DYNAMIC RESOURCE MANAGEMENT IN INTERNET HOSTING PLATFORMS

A Dissertation Presented

by

BHUVAN URGAONKAR

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

DOCTOR OF PHILOSOPHY

September 2005

Computer Science
DYNAMIC RESOURCE MANAGEMENT IN
INTERNET HOSTING PLATFORMS

A Dissertation Presented

by

BHUVAN URGAONKAR

Approved as to style and content by:

_________________________________________
Prashant J. Shenoy, Chair

_________________________________________
Emery D. Berger, Member

_________________________________________
James F. Kurose, Member

_________________________________________
Donald F. Towsley, Member

_________________________________________
Tilman Wolf, Member

_________________________________________
Bruce W. Croft, Department Chair
Computer Science
To Aai, Baba, Rahul, and Ajoba.
With every mistake we must surely be learning...
George Harrison, The Beatles.
ABSTRACT

DYNAMIC RESOURCE MANAGEMENT IN INTERNET HOSTING PLATFORMS

SEPTEMBER 2005

BHUVAN URGAONKAR
B.Tech., INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR, INDIA
M.S., UNIVERSITY OF MASSACHUSETTS AMHERST
Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Prashant J. Shenoy

Internet applications such as on-line news, retail, and financial sites have become commonplace in recent years. Due to the prevalence of these applications, platforms that host them have become an important and attractive business. These platforms, called hosting platforms, typically employ large clusters of servers to host multiple applications. Hosting platforms provide performance guarantees to the hosted applications (such as guarantees on response time or throughput) in return for revenue. Two key features of Internet applications make the design of hosting platforms challenging. First, modern Internet applications are extremely complex—existing resource management solutions rely on very simple abstractions of these applications and are therefore inadequate in several respects. Second, these applications exhibit highly dynamic workloads with multi-time-scale variations. Managing the resources in a hosting platform to realize the often opposing goals of meeting application performance targets and achieving high resource utilization is therefore a diffi-
cult endeavor. In this thesis, we present resource management mechanisms that an Internet hosting platform can employ to address these challenges.

We design and evaluate analytical models of multi-tier Internet applications. Our models accurately capture several key features of Internet applications and serve as the building blocks for some of our resource management algorithms. We devise techniques for automatically deriving the resource requirements of applications. We show how a hosting platform can utilize this information to improve its revenue by careful statistical multiplexing and under-provisioning of its resources. We develop a dynamic capacity provisioning technique for multi-tier Internet applications based on a combination of predictive and reactive mechanisms. We propose a novel virtual machine based hosting platform architecture that enables agile provisioning of servers. Finally, we develop techniques that allow a hosted application to remain operational even under extreme overloads where the arrival rates are an order of magnitude higher than the provisioned capacity. We implement our resource management techniques in a prototype hosting platform based on a cluster of Linux machines. We demonstrate the feasibility of these techniques through extensive experimentation on this prototype using realistic applications and workloads.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xv</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION AND MOTIVATION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction: Internet Applications and Hosting Platforms</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Internet applications</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Data Centers</td>
<td>1</td>
</tr>
<tr>
<td>1.1.3 Hosting Platforms</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Models of Hosting</td>
<td>4</td>
</tr>
<tr>
<td>1.2.1 Dedicated Hosting</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Shared Hosting</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Internet Hosting Platform Design Challenges and Requirements</td>
<td>5</td>
</tr>
<tr>
<td>1.4 The Case for a Novel Resource Management</td>
<td>9</td>
</tr>
<tr>
<td>1.5 Thesis Summary and Contributions</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Overview of Our Hosting Platform Design</td>
<td>13</td>
</tr>
<tr>
<td>1.7 Dissertation Road-map</td>
<td>15</td>
</tr>
<tr>
<td>2. APPLICATION MODELING</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>16</td>
</tr>
<tr>
<td>2.1.1 Motivation</td>
<td>16</td>
</tr>
<tr>
<td>2.1.2 Research Contributions</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Internet Application Architecture</td>
<td>18</td>
</tr>
</tbody>
</table>
2.2 Background and Related Work ......................................................... 20
  2.2.1 Request Processing in Multi-tier Applications ......................... 20
  2.2.2 Related Work ................................................................. 21

2.3 A Model for a Multi-tier Internet Application ................................. 23
  2.3.1 The Basic Queuing Model ................................................... 24
  2.3.2 Deriving Response Times from the Model .............................. 26
  2.3.3 Estimating the Model Parameters ....................................... 28

2.4 Model Enhancements ..................................................................... 31
  2.4.1 Replication and Load Imbalance at Tiers ............................... 31
  2.4.2 Handling Concurrency Limits at Tiers ................................. 33
  2.4.3 Handling Multiple Session Classes ..................................... 36
  2.4.4 Other Salient Features ....................................................... 38

2.5 Model Validation ......................................................................... 40
  2.5.1 Experimental Setup ............................................................. 41
  2.5.2 Performance Prediction ....................................................... 43
  2.5.3 Query Caching at the Database .......................................... 45
  2.5.4 Load Imbalance at Replicated Tiers .................................... 47
  2.5.5 Multiple Session Classes ..................................................... 49

2.6 Applications of the Model ............................................................ 50
  2.6.1 Dynamic Capacity Provisioning and Bottleneck Identification ...... 50
  2.6.2 Session Policing and Class-based Differentiation .................... 53

2.7 Concluding Remarks .................................................................... 55

3. APPLICATION PLACEMENT ........................................................... 57

  3.1 Introduction and Motivation ...................................................... 57
  3.2 The Application Placement Problem ......................................... 58
    3.2.1 Notation and Definitions ................................................. 58
    3.2.2 Related Work ................................................................. 60
  3.3 Hardness of Approximating the APP ......................................... 61
  3.4 Offline Algorithms for APP ....................................................... 68
    3.4.1 Placement without the Capsule Placement Restriction .......... 68
      3.4.1.1 First-fit Based Approximation Algorithm .................. 68
3.4.1.2 Placement of applications whose capsules must be co-located ........................................... 70

3.4.2 Placement with the Capsule Placement Restriction ......................................................... 71

3.4.2.1 Placement of Identical Applications ................................................................. 71
3.4.2.2 Placement of Arbitrary Applications ............................................................. 73

3.5 The On-line APP ........................................................................................................... 76

3.5.1 Online Placement Algorithms ................................................................................. 76
3.5.2 Online Placement with Variable Preference for Nodes ........................................... 79

3.6 Concluding Remarks .................................................................................................... 81

3.6.1 Summary of Results ............................................................................................... 81
3.6.2 Future Work ............................................................................................................ 82

4. DYNAMIC CAPACITY PROVISIONING ........................................................................ 83

4.1 Introduction .................................................................................................................. 83

4.1.1 Motivation .............................................................................................................. 83
4.1.2 The Case for a New Provisioning Technique ......................................................... 84

4.2 Provisioning Algorithm Overview ............................................................................... 87
4.3 How Much to Provision: Modeling Multi-tier Applications ........................................ 88
4.4 When to Provision? ...................................................................................................... 89

4.4.1 Predictive Provisioning for the Long Term ............................................................ 89
4.4.2 Reactive Provisioning: Handling Prediction Errors and Flash Crowds ................. 92
4.4.3 Request Policing ..................................................................................................... 93

4.5 Agile Server Switching using VMMs ........................................................................... 94
4.6 Implementation Considerations ................................................................................... 97
4.7 Experimental Evaluation ............................................................................................ 99

4.7.1 Experimental Setup ............................................................................................... 99
4.7.2 Effectiveness of Multi-tier Model ........................................................................... 100

4.7.2.1 Independent Per-tier Provisioning ................................................................. 100
4.7.2.2 The Black box Approach ............................................................................... 102

4.7.3 Predictive and Reactive Provisioning .................................................................... 103

4.7.3.1 Only Predictive Provisioning ........................................................................ 105
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7.3.2</td>
<td>Only Reactive Provisioning</td>
<td>106</td>
</tr>
<tr>
<td>4.7.3.3</td>
<td>Integrated Provisioning</td>
<td>108</td>
</tr>
<tr>
<td>4.7.4</td>
<td>VM-based Switching of Server Resources</td>
<td>109</td>
</tr>
<tr>
<td>4.7.5</td>
<td>System Overheads</td>
<td>110</td>
</tr>
<tr>
<td>4.8</td>
<td>Concluding Remarks</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction and Motivation</td>
<td>113</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Research Contributions</td>
<td>115</td>
</tr>
<tr>
<td>5.1.2</td>
<td>System Model</td>
<td>117</td>
</tr>
<tr>
<td>5.2</td>
<td>Automatic Derivation of Application QoS Requirements</td>
<td>118</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Application QoS Requirements: Definitions</td>
<td>119</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Kernel-based Profiling of Resource Usage</td>
<td>121</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Empirical Derivation of the QoS Requirements</td>
<td>122</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Profiling Server Applications: Experimental Results</td>
<td>124</td>
</tr>
<tr>
<td>5.3</td>
<td>Resource Under-provisioning and Capsule Placement in Hosting Platforms</td>
<td>128</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Resource Under-provisioning Techniques</td>
<td>129</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Capsule Placement Algorithms</td>
<td>130</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Handling Dynamically Changing Resource Requirements</td>
<td>130</td>
</tr>
<tr>
<td>5.4</td>
<td>Implementation Considerations</td>
<td>133</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Providing Application Isolation at Run Time</td>
<td>133</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Prototype Implementation</td>
<td>135</td>
</tr>
<tr>
<td>5.5</td>
<td>Experimental Evaluation</td>
<td>136</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Efficacy of Resource Under-provisioning</td>
<td>137</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Capsule Placement Algorithms</td>
<td>139</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Effectiveness of Kernel Resource Allocation Mechanisms</td>
<td>141</td>
</tr>
<tr>
<td>5.6</td>
<td>Related Work</td>
<td>144</td>
</tr>
<tr>
<td>5.7</td>
<td>Concluding Remarks</td>
<td>146</td>
</tr>
<tr>
<td>6.</td>
<td>SHARC: MANAGING RESOURCES IN SHARED HOSTING PLATFORMS</td>
<td>148</td>
</tr>
</tbody>
</table>

---

xi
# 6. Introduction

## 6.1 Introduction

- **6.1.1 Motivation**
- **6.1.2 Research Contributions**

## 6.2 Resource Management in Shared Clusters: Requirements

## 6.3 Sharc Architecture Overview

- **6.3.1 The Control Plane**

## 6.4 Sharc Mechanisms and Policies

- **6.4.1 Resource Requirement Inference**
- **6.4.2 Trading Resources based on Capsule Needs**

## 6.5 Failure Handling in Sharc

- **6.5.1 Nucleus Failure**
- **6.5.2 Control Plane Failure**
- **6.5.3 Node and Link Failures**
- **6.5.4 Application Failure**

## 6.6 Implementation Considerations and Complexity

## 6.7 Experimental Evaluation

- **6.7.1 Experimental Setup**
- **6.7.2 Predictable Resource Allocation and Application Isolation**
- **6.7.3 Performance of a Scientific Application Workload**
- **6.7.4 Application Isolation in Sharc**
- **6.7.5 Impact of Resource Trading**
- **6.7.6 Scalability of Sharc**
- **6.7.6.1 Overheads Imposed by the Nucleus**
- **6.7.6.2 Control Plane Overheads**
- **6.7.7 Effect of Tunable Parameters**
- **6.7.8 Handling Failures**

## 6.8 Related Work

## 6.9 Concluding Remarks

## 7. OVERLOAD MANAGEMENT

## 7.1 Introduction

- **7.1.1 Motivation**
- **7.1.2 Research Contributions of this Chapter**
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Summary of contributions.</td>
</tr>
<tr>
<td>2.1</td>
<td>Notation used in describing the MVA algorithm.</td>
</tr>
<tr>
<td>4.1</td>
<td>Performance of VM-based switching; “n/a” stands for “not applicable”.</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of profiles. Although we profiled both CPU and network usage for each application, we only present results for the more constraining resource. Abbreviations: WS=Apache, SMS=streaming media server, GS=Quake game server, DBS=database server, k=number of clients, dyn.=dynamic, Res.=Resource.</td>
</tr>
<tr>
<td>5.2</td>
<td>Effectiveness of kernel resource allocation mechanisms. All results are shown with 95% confidence intervals.</td>
</tr>
<tr>
<td>6.1</td>
<td>Capsule Placement and Reservations.</td>
</tr>
<tr>
<td>6.2</td>
<td>Capsule Placement and Reservations.</td>
</tr>
<tr>
<td>6.3</td>
<td>Failure Handling Times (with 95% Confidence Intervals).</td>
</tr>
<tr>
<td>7.1</td>
<td>A sample service-level agreement.</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Dedicated hosting platform architecture.</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>A three-tier application.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Request processing in an online auction application.</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Modeling a multi-tier application using a network of queues.</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Response time of Rubis with 95% confidence intervals. A concurrency limit of 150 for Apache and 75 for the middle Java tier is used. Figure (a) depicts the deviation of the baseline model from observed behavior when concurrency limit is reached. Figure (b) depicts the ability of the enhanced model to capture this effect.</td>
<td>34</td>
</tr>
<tr>
<td>2.5</td>
<td>Multi-tier application model enhanced to handle concurrency limits. Since each tier has only one replica, we use only one subscript in our notation.</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>Rubis based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.</td>
<td>43</td>
</tr>
<tr>
<td>2.7</td>
<td>Rubis based on Java servlets: bottleneck at CPU of database tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.</td>
<td>44</td>
</tr>
<tr>
<td>2.8</td>
<td>Rubis based on EJB: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.</td>
<td>46</td>
</tr>
<tr>
<td>2.9</td>
<td>Rubbos based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.</td>
<td>46</td>
</tr>
<tr>
<td>2.10</td>
<td>Caching at the database tier of Rubbos.</td>
<td>48</td>
</tr>
</tbody>
</table>
2.11 Load imbalance at the middle tier of Rubis. (a) and (b) present number of requests and response times classified on a per-replica basis; (c) presents response times classified according to most loaded, second most loaded, and most loaded replicas and overall average response times.

2.12 Rubis serving sessions of two classes. Sessions of class 1 were generated using workload $W_1$ while those of class 2 were generated using workload $W_2$.

2.13 Model-based dynamic provisioning of servers for Rubis.

2.14 Maximizing revenue via differentiated session policing in Rubis. The application serves two classes of sessions.

3.1 An example of the gap-preserving reduction from the Multi-dimensional Knapsack problem to the general offline placement problem.

3.2 An example of striping-based placement.

3.3 A bipartite graph indicating which capsules can be placed on which nodes.

3.4 An example of reducing the minimum-weight maximum matching problem to the minimum-weight perfect matching problem.

4.1 Strawman approaches for provisioning a multi-tier application.

4.2 The workload prediction algorithm.

4.3 Virtual Machine Based Hosting Platform Architecture.

4.4 Rubbos: Independent per-tier provisioning.

4.5 Rubbos: Provision only the Tomcat tier.

4.6 Rubbos: Model-based multi-tier provisioning.

4.7 Rubis: Blackbox provisioning.

4.8 Rubis: Model-based multi-tier provisioning.

4.9 Provisioning on day 6—typical day.

4.10 Provisioning on day 7—moderate overload.
4.11 Provisioning on day 8—extreme overload ........................................ 109

5.1 Architecture of a shared hosting platform. Each application runs on one or
more nodes and shares resources with other applications. .............. 118

5.2 An example of an On-Off trace......................................................... 122

5.3 Derivation of the usage distribution and token bucket parameters. ....... 123

5.4 Profile of the Apache Web server using the default SPECWeb99
configuration................................................................. 125

5.5 Profiles of Various Server Applications ...................................... 127

5.6 Demonstration of how an application overload may be detected by
comparing the latest resource usage profile with the original offline
profile. ................................................................. 130

5.7 Benefits of resource under-provisioning for a bursty Web server
application, a less bursty streaming server application and for
application mixes. ............................................................. 138

5.8 Performance of various capsule placement strategies...................... 140

5.9 Effect of different levels of provisioning on the PostgreSQL server CPU
profile. ................................................................. 143

6.1 Sharc architecture and abstractions. Figure (a) shows the overall Sharc
architecture. Figure (b) shows a sample cluster-wide virtual hierarchy,
a physical hierarchy on a node and the relationship between the
two. ................................................................. 154

6.2 Various scenarios that occur while trading resources among capsules. .... 158

6.3 Predictable CPU allocation and trading. Figures (a) and (b) show the CPU
allocation for the database server and the Web server capsules. Figure
(c) shows the progress of the two bursts processed by these database
servers. ................................................................. 169

6.4 Predictable network allocation and trading. Figure (a), (b) and (c) depict
network allocations of capsules of the File download application. ......... 170

6.5 Predictable allocation and resource trading. Figure (a), (b) and (c) depict
CPU usages and allocations of capsules residing on node 3. ............. 172
6.6 Application Isolation in Sharc. The allocations of all capsules on the three nodes are shown (due to space constraints, CPU usages of these capsules have been omitted). .................................................. 173

6.7 Impact of resource trading. Figure (a) shows the number of playback discontinuities seen by the three clients of the overloaded video server with and without the trading of network bandwidth. Figures (b) and (c) show a portion of the reception and playback of the second stream for the two cases. .............................................................. 175

6.8 Overheads imposed by the nucleus............................................. 176

6.9 Overheads imposed by the control plane.................................. 177

6.10 Impact of tunable parameters on capsule allocations............... 179

7.1 The Hosting Platform Architecture........................................... 187

7.2 Working of the sentry. First, the class a request belongs to is determined. If the request confirms to the leaky bucket for its class, it is admitted to the application without any further processing. Otherwise, it is put into its class-specific queue. The admission control processes the requests in various queues at frequencies given by the class-specific delays. A request is admitted to the application if there is enough capacity, else it is dropped. .............................................................. 193

7.3 Demonstration of the working of the admission control during an overload................................................................. 203

7.4 Scalability of the admission control....................................... 204

7.5 Performance of the threshold-based admission control. At t=135 sec, the threshold was set to reject all Bronze requests; at t=180 sec, it was updated to reject all Bronze and Silver requests; at t=210 sec it was updated to also reject Gold requests with a probability 0.5; finally, at t=390 sec, it was again set to reject only Bronze requests. ............... 205

7.6 Dynamic provisioning of sentries. [S=n] means the number of sentries is n now............................................................... 206

7.7 Dynamic provisioning and admission control: Performance of Applications 1 and 2. D: Default invocation of provisioning, T: Provisioning triggered by excessive drops, [N=n]: size of the server set is n now. Only selected provisioning events are shown. ............... 208
CHAPTER 1
INTRODUCTION AND MOTIVATION

1.1 Introduction: Internet Applications and Hosting Platforms

1.1.1 Internet applications

An Internet application is an application delivered to users from a server over the Internet. A popular class of Internet applications consists of Web applications such as Web-mail, online retail sales, online auctions, wikis, discussion boards, Web-logs etc. Web applications are popular due to the ubiquity of the Web browser as a client, sometimes called a thin client. The ability to update and maintain Web applications without distributing and installing software on potentially thousands of client computers is a key reason for their popularity. Not all Internet applications are Web based, for example some streaming media servers [105] or game servers [45]. During the past decade we have increasingly come to rely on these applications to conduct both our personal and business affairs. We use the terms Internet application and Internet service interchangeably in this thesis.

1.1.2 Data Centers

A data center is a facility used for housing a large amount of electronic equipment, typically computers and communications equipment. As the name implies, a data center is usually maintained by an organization for the purpose of handling the data necessary for its operations. A bank for example may have a data center, where all its customers’ account information is maintained and transactions involving this data are carried out. Practically

\footnote{Notice that our focus is exclusively on applications based on the client-server model. We do not consider the recently popular peer-to-peer applications [47, 80] in this work.}
every company mid-sized and upwards has some kind of data center, and large companies often have dozens of data centers. Most large cities have many purpose-built data center buildings built to provide data center space in secure locations close to telecommunications services.

**Physical Layout:** A data center can occupy one room of a building, one or more floors, or up to the whole building. Most of the equipment is often in the form of 1U servers (so-called “pizza boxes”) racked up in 19 inch rack cabinets, which are usually placed in single rows forming corridors between them. This allows people access to the front and rear of each cabinet. Some equipment such as mainframe computers and storage devices are often as big as the racks themselves, and are placed alongside them.

- Air conditioning is used to keep the room cool, generally around 17 degrees Celsius (about 63 degrees Fahrenheit). This is crucial since electronic equipment in a confined space generates much excess heat, and tends to malfunction if cooling is not handled.

- Backup power is often available. This can include one or more uninterrupted power supplies and diesel generators located close by.

- To prevent single points of failure, all elements of the electrical systems, including backup system, are typically fully duplicated, and critical servers are connected to both the "A-side" and "B-side" power feeds.

- Data centers typically have raised flooring made up of 2 foot (600mm) removable square tiles. These provide a plenum for air to circulate below the floor, as part of the air conditioning system, as well as providing space for power cabling. Data cabling is typically routed through overhead cable trays in modern data centers.

- Data centers often have elaborate fire prevention and fire extinguishing systems.
Physical security also plays a large role with data centers. Physical access to the site is usually restricted to selected personnel. Video camera surveillance and permanent security guards are almost always present if the data center is large or contains sensitive information on any of the systems within.

**Network Infrastructure:** Communications in data centers today are most often based on networks running the IP protocol suite. Data centers contain a set of routers and switches that transport traffic between the servers and to the outside world. Some of the servers at the data center are used for running the basic Internet and Intranet services needed by internal users in the organization: email servers, proxy servers, DNS servers, etc. Network security elements are also usually deployed: firewalls, VPN gateways, Intrusion detection systems, etc. Also common are monitoring systems for the network and some of the applications.

### 1.1.3 Hosting Platforms

Due to the prevalence of Internet applications, data centers that host them have become an important and attractive business. We refer to such data centers as *hosting platforms*. To make an application available to the Internet community, it needs to be hosted on one or more servers. For example, a Web site needs to be hosted on a Web server which is a powerful computer that can accommodate thousands of requests for the Web site pages. A Web server has to be connected to the Internet all day and night so that users can access it anytime. The high complexity and cost of maintaining a hosting platform infrastructure has resulted in a growing trend among businesses and institutions to have their applications hosted on platforms managed by another party. Such hosting platforms are referred to as *third-party hosting platforms*. A Web hosting provider is an example of a third-party hosting platform that sells space on its servers to Website owners. They provide a full-time, high-bandwidth connection to the Internet, so that visitors can access the sites easily. An example is Yahoo’s *Small Business Web hosting* service [126]. We list below some examples of the complexity and cost involved in maintaining a hosting platform:
1. Servers and software (Web server, mail server, firewall, virus protection etc.) can be expensive.

