion threshold, our scheme can reduce the number of required instances dramatically with very small performance penalty. In particular, we choose deletion threshold 150 req/s throughout our subsequent experiments as it obtains factor of 5-7 reduction in the number of application instances while keeping the penalty under 8% for all loads.
4.5.5 Hysteresis Placement Effects

As discussed in Section 4.3, we use hysteresis ratio to reduce the number of application placement changes during policy recomputations. We now evaluate how hysteresis ratio affects the system behavior. We use deletion threshold of 150 req/s (see Section 4.5.4) and prefix length 3 (see Section 4.5.3) in the experiments of this subsection.

We use the following workload to explore this problem. At the first logical second, we generate a demand, and let the decision component compute application placement based on this demand. At the next second, we change the load pattern by remapping the regional applications randomly to regions and recompute the demand. At all the subsequent seconds, we recompute the demand with new random coin tosses but keep the same pattern. So the workload changes dramatically in the second second, but keeps stable (except for statistical variations) in the remaining time. The decision component computes the application placement every second: with our focus on placement changes, shortening the interval between decisions allows us to shorten the experiment without affecting the results. The experiment lasts 20s. In the graphs below, each data point represents the average result over five simulation runs with different seeds for random number generation.

Figure 4.9a shows the number of placement changes as the hysteresis ratio increases. For comparison, the figure also includes results for a heuristic application placement from [69] at 0 point on the x-axis. We can see that with the increase of the hysteresis ratio, the number of placement updates drops but the total number of instance increases. Without hysteresis placement (i.e., when hysteresis rate is 1), the number of placement updates of our approach is larger than in the existing placement algorithm. However, when hysteresis ratio increases towards 3, according to Figure 4.9a, our approach results in fewer placement updates than the heuristic algorithm, even though the latter computes the next placement by adjusting the current configuration and our approach recomputes the placement from
scratches. Admittedly, as Figure 4.9b shows, this comes at the expense of a certain increase in the number of instances, especially at higher load factors. This figure reports the average number of application instances over the course of a 20-second run and it indicates that our scheme may create up to around a third more instances than [69]. We argue that this modest increase is justified by a significantly better performance of our approach, as we will see in the Section 4.5.6. Interestingly, the delay penalty is negligible - less than 2% – and is not shown here. We choose hysteresis ratio 3 for the rest of our experiments.

4.5.6 Policy Evaluation

In this section, we compare the quality of the policies produced by our approach with prior work. To the best of our knowledge, the only works that jointly address the problems of demand distribution and application placement are [69] and [66]. Since [66] uses different objectives in formulating their policies (see Section 4.7), we compare our approach with [69]. The latter represents a drastically different approach from ours: instead of solving a global optimization problem, it repeatedly adjusts current placement by replicating or migrating instances and modifies server selection strategy according to the observed demand.

In these experiments, we use the dynamic load patterns specified in Section 4.5.2 with load factor 0.5 and regional ratio 0.9 (except in the experiment of Figure 4.12 where regional ratio varies). Experiments start with full deployment and every request is forwarded to the closest data center. We generate the initial demand according to the Section 4.5.2. For the first 15 logical seconds, the system is in a warm-up stage, where we update the policies every second so that the policies reflect the initial demand pattern after this stage. This is done for fairness to the scheme from [69] as it adopts to the desired configuration incrementally. Then the system goes into the measurement stage, lasting 900s, in which the demand is recomputed every 150 logical seconds according to the dynamic load pattern
used, and the policies are updated every 30 logical seconds. When computing the policies, we collect input request rates through the statistics from all data centers as in reality. We use exponential moving average with smoothing factor 0.6 to maintain these statistics.

Figures 4.10, 4.11 and 4.12 show the average response time and the number of dropped requests for the two approaches, for the three dynamic workloads we consider. The curves corresponding to our approach and the approach of [69] are labeled, respectively, “min-cost” and “heuristic”. The results show dramatic performance advantage of our approach.
for both metrics. The average response time shows improvements at least by a factor of 2, and dropped requests reduce by orders of magnitude.

4.5.7 Scalability

We now evaluate the scalability of our approach. Our baseline setup of the system includes 100,546 client clusters, 20 data centers and 100 applications. We measure the execution time of our algorithms by increasing one of these parameters and keeping the other two constant. For the purpose of simulations, whenever we add a new entity to the setup and need a network delay between it and other entities, we pick the delay at random between 0 and 500ms. We utilize Dell PowerEdge 2950 server with 8 cores and 16G memory for these experiments.