2. The server needs a 24/7 high speed connection to the Internet, which is relatively costly.

3. Setting up all the configurations including mail server, FTP server, and DNS server can be complicated.

4. Server maintenance requires twenty-four hour support, special skills, and knowledge.

Henceforth, we refer to such a third-party hosting platform as simply a hosting platform. Hosting platforms enable entrepreneurs and emerging organizations to focus on their business rather than technology. Hosting platforms are typically expected to provide performance guarantees to the hosted applications (such as guarantees on response time or throughput) in return for revenue [94]; these contracts are expressed using service-level agreements.

1.2 Models of Hosting

Due to the rapid advances in computing and networking technologies and falling hardware prices, server clusters built using commodity hardware have become an attractive alternative to the traditional large multiprocessor servers for constructing hosting platforms. Depending on the resource requirements of the applications and the strictness of the performance or resource guarantees they require, a platform may employ a shared or a dedicated model for hosting them. We elaborate on these two models of hosting applications next. Henceforth, we use the terms server and node interchangeably.

1.2.1 Dedicated Hosting

In dedicated hosting each application runs on a subset of the servers and a server is allocated to at most one application component at any given time. Dedicated hosting is
used for running large clustered applications where server sharing is infeasible due to the workload demand imposed on each individual application. In dedicated hosting either an entire cluster runs a single application (such as a Web search engine), or each individual processing element in the cluster is dedicated to a single application (as in the “managed hosting” services provided by some data centers [73]).

### 1.2.2 Shared Hosting

Shared hosting platforms run a large number of different third-party applications (Web servers, streaming media servers, multi-player game servers, e-commerce applications, etc.), and the number of applications typically exceeds the number of nodes in the cluster. More specifically, each application runs on a subset of the nodes and these subsets may overlap. Whereas dedicated hosting platforms are used for many niche applications that warrant their additional cost, economic reasons of space, power, cooling, and cost make shared hosting platforms an attractive choice for many application hosting environments. For example, now-a-days Web hosting is very cheap (usually starting from under $5/month). There are free Web hosting companies also that recover their costs by showing advertisements on the hosted Websites.

### 1.3 Internet Hosting Platform Design Challenges and Requirements

The objective of a hosting platform is to maximize the revenue generated from the hosted applications while satisfying the service-level agreements. Designing a hosting platform is made challenging by the following characteristics of Internet applications and their workloads.

**Application and Platform Idiosyncrasies**

1. *Complex multi-tier software architecture:* Modern Internet applications are complex, distributed software systems designed using multiple *tiers*. A multi-tier architecture
provides a flexible, modular approach for designing such applications. Each application tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. The various tiers participate in the processing of each incoming request during its lifetime in the system. Additionally, these applications may employ replication and caching at one or more tiers. These characteristics of Internet applications make inferring requirements and provisioning capacity non-trivial tasks.

2. *Dynamic content*: An increasing fraction of the content delivered by Internet applications is generated dynamically [100]. Generation of dynamic content is significantly more resource intensive than static content which accounted for the bulk of the Internet traffic a few years ago.

3. *Diverse software components*: Internet applications are built using diverse software components. For example, a typical e-commerce application consists of three tiers—a front-end Web tier that is responsible for HTTP processing, a middle tier Java enterprise server that implements core application functionality, and a backend database that stores product catalogs and user orders. These application have vastly different performance characteristics.

4. *Heterogeneous hardware*: In most hosting platforms, hardware resources get added or removed incrementally resulting in heterogeneity in the hardware.

**Internet Workload Characteristics**

1. *Multi-time-scale workload variations*: Internet applications see dynamically changing workloads that contain long-term variations such as time-of-day effects [52] as well as short-term fluctuations such as transient overloads [1]. Predicting the peak workload of an Internet application and capacity provisioning based on this estimate are known to be notoriously difficult.
2. **Extreme overloads**: There are numerous documented examples of Internet applications that faced an outage due to an unexpected overload. For instance, the normally well-provisioned Amazon.com site suffered a forty-minute down-time due to an overload during the popular holiday season in November 1999. The load seen by on-line brokerage Web sites during the unexpected 1999 stock market crash was several times greater than the normal peak load, resulting in degraded performance and possible financial losses to users.

3. **Session-based workloads**: Modern Internet workloads are often session-based, where each session comprises a sequence of requests with intervening think-times. For instance, a session at an online retailer comprises the sequence of user requests to browse the product catalog and to make a purchase. Sessions are stateful from the perspective of the application.

4. **Multiple session classes**: Internet applications typically classify incoming sessions into multiple classes. To illustrate, an online brokerage Web site may define three classes and may map financial transactions to the *Gold* class, customer requests such as balance inquiries to the *Silver* class, and casual browsing requests from non-customers to the *Bronze* class. Typically such classification helps the application to preferentially admit requests from more important classes during overloads and drop requests from less important classes.

To meet its goal of maximizing revenue given the above challenges, a hosting platform needs to carefully multiplex its resources among the hosted applications. For this, a hosting platform requires the following mechanisms.

1. **Requirement inference**: A hosting platform should be able to accurately infer the resource requirements of applications. While underestimating the resource requirements of an application can cause violations of its performance guarantees (e.g.,
degraded response times), overestimation of requirements will result in wastage platform resources and revenue. Requirement inference may be based on analytical models of applications or on empirical observations.

2. Application placement: Application placement refers to the problem of determining where on the cluster the various components of a newly arrived application would run. It is desirable for a hosting platform to employ a placement algorithm that allows it to maximize the revenue generated by the hosted applications.

3. Workload prediction: Being able to predict the workloads of the hosted applications is desirable for determining their changing resource demands. This allows the hosting platform to decide which applications to divert its resources to during a given time period.

4. Dynamic capacity provisioning: A hosting platform should employ mechanisms to be able to dynamically change the allocation of resources to the hosted applications to match their dynamic workloads. In a dedicated hosting platform, this would mean changing the number of servers assigned to an application; in a shared hosting platform, dynamic capacity provisioning might imply changing the CPU shares (and possibly shares of other resources) of applications on some nodes.

5. Policing: To protect the applications from unanticipated overloads, a hosting platform should employ request policing mechanisms. A policer allows an application to discard excessive requests so that the admitted requests continue to experience desired performance even during overloads. Further, it is desirable for a hosting platform to preferentially admit more important requests during overloads—this is in accordance with the goal of maximizing the platform’s revenue.

6. Appropriate resource sharing OS mechanisms: A shared hosting platform needs support from the operating systems on the constituent nodes to effectively partition re-
sources such as CPU, network bandwidth, memory etc. among the hosted application components.

Additionally, a hosting platform should be robust. We elaborate on what we mean by this below.

1. **Scalability**: The hosted applications should be able to operate even when the request arrival rate is much higher than the anticipated workload.

2. **Failure handling**: The hosting platform should employ mechanisms to handle various kinds of software and hardware failures that may occur.

### 1.4 The Case for a Novel Resource Management Approach: Inadequacies of Existing Work

During the past decade, several researchers have contributed to different facets of the resource management problem in hosting platforms. In this section (i) we describe the problems that have been solved (and that our thesis builds on) and (ii) we argue that there are several problems that this body of work has either not addressed at all or not solved to satisfaction.

Predictable resource allocation within a single machine is a well-researched topic. Several techniques for predictable allocation of resources within a single machine have been developed over the past decade. New ways of defining resource principals have been proposed that go beyond the traditional approach of equating resource principals with entities like processes and threads. Banga et al. provide a new operating system abstraction called a *resource container* which enables fine grained allocation of resources and accurate accounting of resource consumption in a single server [14]. *Scheduling domains* in the Nemesis operating system [68], *activities* in Rialto [59], and *Software Performance Units* [116] are other examples. Numerous approaches have been proposed for predictable
scheduling of CPU cycles and network bandwidth on a single machine among competing applications. These include proportional-share schedulers such as Borrowed Virtual Time [37] and Start-time Fair Queuing [50], and reservation-based schedulers as in Rialto [59] and Nemesis [68].

There has also been work on predictable allocation of memory, disk bandwidth and shared services in single servers. Verghese et al. [116] address the problem of managing resources in a shared-memory multiprocessor to provide performance guarantees to high-level logical entities (called software performance units (SPUs)) such as a group of processes that comprise a task. Their resource management scheme, called “performance isolation”, has been implemented on the Silicon Graphics IRIX operating system for three system resources: CPU, memory, and disk bandwidth. Of particular interest is their mechanism for providing isolation with respect to physical memory, which works by having dynamically adjustable limits on the number of pages that different SPUs are entitled to based on their usage and importance. They also implement some mechanisms for managing shared kernel resources such as spinlocks and semaphores. Reumann et al. [60] propose an OS abstraction called Virtual Service (VS) to eliminate the performance interference caused by shared services such as DNS, proxy cache services, time services, distributed file systems, and shared databases. VSs provide per-service resource partitioning and management by dynamically deciding resource bindings for shared services in a manner transparent to the applications. Also the resource bindings for shared services are delayed until it is known who they work for.

In our work we build on such single-node resource management mechanisms and extend their benefits to distributed applications running on a cluster.

**Current application models are too simplistic.** Most of the existing work on modeling Internet applications has looked at single-tier applications such as replicated Web servers [36, 23, 69, 3, 74]. Since these efforts focus primarily on single-tier Web servers, they are not directly applicable to applications employing multiple tiers, or to components
such as Java enterprise servers or database servers employed by multi-tier applications. Further, many of the above efforts assume static Web content, while multi-tier applications, by their very nature, serve dynamic Web content. Although a few recent efforts have focused on the modeling of multi-tier applications, many of these efforts either make simplifying assumptions or are based on simple extensions of single-tier models [118, 91, 61].

These models are not sophisticated enough to capture the various application idiosyncrasies we had described earlier.

**Dynamic capacity provisioning has been studied only in the context of single-tier applications.** Several papers have addressed the problem of dynamic resource allocation to competing applications running on a single server. Chandra et al. [24] propose a system architecture that combines online measurements with workload prediction and resource allocation techniques. The goal of their technique is to react to changing workloads by dynamically varying the resource shares of applications. Pradhan et al. [87] propose an observation-based approach that has the goal of designing self-managing Web servers that can adapt to changing workloads while maintaining QoS requirements of different request classes. While Chandra et al. [24] consider dynamic management of CPU, Pradhan et al. [87] manage CPU and the accept queue. Doyle et al. [36] present an approach for provisioning memory and storage resources based on simple queuing theoretic models of service behavior to predict resource requirements under changing load.

All these techniques focus on resource allocation for applications running on a single server and are inadequate for platforms hosting multi-tiered applications with components distributed across multiple nodes.

**Existing policing mechanisms do not scale.** Although a lot of research has been conducted on developing admission control algorithms for Internet applications [29, 42, 62, 70, 117, 123], the issue of the scalability of the policer itself has been unaddressed. During extreme overloads, the policer units can become bottlenecks resulting in indiscriminate, class-unaware dropping of requests causing loss in revenue.
1.5 Thesis Summary and Contributions

Having discussed the shortcomings of existing work, we describe the contributions made by our thesis.

Analytical Models for Multi-tier Applications

In this thesis, we propose analytical models of multi-tier Internet applications. Modeling single-tier applications such as vanilla Web servers (e.g., Apache) is well-studied [36, 74, 102]. In contrast, modeling multi-tier applications is less well-studied, even though this flexible architecture is widely used for constructing Internet applications and services. Extending single-tier models to multi-tier scenarios is non-trivial. Our models can handle applications with an arbitrary number of tiers and tiers with significantly different performance characteristics. Our models are designed to handle session-based workloads and can account for application idiosyncrasies such as replication at tiers, load imbalances across replicas, caching effects, and concurrency limits at each tier.

Application Placement

We formulate the application placement problem that arises in shared hosting platforms. We study the theoretical properties of this problem and develop online algorithms.

Dynamic Capacity Provisioning in Dedicated Hosting Platforms

Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. Dynamic provisioning of resources—allocation and deallocation of servers to replicated applications—has been studied in the context of single-tier applications, of which clustered HTTP servers are the most common example. However, it is non-trivial to extend provisioning mechanisms designed for single-tier applications to multi-tier scenarios. We design a dynamic capacity provisioning approach for multi-tier Internet applications based on a combination of predictive and reactive mech-
Managing Resources in Shared Hosting Platforms

Shared hosting environments present us with some distinct resource management challenges and opportunities. In particular, unlike dedicated environments (i) we need mechanisms to isolate collocated application components from each other and (ii) it is possible to achieve finer grain multiplexing of resources. We devise an offline profiling based technique to infer the resource needs of applications and show how a shared platform may improve its revenue by careful under-provisioning of its resources.

Overload Management

We propose overload management mechanisms that allow a hosting platform to remain operational even under extreme overloads. Our mechanisms allow an application to handle request arrival rates of several thousand requests/sec.

Table 1.1 summarizes the contributions of this thesis.

1.6 Overview of Our Hosting Platform Design

We implement all our resource management algorithms in a prototype hosting platform based on a cluster of Linux machines and evaluate them using realistic applications and
workloads. We present the architecture of our hosting platform in Figure 1.1. We also introduce some terminology that we use throughout this thesis.

Our hosting platform consists of two main components—the control plane and the nucleus—that are responsible for managing resources in the cluster. The control plane manages resources on a cluster-wide basis—it implements the application models, and the algorithms for application placement and dynamic provisioning. The nucleus is responsible for managing resources on each individual node. It takes various measurements that are needed by the placement, provisioning, and policing algorithms. Architecturally, the nucleus is distinct from the operating system kernel on a node. Moreover, unlike a middleware, the nucleus does not sit between applications and the kernel; rather it complements the functionality of the operating system kernel. We describe the design of these components in Chapters 4 and 6.

As shown, an application may consist of multiple tiers. The figure shows a dedicated platform with each tier running on its own server. In a shared platform, we allow multiple application components to share a single server. The rest of the architecture is identical for both hosting models.
Each application is guarded by a sentry which performs admission control to turn away excess requests during overloads. We elaborate on the design of a sentry in Chapter 7.

We borrow terminology from Roscoe and Lyles [94] and refer to that component of an application that runs on an individual node as a capsule. Each application has at least one capsule and more if the application is distributed. Each capsule consists of one or more resource principals (processes, threads), all of which belong to the same application. Capsules provide a useful abstraction for logically partitioning an application into sub-components and for exerting control over the distribution of these components onto different nodes. To illustrate, consider an e-commerce application consisting of a Web server, a Java application server, and a database server. If all three components need to be collocated on a single node, then the application will consist of a single capsule with all three components. On the other hand, if each component needs to be placed on a different node, then the application should be partitioned into three capsules. Depending on the number of its capsules, each application runs on a subset of the platform nodes and these subsets can overlap with one another in shared hosting.

1.7 Dissertation Road-map

The rest of this thesis is structured as follows. In Chapter 2, we present analytical models for Internet applications. In Chapter 3, we discuss the issue of application placement in hosting platforms. Chapter 4 considers the problem of dynamic capacity provisioning for Internet applications in a dedicated hosting environment. Chapters 5 and 6 presents resource management solutions unique to a shared hosting environment. Chapter 7 addresses overload management in hosting platforms. We conclude with a summary of our research contributions in Chapter 8.
CHAPTER 2
APPLICATION MODELING

2.1 Introduction

2.1.1 Motivation

Modern Internet applications are complex software systems that employ a multi-tier architecture and are replicated or distributed on a cluster of servers. Each tier provides a certain functionality to its preceding tier and makes use of the functionality provided by its successor to carry out its part of the overall request processing. For instance, a typical e-commerce application consists of three tiers—a front-end Web tier that is responsible for HTTP processing, a middle tier Java enterprise server that implements core application functionality, and a backend database that stores product catalogs and user orders. In this example, incoming requests undergo HTTP processing, processing by Java application server, and trigger queries or transactions at the database.

This chapter focuses on analytically modeling the behavior of multi-tier Internet applications. Such a model is important for the following reasons: (i) capacity provisioning, which enables a server farm to determine how much capacity to allocate to an application in order for it to service its peak workload; (ii) performance prediction, which enables the response time of the application to be determined for a given workload and a given hardware and software configuration, (iii) application configuration, which enables various configuration parameters of the application to be determined for a certain performance goal, (iv) bottleneck identification and tuning, which enables system bottlenecks to be identified for purposes of tuning, and (v) request policing, which enables the application to turn away excess requests during transient overloads.
Modeling single-tier applications such as vanilla Web servers (e.g., Apache [5]) is well studied [36, 74, 102]. In contrast, modeling of multi-tier applications is less well studied, even though this flexible architecture is widely used for constructing Internet applications. Extending single-tier models to multi-tier scenarios is non-trivial due to the following reasons. First, various application tiers such as Web, Java, and database servers have vastly different performance characteristics and collectively modeling their behavior is a difficult task. Further, in a multi-tier application, (i) some tiers may be replicated while others are not, (ii) the replicas may not be perfectly load balanced, and (iii) caching may be employed at intermediate tiers—all of which complicate the performance modeling. Finally, modern Internet workloads are session-based, where each session comprises a sequence of requests with think-times in between. For instance, a session at an online retailer comprises the sequence of user requests to browse the product catalog and to make a purchase. Sessions are stateful from the perspective of the application, an aspect that must be incorporated into the model. The design of an analytical model that can capture the impact of these factors is the focus of this chapter.

2.1.2 Research Contributions

This chapter presents a model of a multi-tier Internet application based on a network of queues, where the queues represent different tiers of the application. Our model can handle applications with an arbitrary number of tiers and those with significantly different performance characteristics. A key contribution of our work is that the complex task of modeling a multi-tier application is reduced to the modeling of request processing at individual tiers and the flow of requests across tiers. Our model is inherently designed to handle session-based workloads and can account for application idiosyncrasies such as replication at tiers, load imbalances across replicas, caching effects, and concurrency limits at each tier.

We validate the model using two open-source multi-tier applications running on a Linux-based server cluster. We demonstrate the ability of our model to accurately cap-
ture the effects of a number of commonly used techniques such as query caching at the database tier and class-based service differentiation. For a variety of scenarios, including an online auction application employing query caching at its database tier, the average response times predicted by our model were within the 95% confidence intervals of the observed average response times. We conduct a detailed experimental study using our prototype to demonstrate the utility of our model for the purposes of dynamic provisioning, response time prediction, application configuration, and request policing. Our experiments demonstrate the ability of our model to correctly identify bottlenecks in the system and the shifting of bottlenecks due to variations in the Internet workload.

The remainder of this chapter is structured as follows. Section 2.2 provides an overview of multi-tier applications and related work. We describe our model in Sections 2.3 and 2.4. Sections 2.5 and 2.6 present experimental validation of the model and an illustration of its applications respectively. Finally, Section 2.7 presents our conclusions.

This section provides an overview of multi-tier applications. We also discuss related work in the area.

### 2.1.3 Internet Application Architecture

Modern Internet applications are designed using multiple tiers. A multi-tier architecture provides a flexible, modular approach for designing such applications. Each application tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. The various tiers participate in the processing of each incoming request during its lifetime in the system. Request processing at each tier consists of an interleaving of periods where the tier performs some work on the request and periods where it awaits some service it requested from its successor tier, if any. A request circulates among the various application tiers during its lifetime and may get processed at certain tiers multiple times before the final response is constructed and returned to the client. Depending on the processing demand, a tier may
be replicated using clustering techniques. In such an event, a dispatcher is used at each replicated tier to distribute requests among the replicas for the purpose of load balancing. Figure 2.1 depicts a three-tier application where the first two tiers are replicated, while the third one is not. Such an architecture is commonly employed by e-commerce applications where a clustered Web server and a clustered Java application server constitute the first two tiers, and the third tier consists of a non-replicable database.

The workload of an Internet application is assumed to be session-based, where a session consists of a succession of requests issued by a client with think times in between. If a session is stateful, which is often the case, successive requests will need to be serviced by the same server at each tier, and the dispatcher will need account for this server state when redirecting requests.

As shown in Figure 2.1, each application employs a sentry that polices incoming sessions to an application’s server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads.

---

1Traditionally database servers have employed a shared-nothing architecture that does not support replication. However, certain new databases employ a shared-everything architecture [82] that supports clustering and replication but with certain constraints.
In this work, we assume a dedicated hosting model. That is, we assume that each tier of an application (or each replica of a tier) runs on a separate server. Our approach can be easily extended to model applications hosted on a shared hosting platform if the nodes in the platform have the following property—each node employs schedulers that guarantee to provide specified fractions of its resources to the capsules running on that node. An example of such a scheduler is a reservation-based scheduler. A reservation-based scheduler allows resource requirements to be specified in absolute terms. Numerous reservation-based schedulers such as Nemesis [68], Rialto [59] and DSRT [71] have been proposed recently. A typical resource specification for such schedulers is a pair \((x, y)\), where \(x\) units of the resource are requested every \(y\) time units (effectively requesting \(x/y\) fraction of the resource). Analytical modeling of applications in a shared hosting environment where the nodes do not employ such schedulers (e.g., nodes running vanilla Linux with its time-sharing CPU scheduler) is a harder problem and is beyond the scope of this thesis. We propose an alternate empirical approach for modeling such applications in shared environments in Chapter 5.

Given an Internet application, we assume that it specifies its desired performance requirement in the form of a service-level agreement (SLA). The SLA assumed in this work is a bound on the average response time that is acceptable to the application. For instance, the application SLA may specify that the average response time should not exceed one second regardless of the workload.

2.2 Background and Related Work

2.2.1 Request Processing in Multi-tier Applications

Consider a multi-tier application consisting of \(M\) tiers denoted by \(T_1, T_2 \text{ through } T_M\). In the simplest case, each request is processed exactly once by tier \(T_i\) and then forwarded to tier \(T_{i+1}\) for further processing. Once the result is computed by the final tier \(T_M\), it is sent back to \(T_{M-1}\), which processes this result and sends it to \(T_{M-2}\) and so on. Thus, the
result is processed by each tier in the reverse order until it reaches $T_1$, which then sends it to the client. Figure 2.2 illustrates the steps involved in processing a “bid” request at a three-tier online auction site. The figure shows how the request trickles downstream and how the result propagates upstream through the various tiers

More complex processing at the tiers is also possible. In such scenarios, each request can visit a tier *multiple* times. As an example, consider a keyword search at an online superstore, which triggers a query on the music catalog, a query on the book catalog and so on. These queries can be issued to the database tier sequentially, where each query is issued after the result of the previous query has been received, or in parallel. Thus, in the general case, each request at tier $T_i$ can trigger multiple requests to tier $T_{i+1}$. In the sequential case, each of these requests is issued to $T_{i+1}$ once the result of the previous request has finished. In the parallel case, all requests are issued to $T_{i+1}$ at once. In both cases, all results are merged and then sent back to the upstream tier $T_{i-1}$.

### 2.2.2 Related Work

Modeling single-tier Internet applications, of which HTTP servers are the most common example, has been studied extensively. Slothouber proposes a queuing model of a

---

2We describe the various software technologies used to build this example application in Section 2.5.1.
Web server serving static content [102]. The model employs a network of four queues—two modeling the Web server itself, and the other two modeling the Internet communication network. Doyle et al. propose a queuing model for performance prediction of single-tier Web servers with static content [36]. This approach (i) explicitly models CPU, memory, and disk bandwidth in the Web server, (ii) utilizes knowledge of file size and popularity distributions, and (iii) relates average response time to available resources. Chandra et al. present a GPS-based (generalized processor sharing [84]) queuing model of a single resource, such as the CPU, at a Web server [23]. The model is parameterized by online measurements and is used to determine the resource allocation needed to meet desired average response time targets. The paper proposes an extension to multiple resources by splitting response time into per-resource delays and using the single-resource model. However, the problem of partitioning the response time is unaddressed. We propose a G/G/1 queuing model for replicated single-tier applications (e.g., clustered Web servers) [112]. Levy et al. present the architecture and prototype implementation of a performance management system for cluster-based Web services [69]. The work employs an M/M/1 queuing model to compute responses times of Web requests. Abdelzaher et al. study a model of a Web server for the purpose of performance control using classical feedback control theory [3]; they also present an implementation and evaluation using the Apache Web server in this work. Menasce employs a combination of a Markov chain model and a queuing network model to capture the operation of a Web server—the former model represents the software architecture employed by the Web server (e.g. process-based versus thread-based) while the latter computes the Web server’s throughput [74].

Since these efforts focus primarily on single-tier Web servers, they are not directly applicable to applications employing multiple tiers, or to components such as Java enterprise servers or database servers employed by multi-tier applications. Furthermore, many of the above efforts assume static Web content, while multi-tier applications, by their very nature, serve dynamic Web content.
A few recent efforts have focused on the modeling of multi-tier applications. However, many of these efforts either make simplifying assumptions or are based on simple extensions of single-tier models. A number of papers have taken the approach of modeling only the most constrained or the most bottlenecked tier of the application. For instance, Villela et al. consider the problem of provisioning servers for only the Java application tier; they use an M/G/1/PS model\(^3\) for each server in this tier [118]. Similarly, Ranjan et al. model the Java application tier of an e-commerce application with \(N\) servers as a G/G/N queuing system [91]. Other efforts have modeled the entire multi-tier application using a single queue. For example, Kamra et al. use a M/GI/1/PS model for an e-commerce application [61]. While these approaches are useful for specific scenarios, they have many limitations. For instance, modeling only a single bottlenecked tier of a multi-tier application will fail to capture caching effects at other tiers. Such a model cannot be used for capacity provisioning of other tiers. Finally, as we show in our experiments, system bottlenecks can shift from one tier to another with changes in workloads. Under these scenarios, there is no single tier that is the most constrained. We present several shortcomings of such models in Chapter 4 using both thought experiments and experiments on a real prototype hosting platform. In this chapter, we present a model of a multi-tier application that overcomes these drawbacks. Our model explicitly accounts for the presence of all tiers and also captures application artifacts such as session-based workloads, tier replication, load imbalances, caching effects, and concurrency limits.

### 2.3 A Model for a Multi-tier Internet Application

In this section, we present a baseline queuing model for a multi-tier Internet application, followed by several enhancements to the model to capture certain application idiosyncrasies.

---

\(^3\)PS stands for the processor sharing queuing discipline [63].
Figure 2.3. Modeling a multi-tier application using a network of queues.

2.3.1 The Basic Queuing Model

Consider an application with $M$ tiers denoted by $T_1, \cdots, T_M$. Initially we assume that no tier is replicated—each tier is assumed to run on exactly one server, an assumption that is relaxed later.

**Modeling Multiple Tiers:** We model the application using a network of $M$ queues, $Q_1, \cdots, Q_M$ (see Figure 2.3). Each queue represents an application tier and the underlying server that it runs on. We assume a processor sharing (PS) discipline at each queue, since it closely approximates the scheduling policies employed by most commodity operating systems (e.g., Linux CPU time-sharing).

When a request arrives at tier $T_i$ it triggers one or more requests at its subsequent tier $T_{i+1}$; recall the example of a keyword search that triggers multiple queries at different product catalogs. In our queuing model, we capture this phenomenon by allowing a request to make multiple visits to each of the queues during its overall execution. This is achieved by introducing a transition from each queue to its predecessor, as shown in Figure 2.3. A request, after some processing at queue $Q_i$, either returns to $Q_{i-1}$ with a certain probability $p_i$ or proceeds to $Q_{i+1}$ with probability $(1 - p_i)$. The only exceptions are the last tier queue $Q_M$, where all requests return to the previous queue, and the first queue $Q_1$, where a transition to the preceding queue denotes request completion. As argued in Section 2.3.2,
our model can handle multiple visits to a tier regardless of whether they occur sequentially or in parallel.

Observe that caching effects are naturally captured by this model. If caching is employed at tier \( T_i \), a cache hit causes the request to immediately return to the previous queue \( Q_{i-1} \) without triggering any work in queues \( Q_{i+1} \) or later. Thus, the impact of cache hits and misses can be incorporated by appropriately determining the transition probability \( p_i \) and the service time of a request at \( Q_i \).

**Modeling Sessions:** Recall from Section 2.2 that Internet workloads are session-based. A session issues one or more requests during its lifetime, one after another, with intervening think times (we refer to the latter as the user think times). Typical sessions in an Internet application may last several minutes. Thus, our model needs to capture the relatively long-lived nature of sessions as well as the response times of individual requests within a session. We do so by augmenting our queuing network with a subsystem modeling the active sessions of the application. We model sessions using an infinite server queuing system, \( Q_0 \), that feeds our network of queues and forms the closed-queuing system shown in Figure 2.3. The servers in \( Q_0 \) capture the session-based nature of the workload as follows. Each active session is assumed to “occupy” one server in \( Q_0 \). As shown in Figure 2.3, a request issued by a session emanates from a server in \( Q_0 \) and enters the application at \( Q_1 \). It then moves through the queues \( Q_1, \ldots, Q_M \), possibly visiting some queues multiple times (as captured by the transitions from each tier to its preceding tier) and getting processed at the visited queues. Eventually, its processing completes, and it returns to a server in \( Q_0 \). The time spent at \( Q_0 \) corresponds to the think time of the user; the next request of the session is issued subsequently. The infinite server system also enables the model to capture the independence of the user think times from the request service times at the application.

Let \( S_i \) denote the service time of a request at \( Q_i \) \( (1 \leq i \leq M) \). Also, \( p_i \) denotes the probability that a request makes a transition from \( Q_i \) to \( Q_{i-1} \) (note that \( p_M = 1 \)); \( p_1 \) denotes the transition probability from \( Q_1 \) to \( Q_0 \). Finally, let \( Z \) denote the service time at any server.
in \( Q_0 \) (which is essentially the user think time). Our model requires these parameters as inputs in order to compute the average end-to-end response time of a request.

Our discussion thus far has implicitly assumed that sessions never terminate. In practice, the number of sessions being serviced will vary as existing sessions terminate and new sessions arrive. Our model can compute the mean response time for a given number of concurrent sessions \( N \). This property can be used for admission control at the application sentry, as discussed in Section 2.6.2.

### 2.3.2 Deriving Response Times from the Model

The Mean-Value Analysis (MVA) algorithm [92] for closed-queuing networks can be used to compute the mean response time experienced by a request in our network of queues. The MVA algorithm is based on the following key queuing theory result: In product-form closed queuing networks\(^4\), when a request moves from queue \( Q_i \) to another queue \( Q_j \), it sees, at the time of its arrival at \( Q_j \), a system with the same statistics as a system with one less customer. Consider a product-form closed-queuing network with \( N \) customers. Let \( \bar{A}_m(N) \) denote the average number of customers in queue \( Q_m \) seen by an arriving customer. Let \( \bar{L}_m(N) \) denote the average length of queue \( Q_m \) in such a system. Then, the above result implies

\[
\bar{A}_m(N) = \bar{L}_m(N - 1) \tag{2.1}
\]

Given this result, the MVA algorithm iteratively computes the average response time of a request. The MVA algorithm uses Equation (2.1) to introduce customers into the queuing network, one by one, and determines the resulting average delays at various queues at

---

\(^4\)The term product-form applies to any queuing network in which the expression for the equilibrium probability has the form of \( P(n_1, \cdots, n_M) = \frac{1}{G(N)} n_i=1 f_i(n_i) \) where \( f_i(n_1) \) is some function of the number of jobs at the \( i^{th} \) queue, \( G(N) \) is a normalizing constant. Product form solutions are known to exist for a broad class of networks, including ones where the scheduling discipline at each queue is processor sharing (PS).
each step. It terminates when all $N$ customers have been introduced, and yields the average response time experienced by $N$ concurrent customers. Note that a session in our model corresponds to a customer in the result described by Equation (2.1). The MVA algorithm for an $M$-tier Internet application servicing $N$ sessions simultaneously is presented in Algorithm 1 and the associated notation is in Table 2.1.

| input | $N, S_m, V_m, 1 \leq m \leq M; \bar{Z}$ |
| output | $\bar{R}_m$ (avg. delay at $Q_m$), $\bar{R}$ (avg. resp. time) |

**Algorithm 1**: Mean-value analysis algorithm for an $M$-tier application.

The algorithm uses the notion of a visit ratio for each queue $Q_1, \cdots, Q_M$. The visit ratio $V_m$ for queue $Q_m$ ($1 \leq m \leq M$) is defined as the average number of visits made by a request to $Q_m$ during its processing (that is, from when it emanates from $Q_0$ and when it returns to it). Visit ratios are easy to compute from the transition probabilities $p_1, \cdots, p_M$ and provide an alternate representation of the queuing network. The use of visit ratios in lieu of transition probabilities enables the model to capture multiple visits to
Table 2.1. Notation used in describing the MVA algorithm.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>Number of servers</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of sessions</td>
</tr>
<tr>
<td>$Q_m$</td>
<td>Queue representing tier $T_m$ ($1 \leq m \leq M$)</td>
</tr>
<tr>
<td>$Q_0$</td>
<td>Inf. server system to capture sessions</td>
</tr>
<tr>
<td>$Z$</td>
<td>User think time</td>
</tr>
<tr>
<td>$\bar{s}_m$</td>
<td>Avg. per-request service time at $Q_m$</td>
</tr>
<tr>
<td>$\bar{L}_m$</td>
<td>Avg. length of $Q_m$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Throughput</td>
</tr>
<tr>
<td>$\bar{R}_m$</td>
<td>Avg. per-request delay at $Q_m$</td>
</tr>
<tr>
<td>$\bar{R}$</td>
<td>Avg. per-request response time</td>
</tr>
<tr>
<td>$\bar{D}_m$</td>
<td>Avg. per-request service demand at $Q_m$</td>
</tr>
<tr>
<td>$V_m$</td>
<td>Visit ratio for $Q_m$</td>
</tr>
<tr>
<td>$A_m$</td>
<td>Avg. num. customers in $Q_m$ seen by an arriving customer</td>
</tr>
</tbody>
</table>

a tier regardless of whether they occur sequentially or in parallel—the visit ratio is only concerned with the mean number of visits made by a request to a queue and not *when* or in *what order* these visits occur.

Thus, given the average service times and visit ratios for the queues, the average think time of a session, and the number of concurrent sessions, the algorithm computes the average response time $\bar{R}$ of a request.

### 2.3.3 Estimating the Model Parameters

In order to compute the response time, the model requires several parameters as inputs. In practice, these parameters can be estimated by monitoring the application as it services its workload. To do so, we assume that the underlying operating system and application software components (such as the Apache Web server) provide monitoring hooks to enable accurate estimation of these parameters. Our experience with the Linux-based multi-tier applications used in our experiments is that such functionality is either already available or
can be implemented at a modest cost. The rest of this section describes how the various model parameters can be estimated in practice.

**Estimating visit ratios:** The visit ratio for any tier of a multi-tier application is the average number of times that tier is invoked during a request’s lifetime. Let $\lambda_{req}$ denote the number of requests serviced by the entire application over a duration $t$. Then the visit ratio for tier $T_i$ can be simply estimated as

$$V_i \approx \frac{\lambda_i}{\lambda_{req}}$$

where $\lambda_i$ is the number of requests serviced by that tier in that duration. By choosing a suitably large duration $t$, a good estimate for $V_i$ can be obtained. We note that the visit ratios are easy to estimate in an online fashion. The number of requests serviced by the application $\lambda_{req}$ can be monitored at the application sentry. For replicated tiers, the number of requests serviced by all servers of that tier can be monitored at the dispatchers. Monitoring of both parameters requires simple counters at these components. For non-replicated tiers that lack a dispatcher, the number of serviced requests can be determined by real-time processing of the tier logs. In the database tier, for instance, the number of queries and transactions processed over a duration $t$ can be determined by processing the database log using a script.

**Estimating service times:** Application components such as Web, Java, and database servers all support extensive logging facilities and can log a variety of useful information about each serviced request. In particular, these components can log the residence time of individual requests as observed at that tier—the residence time includes the time spent by the request at this tier and all the subsequent tiers that processed this request. This logging facility can be used to estimate per-tier service times. Let $\bar{X}_i$ denote the average per-request residence time at tier $T_i$. We start by estimating the mean service time at the last tier. Since this tier does not invoke services from any other tiers, the request execution time at this tier under lightly loaded conditions is an excellent estimate of the service time. Thus, we have,
\[ S_M \approx X_M \]

Let \( S_i, X_i, \) and \( n_i \) be random variables denoting the service time of a request at a tier \( T_i, \) residence time of a request at tier \( T_i, \) and the number of times \( T_i \) requests service from \( T_{i+1} \) as part of the overall request processing, respectively. Then, under lightly loaded conditions,

\[ S_i = X_i - n_i \cdot X_{i+1}, \quad 1 \leq i < M. \]

Taking averages on both sides, we get,

\[ \bar{S}_i = \bar{X}_i - E[n_i \cdot X_{i+1}] \]

Since \( n_i \) and \( X_{i+1} \) are independent, this gives us,

\[ \bar{S}_i = \bar{X}_i - \bar{n}_i \cdot \bar{X}_{i+1} = \bar{X}_i - \left( \frac{V_{i+1}}{V_i} \right) \cdot \bar{X}_{i+1} \]

Thus, the service times at tiers \( T_1, \ldots, T_{M-1} \) can be estimated.

**Estimating think times:** The average user think time, \( \bar{Z} \), can be obtained by recording the arrival and finish times of individual requests at the sentry. \( \bar{Z} \) is estimated as the average time elapsed between when a request finishes and when the next request (belonging to the same session) arrives at the sentry. By using a sufficient number of observations, we can obtain a good estimate of \( \bar{Z} \).

**Increased Service Times During Overloads:** Our estimation of the tier-specific service times assumed lightly loaded conditions. As the load on a tier grows, software overheads such as waiting on locks, virtual memory paging, and context switch overheads, that are not captured by our model, can become significant components of the request processing time.
Incorporating the impact of increased context switching overhead or contention for memory or locks into our model is non-trivial. Rather than explicitly modeling these effects, we implicitly account for their impact by associating increased service times with requests under heavy loads. We use the Utilization Law \[ 67 \] for a queuing system which states that \( S = \rho / \tau \), where \( \rho \) and \( \tau \) are the queue utilization and throughput, respectively. Consequently, we can improve our estimate of the average service time at tier \( T_i \) as

\[
S'_{i} = \max \left( S_i, \frac{\rho_i}{\tau_i} \right)
\]

where \( \rho_i \) is the utilization of the busiest resource (e.g. CPU, disk, or network interface) and \( \tau_i \) is the tier throughput. Since all modern operating systems support facilities for monitoring system performance (e.g., the sysstat package in Linux \[ 96 \]), the utilizations of various resources is easy to obtain online. Similarly, the tier throughput \( \rho_i \) can be determined at the dispatcher (or from logs) by counting the number of completed requests in a duration \( t \).

### 2.4 Model Enhancements

This section proposes enhancements to our baseline model to capture four application artifacts—replication and load imbalance at tiers, concurrency limits, and multiple session classes.

#### 2.4.1 Replication and Load Imbalance at Tiers

Recall that our baseline model assumes a single server (queue) per tier and consequently does not support the notion of replication at a tier. We now enhance our model to handle this scenario. Let \( r_i \) denote the number of replicas at tier \( T_i \). Our approach to capture replication at tier \( T_i \) is to replace the single queue \( Q_i \) with \( r_i \) queues, \( Q_{i,1}, \ldots, Q_{i,r_i} \), one for each replica. A request in any queue can now make a transition to any of the \( r_{i-1} \) queues of the previous tier or to any of the \( r_{i+1} \) queues of the next tier.
In general, whenever a tier is replicated, a dispatcher is necessary to distribute requests to replicas. The dispatcher determines which request to forward to which replica and directly influences the transitions made by a request.

The dispatcher is also responsible for balancing load across replicas. In a perfectly load balanced system, each replica processes \( \frac{1}{r_i} \) fraction of the total workload of that tier. In practice, however, perfect load balancing is difficult to achieve for the following reasons. First, if a session is stateful, successive requests will need to be serviced by the same stateful server at each tier; the dispatcher is forced to forward all requests from a session to this replica regardless of the load on other replicas. Second, if caching is employed by a tier, a session and its requests may be preferentially forwarded to a replica where a response is likely to be cached. Thus, sessions may have affinity for particular replicas. Third, different sessions impose different processing demands. This can result in variability in resource usage of sessions, and simple techniques such as forwarding a new session to the least-loaded replica may not be able to counter the resulting load imbalance. Thus, the issues of replication and load imbalance are closely related. Our enhancement captures the impact of both these factors.

In order to capture the load imbalance across replicas, we explicitly model the load at individual replicas. Let \( \lambda_i^j \) denote the number of requests forwarded to the \( j^{th} \) most loaded replica of tier \( T_i \) over some duration \( t \). Let \( \lambda_i \) denote the total number of requests handled by that tier over this duration. Then, the imbalance factor \( \beta_i^j \) is computed as

\[
\beta_i^j = \left( \frac{\lambda_i^j}{\lambda_i} \right)
\]

We use exponentially smoothed averages of these ratios \( \bar{\beta}_i^j \) as measures of the load imbalance at individual replicas. The visit ratios of the various replicas are then chosen as

\[
V_{i,j} = V_i \bar{\beta}_i^j
\]
The higher the load on a replica, the higher the value of the imbalance factor, and the higher its visit ratio. In a perfectly load-balanced system, $\beta_i^j = \frac{1}{r_i}, \forall j$. Observe that the number of requests forwarded to a replica $\lambda_i^j$ and the total number of requests $\lambda_i$ can be measured at the dispatcher using counters. The MVA algorithm can then be used with these modified visit ratios to determine the average response time.

2.4.2 Handling Concurrency Limits at Tiers

The software components of an Internet application have limits on the amount of concurrency they can handle. For instance, the Apache Web server uses a configurable parameter to limit the number of concurrent threads or processes that are spawned to service requests. This limit prevents the resident memory size of Apache from exceeding the available RAM and prevents thrashing. Connections are turned away when this limit is reached. Other tiers impose similar limits.

The model developed thus far assumes that each replica at any tier can service an unbounded number of simultaneous requests and fails to capture the behavior of the application when the concurrency limit is reached at any software component. This is depicted in Figure 2.4(a), which shows the response time of a three-tier application called Rubis that is configured with a concurrency limit of 150 for the Apache Web server and a limit of 75 for the middle Java tier (details of the application appear in Section 2.5.1). As shown, the response times predicted by the model match the observed response times until the concurrency limit is reached. Beyond this point, the model continues to assume an increasing number of simultaneous requests being serviced and predicts an increase in response time, while the actual response time of successful requests shows a flat trend due to an increasing number of dropped requests.

In general, when the concurrency limit is reached at a software component in tier $T_i$, one of two actions are possible: (1) it can silently drop additional requests and rely upon a timeout mechanism in the software component in tier $T_{i-1}$ that issued this request to detect
Figure 2.4. Response time of Rubis with 95% confidence intervals. A concurrency limit of 150 for Apache and 75 for the middle Java tier is used. Figure (a) depicts the deviation of the baseline model from observed behavior when concurrency limit is reached. Figure (b) depicts the ability of the enhanced model to capture this effect.

these drops, or (2) it can explicitly notify tier \( T_{i-1} \) of its inability to serve the request (by returning an error message). In either case, tier \( T_{i-1} \) may reissue the request some number of times before abandoning its attempts. It will then either drop the request or explicitly notify its preceding tier. Finally, tier \( T_1 \) can notify the client of the failure.

Rather than distinguishing these possibilities, we employ a general approach for capturing these effects. As before we use \( V_{i,j} \) to denote the visit ratio to the replica \( Q_{i,j} \) in tier \( T_i \). Notice that the online technique described in Section 2.3.3 will not accurately estimate the visit ratio at a software component if its concurrency limit has been reached. This is because the concurrency limit will cause some requests to be dropped whereas the technique presented in Section 2.3.3 is based on the assumption that all requests arriving at a software component are successfully serviced by it. Therefore, our enhancement relies on visit ratios estimated using offline measurements conducted with all concurrency limits set to sufficiently high values. These visit ratios are corrected to capture load imbalances at replicated tiers exactly as described in Section 2.4.1. Let \( K_i \) denote the concurrency limit at \( Q_{i,j} (1 \leq j \leq r_i) \). To capture requests that are dropped at \( Q_{i,j} \) when its concurrency
limit is reached, we add an additional transition to the model developed thus far. At the entrance of $Q_{i,j}$, we add a transition into an infinite server queuing subsystem $Q^\text{drop}_{i,j}$. Let $V^\text{drop}_{i,j}$ denote the visit ratio for $Q^\text{drop}_{i,j}$ as shown in Figure 2.5. For sake of clarity we have only shown one replica at each tier. $Q^\text{drop}_{i,j}$ has a mean service time of $S^\text{drop}_i$; notice that this is the same for all the replicas in the tier $T_i$. This enhancement allows us to distinguish between the processing of requests that get dropped due to concurrency limits and those that are processed successfully. Requests that are processed successfully are modeled exactly as in the basic model. Requests that are dropped at $Q_{i,j}$ experience some delay in the subsystem $Q^\text{drop}_{i,j}$ before returning to $Q_0$—this models the delay between when a request is dropped at tier $T_i$ and when this information gets propagated to the client that initiated the request.

Like in the baseline model, we can use the MVA algorithm to compute the response time of a request. The algorithm computes the fraction of requests that finish successfully and those that encounter failures, as well as the delays experienced by both types of requests. To do so, we need to estimate the additional parameters that we have added to our basic model, namely, $V^\text{drop}_{i,j}$ for each replica in tier $T_i$ and $S^\text{drop}_i$ for each tier $T_i$.

**Estimating $V^\text{drop}_{i,j}$:** Our approach to estimate $V^\text{drop}_{i,j}$ consists of the following two steps.

**Step 1:** Estimate throughput of the queuing network if there were no concurrency limits:
Solve the queuing network shown in Figure 2.5 using the MVA algorithm using $V^\text{drop}_{i,j} = 0$ (i.e., assuming that the queues have no concurrency limits). Let $\lambda$ denote the throughput computed by the MVA algorithm in this step.