The results are presented in Figure 4.13. They show that the execution time grows almost linearly with the number of client clusters and applications, but superlinearly with the number of data centers. The latter makes sense since, for the prefix length 3 we used, the size of the min-cost flow model of Figure 4.3 grows as the power of 4 of the number of data centers (see Section 4.2.2). At the same time, when the number of client clusters and
Figure 4.13. Scalability
applications increases, the size of the min-cost flow problem used in the first phase (optimization with full deployment) does not change, but the size of the problems used in the second phase (application placement) and in the flow-splitting phase increase linearly. Most importantly, these results support the practicality of our approach for application placement in realistic-size cloud platforms. Indeed, the execution time remains within tens of seconds for thousands of applications, millions of client clusters, and tens of data centers. Another interesting point is that the execution time is largely independent of the load factor.

We argue that this represents realistic platform sizes and acceptable execution time. Indeed, Krishnamurthy et al. found on the order of 400K client clusters on the Internet [51], and most infrastructure providers, such as Limelight and AT&T, operate up to 20-30 data centers.\(^2\) The execution time in the order of tens of seconds appears acceptable based on prior studies: Oppenheimer et al., considering three real application workloads (OpenDHT, Coral, and CoDeen), recommend the application placement be done in the order of 30 minutes interval [61]; Wendell et al., using a different Coral dataset, found demand patterns to be fairly stable on the 10-minute time scale [82].

### 4.6 Prototype

We have implemented our approach using MyXDNS [18], an authoritative DNS server that allows one to plug in an external server selection policy. In the prototype, a decision component collects utilization and demand distribution from data centers, periodically computes placement and server selection policies using our approach, and uploads the new server selection policy into MyXDNS.

To demonstrate the operation of our system, we deploy a testbed that mimics a global platform. We use five machines to emulate five data centers: one in Japan, one in UK, one

\(^2\)As mentioned earlier, our approach is not suitable for highly distributed platforms such as Akamai, with thousands of locations.
in Australia, and two in US (one in California and one in New York). We use another five machines to mimic the clients distributed at these locations. To emulate global deployment, we hard-code the distances between the machines representing clients and data centers using measured ping RTTs between PlanetLab nodes in the mimicked locations. MyXDNS with our decision component runs on a separate machine, updating the policy every 30 seconds. On machines that mimic data centers, we install the Websphere application server running the TPC-W benchmark (with the browsing mix workload) as the application. We set server capacity to 100 req/s and capacity watermark at 70%. Yet we report results in terms of actual server utilization, thus justifying (at least for this application) our assumption about feasibility of using request rates as measure of demand and utilization. We use the following two scenarios to demonstrate how our system responds to the dynamically changing demand.

4.6.1 Demand Shift

Our first scenario shows the ability of the system to handle demand shifts from one region to another. In this scenario, we generate requests from only one location at a time and at a level that a single data center can cope with, but we change the location every time period (120s).

Figure 4.14(a) shows the CPU utilization of the five machines imitating data centers. It indicates that the system handles this scenario successfully, i.e., as one would have liked. Indeed, as seen from the figure, the application placement follows the demand after the delay induced by the periodicity of policy updates. Only one instance of the application is deployed at a time except during transitions, since our prototype is careful not to enact instance deletion until it completes pending requests - this is seen from an overlap in utilization curves.
4.6.2 Flash Crowd

Our second scenario imitates a flash crowd coming from one region. In this experiment, we generate requests from a single fixed location throughout the experiment but the amount of requests increases in the first 220s, then stays constant for 120s, and then drops during the final 220s. The application is initially placed in the data center in the region that generates the demand.

Figure 4.14(b) shows the CPU utilization of the five data centers in this scenario and again shows that the system handles this scenario successfully. Initially, the data center $DC_1$, the nearest to client demand, is sufficient to handle all the workload. As the demand increases, the data center utilization exceeds the watermark and at time 60 second and 180 second, the system deploys the application at two more data centers to deal with the workload - first at the second closest data center $DC_2$ and then at $DC_3$ as the third closest. Once the flash crowd subsides, the system removes the application from the two more distant data centers, first from the more distant $DC_3$ at 360 seconds and then from $DC_2$ at 450 seconds. Note that during the flash crowd, the two closest data centers are utilized equally (as determined by their capacity watermark) and the more distant data center $DC_3$ receives...
only the overflow demand. Also note transient effects around 60 and 150 seconds, due to periodicity in policy recomputation (hence an inherent lag in reaction to changing demand) and an occasional unpredictable change in demand. Indeed, at around 120 sec, the request rate produced by the demand generator unexpectedly dropped (not following the workload pattern), causing the system to lower selection probability to $DC_2$. But immediately after that, the workload increased back to normal, leading to spike in utilization of $DC_1$, while leaving $DC_2$ only modestly utilized.

4.7 Related Work

While many algorithms have been proposed for application placement and server selection, most of them consider only one or the other of these two problems. Algorithms in [26, 32, 46, 53, 56, 68] address the placement problem assuming client requests are always forwarded to the closest replica. This makes these approaches inherently sub-optimal as servers have limited capacity in practice. Some of these works formulate global optimization problems [26, 68] but use them only as the basis for comparison with heuristic approaches because the optimal solutions are impractical due to computational cost. As we showed, our prefix clustering approach makes global optimization practical in many cases.