**Step 2:** Estimate $V^\text{drop}_{i,j}$: Treat $Q_{i,j}$ as an open, finite-buffer M/M/1/$K_i$ queue with arrival rate $\lambda V^\text{drop}_{i,j}$ (using the $\lambda$ computed in Step 1). Let $p^\text{drop}_{i,j}$ denote the probability of buffer overflow in this M/M/1/$K_i$ queue [63]. Then $V^\text{drop}_{i,j}$ is estimated as: $V^\text{drop}_{i,j} = p^\text{drop}_{i,j} \cdot V_{i,j}$. Also, $V_{i,j}$ is updated as: $V_{i,j} = (1 - p^\text{drop}_{i,j}) \cdot V_{i,j}$.

**Estimating $S^\text{drop}_i$:** An estimate of $S^\text{drop}_i$ is application-specific and depends on the manner in which information about dropped requests is conveyed to the client, and how the
Figure 2.5. Multi-tier application model enhanced to handle concurrency limits. Since each tier has only one replica, we use only one subscript in our notation.

client responds to it. In our current model we make the simplifying assumption that upon detecting a failed request, the client reissues the request. This is captured by the transitions from $Q_{i,j}^{\text{drop}}$ ($1 \leq j \leq r_i$) back to $Q_0$ in Figure 2.5. Our approach for estimating $S_i^{\text{drop}}$ is to subject the application to an offline workload that causes the limit to be exceeded only at tier $T_i$ (this can be achieved by setting a low concurrency limit at that tier and sufficiently high limits at all the other tiers), and then record the response times of the requests that do not finish successfully. $S_i^{\text{drop}}$ is then estimated as the difference between the average response time of these unsuccessful requests and the sum of the service times at tiers $T_1$, $\cdots$, $T_{i-1}$, multiplied by their respective visit ratios.

In Figure 2.4(b) we plot the response times for Rubis as predicted by our enhanced model. We find that this enhancement enables us to capture the behavior of the Rubis application even when its concurrency limit is reached.

2.4.3 Handling Multiple Session Classes

Internet applications typically classify incoming sessions into multiple classes. To illustrate, an online brokerage Web site may define three classes and may map financial
transactions to the *Gold* class, customer requests such as balance inquiries to the *Silver* class, and casual browsing requests from non-customers to the *Bronze* class. Typically such classification helps the application sentry to preferentially admit requests from more important classes during overloads and drop requests from less important classes.

We can extend our baseline model to account for the presence of different session classes and to compute the response time of requests within each class. Consider an Internet application with $C$ session classes: $C_1, C_2, \ldots, C_C$. Assume that the sentry implements a classification algorithm to map each incoming session to one of these classes. We can use a straightforward extension of the MVA algorithm to deal with multiple session classes. This is presented in Algorithm 2. The notation used in this algorithm is a simple extension of that used in Algorithm 1 with an additional subscript $c$ for requests of class $c$. $N_c$ denotes the number of sessions of class $c$. We denote the total number of sessions by $N$ as before, so $N = \sum_{c=1}^{C} N_c$. This algorithm is based on the following extension of the result (2.1). Let $N - 1_c = (N_1, \ldots, N_{c-1}, N_c - 1, N_{c+1}, \ldots, N_C)$. For closed product form networks, 

$$\bar{A}_{c,m}(N) = \bar{L}_m(N - 1_c)$$ (2.2)

The notion of feasible population used in Algorithm 2 needs explanation. A feasible population with $n$ sessions total is a set of sessions such that the number of sessions within each class $c$ is between 0 and $N_c$, and the sum of the number of jobs in all classes is $n$.

We note that this algorithm requires the visit ratios, service times, and think time to be measured on a *per-class basis*. For handling load imbalances at replicated tiers, we propose to correct the per-class visit ratios by employing load imbalance factors determined using the heuristic described in Section 2.4.1. Our approach makes the simplifying assumption of identical load imbalance factors for the various classes at each tier. Finally, we refine our technique for dealing with concurrency limits presented in Section 2.4.2 to accommodate multiple classes. We estimate $S_i^{drop}$ exactly as in Section 2.4.2 because this parameter is
independent of the class of a request. The estimation of the drop probabilities, however, needs to be done on a per-class basis. We do this by enhancing the two step procedure described in Section 2.4.2. Let us denote by $V_{c,i,j}$ the visit ratio for class $c$ requests at $Q_{i,j}$ and by $V_{c,i,j}^{drop}$ the visit ratio for class $c$ requests at $Q_{i,j}^{drop}$.

**Step 1:** *Estimate throughput of the queuing network if there were no concurrency limits.*

Solve the queuing network using the multi-class MVA algorithm with $V_{c,i,j}^{drop} = 0, 1 \leq c \leq C$ (i.e., assuming that the queues have no concurrency limits). Let $\lambda = \sum_{c=1}^{C} \lambda_c$ denote the throughput computed by the MVA algorithm in this step.

**Step 2:** *Estimate $V_{c,i,j}^{drop}$.*

Treat $Q_{i,j}$ as an open, finite-buffer M/M/1/$K_i$ queue with arrival rate $\lambda V_{i,j}$ (using the $\lambda$ computed in Step 1). Let $p_{i,j}^{drop}$ denote the probability of buffer overflow in this M/M/1/$K_i$ queue [63]. Then $V_{c,i,j}^{drop}$ is estimated as: $V_{c,i,j}^{drop} = p_{i,j}^{drop} \cdot V_{c,i,j} \cdot \frac{\lambda_c}{\lambda}$.

Also, $V_{c,i,j}$ is updated as: $V_{c,i,j} = (1 - p_{i,j}^{drop}) \cdot V_{c,i,j} \cdot \frac{\lambda_c}{\lambda}$.

Given a $C$-tuple $(N_1, \cdots, N_C)$ of sessions belonging to the $C$ classes that are simultaneously serviced by the application, the algorithm can compute the average delays incurred at each queue and the end-to-end response time on a per-class basis. In Section 2.6.2 we discuss how this algorithm can be used to flexibly implement session policing policies in an Internet application.

### 2.4.4 Other Salient Features

Our closed queuing model has several desirable features.

**Simplicity:** For an $M$-tier application with $N$ concurrent sessions, the MVA algorithm has a time complexity of $O(MN)$. The algorithm is simple to implement, and as argued earlier, the model parameters are easy to measure online.

**Generality:** Our model can handle an application with arbitrary number of tiers. Further, when the scheduling discipline is processor sharing (PS), the MVA algorithm works without making any assumptions about the service time distributions of the customers [67]. This feature is highly desirable for two reasons: (1) it is representative of scheduling poli-
Algorithm 2: Mean-value analysis algorithm for an $M$-tier application with $C$ classes.
cies in commodity operating systems (e.g., Linux’s CPU time-sharing), and (2) it implies that our model is sufficiently general to handle workloads with an arbitrary service time requirements.\footnote{The applicability of the MVA algorithm is more restricted with some other scheduling disciplines. E.g., in the presence of a FIFO scheduling discipline at a queue, the service time at a queue needs to be exponentially distributed for the MVA algorithm to be applicable.}

While our model is able to capture a number of application idiosyncrasies, certain scenarios are not explicitly captured.

**Multiple resources:** We model each server occupied by a tier using a single queue. In reality, the server contains various resources such as the CPU, disk, memory, and the network interface. Our model currently does not capture the utilization of various server resources by a request at a tier. An enhancement to the model where various resources within a server are modeled as a network of queues is the subject of future work.

**Resources held simultaneously at multiple tiers:** Our model essentially captures the passage of a request through the tiers of an application as a juxtaposition of periods, during each of which the request utilizes the resources at exactly one tier. Although this is a reasonable assumption for a large class of Internet applications, it does not apply to certain Internet applications such as streaming video servers. A video server that is constructed as a pipeline of processing modules will have all of its modules or “tiers” active as it continuously processes and streams a video to a client. Our model does not apply to such applications.

### 2.5 Model Validation

In this section we present our experimental setup followed by our experimental validation of the model.
2.5.1 Experimental Setup

Applications: We use two open-source multi-tier applications in our experimental study. Rubis implements the core functionality of an eBay-like auction site: selling, browsing, and bidding. It implements three types of user sessions, has nine tables in the database and defines 26 interactions that can be accessed from the clients’ Web browsers. Rubbos is a bulletin-board application modeled after an online news forum like Slashdot. Users have two different levels of access: regular user and moderator. The main tables in the database are the users, stories, comments, and submissions tables. Rubbos provides 24 Web interactions. Both applications were developed by the DynaServer group at Rice University [38]. Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate the workloads and take the measurements needed by our experiments. We chose an average duration of 5 min for the sessions of both Rubis and Rubbos. For both applications, the think time was chosen from an exponential distribution with a mean of 1 sec.

We used 3-tier versions of these applications. We first provide an overview of the various software technologies used to build these applications.

- Java servlets: Early in the WWW’s history, the Common Gateway Interface (CGI) [22] was defined to allow Web servers to process user input and serve dynamic content. CGI programs can be developed in any script or programming language, but Perl is by far the most common language. CGI is supported by virtually all Web servers and many Perl modules are available as freeware or shareware to handle most tasks. But CGI is not without drawbacks. Performance and scalability are big problems since a new process is created for each request, quickly draining a busy server of resources. Sharing resources such as database connections between scripts or multiple calls to the same script is far from trivial, leading to repeated execution of expensive operations. The Java Servlet API was developed to leverage the advantages of the Java platform to solve the issues of CGI and proprietary APIs. It is a simple API
supported by virtually all Web servers and even load-balancing, fault-tolerant application servers. It solves the performance problem by executing all requests as threads in one process, or in a load-balanced system, in one process per server in the cluster. A servlet is a Java class and therefore needs to be executed in a Java VM by a service called a servlet container. Some Web servers, such as Sun’s Java Web Server (JWS) [57] are implemented in Java and have a built-in servlet container. Other Web servers, such as the Apache Web server [5], require a servlet container add-on module. The add-on intercepts all requests for servlets, executes them and returns the response through the Web server to the client. Tomcat [110] is the servlet container that is used in the official Reference Implementation for the Java Servlet technology.

- **Enterprise Java Beans:** Enterprise JavaBeans (EJB) [41] technology is the server-side component architecture for the Java 2 Platform, Enterprise Edition (J2EE) platform [55]. Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing component-based multitier enterprise applications. Features include Web services support and development tools (SDK). EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology. The JBoss Application Server [58] is a popular EJB container.

The front tier was based on Apache 2.0.48 Web server. We experimented with two implementations of the middle tier for Rubis—(i) based on Java servlets, and (ii) based on Sun’s J2EE Enterprise Java Beans (EJBs). The middle tier for Rubbos was based on Java servlets. We employed Tomcat 4.1.29 as the servlets container and JBoss 3.2.2 as the EJB container. We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [66] to implement the application sentry. ktcpvs is an open-source, Layer-7 request dispatcher implemented as a Linux kernel module. A round-robin load balancer implemented in ktcpvs was used for Apache. Request dispatching for the middle tier was performed by *mod_jk*, an Apache
module that implements a variant of round robin request distribution while taking into account session affinity. Finally, the database tier was based on the Mysql 4.0.18 database server [79].

**Hosting environment:** We conducted experiments with the applications hosted on two different kinds of machines. The first hosting environment consisted of IBM servers (model 6565-3BU) with 662 MHz processors and 256MB RAM connected by 100Mbps ethernet. The second setting, used for experiments reported in Section 2.6, had Dell servers with 2.8GHz processors and 512MB RAM interconnected using gigabit ethernet. This served to verify that our model was flexible enough to capture applications running on different types of machines. Finally, the workload generators were run on machines with Pentium-III processors with speeds 450MHz-1GHz and RAM sizes in the range 128-512MB. All the machines ran the Linux 2.4.20 kernel.

### 2.5.2 Performance Prediction

We conduct a set of experiments with the purpose of ascertaining the ability of our model to predict the response time of multi-tier applications. We experiment with (i)
two kinds of applications (Rubis and Rubbos), (ii) two different implementations of Rubis (based on Java servlets and EJBs), and (iii) different workloads for Rubis. Each of the three application tiers are assigned one server except in the experiments reported in Section 2.5.4. We vary the number of concurrent sessions seen by the application and measure the average response times of successfully finished requests over 30 sec intervals. Each experiment lasts 30 min. We compute the average response time and the 95% confidence intervals from these observations.

Our first experiment uses Rubis with a Java servlets-based middle tier. We use two different workloads—$W_1$: CPU-intensive on the Java servlets tier, and $W_2$: CPU-intensive on the database tier. These were created by modifying the Rubis client so that it generated an increased fraction of requests that stressed the desired tier. Earlier, in Figure 2.4(b) we had presented the average response time and 95% confidence intervals for sessions varying from 1 to 500 for the workload $W_1$. Also plotted were the average response times predicted by our basic model and our model enhanced to handle concurrency limits. Additionally, we present the observed and predicted residence times in Figure 2.6(a). Figure 2.6(b) shows that the CPU on the Java servlets tier becomes saturated beyond 100 sessions for this workload. As already explained in Section 2.4.2, the basic model fails to capture the response times for workloads higher than about 100 sessions due to an increase in the
fraction of requests that arrive at the Apache and servlets tiers only to be dropped because of the tiers operating at their concurrency limits. We find that our enhanced model is able to capture the effect of dropped requests at these high workloads and continues to predict response times well for the entire workload range.

Figure 2.7 plots the response times, the residence times, and the server CPU utilizations for servlets-based Rubis subjected to the workload $W^2$ with varying number of sessions. As shown in Figure 2.7(c), the CPU on the database server is the bottleneck resource for this workload. We find that our basic model captures response times well. The predicted response times are within the 95% confidence interval of the observed average response time for the entire workload range.

Next, we repeat the experiment described above with Rubis based on an EJB-based middle tier. Our results are presented in Figure 2.8. Again, our basic model captures the response time well until the concurrency limits at Apache and JBoss are reached. As the number of sessions grows beyond this point, increasingly large fractions of requests are dropped, the request throughput saturates, and the response time of requests that finish successfully shows a flat trend. Our enhancement to the model is again found to capture this effect well.

Finally, we repeat the above experiment with the Rubbos application. We use a Java servlets based middle tier for Rubbos and subject the application to the workload $W^1$ that is CPU-intensive on the servlets tier. Figure 2.9 presents the observed and predicted response times as well as the server CPU utilizations. We find that our enhanced model predicts response times well over the chosen workload range for Rubbos.

2.5.3 Query Caching at the Database

Recent versions of the Mysql server feature a query cache. When in use, the query cache stores the text of a SELECT query together with the corresponding result that was sent to the client. If the identical query is received later, the server retrieves the results
Figure 2.8. Rubis based on EJB: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.

Figure 2.9. Rubbos based on Java servlets: bottleneck at CPU of middle tier. The concurrency limits for the Apache Web server and the Java servlets container were set to be 150 and 75, respectively.
from the query cache rather than parsing and executing the query again. Query caching at the database has the effect of reducing the average service time at the database tier. We conduct an experiment to determine how well our model can capture the impact of query caching on response time. We subject Rubbos to a workload consisting of 50 simultaneous sessions. To simulate different degrees of query caching at Mysql, we use a feature of Mysql queries that allows the issuer of a query to specify that the database server not use its cache for servicing this query\(^6\). We modified the Rubbos servlets to make them request different fractions of the queries with this option. For each degree of caching we plot the average response time with 95% confidence intervals in Figure 2.10. As expected, the observed response time decreases steadily as the degree of query caching increases—the average response time is nearly 1400 msec without query caching and reduces to about 100 msec when all the queries are cached. In Figure 2.10 we also plot the average response time predicted by our model for different degrees of caching. We find that our model is able to capture the impact of the reduced query processing time with increasing degrees of caching on average response time. The predicted response times are found to be within the 95% confidence interval of the observed response times for the entire range of query caching.

2.5.4 Load Imbalance at Replicated Tiers

We configure Rubis using a replicated Java servlets tier—we assign three servers to this tier. We use the workload \(W_1\) with 100 simultaneous sessions. The user think times for a session are chosen using an exponential distribution whose mean is chosen uniformly at random from the set \(\{1\text{ sec, 5 sec}\}\). We choose short-lived sessions with a mean session duration of 1 minute. Our results are presented in Figure 2.11. Note that replication at the middle tier causes the response times to be significantly smaller than in the experi-

\(^6\)Specifically, replacing a SELECT with SELECT SQL_NO_CACHE ensures that Mysql does not cache this query.
ment depicted in Figure 2.6(a). Further, choosing sessions with two widely different think times ensures variability in the workload imposed by individual sessions and creates load imbalance at the middle tier.

Figure 2.11(a) plots the number of requests passing through each of the three servers in the servlets tier over 30 sec intervals during a 10 min run of this experiment; Figure 2.11(b) plots the average end-to-end response times for these requests. These figures show the imbalance in the load on the three replicas. Also, the most loaded server changes over time—choosing a short session duration causes the load imbalance to shift among replicas frequently. Figure 2.11(c) plots the average response times observed for requests passing through the three servers—instead of presenting response times corresponding to specific servers, we plot values for the least loaded, the second least loaded, and the most loaded server. Figure 2.11(c) also shows the response times predicted by the model assuming perfect load balancing at the middle tier. Under this assumption, we see a deviation between the predicted values and the observed response times. Next, we use the model enhancement described in Section 2.4.1 to capture load imbalance. For this workload the values for the load imbalance factors used by our enhancement were determined to be $\beta_1 = 0.25$, $\beta_2 = 0.32$, and $\beta_3 = 0.43$. We plot the response times predicted by the enhanced model at the extreme right in Figure 2.11(c). We observe that the use of these additional parameters

![Figure 2.10. Caching at the database tier of Rubbos.](image-url)
Figure 2.11. Load imbalance at the middle tier of Rubis. (a) and (b) present number of requests and response times classified on a per- replica basis; (c) presents response times classified according to most loaded, second most loaded, and most loaded replicas and overall average response times.

Improves our prediction of the response time. The predicted average response time (1350 msec) closely matched the observed value (1295 msec); with the assumption of perfect load balancing the model underestimated the average response time to be 950 msec.

2.5.5 Multiple Session Classes

We created two classes of Rubis sessions using the workloads W1 and W2 respectively. Recall that the requests in these classes have different service time requirements at different tiers—W1 is CPU-intensive on the Java servlets tier while W2 is CPU intensive on the database tier. We conduct two sets of experiments, each of which involves keeping the number of sessions of one class fixed at 10 and varying the number of sessions of the other class. We then compute the per-class average response time predicted by the multi-class version of our model (Section 2.4.3). We plot the observed and predicted response times for the two classes in Figure 2.12. While the predicted response times closely match the observed values for the first experiment, in the second experiment (Figure 2.12(b)), we observe that our model underestimates the response time for class 1 for 50 sessions—we attribute this to an inaccurate estimation of the service time of class 1 requests at the servlets tier at this load.

(a) Num. requests (per- replica)  (b) Resp. times (per- replica)  (c) Resp. times (based on load)
2.6 Applications of the Model

In this section we demonstrate some applications of our model for managing resources in a hosting platform. We also discuss some important issues related to the online use of our model.

2.6.1 Dynamic Capacity Provisioning and Bottleneck Identification

Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. The goal of dynamic provisioning is to dynamically allocate sufficient capacity to the tiers of an application so that its response time needs can be met even in the presence of the peak workload. Two key components of a dynamic provisioning technique are: (i) predicting the workload of an application, and (ii) determining the capacity needed to serve this predicted workload. The former problem has been addressed in papers such as [52]. The workload estimates made by such predictors can be used by our model to address the issue of how much capacity to provision. In Chapter 4, we extensively study the problem of dynamic capacity provisioning in hosting platforms. Here we conduct an exploratory experiment to evaluate the utility of our model for dynamic capacity provisioning.

Figure 2.12. Rubis serving sessions of two classes. Sessions of class 1 were generated using workload $W_1$ while those of class 2 were generated using workload $W_2$. 

(a) Ten class-1 sessions

(b) Ten class-2 sessions
Observe that the inputs to our model-based provisioning technique are the workload characteristics, number of sessions to be serviced simultaneously, and the response time target, and the desired output is a capacity assignment for the application. We start with an initial assignment of one server to each tier. We use the MVA algorithm to determine the resulting average response time as described in Sections 2.3 and 2.4. In case this is worse than the target, we use the MVA algorithm to determine, for each replicable tier, the response time resulting from the addition of one more server to it. We add a server to the tier that results in the most improvement in response time. We repeat this till we have an assignment for which the predicted response time is below the target—this assignment yields the capacity to be assigned to the application’s tiers. The above provisioning procedure has a time complexity of $O(kMN)$, where $k$ is the number of servers that the application is eventually assigned, $M$ is the number of tiers, and $N$ is the number of sessions. Since provisioning decisions are typically made over periods of tens of minutes or hours, this overhead is practically feasible.

We conduct an experiment to demonstrate the application of our model to dynamically provision Rubis configured using Java servlets at its middle tier. We assume an idealized workload predictor that can accurately forecast the workload for the near future. We generated a 1-hour long session arrival process based on a Web trace from the 1998 Soccer World Cup site [6]; this is shown in Figure 2.13(a). Sessions are generated according to this arrival process using workload $W_1$. This trace contained the number of arrivals per minute to this Web site during one day of the World Cup event. Based on this trace we created a smaller trace to drive our experiment. This trace was obtained by compressing the original 24-hr long trace to 1 hr—this was done by picking arrivals for every 24th minute and discarding the rest. This enabled us to capture the time-of-day effect as a “time-of-hour” effect.

Note that our current discussion assumes that it is always possible to meet the response time target by adding enough servers. Sometimes this may not be possible (e.g., due to the workload exceeding the entire available capacity, or a non-replicable tier becoming saturated) and we may have to employ admission control in addition to provisioning. This is discussed in Section 2.6.2.
Further, the request arrival rate was scaled down so it could be sustained by our hardware. Sessions for Rubis were then generated according to the arrival process described in this trace (shown in Figure 2.13(a)) using workload $W_1$.

We implemented a provisioning unit that invokes the model-based procedure described above every 10 min to determine the capacity required to handle the workload during the next interval. Our goal was to maintain an average response time of 1 sec for Rubis requests. Since our model requires the number of simultaneous sessions as input, the provisioning unit converted the peak rate during the next interval into an estimate of the number of simultaneous sessions for which to allocate capacity using Little’s Law \cite{63} as $N = \Lambda \cdot d$, where $\Lambda$ is the peak session arrival rate during the next interval as given by the predictor and $d$ is the average session duration. The provisioning unit ran on a separate server. It implemented scripts that remotely log on to the application sentry and the dispatchers for the affected tiers after every re-computation to enforce the newly computed allocations. The concurrency limits of the Apache Web server and the Tomcat servlets container were both set to 100. We present the working of our provisioning unit and the performance of Rubis in Figure 2.13(b). The provisioning unit is successful in changing the capacity of the servlets tier to match the workload—recall that workload $W_1$ is CPU intensive on this tier. The session arrival rate goes up from about 10 sess/min at $t = 20$ min to nearly 30 sess/min at $t = 40$ min. Correspondingly, the request arrival rate increases from about 1500 req/min to about 4200 req/min. The provisioning unit increases the number of Tomcat replicas from 2 to a maximum of 7 during the experiment. Further, at $t = 30$ min, the number of simultaneous sessions during the upcoming 10 min interval is predicted to be higher than the concurrency limit of the Apache tier. To prevent new sessions being dropped due to the connection limit being reached at Apache, a second Apache server is added to the application. Thus, our model-based provisioning is able to identify potential bottlenecks at different tiers (connections at Apache and CPU at Tomcat) and maintain response time targets by adding capacity appropriately. We note that the single-tier models described in
Section 2.2.2 will only be able to add capacity to one tier and will fail to capture such changing bottlenecks.

2.6.2 Session Policing and Class-based Differentiation

As mentioned in Chapter 1, an important component of an Internet application in our proposed hosting platform architecture is a sentry that polices incoming sessions to an application’s server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads. In an application supporting multiple classes of sessions, with possibly different response time requirements and revenue schemes for different classes, it is desirable to design a sentry that, during a flash crowd, can determine a subset of sessions admitting which would optimize a meaningful metric. An example of such a metric could be the overall expected revenue generated by the admitted sessions while meeting their response time targets (this constraint on response times will be assumed to hold in the rest of our discussion without being stated). Formally, given \( L \) session classes, \( C_1, \ldots, C_L \), with up to \( N_i \) sessions of class \( C_i \) and using overall revenue as the metric to be optimized, the goal of the sentry is to determine an \( L \)-tuple \((N_1^{\text{admit}}, \ldots, N_L^{\text{admit}})\) such that
∀n_i ≤ N_i (1 ≤ i ≤ L), \( \sum_i \text{rev}_i(N_i^{admit}) ≥ \sum_i \text{rev}_i(n_i) \)

where \( \text{rev}_i(n_i) \) denotes the revenue generated by \( n_i \) admitted sessions of \( C_i \).

Our multi-class model described in Section 2.4.3 provides a flexible procedure for realizing this. First observe that the inputs to this procedure are the workload characteristics of various classes and the capacity assigned to the application tiers, and the desired output is the number of sessions of each class to admit. In theory, we could use the multi-class MVA algorithm to determine the revenue yielded by every admissible \( L \)-tuple. Clearly this would be computationally prohibitive. Instead, we use a heuristic that considers the session classes in a non-increasing order of their revenue-per-session. For the class under consideration, it adds sessions till either all available sessions are exhausted, or adding another session would cause the response time of at least one class, as predicted by the model, to violate its target. The outcome of this procedure is an \( L \)-tuple of the number of sessions that can be used by the policer to make admission control decisions.

We now describe our experiments to demonstrate the working of the session policer for Rubis. We configured the servlets version of Rubis with 2 replicas of the servlets tier. Similar to Section 2.4.3, we chose \( W_1 \) and \( W_2 \) to construct two session classes \( C_1 \) and \( C_2 \) respectively. The response time targets for the two classes were chosen to be 1 sec and 2 sec; the revenue yielded by each admitted session was assumed to be $0.1 and $1 respectively. We assume session durations of exactly 10 min for illustrative purposes. We create the following flash crowd scenarios. We assume that 150 sessions of \( C_1 \) and 10 sessions of \( C_2 \) arrive at \( t = 0 \); 50 sessions each of \( C_1 \) and \( C_2 \) are assumed to arrive at \( t = 10 \) min. Figure 2.14(a) presents the working of our model-based policer. At \( t = 0 \), based on the procedure described above, the policer first admits all 10 sessions of the class with higher revenue-per-session, namely \( C_2 \); it then proceeds to admit as many sessions of \( C_1 \) as it can (90) while keeping the average response times under target. At \( t = 10 \) min, the policer first admits as many sessions of \( C_2 \) as it can (21); it then admits 5 sessions
of $C_1$—admitting more would, according to the model, cause the response time of $C_2$ to be violated. We find from Figure 2.14(a) that the response time requirements of both the classes are met during the experiment. We make two additional observations: (i) during $[0, 10]$ min, the response time of $C_2$ is well below its target of 2 sec—this is because there are only 10 sessions of this class, less than the capacity of the database tier for the desired response time target; since the 90 sessions of $C_1$ stress mainly the servlets tier (recall the nature of $W_1$ and $W_2$), they have minimal impact on the response time of $C_2$ sessions, which mainly exercise the database tier, and (ii) during $(10, 20]$ min, the response time of $C_1$ is well below its target of 1 sec—this is because the policer admits only 5 $C_1$ sessions; the servlets tier is lightly loaded since the $C_2$ sessions do not stress it, and therefore the $C_1$ sessions experience low response times.

Figure 2.14(b) demonstrates the impact of admitting more sessions on application response time. At $t = 0$, the policer admits excess $C_1$ sessions—it admits 140 and 10 sessions respectively. We find that sessions of $C_1$ experience degraded response times (in excess of 2 sec as opposed to the desired 1 sec). Similarly, at $t = 10$ min, it admits excess $C_2$ sessions—it admits 5 and 31 sessions respectively. Now sessions of $C_2$ experience response time violations. Observe that admitting excess sessions of one class does not cause a perceptible degradation in the performance of the other class because they exercise different tiers of the application.

2.7 Concluding Remarks

In this chapter we presented an analytical model for multi-tier Internet applications. Our model is based on using a network of queues to represent how the tiers in a multi-tier application cooperate to process requests. Our model (i) is general enough to capture Internet applications with an arbitrary number of heterogeneous tiers, (ii) is inherently designed to handle session-based workloads, and (iii) can account for application idiosyncrasies such as load imbalances within a replicated tier, caching effects, the presence of multiple classes
of sessions, and limits on the amount of concurrency at each tier. The model parameters are easy to measure and update. We validated the model using two open-source multi-tier applications running on a Linux-based server cluster. Our experiments demonstrated that our model faithfully captures the performance of these applications for a variety of workloads and configurations. We explored the utility of our model in managing resources for Internet applications under varying workloads and shifting bottlenecks. We investigate this in more depth in the subsequent chapters.

As part of future work, we plan to investigate the suitability our model for capturing more diverse workloads (e.g., IO-intensive at certain tiers) and to design enhancements to handle these. Another direction is to extend our model to handle other kinds of scheduling disciplines (such as proportional-share scheduling) at the application servers.

**Figure 2.14.** Maximizing revenue via differentiated session policing in Rubis. The application serves two classes of sessions.
CHAPTER 3
APPLICATION PLACEMENT

3.1 Introduction and Motivation

We study a mapping problem that arises in the design of Internet hosting platforms. Hosting platforms imply a business relationship between the platform provider and the application providers: the latter pay the former for the resources on the platform. In return, the platform provider gives some kind of guarantees of resource availability to applications. This implies that a platform should admit only applications for which it has sufficient resources. In this work, we take the number of applications that a platform is able to host (admit) to be an indicator of the revenue that it generates from the hosted applications. The number of applications that a platform admits is related to the application placement algorithm used by the platform. A platform’s application placement algorithm decides where on the cluster the different components of an application get placed. In this chapter we study properties of the application placement problem (APP) whose goal is to maximize the number of applications that can be hosted on a platform. Notice that the APP is trivial to solve in a dedicated hosting scenario—placing an application simply involves finding the appropriate number of available servers from the pool of free servers. So our discussion henceforth is concerned with the APP in a shared setting. We show that APP is NP-hard. Further, we show that even restricted versions of the APP may not admit polynomial-time approximation schemes. We design and analyze several approximation algorithms for the APP and present algorithms for its online version.

The rest of the chapter is organized as follows. Section 3.2 develops a formal setting for the APP and discusses related work. Section 3.3 establishes the hardness of approximat-
ing the APP. Section 3.4 presents polynomial-time approximation algorithms for various restrictions of the APP. Section 3.5 begins to study the online version of the APP. Section 3.6 discusses directions for further work.

3.2 The Application Placement Problem

3.2.1 Notation and Definitions

Consider a hosting platform built using a cluster of $n$ servers (also called nodes), $N_1, N_2, \ldots, N_n$, each having a given capacity $C_i$ (of available resources). Unless otherwise noted, nodes are homogeneous, in the sense of having the same initial capacities. The application placement problem (APP) appropriates portions of nodes’ capacities to applications. Let $A_1, \ldots, A_m$ be the applications to be placed on the cluster. For our purposes, an application can be viewed as a set of demands for node capacity. As described in Chapter 1, these demands come in discrete uniform units called capsules. We assume that these demands are determined using the analytical models described in Chapter 2 [111].

An example, a typical online bookstore application may consist of three capsules—a Web server responsible for HTTP processing, a middle-tier Java application server that implements the application logic, and a back-end database that stores catalogs and user orders. A capsule may be thought of as the smallest component of an application for the purposes of placement—all the processes, data etc., belonging to a capsule must be placed on the same node. Capsules provide a useful abstraction for logically partitioning an application into sub-components and for exerting control over the distribution of these components onto different nodes. If an application wants certain components to be placed together on the same node (e.g., because they communicate a lot), then it could bundle them as one capsule. Some applications may want their capsules to be placed on different nodes. An important reason for doing this is to improve the availability of the application in the face

---

1 An alternative approach for determining these demands is based on offline profiling. We describe this in Chapter 5.
of node failures—if a node hosting a capsule of the application fails, there would still be capsules on other nodes. An example of such an application is a replicated Web server. We refer to this requirement as the capsule placement restriction. In what follows, we look at the APP both with and without the capsule placement restriction.

In general, each capsule in an application would require guarantees on access to multiple resources. In this work, we consider just one resource, such as the CPU or the network bandwidth. We assume a simple model where a capsule specifies its resource requirement as a fraction of the resource capacity of a node in the cluster; i.e., we assume that the resource requirement of each capsule is less than the capacity of a node. A capsule $C$ can be placed on a node $N$ only if the sum of $C$’s resource requirement and those of the capsules already placed on $N$ does not exceed $N$’s resource capacity. We say that an application can be placed only if all of its capsules can be placed simultaneously. It is easy to see that there can be more than one way in which an application may be placed on a platform. We refer to the total number of applications that a placement algorithm could place as the size of the placement. A node, none of whose resources have been reserved, is referred to as an empty node.

We define two versions of the APP.

**Definition 1 The offline APP:** Given a cluster of $n$ empty nodes $N_1, \ldots, N_n$, and a set of $m$ applications $A_1, \ldots, A_m$, determine a placement of maximum size.

**Definition 2 The online APP:** Given a cluster of $n$ empty nodes $N_1, \ldots, N_n$, and a set of $m$ applications $A_1, \ldots, A_m$, determine a placement of maximum size while satisfying the following conditions.

1. The applications should be considered for placement in increasing order of their indices.

2. Once an application has been placed, it cannot be moved while the subsequent applications are being placed.
Lemma 1  The APP is NP-hard.

Proof: We reduce the well-known bin-packing problem [81] to the APP to show that it is NP-hard. We present the proof in Appendix A.

Definition 3  Polynomial-time approximation scheme (PTAS): A member of the set of algorithms \( A_\epsilon (\epsilon > 0) \) for a problem \( P \), where each \( A_\epsilon \) is a \((1+\epsilon)\)-approximation algorithm and the execution time is bounded by a polynomial in the length of the input. The execution time may depend on the choice of \( \epsilon \).

Definition 4  Approximation ratio: Approximation ratio of an algorithm \( A \), \( RA(A) \), is defined as:

\[
RA(A) = \max_I \frac{A(I)}{OPT(I)}
\]

where \( A(I) \) is the solution found by an approximation algorithm \( A \) and \( OPT(I) \) is the optimum solution for instance \( I \) of a minimization problem. For a maximization problem:

\[
RA = \max_I \frac{OPT(I)}{A(I)}
\]

So, clearly \( R(A) \geq 1 \), and the closer to 1, the better the approximation algorithm.

3.2.2 Related Work

Two generalizations of the classical knapsack problem are relevant to our discussion of the APP. These are the Multiple Knapsack Problem (MKP) and the Generalized Assignment Problem (GAP) [81]. In MKP, we are given a set of \( n \) items and \( m \) bins (knapsacks) such that each item \( i \) has a profit \( p(i) \) and a size \( s(i) \), and each bin \( j \) has a capacity \( c(j) \). The goal is to find a subset of items of maximum profit that has a feasible packing in the bins. MKP is a special case of GAP where the profit and the size of an item can vary based on the...
specific bin that it is assigned to. GAP is APX-hard\(^2\) and Shmoys and Tardos provide a 2-
approximation algorithm for it [101]. This was the best result known for MKP until Chekuri
and Khanna presented a polynomial-time PTAS for it [28]. It should be observed that the
offline APP is a generalization of MKP where an item may have multiple components that
need to be assigned to different bins (the profit associated with an item is 1). Further,
Chekuri and Khanna show that slight generalizations of MKP are APX-hard [28]. This
provides reason to suspect that the APP may also be APX-hard (and hence may not have a
PTAS).

Another closely related problem is a “multidimensional” version of the MKP where
each item has requirements along multiple dimensions, each of which must be satisfied to
successfully place it. The goal is to maximize the total profit yielded by the items that
could be placed. Moser et al. describe a heuristic for solving this problem [78]. However,
the authors evaluate this heuristic only via simulations and do not provide any analytical
results on its performance.

To the best of our knowledge, our work is the first to formulate and study the APP that
arises in hosting platforms.

3.3 Hardness of Approximating the APP

In this section, we demonstrate that even a restricted version of the APP may not admit
a PTAS. The capsule placement restriction is assumed to hold throughout this section.

Definition 5 Gap-preserving reduction: [39] Let \( \Pi \) and \( \Pi' \) be two maximization prob-
lems. A gap-preserving reduction from \( \Pi \) to \( \Pi' \) with parameters \((c, \rho), (c', \rho')\) is a polynomial-
time algorithm \( f \). For each instance \( I \) of \( \Pi \), algorithm \( f \) produces an instance \( I' = f(I) \) of

\(^2\)A problem is APX-hard if there exists some constant \( e > 0 \) such that it is NP-hard to approximate the
problem within a factor of \( (1 + e) \) (meaning that a PTAS is unlikely).
II'. The optima of $I$ and $I'$, say $OPT(I)$ and $OPT(I')$ respectively, satisfy the following property:

$$OPT(I) \geq c \implies OPT(I') \geq c', \quad (3.1)$$

$$OPT(I) < c/\rho \implies OPT(I') < c'/\rho'. \quad (3.2)$$

Here $c$ and $\rho$ are functions of $|I|$, the size of instance $|I|$, and $c'$, $\rho'$ are functions of $|I'|$. Also, $\rho(I), \rho'(I') \geq 1.$

Suppose we wish to prove the inapproximability of problem II'. Suppose further that we have a polynomial time reduction $\tau$ from $SAT$ to $\Pi$ that ensures, for every boolean formula $\phi$:

$$\phi \in SAT \implies OPT(\tau(\phi)) \geq c,$$

$$\phi \notin SAT \implies OPT(\tau(\phi)) < c/\rho.$$

Then composing this reduction with the reduction of Definition 5 gives a reduction $f \circ \tau$ from $SAT$ to $\Pi'$ that ensures:

$$\phi \in SAT \implies OPT(f(\tau(\phi))) \geq c',$$

$$\phi \notin SAT \implies OPT(f(\tau(\phi))) < c'/\rho'.$$

In other words, $f \circ \tau$ shows that achieving an approximation ratio $\rho'$ for $\Pi'$ is NP-hard. So a gap-preserving reduction can be used to exhibit the hardness of approximating a problem.
We now give a gap-preserving reduction from the Multi-dimensional 0-1 Knapsack Problem [25] to a restricted version of the APP. We begin with definition of the former problem (which is also known as the Packing Integer Problem [27]).

**Definition 6** Multi-Dimensional 0-1 Knapsack Problem (MDKP): For a fixed positive integer \( k \), the \( k \)-dimensional knapsack problem is the following:

\[
\text{Maximize } \sum_{i=1}^{n} c_i x_i
\]

Subject to

\[
\sum_{i=1}^{n} a_{ij} x_i \leq b_j, j = 1, \ldots, k
\]

where: \( n \) is a positive integer; each \( c_i \in \{0, 1\} \) and \( \max_i c_i = 1 \); the \( a_{ij} \) and \( b_i \) are non-negative real numbers; all \( x_i \in \{0, 1\} \). Define \( B = \min_i b_i \).

To see why the above maximization problem models a multi-dimensional knapsack problem, think of a \( k \)-dimensional knapsack with the capacity vector \((b_1, \ldots, b_k)\). That is, the knapsack has capacity \( b_1 \) along dimension 1, \( b_2 \) along dimension 2, etc. Think of \( n \) items \( I_1, \ldots, I_n \), each having a \( k \)-dimensional requirement vector. Let the requirement vector for item \( I_j \) be \((a_{j1}, \ldots, a_{jk})\). It is easy to see that the above maximization problem is equivalent to the problem of maximizing the number of \( k \)-dimensional items that can be packed in the \( k \)-dimensional knapsack such that for any \( d \), where \( 1 \leq d \leq k \), the sum of the requirements along dimension \( d \) of the packed items does not exceed the capacity of the knapsack along dimension \( d \).

**Hardness of approximating MDKP**: For fixed \( k \) there is a PTAS for MDKP [44]. Raghavan and Thompson present a randomized rounding technique for large \( k \) that yields integral solutions of value \( \Omega(OPT/d^{1/B}) \) [90]. Chekuri Khanna establish that MDKP is hard to approximate within a factor of \( \Omega(k^{\frac{1}{k+1}-\epsilon}) \) for every fixed \( B \), thus establishing that randomized rounding essentially gives the best possible approximation guarantees [27].
**Theorem 1** Given any $\epsilon > 0$, it is NP-hard to approximate to within $(1 + \epsilon)$ the offline placement problem that has the following restrictions: (1) all the capsules have a positive requirement and (2) there exists a constant $M$, such that $\forall i, j (1 \leq j \leq k, 1 \leq i \leq n), M \geq b_j/a_{ji}$.

**Proof:** We explain later in this proof why the two restrictions mentioned above arise. We begin by describing the reduction.

**The reduction:** Consider the following mapping from instances of $k$-MDKP to offline APP:

Suppose the input to $k$-MDKP is a knapsack with capacity vector $(b_1, \ldots, b_k)$. Also let there be $n$ items $I_1, \ldots, I_n$. Let the requirement vector for item $I_j$ be $(a_{j1}, \ldots, a_{jk})$. We create an instance of offline APP as follows. The cluster has $k$ nodes $N_1, \ldots, N_k$. There are $n$ applications $A_1, \ldots, A_n$, one for each item in the input to $k$-MDKP. Each of these applications has $k$ capsules. The $k$ capsules of application $A_i$ are denoted $c_{i1}^1, \ldots, c_{ik}^k$. Also, we refer to $c_{ij}^s$ as the $j^{th}$ capsule of application $A_i$. We now describe how to assign capacities to the nodes and requirements to the applications we have created. This part of the mapping proceeds in $k$ stages. In stage $s$, we determine the capacity of node $N_s$ and the requirements of the $s^{th}$ capsule of all the applications. Next, we describe how these stages proceed.

**Stage 1:** Assigning capacity to the first node $N_1$ is straightforward. We assign it a capacity $C(N_1) = b_1$. The first capsule of application $A_i$ is assigned a requirement $r_{i1} = a_{i1}$.

**Stage $s$ ($1 < s \leq k$):** The assignments done by stage $s$ depend on those done by stage $s - 1$. We first determine the smallest of the requirements along dimension $s$ of the items in the input to $k$-MDKP, that is, $r_{min}^s = \min_{i=1}^n (a_{is})$. Next we determine the scaling factor for stage $s$, $SF_s$ as follows:

$$SF_s = \lceil C(N_{s-1})/r_{min}^s \rceil + 1. \quad (3.3)$$
Recall that we assume that $\forall s, r^s_{\text{min}} > 0$. Now we are ready to do the assignments for stage $s$. Node $N_s$ is assigned a capacity $C(N_s) = b_i \times SF_s$. The $s^{th}$ capsule of application $A_i$ is assigned a requirement $r^s_i = a_{is} \times SF_s$.

This concludes our mapping. Let us now take a simple example to better explain how this mapping works. Consider the instance of input $T$ to MDKP shown on the left of Figure 3.1. Here we have $k = 3, n = 4$. We create 3 nodes $N_1, N_2$ and $N_3$. We create 4 applications $A_1, A_2, A_3$ and $A_4$, each with 3 capsules. Let us now consider how the 3 stages in our mapping proceed.

**Stage 1:** We assign a capacity of 10 to $N_1$ and requirements of 1 each to the first capsules of all four applications.

**Stage 2:** The scaling factor for this stage $SF_2$ is 11. So we assign a capacity of 110 to $N_2$ and requirements of 11 each to the second capsules of the four applications.

**Stage 3:** The scaling factor for this stage, $SF_3$ is $\lceil 110/s \rceil + 1 = 56$. So we assign $N_3$ a capacity of 560. The third capsules of the four applications are assigned requirements of 280, 112, 280 and 392 respectively.

**Correctness of the reduction:** We show that the mapping described above is a reduction.
(⇒) Assume there is a packing $P$ of size $m \leq n$. Denote the $n$ items in the input to $k$-MDKP as $I_1, \ldots, I_n$. Without loss of generality, assume that the $m$ items in $P$ are $I_1, \ldots, I_m$. Therefore we have,

$$\sum_{i=1}^{m} a_{ij} \leq b_j, \ j = 1, \ldots, k. \quad (3.4)$$

Consider this way of placing the applications that the mapping constructs on the nodes $N_1, \ldots, N_k$. If item $I_i \in P$, place application $A_i$ as follows: for all $j \in \{1, \ldots, k\}$, place capsule $c^j_i$ on node $N_j$. We claim that we will be able to place all $m$ applications corresponding to the $m$ items in $P$. To see why consider any node $N_i$, for $1 \leq i \leq k$. The capacity assigned to $N_i$ is $SF_i$ times the capacity along dimension $i$ of the $k$-dimensional knapsack in the input to $k$-MDKP, where $SF_i \geq 1$. The requirements assigned to the $i^{th}$ capsules of all the applications are also obtained by scaling by the same factor $SF_i$ the sizes along the $i^{th}$ dimension of the items. Multiplying both sides of (3.4) by $SF_i$ we get,

$$SF_i \times \sum_{i=1}^{m} a_{ij} \leq SF_i \times b_j, \ j = 1, \ldots, k.$$

Observe that the term on the right is the capacity assigned to $N_i$. The term on the left is the sum of the requirements of the $i^{th}$ capsules of the applications corresponding to the items in $P$. This shows that node $N_i$ can accommodate the $i^{th}$ capsules of the applications corresponding to the $m$ items in $P$. This implies that there is a placement of size $m$.