The approaches of [19, 22, 71, 72, 82] are examples of the algorithms that focus on the server selection problem assuming a given set of replicas. None of them take into account the distance between server and back-end database, partly because they mostly consider server selection for CDNs, which do not have this issue. In particular, [82] proposes an optimal decentralized server selection algorithm done by a set of mapping nodes. We address a joint placement and selection problem in a centralized manner but handle the scale issue through a novel prefix clustering technique. In [19], the authors use a min-cost flow model to generate the server selection strategy. However, they assume that
the placement of applications is fixed while our approach includes the placement aspect. Furthermore, unlike our clustering technique, their approach to scalability depends on a fortuitous placement configuration.

Among the few works that tackle both placement and server selection, [66] proposes distributed placement and server selection algorithms. However, their server selection aims to balance load without considering proximity. In [69], the authors propose decentralized placement and centralized server selection algorithms that take into account both server load and proximity but compute both policies in isolation. Our unified approach showed performance advantages over it in our experiments. Another heuristic approach uses hashing to map clients to a subset of server replicas [81]. None of these works considers the distance between server and back-end database.

In addition, in our system, we employ the software [11] which implements the scaling push-relabel algorithm for the min-cost flow problem. Due to the specific architecture of the min-cost flow model in our problem (it is a complete bipartite network except the source and sink node), other efficient algorithms exist, e.g., the successive shortest path algorithm and Hungarian algorithm, as discussed at chapter 12 of [13].

4.8 Conclusion

This work addresses a problem of efficient hosting of multiple applications in a globally distributed cloud computing platform and makes two main contributions. First, we design a unified approach for application placement and demand distribution policies and show its promise through both simulation experiments and a prototype testbed demonstration. Second, we propose a novel demand clustering technique and show that it makes policies based on global optimization models practical for realistic-size environments. We hope our clustering technique will be found useful beyond its application to the particular algorithms
discussed here.
Chapter 5

Improving DNS-based Server Selection

DNS-based request routing is widely used to facilitate transparent server replication in today’s large-scale Internet platforms. For example, content delivery networks (CDNs) such as Akamai and Limelight, as well as large-scale content providers such as Google, use DNS-based request routing to forward client requests to nearby content replicas. The same mechanism is also used in cloud computing platforms, such as Google AppEngine.

DNS-based request routing [24] utilizes the need for Web clients to execute a DNS query to resolve the server’s hostname before performing the rest of the interaction. When the authoritative DNS server receives this query, it responds to the client’s DNS query with the IP address of a server that is dynamically selected to be “close” to the originator of the query (where the meaning of “close” is at the discretion of the service provider, and is often part of its core expertise). The subsequent interaction of the client occurs with the nearby server, resulting in better performance. However, the authoritative DNS server only sees the IP address of the client’s DNS server (LDNS - for “local DNS”) and not the client itself, when making the server selection decision. Thus, the effectiveness of this mechanism is inherently limited by how close the clients are located to their LDNS.

The proximity between clients and their LDNS was studied in [57, 73]. In particular,
found that only 16% of the clients were in the same network-aware cluster [52] with their LDNS servers, and about 36% of them were not in the same autonomous system. These mismatches can negatively affect the effectiveness of DNS-based server selection. An obvious way to avoid this problem would be to run the LDNS component on every host, but that would negate the advantage of a shared cache provided by current distinct LDNS configurations. The latter is important because the LDNS cache not only reduces the burden on the DNS infrastructure, but also improves the response time of the DNS queries satisfied from the cache. An alternative approach suggested in [73], which is to carry the IP address of the client requesting the name resolution in the DNS query message, so that the exact location of the client is conveyed to authoritative DNS, similarly negates the shared LDNS cache.

In this work, we propose to run an LDNS component on every host, and at the same time use peer-to-peer (P2P) techniques to share cached DNS resolutions among hosts within the same subnet. In our proposed system, peers sharing the same gateway would form a P2P system, where different peers would be responsible for different domain names according to the distributed hash mapping utilized by the P2P infrastructure. Since all peers in the P2P system are with the same gateway, they are supposed to be close to each other. Also, by sharing resolutions among the peers, we can preserve most of cache benefits of the current system. Through our experiments in section 5.4, we show that the overhead of running the P2P and LDNS components on each host is trivial, and the P2P lookup delay for DNS resolution is acceptable. At the same time, sharing DNS resolutions among proximal neighbors promises clients significant improvements in terms of network delay and bandwidth of their interactions with replicated content servers.

Previously, Cox et al. [34] and Ramasubramanian and Sirer [70] discussed P2P-based mechanisms for the whole DNS infrastructure. Our work does not affect the entire DNS infrastructure and instead only changes the local LDNS setup. CoDNS [65] is client-side
P2P approach but, unlike our scheme, it focuses on reliability and performance of DNS without addressing the client-to-LDNS proximity.