(⇐) Assume that there is a placement $L$ of size $m \leq n$. Let the $n$ applications be denoted $A_1, \ldots, A_n$. Without loss of generality, let the $m$ applications in $L$ be $A_1, \ldots, A_m$. Also denote the set of the $s^{th}$ capsules of the placed applications by $Cap_s$, for $1 \leq s \leq k$.

We make the following key observations:

- For any application to be successfully placed, its $i^{th}$ capsule must be placed on node $N_i$. Due to the scaling by the factor computed in Equation (3.3), the requirements
assigned to the $s^{th}$ capsules of the applications, for $s > 1$, are strictly greater than the capacities of nodes $N_1, \ldots, N_{s-1}$. Consider the $k^{th}$ capsules of the applications first. The only node these can be placed on is $N_k$. Since no two capsules of an application may be placed on the same node, this implies that the $(k-1)^{th}$ capsules of the applications may be placed only on $N_{k-1}$. Proceeding in this manner, we find that the claim holds for all capsules.

- Since for all $s$, where $1 \leq s \leq k$, the node capacities and the requirements of the $s^{th}$ capsules are scaled by the same multiplicative factor, the fact that the $m$ capsules in $Cap_s$ could be placed on $N_s$ implies that the $m$ items $I_1, \ldots, I_m$ can be packed in the knapsack in the $s^{th}$ dimension.

Combining these two observations, we find that a packing of size $m$ must exist.

**Time and space complexity of the reduction:** This reduction works in time polynomial in the size of the input. To wit, it proceeds in $k$ stages. Each stage involves computing a scaling factor (which requires performing a division) and multiplying $n + 1$ numbers (the capacity of the knapsack and the requirements of the $n$ items along the relevant dimension).

Let us consider the size of the input to the offline placement problem produced by the reduction. Due to the scaling of capacities and requirements described in the reduction, the magnitudes of the inputs increase by a (multiplicative) factor of $O(M^j)$ for node $N_j$ and the $j^{th}$ capsules. If we assume binary representation, this implies that the input size increases by a factor of $O(M^{j/2})$, for $1 < j \leq k$. Overall, the input size increases by a factor of $O(M^k)$. For the mapping to be a reduction, we need this to be a constant. Therefore, our reduction works only when we impose the following restrictions on the offline APP: (1) $k$ and $M$ are constants, and (2) all the capsule requirements are positive.

**Gap-preserving property of the reduction:** The reduction presented is gap-preserving because the size of the optimal solution to the offline placement problem is *exactly equal* to the size of the optimal solution to MDKP. More formally, in terms of the terminology
used in Definition 5, we can set $c = c' = \rho = \rho' = 1$. Putting these values in Equations (3.1) and (3.2), we find that the following conditions hold:

\[ \text{OPT(MDKP)} \geq 1 \implies \text{OPT(offline APP)} \geq 1 \]

\[ \text{OPT(MDKP)} < 1 \implies \text{OPT(offline APP)} < 1 \]

This proves that the reduction is gap-preserving. Together, these results prove that the restricted version of the offline APP described in Theorem 1 does not admit a PTAS unless $P = NP$.

3.4 Offline Algorithms for APP

In this section we present and analyze offline approximation algorithms for several variants of the placement problem. Except in Section 3.4.2.2, we assume that the cluster is homogeneous, in the sense specified earlier.

3.4.1 Placement without the Capsule Placement Restriction

We first consider the placement problem without the capsule placement restriction. We present first-fit based placement algorithms for two variants of the placement problem: (i) when any capsule may be placed on any node; (ii) when the capsules of an application must be placed on the same node. We show that each of the resulting algorithms has an approximation ratio of 2.

3.4.1.1 First-fit Based Approximation Algorithm

We consider the most general form of the APP—one in which there is no restriction on the placement of a capsule; i.e., a capsule may be placed on any node that has enough capacity. We show that a placement algorithm based on first-fit gives an approximation ratio approaching 2 as the size of the cluster grows.

The approximation algorithm works as follows. Say that we are given $n$ nodes $N_1, \ldots, N_n$ and $m$ applications $A_1, \ldots, A_m$ with requirements $R_1, \ldots, R_m$. The requirement of an application is the sum of the requirements of its capsules. Assume that the nodes have
unit capacities. The algorithm first orders the applications in nondecreasing order of their requirements. Denote the ordered applications by \( a_1, \ldots, a_m \) and their requirements by \( r_1, \ldots, r_m \). The algorithm considers the applications in this order. An application is placed on the “first” set of nodes where it can be accommodated, i.e., the nodes with the smallest indices that have sufficient resources for all its capsules. The algorithm terminates once it has considered all applications, or as soon as it finds an application that cannot be placed, whichever occurs first. We call this algorithm FF\_MULTIPLE\_RES.

**Lemma 2** FF\_MULTIPLE\_RES has an approximation ratio that approaches 2 as the number of nodes in the cluster grows.

**Proof:** Denote by \( k_{FF} \) the number of applications that FF\_MULTIPLE\_RES could place on \( n \) nodes, completely (meaning that all the capsules of the application could be placed) or partially (meaning that at least one capsule of the application could not be placed). Denote by \( k_{OPT} \) the number of applications that an optimal algorithm could place on the same set of nodes.

If FF\_MULTIPLE\_RES places all the applications on the given set of nodes, then it has matched the optimal algorithm and we are done.

Consider the case when there is at least one application that FF\_MULTIPLE\_RES could not place. Since all capsules have requirements less than the capacity of a node, this implies that there is no empty node after the placement. The set of applications placed by FF\_MULTIPLE\_RES is \( \{ a_1, \ldots, a_{k_{FF}} \} \). Observe that except for the last of these applications, namely \( a_{k_{FF}} \), the algorithm places all the applications completely. The application \( a_{k_{FF}} \) may or may not have been completely placed. In either case, the following key observation would hold: if FF\_MULTIPLE\_RES could not place all the applications, then there can be at most one node that is more than half empty. To see why, assume that there are two nodes \( N_i \) and \( N_j \) that are more than half empty, with \( i < j \). Since the capsules placed on \( N_j \) can be accommodated in \( N_i \), the assumed situation can never arise in a placement found by FF\_MULTIPLE\_RES.
As a result we have

\[ R_1 + \ldots + R_{k_{FF}} + R'_{k_{FF}} \geq \frac{n}{2}, \]

where \( R'_{k_{FF}} \) is the sum of the requirements of the capsules of application \( a_{k_{FF}} \) that could be placed on the cluster. Since \( R'_{k_{FF}} \leq R_{k_{FF}} \), this implies that

\[ R_1 + \ldots + R_{k_{FF}} \geq \frac{n}{2} \]

The best that an optimal algorithm can do is to use up all the capacity on the nodes, so we have

\[ R_1 + \ldots + R_{k_{FF}} + \ldots + R_{k_{OPT}} \leq n \]

Since \( R_{k_{OPT}} \geq \ldots \geq R_{k_{FF}} \geq \ldots \geq R_1 \), the set \( \{c_1, \ldots, c_{FF}\} \) would have at least as many applications as the set \( \{a_{k_{FF}}, \ldots, a_{k_{OPT}}\} \). Discounting \( a_{k_{FF}} \), which may not have been completely placed, we find that FF_MULTIPLE_RES guarantees to place one less than half as many applications as an optimal algorithm can place. As the number of nodes grows, the performance ratio of FF_MULTIPLE_RES thus tends to 2.

3.4.1.2 Placement of applications whose capsules must be co-located

We consider a restricted version of APP in which all capsules of an application must be placed on the same node. This is equivalent to each application’s having exactly one capsule whose resource requirement is equal to the sum of the requirements of all the capsules of the application. We provide a polynomial-time algorithm for this restriction of offline APP, whose placements are within a factor 2 of optimal.

Motivating example: Some highly parallel scientific applications involve a significant amount of communication among their constituent processes. The communication overheads due to placing these processes on separate nodes connected via a network may be
prohibitive. Such applications may desire that all their processes be placed on the same node.

The approximation algorithm works as follows. Say that we are given $n$ nodes $N_1, \ldots, N_n$ and $m$ single-capsule applications $C_1, \ldots, C_m$ with requirements $R_1, \ldots, R_m$. Assume that the nodes have unit capacities. The algorithm first sorts the applications in nondecreasing order of their requirements. Denote the sorted applications by $c_1, \ldots, c_m$ and their requirements by $r_1, \ldots, r_m$. The algorithm considers the applications in this order. An application is placed on the “first” node where it can be accommodated, i.e., the node with the smallest index that has sufficient resources for it. The algorithm terminates once it has considered all the applications or it finds an application that cannot be placed, whichever occurs earlier. We call this algorithm FF\_SINGLE. The following result yields to a proof similar to that of Lemma 2.

**Lemma 3** \( FF\_SINGLE \) has an approximation ratio of 2.

### 3.4.2 Placement with the Capsule Placement Restriction

In this section, we consider the APP with the capsule placement restriction described in Section 3.2.1. We first consider a special case—identical applications. Then we remove this restriction and consider arbitrary applications.

**Motivating example:** A replicated Web server is an example of an application where one might like to have the capsule placement restriction. This restriction forces us to distribute the application over multiple nodes, thereby improving its ability to tolerate node failures.

#### 3.4.2.1 Placement of Identical Applications

Two applications are identical if their sets of capsules are identical. We now present a placement algorithm based on striping applications across the nodes in the cluster and determine the algorithm’s approximation ratio.

**Striping-based placement:** Assume that the applications have $k$ capsules each, with requirements $r_1, \ldots, r_k$, where $r_1 \leq \ldots \leq r_k$. The algorithm works as follows. Denote the
nodes as $N_1, \ldots, N_m$. Divide them into sets of size $k$ each. Letting $t = \lfloor m/k \rfloor \geq 1$, there are $(t + 1)$ such sets, $S_1, \ldots, S_{t+1}$, where $S_{t+1}$ may be empty (if $k$ divides $m$). The preceding inequality holds because $m \geq k$. The algorithm considers the sets in turn and stripes as many unplaced applications on them as it can. The $i^{th}$ iteration of this striping-based algorithm involves trying to place the capsules on nodes $N_i \mod (m+1), \ldots, N_{(i+k) \mod (m+1)}$. The set of nodes under consideration at any moment in this process is referred to as the current set of $k$ nodes.

We illustrate the notion of striping using an example. In Figure 3.2, we have three nodes and a number of identical 3-capsule applications to be placed on them. Striping places the first capsule of $A_1$ on $N_1$, the second on $N_2$, and the third on $N_3$. For the next application $A_2$, it places the first capsule on $N_2$, second on $N_3$, and third on $N_1$.

When the current set of $k$ nodes gets exhausted and there are more applications to place, the algorithm takes the next set of $k$ nodes and continues. The algorithm terminates when the nodes in $S_t$ are exhausted, or all applications have been placed, whichever occurs first. Note that none of the nodes in the (possibly empty) set $S_{t+1}$ are used for placing the applications.

**Lemma 4** The striping-based placement algorithm yields an approximation ratio of $\left(\frac{t + 1}{t}\right)$ for identical applications, where $t = \lfloor m/k \rfloor$. 

![Figure 3.2. An example of striping-based placement.](image)
Proof: It is easy to observe that the striping-based placement algorithm places an optimal number of identical applications on a homogeneous cluster of size $k$ (due to symmetry). Since the striping-based algorithm places applications on the sets $S_1, \ldots, S_t$ and lets $S_{t+1}$ go unused, and since the nodes are homogeneous and the applications are identical, its approximation ratio is strictly less than $\left(\frac{t + 1}{t}\right)$.

3.4.2.2 Placement of Arbitrary Applications

We have thus far considered restricted versions of the offline APP and have presented heuristics that have approximation ratios of 2 or better. In this section we turn our attention to the general offline APP. We let the nodes in the cluster be heterogeneous. We find that it is much harder to compute approximately optimal solutions for this problem than for the restricted cases. We first present a heuristic that works differently from the first-fit based heuristics we have considered so far. We obtain an approximation ratio of $k$ for this heuristic, where $k$ is the maximum number of capsules in any application.

Our heuristic works as follows. It associates with each application a weight which is equal to the requirement of the largest capsule in the application. The heuristic considers the applications in nondecreasing order of their weights. We use a bipartite graph to model the problem of placing an application on the cluster. In this graph, we have one vertex for each capsule in the application and for each node in the cluster. Edges are added between
a capsule and a node if the node has sufficient capacity to host the capsule. In this case, we say that the node is feasible for the capsule. An example is shown in Figure 3.3. In Lemma 5 we show that an application can be placed on the cluster if, and only if, there is a matching of size equal to the number of capsules in the application. We therefore use the maximum matching problem on this bipartite graph [33] to derive a placement. If the matching has size equal to the number of capsules, then we place the capsules of the application on the nodes that the maximum matching connects them to. Otherwise, we say that the application cannot be placed, and the heuristic terminates. We refer to this heuristic as Max-First.

**Lemma 5** An application with $k$ capsules can be placed on a cluster if, and only if, there is a matching of size $k$ in the bipartite graph modeling its placement on the cluster.

**Proof:** We prove each direction in turn.

($\implies$) Consider a matching of size $k$ in the bipartite graph. It must have an edge connecting each capsule to a node. Further, no two capsules could be connected to the same node (since this is a matching). Since edges denote feasibility, this is clearly a valid placement.

($\impliedby$) Suppose there is no matching of size $k$ in the bipartite graph. Then there must be at least one capsule that cannot be assigned to a node independently of the other capsules. In other words, there must be at least one capsule that would need to share a node with some other capsule(s). Therefore this application cannot be placed without violating the capsule placement restriction.

This concludes the proof.

**Lemma 6** The placement heuristic Max-First described above has an approximation ratio of $k$, where $k$ is the maximum number of capsules in an application.
**Proof:** Let \( A \) represent the set of all the applications, so \(|A| = m\). Denote by \( n \) the number of nodes in the cluster and the nodes themselves by \( N_1, \ldots, N_n \). Let us denote by \( H \) the set of applications that Max-First places. Let \( O \) denote the set of applications placed by any optimal placement algorithm. Clearly, \(|H| \leq |O| \leq m\). Represent by \( I = H \cap O \) the set of applications that both \( H \) and \( O \) place. Further, denote by \( R \) the set of applications that neither \( H \) nor \( O \) places.

The basic idea behind this proof is as follows. We focus in turn on the applications that only Max-First and the optimal algorithm place (that is, applications in \((H - I)\) and \((O - I)\)) and compare the sizes of these sets. A relation between the sizes of these sets immediately yields a relation between the sizes of the sets \( H \) and \( O \). (Observe that \((H - I)\) and \((O - I)\) may both be empty, in which case we have the claimed ratio trivially.)

Consider the placement given by Max-First. Remove from this all the applications in \( I \), and deduct from the nodes the resources reserved for the capsules of these applications. Denote the resulting nodes by \( N_{H}^{I}, \ldots, N_{n}^{H-I} \). Do the same for the placement given by the optimal algorithm, and denote the resulting nodes by \( N_{O}^{I}, \ldots, N_{n}^{O-I} \). To understand the relation between the applications placed on these node-sets by Max-First and the optimal algorithm, suppose Max-First places \( y \) applications from the set \((H - I)\) on the nodes \( N_{1}^{H-I}, \ldots, N_{n}^{H-I} \). Let us denote the applications in \((A - I)\) by \( B_1, \ldots, B_y, \ldots, B_{|A-I|} \), where the applications are arranged in nondecreasing order of the size of their largest capsule; that is, \( l(B_1) \leq \ldots \leq l(B_y) \leq \ldots \leq l(B_{|A-I|}) \), where \( l(x) \) is the requirement of the largest capsule in application \( x \). From the definition of Max-First, the \( y \) applications that it places are \( B_1, \ldots, B_y \). Also, the applications that the optimal algorithm places on the set of nodes \( N_{1}^{O-I}, \ldots, N_{n}^{O-I} \) must be from the set \( B_{y+1}, \ldots, B_{|A-I|} \). We make the following observation about the applications in the set \( B_{y+1}, \ldots, B_{|A-I|} \): for each of these applications, the requirement of the largest capsule is at least \( l(B_y) \). Based on this, we infer the following: Max-First will exhibit the worst approximation ratio when all the applications in \((H - I)\) have \( k \) capsules, each with requirement \( l(B_y) \), and all applications in \((O - I)\)
have \((k - 1)\) capsules with requirement 0, and one capsule with requirement \(l(B_y)\). Since the total capacities remaining on the node-sets \(N_{1}^{H-I}, \ldots , N_{n}^{H-I}\) and \(N_{1}^{O-I}, \ldots , N_{n}^{O-I}\) are equal, this implies that in the worst case, the set \(O - I\) would contain \(k\) times as many applications as \(H - I\). Based on the above, we can prove an approximation ratio of \(k\) for Max-First as follows:

\[
|O| = |O - I| + |I| \leq k \cdot |H - I| + |I| \leq k \cdot (|H - I| + |I|) = k \cdot |H|
\]

This concludes our proof.

### 3.5 The On-line APP

In the online version of the APP, the applications arrive one by one. We require the following from any online placement algorithm—*the algorithm must place a newly arriving application on the platform if it can find a placement for it without moving any already placed capsule*. This captures the placement algorithm’s lack of knowledge of the requirements of the applications arriving in the future. We assume a heterogeneous cluster throughout this section.

#### 3.5.1 Online Placement Algorithms

Online placement algorithms consider applications for placement one by one, as they arrive. Consider the situation an online placement algorithm is faced with when a new application arrives. We model this using a graph, in which we have one vertex for each capsule in the application and for each node in the cluster. Edges are added between a capsule and a node if the node has sufficient resources for hosting the capsule. We say that the node is *feasible* for the capsule. This gives us a bipartite graph that we call the *feasibility graph* of the new application. An example of a feasibility graph is shown in Figure 3.3. As described in Section 3.4.2.2, a maximum matching on this graph can be used to find a placement for the application if one exists.
Let us denote by \( \mathcal{A} \) the class of greedy online placement algorithms that work as follows. Any such algorithm considers the capsules of the newly arrived application in non-decreasing order of their degrees in the feasibility graph of the application. If there are no feasible nodes for a capsule, the algorithm terminates. Otherwise, the capsule is placed on one of the nodes feasible for it. After this, all edges connecting any unplaced capsules to this node are removed from the graph. This is repeated until all capsules have been placed or the algorithm cannot find any feasible nodes for some capsule.

We define two members of \( \mathcal{A} \) below.

**Definition 7** Best-fit based Placement (BF): *When more than one node can accommodate the new capsule, BF chooses the node with the least remaining capacity.*

**Definition 8** Worst-fit based Placement (WF): *When more than one node can accommodate the new capsule, WF chooses the node with the most remaining capacity.*

We can show the following regarding the approximation ratios of BF and WF, denoted \( R_{BF} \) and \( R_{WF} \) respectively.

**Lemma 7** BF can perform arbitrarily worse than optimal.

**Proof:** Let \( m \) be the total number of applications and \( n \) the number of nodes and let \( m > n \).

Let all the nodes have a capacity of 1. Suppose that \( n \) single-capsule applications arrive first, each capsule with a requirement \( 1/n \). BF puts them all on the first node. Next, \((m - n)\) \( n \)-capsule applications arrive with each capsule having non-zero requirement. Since the first node has no capacity left, BF will not be able to place any of these. WF would have worked as follows on this input. Each of the first \( n \) single-capsule applications
would have been placed on a separate node, resulting in each of the $n$ nodes having a remaining capacity $(1 - 1/n)$, available for the $n$-capsule applications. Therefore,

$$\exists \text{ input s.t. } \frac{BF}{WF} \geq \frac{m}{n}$$

Also, since $WF$ is optimal for this input, we have

$$R_{BF} \geq \frac{m}{n}$$

Since $m$ can be arbitrarily larger than $n$ (by making the $n$-capsule applications have capsules with requirements tending to 0), $R_{BF}$ cannot be bounded from above.

**Lemma 8** $R_{WF} \geq (2 - 1/n)$ for an $n$-node cluster.

**Proof:** Say that the cluster has $n$ nodes, each with unit capacity. Consider the following sequence of application arrivals. Suppose that $n$ single-capsule applications arrive first, each capsule with a requirement $\epsilon$ that approaches 0. $WF$ places each of these applications on a separate node, resulting in each of the $n$ nodes having a remaining capacity $(1 - \epsilon)$. Next, $n$ single-capsule applications arrive, each capsule with a requirement of 1. Since no node is fully vacant, none of these applications can be placed. Here is how $BF$ would work on this input. The $n$ single-capsule nodes would be placed on the first node. Then, $(n - 1)$ of the subsequently arriving applications would be placed on the $(n - 1)$ fully vacant nodes, and the last application would be turned away. Therefore we have,

$$\exists \text{ input s.t. } \frac{WF}{BF} \geq (2 - \frac{1}{n})$$

Since $BF$ is optimal on this input, this gives us,

$$R_{WF} \geq (2 - \frac{1}{n})$$

This gives the claimed lower bound as $n$ grows without bound.
3.5.2 Online Placement with Variable Preference for Nodes

In some scenarios, it may be useful to be able to honor any preference a capsule may have for one feasible node over another. In this section, we describe how online placement can take such preferences into account. We model such a scenario by enhancing the bipartite graph representing the placement of an application on the cluster by allowing the edges in the graph to have positive weights. An example of such a graph is shown in Figure 3.4. In this graph lower weights mean higher preference. A valid placement corresponds to a placement of size equal to the number of capsules $k$.

The online placement problem therefore is to find the maximum matching of minimum weight in this weighted graph. We show that this can be found by reducing the placement problem to the Minimum-weight Perfect Matching Problem. We will first define this problem and then present the reduction.

**Definition 9 Minimum-weight Perfect Matching Problem:** A perfect matching in a graph $G$ is a subset of edges such that each node in $G$ is met by exactly one edge in the subset. Given a real weight $c_e$ for each edge $e$ of $G$, the minimum weight perfect matching problem is to find a perfect matching $M$ of minimum weight $\sum_{e \in M} c_e$.

Our reduction works as follows. Assume that all the weights in the original bipartite graph are in the range $(0, 1)$ and that they sum to 1. This can be achieved by normalizing
all the weights by the sum of the weights. If an edge $e_i$ had weight $w_i$, its new weight would be $\frac{w_i}{\sum_{e \in E} w_e}$. Denote the number of capsules by $m$ and the number of nodes by $n$, $m \leq n$. Construct $n - m$ capsules and add edges with weight 1 each between them and all the nodes. We call these the dummy capsules.

Figure 3.4 presents an example of this reduction. On the left is a bipartite graph showing the normalized preferences of the capsules $C_1, C_2, C_3$ for their feasible nodes. We add another capsule $C_4$ shown on the right to make the number of capsules equal to the number of nodes. Also shown on the right are the new edges connecting $C_4$ to all the nodes. each of these edges has a weight of 1. The weights of the remaining edges do not change, so they have been omitted from the graph on the right.

**Lemma 9** In the weighted bipartite graph $G$ corresponding to an application with $m$ capsules and a cluster with $n \geq m$ nodes, a matching of size $m$ and cost $c$ exists if, and only if, a perfect matching of cost $(c + n - m)$ exists in the graph $G'$ produced by reduction described above.

**Proof:** ($\Rightarrow$) Suppose that there is a matching $M$ of size $m$ and cost $c$ in $G$. We construct a perfect matching $M'$ in $G'$ as follows. $M'$ has all the edges in $M$. Next we add to $M'$ edges that have the dummy capsules incident on them. For this, we consider the dummy capsules one by one (in any order). For each such capsule, we add to $M'$ an edge connecting it to a node that is not yet on any of the edges in $M'$. Since there is a matching of size $m$ in $G$, and since each dummy capsule is connected to all $n$ nodes, $M'$ will have a matching of size $n$ (that is, a perfect matching). Further, since each edge with a dummy capsule as its end point has a weight of 1, and there are $(n - m)$ such edges, the cost of $M'$ is $c + (n - m) \times 1 = c + n - m$.

($\Leftarrow$) Suppose there is a perfect matching $M'$ of cost $(c + n - m)$ in $G'$. Consider the set $M$ that contains all the edges in $M'$ that do not have a dummy capsule as one of their end points. There would be $m$ such edges. Since $M'$ was a perfect matching, $M$ would be a matching in $G$. Moreover, the cost of $M$ would be the cost of $M'$ minus the sum of the
costs of the \((n - m)\) edges that we removed from \(M'\) to get \(M\). Therefore, the cost of \(M\) is 
\[ c + n - m - (n - m) \times 1 = c. \]

This concludes the proof.

Edmonds presents a polynomial-time algorithm (called the blossom algorithm) for computing minimum-weight perfect matchings [40]. A survey of implementations of the blossom algorithm appears in a paper by Cook and Rohe [31]. The reduction described above, combined with Lemma 9, can be used to find the desired placement. If we do not find a perfect matching in the graph \(G'\), we conclude that there is no placement for the application. Otherwise, the perfect matching minus the edges incident on the newly introduced capsules gives us the desired placement.

3.6 Concluding Remarks

3.6.1 Summary of Results

In this work, we considered the offline and online versions of APP, the problem of placing distributed applications on a cluster of servers. This problem was found to be NP-hard. Barring the results for some special cases, we currently have algorithms with approximation ratio of \(k\) and 2 for the APP with and without the capsule placement restriction (\(k\) denotes the number of capsules in an application). We used a gap preserving reduction from the Multi-dimensional Knapsack Problem to show that even a restricted version of the offline placement problem may not have a PTAS. A heuristic that considered applications in non-decreasing order of their largest component was found to provide an approximation ratio of \(k\), where \(k\) was the maximum number of capsules in any application. We also considered restricted versions of the offline APP in a homogeneous cluster. We found that heuristics based on first-fit or striping could provide an approximation ratio of 2 or better.

For the online placement problem, we provided algorithms based on solving a maximum matching problem on a bipartite graph modeling the placement of a new application on a heterogeneous cluster. These algorithms guarantee to find a placement for a new ap-
application if one exists. We also allowed the capsules of an application to have variable preference for the nodes on the cluster and showed how a standard algorithm for the minimum weight perfect matching problem may be used to find the most preferred of all possible placements for such an application.

3.6.2 Future Work

In this section we describe some directions for future work.

We have focused on the applications’ requirement for a single resource. Realistic applications exercise multiple resources (such as CPU, memory, disk, network bandwidth) on a server, and hence may want guarantees on access to more than one resource. Our approach for online placement can be extended in a straightforward manner to this scenario. Recall that in the online version of the problem we were satisfied with finding a placement for a new application if one existed. We can ensure this even when applications have requirements for multiple resources. A node is now said to be feasible for a capsule if, and only if, it has enough resources of each type to be able to meet the capsule’s requirement. A maximum matching on the resulting bipartite graph would yield a placement for a new application if one exists. For the offline placement, however, our goal was to maximize the number of applications that we could place on the cluster. Solving the offline problem when multiple resources are involved would be interesting future work.
CHAPTER 4
DYNAMIC CAPACITY PROVISIONING

4.1 Introduction

4.1.1 Motivation

Internet applications tend to see dynamically varying workloads that contain long-term variations such as time-of-day effects as well as short-term fluctuations due to flash crowds. Predicting the peak workload of an Internet application and capacity provisioning based on these worst case estimates is notoriously difficult. Under-estimating the peak workload can result in an application overload, causing the application to crash or become unresponsive. There are numerous documented examples of Internet applications that faced an outage due to an unexpected overload. For instance, the normally well-provisioned Amazon.com site suffered a forty-minute down-time due to an overload during the popular holiday season in November 2000.

Given the difficulties in predicting peak Internet workloads, an application needs to employ a combination of dynamic provisioning and request policing to handle workload variations. Dynamic provisioning enables additional resources—such as servers—to be allocated to an application on-the-fly to handle workload increases, while policing enables the application to temporarily turn away excess requests while additional resources are being provisioned.

As described in Chapter 2, typical Internet applications employ a multi-tier architecture, with each tier providing a certain functionality. In this chapter, we focus on dynamic resource provisioning for Internet applications that employ a multi-tier architecture. We argue that (i) provisioning multi-tier applications raises new challenges not addressed by
prior work on provisioning single-tier applications, and (ii) agile, proactive provisioning techniques are necessary to handle both long-term and short-term workload fluctuations seen by Internet applications. To address these issues, we present predictive and reactive provisioning mechanisms as well as a novel hosting platform architecture based on virtual machine monitors.

4.1.2 The Case for a New Provisioning Technique

Dynamic provisioning of resources—allocation and deallocation of servers to replicated applications—has been studied in the context of single-tier applications, of which clustered HTTP servers are the most common example. Fox et al. were the first to propose the notion of *hot spares* and their allocation to cluster-based applications *on-demand* [43]. The Muse project proposed a utility-based approach based on an economic theory for allocation and deallocation of servers to clustered Web servers [26]. Doyle et al. proposed a model-based approach for resource provisioning in single-tier Web servers [36]. Kamra et al. model a multi-tier ecommerce application as a single M/GI/1 server and present a PI-controller based admission control for maintaining response time targets [61]. Whereas multi-tier Internet applications have been studied in the context of SEDA [123, 124], the effort focused on admission control issues to maintain target response times and did not explicitly consider provisioning issues.

It is non-trivial to extend provisioning mechanisms designed for single-tier applications to multi-tier scenarios. To understand why, we consider two strawman approaches that are simple extensions of the above single-tier methods and demonstrate their limitations for multi-tier applications.

Since many single-tier provisioning mechanisms have already been proposed, a straightforward extension is to employ such an approach at each tier of the application. This enables provisioning decisions to be made independently at each tier based on local observations. Thus, our first strawman is to provision additional servers at a tier when the
incoming request rate at that tier exceeds the currently provisioned capacity; this can be inferred by monitoring queue lengths, tier-specific response times, or request drop rates. We refer to this approach as *independent per-tier provisioning*.

**Example 1:** Consider a three-tier Internet application depicted in Figure 4.1(a). Initially, we assume that one server each is allocated to the three tiers, and that this enables the application to service 15, 10 and 10.5 request/s at each tier (since a user request may impose different demands at different tiers, the provisioned capacity at each tier may be different). Let the incoming request rate be 14 req/s. Given the above capacities, all requests are let in through the first tier, and 4 req/s are dropped at the second tier. Due to these drops, the third tier sees a reduced request rate of 10 req/s and is able to service them all. Thus, the effective goodput is 10 req/s. Since request drops are only seen at the second tier, this tier is perceived to be the bottleneck. The provisioning algorithm at that tier will allocate an additional server, doubling its effective capacity to 20 req/s. At this point, the first two tiers are able to service all incoming requests and the third tier now sees a request rate of 14 req/s (see Figure 4.1(b)). Since its capacity is only 10.5 req/s, it drops 3.5 req/s. Thus, the

---

**Figure 4.1.** Strawman approaches for provisioning a multi-tier application.
bottleneck shifts to the third tier, and the effective goodput only increases from 10 to 10.5 reqs/s.

This simple example demonstrates that increasing the number of servers allocated to the bottleneck tier does not necessarily increase the effective goodput of the application. Instead, it may merely shift the bottleneck to another downstream tier. Although the provisioning mechanism at this downstream tier will subsequently increase its capacity, such shifting bottlenecks may require a number of independent provisioning steps at various tiers before the effective application capacity is actually increased. In the worst case, up to \( k \) provisioning steps, one at each tier, may be necessary in a \( k \)-tier application. Since allocation of servers to a tier entails overheads of several minutes or more [26], and since Internet workloads may spike suddenly, independent per-tier provisioning may be simply too slow to effectively respond to such workload dynamics.

Our second strawman models the multi-tier application as a black box and allocates additional servers whenever the observed response time exceeds a threshold.

**Example 2:** Consider the same three-tier application from Example 1 with tier-specific capacities of 15, 20 and 10.5 req/s as depicted in Figure 4.1(b). We ignore admission control issues in this example. Since the incoming request rate is 14 req/s, the first two tiers are able to serve all requests, while the third tier saturates, causing request queues to build up at this tier. This queue build up increases the end-to-end response time of the application beyond the threshold. Thus, as in the single-tier case, a black box approach can successfully detect when additional servers need to be provisioned for the multi-tier application.

However, determining *how many* servers to provision and *where* is far more complex for multi-tier applications. First, since the application is treated as a black box, the provisioning mechanism can only detect an increase in end-to-end response times but cannot determine which tier is responsible for this increase. Second, for single-tier applications, an application model is used to determine how many servers are necessary to service all
incoming requests with a certain response time threshold [36]. Extending such models to multi-tier applications is non-trivial, since each tier has different characteristics. In a typical e-commerce application, for instance, this implies collectively modeling the effects of HTTP servers, Java application servers and database servers—a complex task. Third, not all tiers of the application may be replicable. For instance, the database tier is typically difficult to replicate on-the-fly. In the above example, if the third tier is a non-replicable database, the black box approach, which has no knowledge of individual tiers, will incorrectly signal the need to provision additional servers, when the correct action is to trigger request policing and let no more than 10.5 req/s into the “black box”.

The above example demonstrates that due to the very nature of multi-tier applications, it is not possible to treat them as a black box for provisioning purposes. Knowledge of the number of tiers, their current capacities, and constraints on the degree of replication at each tier is essential for making proper provisioning decisions.

Both examples expose the limitations of using variants of single-tier provisioning methods for multi-tier applications. This chapter presents a multi-tier provisioning technique that overcome these limitations.

The remainder of this chapter is structured as follows. Section 4.2 presents an overview of the proposed system. Sections 4.2-4.5 present our provisioning algorithms. We present our prototype implementation in 4.6, our experimental evaluation in Section 4.7, and conclusions in Section 4.8.

4.2 Provisioning Algorithm Overview

The goal of our provisioning algorithm is to allocate sufficient capacity to the tiers of an application so that its SLA can be met even in the presence of the peak workload. At the

---

1The degree of replication of a tier is the maximum number of replicas that tier may consist of.
heart of any provisioning approach lie two issues: how much to provision and when? We provide an overview of our provisioning algorithm from this perspective.

**How much to provision:** We use the application model developed in Chapter 2 to determine how many servers to allocate to each tier and each application. By using this building block, we determine the number of servers necessary at each tier to handle a peak session arrival rate of $\lambda$ and provision resources accordingly. Our approach overcomes the drawbacks of independent per-tier provisioning and the black box approaches: While the capacity needed at each tier is determined separately using our queuing model, the desired capacities are allocated to the various tiers all at once. This ensures that each provisioning decision immediately results in an increase in effective capacity of the application.

**When to Provision:** The decision of when to provision depends on the dynamics of Internet workloads. Internet workloads exhibit long-term variations such as time-of-day or seasonal effects as well as short-term fluctuations such as flash crowds. While long-term variations can be predicted ahead of time by observing past variations, short-term fluctuations are less predictable, or in some cases, not predictable. Our techniques employ two different methods to handle variations observed at different time scales. We use predictive provisioning to estimate the workload for the next few hours and provision for it accordingly. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. Whereas predictive provisioning attempts to “stay ahead” of the anticipated workload fluctuations, reactive provisioning enables the hosting platform to be agile to deviations from the expected workload.

The following sections present our predictive and reactive provisioning methods.

### 4.3 How Much to Provision: Modeling Multi-tier Applications

To determine how many servers to provision for an application, we use the model presented in Chapter 2. Given an average session think-time of $Z$, and an average session
duration of \( \tau \), using Little’s Law [64], we can translate the session arrival rate of \( \lambda \) to an average number of active sessions given by \( \lambda \cdot \tau \).

We then use the model as described in Section 2.6.1 to determine the number of servers \( \eta_1, \ldots, \eta_k \) needed at the \( k \) tiers to handle a peak demand of \( \lambda \). We then increase the capacity of all tiers to these values in a single step, resulting in an immediate increase in effective capacity. In the event \( \eta_i \) exceeds the degree of replication \( M_i \) of a tier the actual allocation is reduced to this limit. Thus, each tier is allocated no more than \( \min(\eta_i, M_i) \) servers. To ensure that the SLA is not violated when the allocation is reduced to \( M_i \), the excess requests must be turned away at the sentry.

4.4 When to Provision?

In this section, we present two methods—predictive and reactive—to provision resources over long and short time-scales, respectively.

4.4.1 Predictive Provisioning for the Long Term

The goal of predictive provisioning is to provision resources over time scales of hours and days. The technique uses a workload predictor to predict the peak demand over the next several hours or a day and then uses the model presented in Chapter 2 to determine the number of servers that are needed to meet this peak demand. Predictive provisioning is motivated by long-term variations such as time-of-day or seasonal effects exhibited by Internet workloads [52]. For instance, the workload seen by an Internet application typically peaks around noon every day and is minimum in the middle of the night. Similarly, the workload seen by online retail Web sites is higher during the holiday shopping months of November and December than other months of the year. These cyclic patterns tend to repeat and can be predicted ahead of time by observing past variations. By employing a workload predictor that can predict these variations, our predictive provisioning technique
can allocate servers to an application well ahead of the expected workload peak. This ensures that application performance does not suffer even under the peak demand.

The key to predictive provisioning is the workload predictor. In this section, we present a workload predictor that estimates the tail of the arrival rate distribution (i.e., the peak demand) for the next few hours. Other statistical workload predictive techniques proposed in the literature can also be used with our predictive provisioning technique [52, 93].

Our workload predictor is based on a technique proposed by Rolia et al. [93] and uses past observations of the workload to predict peak demand that will be seen over a period of $T$ hours. For simplicity of exposition, assume that $T = 1$ hour. In that case, the predictor estimates the peak demand that will be seen over the next one hour, at the beginning of each hour. To do so, it maintains a history of the session arrival rate seen during each hour of the day over the past several days. A histogram is then generated for each hour using observations for that hour from the past several days (see Figure 4.2). Each histogram yields a probability distribution of the arrival rate for that hour. The peak workload for a particular hour is estimated as a high percentile of the arrival rate distribution for that hour (see Figure 4.2). Thus, by using the tail of the arrival rate distribution to predict peak demand, the predictive provisioning technique can allocate sufficient capacity to handle the worst-case load, should it arrive. Further, monitoring the demand for each hour of the day enables the predictor to capture time-of-day effects.

In addition to using observations from prior days, the workload seen in the past few hours of the current day can be used to further improve prediction accuracy. Suppose that $\lambda_{pred}(t)$ denotes the predicted arrival rate during a particular hour denoted by $t$. Further let $\lambda_{obs}(t)$ denote the actual arrival rate seen during this hour. The prediction error is simply $\lambda_{obs}(t) - \lambda_{pred}(t)$. In the event of a consistently positive prediction error over the past few hours, indicating that the predictor is consistently underestimating peak demand, the predicted value for the next hour is corrected using the observed error:
The workload prediction algorithm.

\[ \lambda_{\text{pred}}(t) = \lambda_{\text{pred}}(t) + \sum_{i=t-h}^{t-1} \frac{\max(0, \lambda_{\text{obs}}(i) - \lambda_{\text{pred}}(i))}{h} \]

where the second expression denotes the mean prediction error over the past \( h \) hours. We only consider positive errors in order to correct underestimates of the predicted peak demand—negative errors indicate that the observed workload is less than the peak demand, which only means that the worst-case workload did not arrive in that hour and is not necessarily a prediction error.

Using the predicted peak arrival rate for each application, the predictive provisioning technique uses the model to determine the number of servers that should be allocated to each tier of an application. An increase in allocation must be met by borrowing servers from the free pool or underloaded applications—underloaded applications are those whose new allocations are less than their current allocations. If the total number of required servers is less than the servers available in the free pool and those released by underloaded applications, then a utility-based approach [26] can be used to arbitrate the allocation of available servers to needy applications—servers are allocated to applications that benefit most from it as defined by their utility functions.
4.4.2 Reactive Provisioning: Handling Prediction Errors and Flash Crowds

The workload predictor outlined in the previous section is not perfect—it may incur prediction errors if the workload on a given day deviates from its behavior on previous days. Further, sudden load spikes or flash crowds are inherently unpredictable phenomena. Finally, errors in the online measurements of the model parameters can translate into errors in the allocations computed by the model. Reactive provisioning is used to swiftly react to such unforeseen events. Reactive provisioning operates on short time scales—on the order of minutes—checking for workload anomalies. If any such anomalies are detected, then it allocates additional capacity to various tiers to handle the workload increase.

Reactive provisioning is invoked once every few minutes. It can also be invoked on-demand by the application sentry if the observed request drop rate increases beyond a threshold. In either case, it compares the currently observed session arrival rate $\lambda_{obs}(t)$ over the past few minutes to the predicted rate $\lambda_{pred}(t)$. If the two differ by more than a threshold, corrective action is necessary. Specifically if $\lambda_{obs}(t) > \lambda_{pred}(t)$ or drop rate $> \tau_2$, where $\tau_1$ and $\tau_2$ are application-defined thresholds, then it computes a new allocation of servers. This can be achieved in one of two ways. One approach is to use the model to compute a new allocation of servers for the various tiers that can sustain an arrival rate $\lambda_{obs}(t)$. The second approach is to increase the allocation of all tiers that are at or near saturation by a constant amount (e.g., 10%). The new allocation needs to ensure that the bottleneck does not shift to another downstream tier; the capacity of any such tiers may also need to be increased proportionately. The advantage of using the model to compute the new allocation is that it yields the new capacity in a single step, as opposed to the latter approach that increases capacity by a fixed amount. The advantage of the latter approach is that it is independent of the model and can handle any errors in the measurements used to parameterize the model. In either case, the effective capacity of the application is raised to handle the increased workload.
The additional servers are borrowed from the free pool if available. If the free pool is empty or has insufficient servers, then these servers need to be borrowed from other underloaded applications running on the hosting platform. An application is said to be underloaded if its observed workload is significantly lower than its provisioned capacity: if \( \frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} < \tau_{low}, \) where \( \tau_{low} \) is a low watermark threshold.

Since a single invocation of reactive provisioning may be insufficient to bring sufficient capacity online during a large load spike, repeated invocations may be necessary in quick succession to handle the workload increase.

Together, predictive and reactive provisioning can handle long-term predictable workload variations as well as short term fluctuations that are less predictable. Predictive provisioning allocates capacity ahead of time in anticipation of a certain peak workload, while reactive provisioning takes corrective action after an anomalous workload increase has been observed. Put another way, predictive provisioning attempts to stay ahead of the workload fluctuations, while reactive provisioning follows workload fluctuations correcting for errors.

### 4.4.3 Request Policing

Request policing enables the hosted applications to temporarily turn away excess requests while additional resources are being provisioned. Also, sometimes there may not be enough resources in the hosting platform to meet an application’s entire workload—in such an event, the application employs its policer to turn away excess requests so that admitted requests continue meeting the SLA. A simple request policing policy could work as follows. The predictor and reactor convey the peak session arrival rate for which they have allocated capacity to the application’s sentry. This is done every time the allocation is changed. The sentry then ensures that the admission rate does not exceed this threshold—excess sessions are dropped at the sentry.

In Chapter 7, we describe a more sophisticated request policing approach.
4.5 Agile Server Switching using VMMs

A Virtual Machine Monitor (VMM) is a software layer that virtualizes the resources of a physical server and supports the execution of multiple virtual machines (VMs) [16]. Each VM runs a separate operating system and an application capsule within it. The VMM enables servers resources, such as the CPU, memory, disk and network bandwidth, to be partitioned among the resident virtual machines.

Traditionally VMMs have been employed in shared hosting environments to run multiple applications and their VMs on a single server; the VM provides isolation across applications while the VMM supports flexible partitioning of server resources across applications. In dedicated hosting, no more than one application can be active on a given physical server, and as a result, sharing of individual server resources across applications is moot in such environments. Instead, we employ VMMs for a novel purpose—fast server switching.

Traditionally, switching a server from one application to another for purposes of dynamic provisioning has entailed overheads of several minutes or more. Doing so involves some or all of the following steps: (i) wait for residual sessions of the current application to terminate, (ii) terminate the current application, (iii) scrub and reformat the disk to wipe
out sensitive data, (iv) reinstall the OS, (v) install and configure the new application. Our hosting platform runs a VMM on each physical server. Doing so enables it to eliminate many of these steps and drastically reduces switching time.

This work assumes a dedicated hosting model, where each application runs on a subset of the servers and a server is allocated to at most one application at any given time (except in special circumstances that we explain momentarily). Each capsule runs inside a virtual machine and each server runs a virtual machine monitor that executes this virtual machine. Depending on whether the capsule is replicable or not, the server may get classified as an Elf or an Ent. Elf servers runs replicable caspules, while Ent run non-replicable components of an application\(^2\). Unlike Ents, an Elf can be reassigned from one application to another. Multiple VMs and their associated capsules may reside on an Elf, although only one of these VMs can be active at any given time, as per the dedicated hosting model. The remaining VMs are dormant and are assigned minimal server resources. Each VM also runs a nucleus—a software component that performs online measurements of the capsule workload, its performance and resource usage; these statistics are periodically conveyed to the control plane. Figure 4.3 presents the architecture of a virtual machine based hosting platform.

We assume that each Elf server runs multiple virtual machines and capsules of different applications within it. Only one capsule and its virtual machine is active at any time—this is the capsule to which the server is currently allocated. Other virtual machines are dormant—they are allocated minimal server resources by the underlying VMM and most server resources are allocated to the active VM. If the server belongs to the free pool, all of its resident VMs are dormant.