5.1 Motivation

To get a sense of the importance of the problem we are addressing, we provide an indication of the effect of the separation between clients and their LDNS on server selection by content delivery networks, which carry a significant portion of the entire Web accesses.\footnote{Indeed, Akamai alone claims to deliver 20% of the total Web content, the claim that our parallel ongoing study of our university traffic finds conservative.} In a parallel project, we use a technique from [57] to instrument a popular Web site and harvest provably correct associations between Web clients and their LDNS servers. We have collected nearly 161K distinct client/LDNS pairs and utilized a GeoIP database [12] to map them to geographical locations. After removing the pairs where at least one of the hosts in a pair could not be mapped, we ended up with nearly 154K mapped pairs. We then modeled server locations for the two leading CDNs - Akamai and Limelight - and estimated how much off their server selection would be for the clients in our dataset from the perspective of geographic proximity. Obviously, CDNs use more sophisticated considerations for server selection, but our back-of-the-envelop estimation indicates the impact of the issue we are considering.

We model CDN server locations as follows. For Akamai, which is known to have presence in most sizable cities in the world, our model consists of 1033 server locations, with 754 major cites in the US and 279 major cities in the rest of the world (Akamai claims around 7000 locations in total). On the other hand, Limelight has only a couple dozen data centers and we modeled their actual data center locations as reported in [41]. In either case, we select the closest CDN server based on the client’s location (which would be the desired choice according to geographic proximity) and the closest CDN server based on...
the location of the corresponding LDNS (which would be the currently practical choice for
CDNs). Then we obtain the extra miles the client’s packets need to travel because of the
potentially wrong choice of the CDN server.

Figure 5.1 shows the cumulative distribution function (CDF) of the extra miles. As
expected, more Akamai clients incur distance penalty at low penalty values: Akamai’s
numerous server locations allow for finer-grained geographical server selection but by the
same token make it more sensitive to the inaccurate representation of client’s location by
its LDNS server.\(^2\) Most important, however, is that roughly 30\% of all clients incur over
500 miles extra travel for both models. Furthermore, our results confirm an early finding
from [57] that many clients reside in an autonomous system that is different from that of
their LDNS servers - we had around 21\% of such clients in our dataset.

\[\text{Figure 5.1. Extra air miles between clients and CDN servers due to LDNS-based server}
\]\n
\[\text{selection}\]

\(^2\)This should not be construed as Limelight’s advantage because this penalty represents the extra distance
over the distance to closest server and not the absolute distance between the clients and CDN servers.
These results indicate that a significant portion of clients are configured with LDNS far enough to adversely affect proximity-based CDN server selection. In principle, there could be a variety of reasons for this phenomenon, such as: (i) ISPs deploy a relatively small number of LDNSs in their networks to save administration and equipment costs; (ii) LDNSs are not uniformly distributed in ISP’s networks, even if their overall number might be large; (iii) when configuring the LDNS servers for clients, ISPs do not specifically attempt to reduce the proximity between clients and the LDNS servers assigned to them; and (iv) ISPs assign a large number of clients to the same shared LDNS server in an attempt to increase its cache utilization, and a large population of clients stipulates assembling them from a large geographical area.

Although all these reasons are theoretically possible, we speculate that reason (i) is the most likely. Indeed, previous work (supported by our results below) indicates that very high DNS cache hit rates are achieved with modest number of clients sharing the LDNS cache [50]. We can discount other reasons by assuming that ISPs would be willing to improve the service for their clients when it can be done with little effort on their part.

Given the above reasoning, one potential solution for the ISPs is to add LDNS functionality to the boxes that are deployed more widely, such as access routers at the points of presence. For example, each access router can run the LDNS software and all the clients sharing the same access router would be configured to use it for DNS resolutions. Since PoPs are supposed to be close to their clients, they are well positioned to act on behalf of their clients for server selection. However, this approach would require ISPs to customize and redeploy these boxes in their networks, which may incur high replacement cost. Furthermore, these boxes are often highly optimized specialized pieces of equipment, and loading them with significant new functionality may not be feasible.

Another way to address this problem is proposed in [73]. Authors propose to carry the IP address of the client requesting the name resolution in the DNS query message. In this
In our proposed system, clients themselves run an LDNS component and at the same time use peer-to-peer (P2P) techniques to share cached DNS resolutions with their neighbors. This approach requires no modifications to the DNS protocol and minimal effort/investment on the part of the ISPs. In the remaining sections, we describe the architecture and implementation of our system and present an evaluation study of its performance.
5.2 Architecture

This section outlines the architecture of our system. We also describe several issues and our design decisions to address them.

5.2.1 Overview

As shown in Figure 5.2, every host in our architecture runs a LDNS component and a P2P component. When a host starts up, in addition to the gateway and other information the host normally gets from the DHCP server, the host will also get the IP address of the seed server that is maintained by the ISP and which keeps track of various client clusters. The members of each cluster form a P2P system within their cluster. The host sends the gateway IP address to the seed server, the latter selects a client cluster for the host to join based on the gateway information and returns seeds (a few hosts that are already members of the selected cluster) to the host. Then the P2P component on the host sends a JOIN message to one of the seeds to join the P2P system of the corresponding cluster. In selecting clusters for each new client, the seed server ensures that clusters comprise only clients that are topologically close to each other.

When an application at host \( i \), e.g., web browser, needs to resolve a domain name, the application sends the DNS query to its co-located P2P component, which sends a LOOKUP message into the P2P system. The lookup message is routed through the cluster according to the distributed hash table employed by the P2P network and using the requested domain name as the key, until it arrives to the destination host \( k \), which is the host responsible for the given domain name. The P2P component on host \( k \) asks its LDNS component to perform an actual DNS query (in the same way as current LDNS servers do). After obtaining the resolution, host \( k \) puts it in a RESPONSE message and sends it to host \( i \). The P2P component at host \( i \) then returns the resolution to the application.
P2P sharing of DNS resolutions allows hosts to benefit from a shared DNS cache. Each host maintains a local cache of DNS responses, whether obtained previously by its own applications or as a result of processing a lookup message from a peer. Before sending a lookup message, the P2P component checks its local DNS cache and responds to the application immediately if the requested DNS record is available there and is still valid. Similarly, the destination peer of a lookup message checks its local DNS cache and uses the cached DNS record to construct its RESPONSE message if the valid DNS record is available in its local cache.

Finally, the internal reliability mechanisms of P2P networks, which are designed for frequent peer disconnections, provide for continued operation of our system in the face of unreliable peers. In particular, our prototype uses Bamboo, an P2P system designed to tolerate high peer churn [2].

5.2.2 Traversing Firewalls and NAT Boxes

Network address translation (NAT) devices present two challenges to our system. First, a host behind a NAT box obtains a private IP address of the NAT box (e.g., home router) as its gateway during startup. While this is acceptable in most organizational settings, where the entire P2P cluster can be formed behind the firewall using private IP addresses (and where our system may in fact not be needed if the organization runs its own LDNS within its local network, in which case hosts would be already close to their LDNS), residential clients may need to form clusters that span multiple home networks. In this case, the host must provide the IP address of its ISP access router, and not of its home router, to the seed server.

To deal with this issue, we note that home routers obtain the IP address of their ISP access router by executing their own DHCP protocol over the public side of the network.

---

3Firewalls present the same issues and our discussion applies to them too.
Furthermore, most home routers (e.g., linksys) provide an API (usually HTTP-based) to the hosts on the private side that allows hosts to obtain this information as an HTML page. Thus, the host can parse the IP address of its ISP access router from the HTML text and submit it to the seed server.

The second issue is typical for all P2P networks and has to do with NAT boxes preventing a peer from being contacted from outside hosts. We use standard techniques to address this issue. The NAT boxes can be configured with a permanent port mapping allowing outside communication (in fact, this can be done by a script using the NAT’s API). The closed nature of our P2P network alleviates attendant security issues.\textsuperscript{4}

5.2.3 Degenerate Clusters

Given that many clusters in our envisioned system would be residential, their size can drop significantly during periods of low activity, e.g., at night when home users may turn off their computers. When this happens, as shown in section 5.4, the DNS hit ratio can degrade significantly. While this would have limited effect on the load on the DNS infrastructure because the hit rate reduction would be canceled by the overall low level of use, the users may experience higher response time in their Internet accesses. We considered two ways to address this problem: (i) merge extremely small clusters with other nearby clusters - in this case hosts in degraded clusters would query the seed server for a nearby cluster to join, and (ii) let the P2P component \textit{proactively refresh} expired DNS resolutions when cached records become invalid (as suggested in [23] in a different context). The first method tries to increase the cluster size with the relaxation of the proximity requirement to some extent. In the second method, whenever an DNS resolution expires, it is updated by querying the

\textsuperscript{4}Furthermore, permanent port mappings and the presence of the seed server as coordinator allow outside communication to be permitted on demand only: the seed server alerts two peers that must establish a link; the peers send each other join messages, each of which opening a hole in the sender’s NAT box. The second message to arrive at the destination peer will find the hole open and establish the communication channel, which can then be maintained with periodic keep-alive messages.
DNS system. In this way, the cached DNS resolutions are always valid unless the client is visiting a new domain name. Admittedly, it would introduce additional DNS load, but again, being counterbalanced by the overall low activity level.

The tradeoffs between these approaches are that, with cluster merging, a degraded cluster may not find a sufficiently close neighbor cluster to join. At the same time, the efficacy of proactive DNS refresh depends on how much clients revisit old domain names. Our experiments of Section 5.4 show that roughly half the clients get over 80% DNS hit rate with DNS pre-refresh even when the cluster reduces to a single peer. We also note that the effectiveness of cluster DNS cache sharing increases rapidly with the cluster size. As we will see, a cluster with only six hosts has DNS cache hit rate of at least 85% even without cache refreshing. Thus, we consider handling of degraded clusters as more of a precautionary mechanism.

5.3 Prototype Implementation

We implemented our system and deployed it both in our lab and the Emulab testbed. Our implementation uses bind-9.5.0 [7] as the LDNS component. We built our P2P component on top of Bamboo DHT [2] table. In the prototype, we modified Firefox to send DNS queries directly to the P2P component instead of LDNS. In real implementation, all the components of our system would be wrapped in a daemon listening on port 53 and mimicking a regular LDNS server. Then, to route local DNS queries from all of the host’s applications to our P2P component, each host would simply be configured to use localhost as its LDNS server.
5.4 Evaluation

To estimate the performance of our approach, we first evaluate the DNS hit ratio in client clusters of various sizes. Then we evaluate the overhead, in terms of CPU and memory, for running the P2P component and DNS component on the host. Next, we measure the delay for looking up DNS resolutions in the P2P system in client clusters of varying size. Finally, we measure the end-to-end improvement when employing our system.

5.4.1 Hit Ratio

We use trace-driven simulation to explore DNS hit rates in our system. We used an anonymized 4-hour HTTP trace collected on May 9, 2006 at Case Western Reserve University. The trace comprises over 5.8 million HTTP downloads from over 6,600 clients. We extracted the hostnames from the trace and obtained TTLs of the corresponding type-A DNS responses by performing actual DNS queries for each name. When evaluating the hit ratio of a client cluster with a specific size $s$, we randomly choose $s$ clients from the trace and simulate each access from these clients while mimicking the DNS cache behavior, which we can do faithfully by using the actual TTLs obtained. We repeat each simulation...
20 times for different random client clusters and report the average hit ratio.

Figure 5.3(a) shows the DNS hit rates for different sizes of client clusters. The results (which are consistent with the previous findings from [50]) show that even small client clusters exhibit high DNS hit rates: as long as the size of client clusters is bigger than 6, the difference of hit ratio is within 7% of the maximum possible, when the cache is shared among all the clients in the trace.

Our next experiment considers hit rates achievable by degenerate clusters with DNS refresh discussed in Section 5.2. We consider the extreme case of clusters with only one host and obtain the DNS hit rate of each client in the trace assuming every DNS record in the cache is refreshed upon expiration and hence available to the client when requested. We did not evaluate the extra DNS traffic this would cause as this mechanism would only be engaged during periods of low activity. Figure 5.3(b) shows the CDF of hit ratios with refresh for all clients (recall that as long as a domain name is visited before, it would be a hit due to the proactive refreshing). According to the figure, roughly half the clients would have 80% or larger hit ratio. Still, many clients would have low hit rate, with almost 20% of clients exhibiting under 30% hit rate. We note, however, that this result is conservative because the cache contents of individual client are limited by the short duration of our trace.

5.4.2 Resource Consumption

In this subsection, we evaluate the resource consumption of the new components on the client machines, focusing on the CPU and memory. We deployed our prototype at Emulab [9] for this experiment. To overcome the difficulty of scheduling a large number of machines, we use the following setup. We allocate 28 physical machines, each with 2 core CPU and 2G memory. On 10 of them, we deploy our system directly. On the remaining 18 physical machines, we allocate the total of 190 virtual machines and deploy our system on each VM. We compose a client cluster of size $s$ with the 10 physical machines and $(s - 10)$
randomly selected virtual machines. Thus, we can evaluate the behavior of a large cluster – up to 200 hosts – while measuring the actual resource consumption on the 10 physical machines. Our results reflect the average numbers over the 10 physical machines.

In the experiment, the machines form a cluster using a separately deployed seed server. The cluster is formed in a random manner: as nodes join, each new node receives the full set of previously joined peers from the seed server and picks a random seed to send a join message. Once the cluster is formed, every node sends out DNS queries at the rate computed as follows. We first obtain the maximum request rate within a 5 second interval for each client according to the trace. Then we order the clients by these rates and use the average request rate among clients in the 50% percentile (e.g., the most active 50% of all the clients), which is about 7.1 req/s. During the experiment, each client sends out 7.1 DNS queries per second for the domain names taken from the list of unique domain names extracted from the trace. The resource consumption of our system components was found to be negligible – within 1.3% of one core for CPU and about 3% for memory – and barely grows with the cluster size. Given these small overhead levels, we omit the graphs due to space limitations.

5.4.3 P2P Lookup Delay

We now turn to the delay overhead in our approach due to the P2P lookup. The testbed setup is the same as in Section 5.4.2. Since all machines in Emulab are on the same LAN, the delay between any pair of hosts is around 0.3ms. Thus, any time a P2P routing hop involves two peers running on virtual machines that happen to reside on the same physical machine, we add 0.3ms to the overlay delay. To focus on how much additional delay occurs due to the P2P nature of our system, we (1) turn off DNS caching, so every lookup leads to the P2P communication and (2) exclude the external delay the DNS queries spend outside the cluster, as this delay would be the same in our and the current systems. Consequently,
when the LOOKUP message reaches its target peer, the latter constructs a RESPONSE message with the priori known IP address and sends this message to the source host immediately.

In this experiment, we pick a random cluster of clients of a given size and assign each client to each peer in the testbed. We then extract HTTP accesses of these clients from the trace, and for each access, perform a DNS lookup for the URL’s hostname from the corresponding peer. Bamboo DHT offers two options for message routing, “send-back” where response messages are sent back directly from the target to source peer, and “route-back” where response messages are routed from the target to source peer through the overlay network. In our experiments, we consider both options.

Figure 5.4 shows the results. We can see that the lookup delay is between 2ms and 3.5ms. Furthermore, the delay is virtually flat as the cluster size increases in the case of the route-back, and grows only marginally in the case of send-back (we are puzzled by the higher delay of the send-back option; we can only speculate that it is an implementation artifact of Bamboo). This, along with the fact that the send-back, which incurs fewer P2P routing hops, exhibits higher delay, suggests that the overall delay is dominated by processing at the source and target peers rather than by the routing itself.

Obviously, the delay cost of a P2P routing hop will be higher for some residential clusters, such as those with DSL-connected peers. Yet other popular access technologies, namely cable modems, represent broadcast medium with very low RTT between hosts on the same coaxial cable. Because one cable typically connects hundreds of homes, these neighborhoods would be prime candidates to form a P2P cluster. Further, newer access network technologies such as fiber-to-the-home (FTTH) are gaining rapid adoption and exhibit low RTTs similar to organizational LANs. Finally, note that a host would generate a lookup message only for local cache misses, and according Figure 5.4, the local hit rate for a stand-alone host is around 60%, meaning that most lookups will not incur any delay. In
fact, the shared cache hit ratio grows to 80% for three hosts, suggesting a possibility for hosts in individual home networks to form their own clusters. In summary, the P2P routing delay in our approach is negligible for clusters of hosts with cable modem or FTTH Internet connectivity and acceptable in multi-computer home networks with DSL connectivity. A further study is needed to assess its suitability for clusters of individual DSL-connected hosts.

5.4.4 End-to-end Improvement

Finally, we study the improvement in user-perceived service quality our system promises, depending on the distance of the clients from their LDNSs. We investigate this problem by comparing the performance of Akamai CDN servers returned by LDNSs with different network distances to the clients. We consider two metrics here: network delay and the effective bandwidth of a web download (BW). We select 100 PlanetLab nodes distributed
around the world as clients and obtain the set of LDNS servers as follows. We first get a
list of Gnutella peer IP addresses from [6] and a list of web server IP addresses from [10].
Then we perform reverse DNS queries to get the authoritative DNS server (ADNS) for each
of these IPs and filter out those ADNSs that reject external recursive DNS queries as they
could not be used in our experiment. As a result, we were able to identify 13420 ADNSs
that could act as LDNS for our clients.

In the experiment, each client first obtains the ping distance to all the LDNSs. Then
it sends DNS query for the domain name of a URL that is delivered by Akamai to each
LDNS and gets the corresponding Akamai server IP address selected by Akamai. After
that, the client collects the ping distance to each Akamai server (averaged over 3 tries) and
the effective bandwidth of downloading the URL from that server (using the curl command,
averaged over 5 tries). To ensure every download occurs from the Akamai server’s cache
and not from the original Web site, we download the object first from each Akamai server
(thus bringing the object into the server’s cache) before performing the actual measurement.

Since the effective download bandwidth depends on the object size, our experiment used
three Akamai-delivered objects with sizes 6.8k, 54k and 2M. Among 100 planetlab nodes,
we were only able to get complete results from 85 of them due to node failures and other
unknown errors. We also eliminate the results in which we fail to get the ping distance
from clients to LDNS or Akamai servers due to the filtering of ICMP packets.

Figures 5.5 – 5.7 show CDFs of the reduction in latency and increase in download band-
width of the Akamai server selected by Akamai when we used the LDNS co-located with
the client over Akamai servers selected when using decoupled LDNSs. We can see sig-
nificant end-to-end improvement by running LDNS on local machine in the overwhelming
majority of cases.

To see how the improvements depend on the distance between the client and LDNS,
we calculate the Spearman’s rank correlation between the distance from a given client to
each LDNS and our two quality metrics of the CDN servers selected by Akamai when we used that LDNS. Table 5.1 presents the average values of the Spearman’s correlation coefficient for these two metrics computed over all the clients and all three object sizes. The results show very strong correlation between the client/LDNS distance and both server quality metrics we used. We conclude that bringing LDNS as close as possible to the clients directly improves the quality of server selection by Akamai.
Figure 5.7. End-to-end improvement (object size 2M)

Table 5.1. Ranking correlation

<table>
<thead>
<tr>
<th>Image size</th>
<th>Delay-delay correlation</th>
<th>Delay-BW correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8k</td>
<td>0.881</td>
<td>-0.869</td>
</tr>
<tr>
<td>54k</td>
<td>0.882</td>
<td>-0.861</td>
</tr>
<tr>
<td>2M</td>
<td>0.826</td>
<td>-0.729</td>
</tr>
</tbody>
</table>

5.5 Conclusion

In this work, we observed that the distance between clients and their LDNS servers can have a significant negative impact on DNS-based server selection, which is widely used by content deliver networks (CDNs) and other platforms. We then proposed a novel peer-to-peer client-side DNS mechanism that moves LDNS close to their clients while still allowing nearby clients to share the common DNS cache. Through trace-driven simulations and tests on a real prototype setup, we showed that our approach holds significant promise of facilitating better server selection by CDNs.
Chapter 6

Conclusion

In this chapter, we first summarize the contributions of this thesis. Then we discuss the directions of our future work.

6.1 Summary of Contributions

This thesis investigates the dynamic resource management for Internet applications at service-oriented computing platforms. It addresses problems of dynamic resource provisioning both within a data center and across multiple data centers, and makes the following contributions.

Firstly, it proposed a resource reassignment mechanism based on ghost/suspended VMs to address the agile resource allocation problem in virtualized data centers. Different from existing works, it focuses on the agile execution of the resource provisioning decisions - both allocation of and reclaiming of resources to/from the applications. We implemented our approach at two testbed with both VMware Server and ESX, and showed that our system can react to the demand surging within 18s (including the detecting and decision making time). Also, our experiments showed that our system performs much better than the legacy system when the demand increases fast.
Secondly, it outlines a hierarchical resource management solution to deal with the scalability problem in mega data centers and a new data center architecture to balance the load among the access links. In our hierarchical resource management, existing technique are employed to manage resource within a pod, which consists of about 5k servers, and meanwhile multiple strategies are proposed to balance the load among the pods. Also, a new layer consisting of load balance switches is added at the access network of the data centers. With this new architecture, we apply DNS-based mechanism to selectively expose IPs of applications to Internet clients so that the network traffics would go through lightly-loaded access links.

Thirdly, a unified approach is introduced to handle the global application placement and server selection problem across multiple geo-distributed data centers. It solved the two problems systematically based on a min-cost network flow model, and invented a novel demand clustering technique to reduce size of the min-cost problem to be manageable. Through simulation and evaluation on testbed, we shows that our approach is both practical and effective in handling the global application placement and server selection problem with varying demand of hosted applications.

Finally, a new client-side local DNS architecture is proposed to bring LDNS close to their client, which makes DNS-based server selection precise and more effective. Our architecture applies P2P technology, and does not require any change of the underlying network infrastructure and existing network protocols. Our experiments shows that by applying our architecture, Internet clients can achieve significant improvement of quality of services when accessing Internet services.
6.2 Future Work

We would continue to investigate and evaluate our approaches for the resource management in mega data centers. We plan to extend solution space of the problem, study the tradeoffs of all the possible solutions, and conduct extensive quantitative evaluation.

Also, we would explore the hint-based local resource management. With our global resource management, it is possible to estimate how much demand each data center would receive for each application. Thereby, in stead of depending on pure monitoring and estimation, we can get some hints from the global resource manager about the status of the demand, and making more intelligent decisions, e.g., prepare more ghost VMs for applications that are about to receive large demand, etc.

Finally, we would further study the dynamic resource management problem under more conditions and constraints, like accounting for inter-dependencies among the hosted applications, considering different priorities of applications, etc.
Bibliography


[43] Using VMware ESX Server with IBM WebSphere Application Server.


[47] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting
utility platforms. In HPDC ’03: Proceedings of the 12th IEEE International Symposium on
High Performance Distributed Computing (HPDC’03), 2003.

[48] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks:
characterization and implications for cdns and web sites. In Proceedings of the 11th interna-
tional conference on World Wide Web, WWW ’02, pages 293–304, New York, NY, USA,
2002. ACM.

[49] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks:
characterization and implications for CDNs and web sites. In WWW ’02: Proceedings of the

[50] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of


[52] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. ACM SIG-

582, 2000.

[54] H. Lagar-Cavilla, J. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. Rumble,
M. Satyanarayanan, and A. Scannell. SnowFlock: Virtual Machine Cloning as a First-Class
Cloud Primitive. ACM TOCS, 29(1), 2011.

Rumble, M. Satyanarayanan, and A. Scannell. Snowflock: Virtual machine cloning as a first-


[80] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for