In such a scenario, switching an Elf server from one application to another corresponds to deactivating a VM by reducing its resource allocation to \(\epsilon\), and reactivating a dormant

\(^2\)Elves are a swift and athletic race in J.R.R. Tolkien’s The Lord of the Rings as opposed to the bulky, tree-like Ents.
VM by increasing its allocation to 100-ε% of the server resources\(^3\). This only involves adjusting the allocations in the underlying VMM and incurs overheads in the order of tens of milliseconds. Thus, in theory, our hosting platform can switch a server from one application to another in a few milliseconds. In practice, however, we need to consider the residual state of the application before it can be made dormant.

To do so, we assume that once the predictor or the reactor decide to reassign a server from an underloaded to an overloaded application, they notify the load balancing element of the underloaded application tier. The load balancing element stops forwarding new sessions to this server. However, the server retains state of existing sessions and new requests may arrive for those sessions until they terminate. Consequently, the underloaded application tier will continue to use some server resources and the amount of resources required will diminish over time as existing sessions terminate. As a result, the allocation of the currently active VM cannot be instantaneously ramped down; instead the allocation needs to be reduced gradually, while increasing the allocation of the VM belonging to the overloaded application. Two strategies for ramping down the allocation of the current VM are possible.

- **Fixed rate ramp down:** In this approach, the resource allocation of the underloaded VM is reduced by a fixed amount \(\delta\) every \(t\) time units until it reduces to \(\epsilon\); the allocation of the new VM is increased correspondingly. The advantage of this approach is that it switches the server from one application to another in a fixed amount of time, namely \(t/\delta\). The limitation is that long-lived residual sessions will be forced to terminate, or their performance guarantees will be violated if the allocation decreases beyond that necessary to service them.

- **Measurement-based ramp down:** In this approach, the actual resource usage of the underloaded VM is monitored online. As the resource usage decreases with termi-

\(^3\)\(\epsilon\) is a small value such that the VM consumes negligible server resources and its capsule is idle and swapped out to disk.
nating sessions, the underlying allocation in the VMM is also reduced. The approach requires monitoring of the CPU, memory, network and disk usage so that the allocation can match the falling usage. The advantage of this approach is that the ramp-down is more conservative and less likely to violate performance guarantees of existing sessions. The drawback is that long-lived sessions may continue to use server resources, which increases the server switching time.

In either case, use of VMMs enables our hosting platform to reduce system switching overheads to a negligible value. The switching time is solely dominated by application idiosyncrasies. If the application has short-lived sessions or the application tier is stateless, the switching overhead is small. Even when sessions are long-lived, the overloaded application immediately gets some resources on the server, which increases its effective capacity; more resources become available as the current VM ramps down.

As a final detail, observe that we have assumed that sufficient dormant VMs are always available for various tiers of an overloaded application to arbitrarily increase its capacity. The hosting platform needs to ensure that there is always a pre-spawned pool of dormant VMs for each application in the system. As dormant VMs of an application are activated during an overload, and the number of dormant VMs falls below a low watermark, additional dormant VMs need to be spawned on other Elf servers, so that there is always a ready pool of VMs that can be tapped [114].

4.6 Implementation Considerations

We implemented a prototype hosting platform on a cluster of 40 Pentium servers connected via a 1Gbps ethernet switch and running Linux 2.4.20. Each machine in the cluster ran one of the following entities: (1) an application capsule (and its nucleus) or load balancer, (2) the control plane, (3) a sentry, (4) a workload generator for an application. The applications used in our evaluation (described in detail in Section 4.7.1) had two replicable tiers—front tier based on the Apache Web server and a middle tier based on Java servlets.
hosted on the Tomcat servlets container. The third tier was a non-replicable Mysql database server.

**Virtual Machine Monitor:** We use Xen 1.2 [16] as the virtual machine monitor in our prototype. The Xen VMM has a special virtual machine called domain0 (virtual machines are called domains in the Xen terminology) that gets created as soon as Xen boots and remains throughout the VMM’s existence. Xen provides a management interface that can be manipulated from domain0 to create new domains, control their CPU, network and memory resource allocations, allocate IP addresses, grant access to disk partitions, and suspend/resume domains to files, etc. The management interface is implemented as a set of library functions (implemented in C) for which there are Python language bindings. We use a subset of this interface—`xc_dom_create.py` and `xc_dom_control.py` provide ways to start a new domain or stop an existing one; the control plane implements a script that remotely logs on to domain0 and invokes these scripts. The control plane also implements scripts that can remotely log onto any existing domain to start a capsule and its nucleus or stop them. `xc_dom_control.py` provides an option that can be used to set the CPU share of an existing domain. The control plane uses this feature for VM ramp up and ramp down.

**Sentry and Load Balancer:** We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [66] to implement the policing mechanisms described in Section 4.4.3. ktcpvs is an open-source, Layer-7 request dispatcher implemented as a Linux module. A round-robin load balancer implemented in ktcpvs was used for Apache. Load balancing for the Tomcat tier was performed by `mod_jk`, an Apache module that implements a variant of round robin request distribution while taking into account session affinity. The sentry keeps record of arrival and finish times of admitted sessions as well as each request within a session. These observations are used to estimate the average session duration $\tau$ and the average think time $Z$. 
4.7 Experimental Evaluation

In this section we present the experimental setup followed by the results of our experimental evaluation.

4.7.1 Experimental Setup

The control plane was run on a dual-processor 450MHz machine with 1GB RAM. Elf and Ent servers had 2.8GHz processors and 512MB RAM. The sentries were run on dual-processor 1GHz machines with 1GB RAM. Finally, the workload generators were run on uniprocessor machines with 1GHz processors. Elves and Ents ran the Xen 1.2 VMM with Linux; all other machines ran Linux 2.4.20. All machines were interconnected by gigabit Ethernet.

We used the two open-source multi-tier applications described in Chapter 2—Rubis and Rubbos in our experimental study. Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate workloads and take measurements needed by our experiments. Rubis and Rubbos sessions had an average duration of 15min and 5min, respectively. For both applications, the average think time was 5sec.

We used 3-tier versions of these applications. The front tier was based on Apache 2.0.48 Web server. The middle tier was based on Java servlets that implement the application logic. We employed Tomcat 4.1.29 as the servlets container. Finally, the database tier was based on the Mysql 4.0.18 database.

Both applications are assumed to require an SLA where the 95\textsuperscript{th} percentile of the response time is no greater than 2 seconds. We use a simple heuristic to translate this SLA into an equivalent SLA specified using the average response time—since the model in Section 4.3 uses mean response times, such a translation is necessary. We use application profiling [115] to determine a distribution whose 95\textsuperscript{th} percentile is 2 seconds and use the mean of that distribution for the new SLA.
4.7.2 Effectiveness of Multi-tier Model

This section demonstrates the effectiveness of our multi-tier provisioning technique over variants of single-tier methods.

4.7.2.1 Independent Per-tier Provisioning

Our first experiment uses the Rubbos application. We use the first strawman described in Example 1 of Section 4.1 for provisioning Rubbos. Here, each tier employs its own provisioning technique. Rubbos was subjected to a workload that increases in steps, once every ten minutes (see Fig. 4.4(a)). The first workload increase occurs at $t=600$ sec and saturates the tier-1 Web server. This triggers the provisioning technique, and an additional server is allocated at $t=900$ sec (see Figure 4.4(b)). At this point, the two tier-1 servers are able to service all incoming requests, causing the bottleneck to shift to the Tomcat tier. The Elf running Tomcat saturates, which triggers provisioning at tier 2. An additional server is allocated to tier 2 at $t=1200$ sec (see Fig. 4.4(b)). The second workload increase occurs at $t=1200$ sec and the above cycle repeats. As shown in Figure 4.4(c), since multiple provisioning steps are needed to yield an effective increase in capacity, the application SLA is violated during this period.

A second strawman is to employ dynamic provisioning only at the most compute-intensive tier of the application, since it is the most common bottleneck [118]. In Rubbos,
the Tomcat tier is the most compute intensive of the three tiers and we only subject this
tier to dynamic provisioning. The Apache and Tomcat tiers were initially assigned 1 and 2
servers respectively. The capacity of a Tomcat server was determined to be 40 simultane-
ous sessions using our model, while Apache was configured with a connection limit of 256
sessions. As shown in Figure 4.5(a), every time the current capacity of the Tomcat tier is
saturated by the increasing workload, two additional servers are allocated. The number of
servers at tier-2 increases from 2 to 8 over a period of time. At t=1800sec, the session ar-
rival rate increases beyond the capacity of the first tier, causing the Apache server to reach
its connection limit of 256. Subsequently, even though plenty of capacity was available at
the Tomcat tier, newly arriving sessions are turned away due to the connection bottleneck
at Apache and the throughput reaches a plateau (see Figure 4.5(b)). Thus, focusing only
on the the commonly bottlenecked tier is not adequate, since the bottleneck will eventually
shift to other tiers.

Next, we repeat this experiment with our multi-tier provisioning technique. Since our
technique is aware of the demands at each tier and can take idiosyncrasies such as connec-
tion limits into account, as shown in Figure 4.6(a), it is able to scale the capacity of both
the Web and the Tomcat tiers with increasing workloads. Consequently, as shown in Fig-

![Figure 4.5. Rubbos: Provision only the Tomcat tier](image)
Figure 4.6. Rubbos: Model-based multi-tier provisioning

ure 4.6(b), the application throughput continues to increase with the increasing workload. Figure 4.6(c) shows that the SLA is maintained throughout the experiment.

Result: Existing single-tier methods are inadequate for provisioning resources for multi-tier applications as they may fail to capture multiple bottlenecks. Our technique anticipates shifting bottlenecks due to capacity addition at a tier and increases capacity at all needy tiers. Further, it can identify different bottleneck resources at different tiers, e.g. CPU at the Tomcat tier and Apache connections at the Web tier.

4.7.2.2 The Black box Approach

We subjected the Rubis application to a workload that increased in steps, as shown in Figure 4.8(a). First, we use the black box provisioning Approach described in Example 2 of Section 4.1. The provisioning technique monitors the per-request response times over 30s intervals and signals a capacity increase if the $95^{th}$ percentile response time exceeds 2sec. Since the black box technique is unaware of the individual tiers, we assume that two Tomcat servers and one Apache server are added to the application every time a capacity increase is signaled. As shown in Figure 4.7(a) and (c), the provisioned capacity keeps increasing with increasing workload and whenever the $95^{th}$ percentile of response time is over 2 seconds. However, as shown in Figure 4.7(d), at $t=1100$sec, the CPU on the Ent running the database saturates. Since the database server is not replicable, increasing capacity of the other two
tiers beyond this point does not yield any further increase in effective capacity. However, the black box approach is unaware of where bottleneck lies and continues to add servers to the first two tiers until it has used up all available servers. The response time continues to degrade despite this capacity addition as the Java servlets spend increasingly larger amounts of time waiting for queries to be returned by the overloaded database (see Figures 4.7(c) and (d)).

We repeat this experiment using our multi-tier provisioning technique. Our results are shown in Figure 4.8. As shown in Figure 4.8(b), the control plane adds servers to the application at t=390sec in response to the increased workload. However, beyond this point, no additional capacity is allocated. Our technique correctly identifies that the capacity of the database tier for this workload is around 600 simultaneous sessions. Consequently, when this capacity is reached and the database saturates, it triggers policing instead of provisioning. The admission control is triggered at t=1070sec and drops any sessions in excess of this limit during the remainder of the experiment. Figure 4.8(d) shows that our provisioning is able to maintain a satisfactory response time throughout the experiment.

Result: Our provisioning technique is able to take constraints imposed by non-replicable tiers into account. It can maintain response time targets by invoking the admission control when capacity addition does not help.

4.7.3 Predictive and Reactive Provisioning

In this section we present experiments to demonstrate the need to have both predictive and reactive provisioning mechanisms. We used Rubis in these experiments. The workload was generated based on the Web traces from the 1998 Soccer World Cup site [6]. These traces contained the number of arrivals per minute to this Web site over an 8-day period. Based on these we created several smaller traces to drive our experiments. These traces were obtained by compressing the original 24-hr long traces to 1 hr—this was done by picking arrivals for every 24th minute and discarding the rest. This enables us to capture
Figure 4.7. Rubis: Blackbox provisioning

the time-of-day effect as a “time-of-hour” effect. The experiment invoked predictive provisioning once every 15min over the one hour duration and we refer to these periods as Intervals 1-4; reactive provisioning was invoked on-demand or once every few minutes.

For the sake of convenience, in the rest of the section, we will simply refer to these traces by the day from which they were constructed (even though they are only 1 hr long). We present three of these traces: (i) Figure 4.9(a) shows the workload for day 6 (a typical day), (ii) Figure 4.10(a) shows the workload for day 7, (moderate overload), and (iii) Figure 4.11(a) shows the workload for day 8 (extreme overload). Throughout this experiment, we
will assume that the database tier has sufficient capacity to handle the peak observed on day 8 and does not become a bottleneck. The average session duration in our trace was 5min.

**4.7.3.1 Only Predictive Provisioning**

Figure 4.9 presents the performance of the system during day 6 with the control plane employing only predictive provisioning (with reactive provisioning disabled). Day 6 was a “typical” day, meaning the workload closely resembled that observed during the previous days. The prediction algorithm was successful in exploiting this and was able to assign sufficient capacity to the application at all times. In Figure 4.9(b) we observe that the
predicted arrivals closely matched the actual arrivals. The control plane added servers at t=30min—well in time for the increased workload during the second half of the experiment. The application experienced satisfactory response time throughout the experiment (Figure 4.9(c)).

Result: Our predictive provisioning works well on “typical” days.

4.7.3.2 Only Reactive Provisioning

In Figure 4.10 we present the results for day 7. Comparing this workload with that on day 6, we find that the application experienced a moderate overload on day 7, with the arrival rate going up to about 150 sessions/min, more than twice the peak on day 6. The workload showed a monotonically increasing trend for the first 40min.

We first let the control plane employ only predictive provisioning. Figure 4.10(b) shows the performance of our prediction algorithm, both with and without using recent trends to correct the prediction. We find that it severely underestimated the number of arrivals in Interval 2. The use of recent trends allowed it to progressively improve its estimate in Intervals 3 and 4 (predicted arrivals were nearly 80% of the actual arrivals in Interval 3 and almost equal in Interval 4). In Figure 4.10(c) we observe that the response time target was violated in Interval 2 due to under allocation of servers.
Next, we repeated the experiment with the control plane using only reactive provisioning. Figure 4.10(d) presents the application performance. Consider Interval 2 first—we observe that, unlike predictive provisioning, the reactive mechanism was able to pull additional servers at t=15min in response to the increased arrival rate, thus bringing down the response time within target. However, as the experiment progressed, the server allocation lagged behind the continuously increasing workload. Since reactive provisioning only responded to very recent workload trends, it could not anticipate future requirement well and took multiple allocation steps to add sufficient capacity. Meanwhile, the application experienced repeated violations of SLA during Intervals 2 and 3.
Result: We need reactive mechanisms to deal with large flash crowds. However, reactive provisioning alone may not be effective, since its actions lag the workload.

4.7.3.3 Integrated Provisioning

We used the workload on day 8 where the application experienced an extremely large overload (Figure 4.11(a)). The peak workload on this day was an order of magnitude (about 20 times) higher than on a typical day. Figure 4.11(b) shows how the prediction algorithm performed during this overload. The algorithm failed to predict the sharp increase in the workload during Interval 1. In Interval 2 it could correct its estimate based on the observed workload during Interval 1. The workload increased drastically (reaching up to 1200 sess/sec) during Intervals 3 and 4, and the algorithm failed to predict it.

In Figure 4.11(c) we show the performance of Rubis when the control plane employs both predictive and reactive mechanisms and session policing is disabled. In Interval 1, the reactive mechanism successfully added additional capacity (at t=8min) to lower the response time. It was invoked again at t=34min (Observe that predictive provisioning was operating in concert with reactive provisioning; it resulted in the server allocations at t=15, 30, 45min). However, by this time (and for the remainder of the experiment) the workload was simply too high to be serviced by the servers available. We imposed a resource limit of 13 servers for illustrative purposes. Beyond this, excess sessions must be turned away to continue meeting the SLA for admitted sessions. The lack of session policing caused response times to degrade during Intervals 3 and 4.

Next, we repeated this experiment with the session policing enabled. The performance of Rubis is shown in Figure 4.11(d). The behavior of our provisioning mechanisms is exactly like above. However, by turning away excess sessions, the sentry was able to maintain the SLA throughout.
Result: Predictive and reactive mechanisms and policing are all integral components of an effective provisioning technique. Our hosting platform integrates all of these, enabling it to handle diverse workloads.

4.7.4 VM-based Switching of Server Resources

We present measurements on our testbed to demonstrate the benefits that our VM-based switching can provide. We switch a server from a Tomcat capsule of Rubis to a Tomcat capsule of Rubbos. We compare five different ways of switching a server to illustrate the salient features of our scheme:
Scenario 1: New server taken from the free pool of servers; capsule and nucleus have to be started on the server.

Scenario 2: New server taken from the free pool of servers; capsule already running on a VM.

Scenario 3: New server taken from another application with residual sessions; we wait for all residual sessions to finish.

Scenario 4: New server taken from another application with residual sessions; we let the two VMs share the CPU equally while the residual sessions exist.

Scenario 5: New server taken from another application with residual sessions; we change the CPU shares of the involved VMs using the “fixed rate ramp down” strategy in Section 4.5.

Table 4.1 presents the switching time and the performance of residual sessions of Rubis in each of the above scenarios. Comparing scenarios 2 and 3, we find that in our VM-based scheme, the time to switch a server is solely dependent on the residual sessions—the residual sessions of Rubis took about 17min to finish resulting in the large switching time in scenario 3. Scenarios 4 and 5 show that by letting the two VMs coexist while the residual sessions finish, we can eliminate this switching time. However, it is essential to continue providing sufficient capacity to the residual sessions during the switching period to ensure good performance—in scenario 4, new Rubbos sessions deprived the residual sessions of Rubis of the capacity they needed, thus degrading their response time.

Result: Use of virtual machines can enable agile switching of servers. Our adaptive techniques improve upon the delays in switching caused by residual sessions.

4.7.5 System Overheads

Two sources of overhead in the proposed system are the virtual machines that run on the Elf nodes and the nuclei that run on all nodes. Measurements on our prototype indicate that the CPU overhead and network traffic caused by the nuclei is negligible. The control
plane runs on a dedicated node and its scalability is not a cause of concern. We chose the Xen VMM to implement our switching scheme since the performance of Xen/Linux has been shown to be consistently close to native Linux [16]. Further, Xen has been shown to provide good performance isolation when running multiple VMs simultaneously, and is capable of scaling to 128 concurrent VMs.

### 4.8 Concluding Remarks

In this chapter, we argued that dynamic provisioning of multi-tier Internet applications raises new challenges not addressed by prior work on provisioning single-tier applications. We proposed a novel dynamic provisioning technique for multi-tier Internet applications that employs (i) a flexible queuing model to determine how much resources to allocate to each tier of the application, and (ii) a combination of predictive and reactive methods that determine when to provision these resources, both at large and small time scales. We proposed a novel hosting platform architecture based on virtual machine monitors to reduce provisioning overheads. Our experiments on a forty machine Linux-based hosting platform demonstrate the responsiveness of our technique in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes while maintaining response time targets. Our technique also reduced the overhead of switching servers.
across applications from several minutes or more to less than a second, while meeting the performance targets of residual sessions.
5.1 Introduction and Motivation

In this chapter and the next, we focus on the design, implementation, and evaluation of resource management mechanisms in a shared cluster—a commodity cluster where the number of applications is significantly larger than the number of nodes, necessitating resource sharing among applications. In a shared hosting platform, the central challenges for the platform provider are the ability to reserve resources for individual applications, the ability to isolate applications from other misbehaving or overloaded applications, and the ability to provide performance guarantees to applications [94].

Arguably, the widespread deployment of shared hosting platforms has been hampered by the lack of effective resource management mechanisms that meet these requirements. Most hosting platforms in use today adopt one of two approaches.

The first avoids resource sharing altogether by employing a dedicated model. This delivers useful resources to application providers, but is expensive in machine resources. The second approach shares resources in a best-effort manner among applications, which consequently receive no resource guarantees. While this is cheap in resources, the value delivered to application providers is limited. Consequently, both approaches imply an economic disincentive to deploy viable hosting platforms.

Recently, several resource management mechanisms for shared hosting platforms have been proposed [8, 9, 26, 113]. This chapter reports work performed in this context, but with two significant differences in goals.
Firstly, we seek from the outset to support a diverse set of potentially antagonistic network services simultaneously on a platform. The services will therefore have heterogeneous resource requirements; Web servers, continuous media processors, and multi-player game engines all make different demands on the platform in terms of resource bandwidth and latency. We evaluate our system with such a diverse application mix.

Secondly, we aim to support resource management policies based on yield management techniques such as those employed in the airline industry [103]. Yield management is driven by the business relationship between a platform provider and many application providers, and results in different short-term goals. In traditional approaches the most important aim is to satisfy all resource contracts while making efficient use of the platform. Yield management by contrast is concerned with ensuring that as much of the available resource as possible is used to generate revenue, rather than being utilized “for free” by a service (since it would otherwise be idle).

An analogy with air travel may clarify the point: instead of trying to ensure that every ticketed passenger gets to board their chosen flight, we try to ensure that no plane takes off with an empty seat (which is achieved by overbooking seats).

An immediate consequence of this goal is our treatment of “flash crowds”—a shared hosting platform should react to an unexpected high demand on an application only if there is an economic incentive for doing so. That is, the platform should allocate additional resources to an application only if it enhances revenue. Further, any increase in resource allocation of an application to handle unexpected high demands should not be at the expense of contract violations for other applications, since this is economically undesirable. Hence, in contrast to systems aimed at an enterprise environment like the oft-cited CNN server farm, where a wholesale redistribution of resources is feasible, our system responds to a flash crowd only with resources which cannot be employed to generate revenue elsewhere, unless a prenegotiated arrangement between the application provider and platform provider exists to justify the disruption.
5.1.1 Research Contributions

The contribution of this chapter is threefold. First, we show how the resource requirements of an application can be derived using offline profiling. Second, we demonstrate the efficiency benefits to the platform provider of under-provisioning the hosted applications, and how this can be usefully done without adversely impacting the guarantees offered to application providers. Thirdly, we show how untrusted and/or mutually antagonistic applications in the platform can be isolated from one another. The rest of this section presents these contributions in detail.

Automatic derivation of QoS requirements: We discuss techniques for empirically deriving an application’s resource needs. The effectiveness of a resource management technique is crucially dependent on the ability to reserve appropriate resources for each application—overestimating an application’s resource needs can result in idling of resources, while underestimating them can degrade application performance. Consequently a shared hosting platform can significantly enhance its utility to users by automatically deriving the QoS requirements of an application. Automatic derivation of QoS requirements involves (i) monitoring an application’s resource usage, and (ii) using these statistics to derive QoS requirements that conform to the observed behavior.

We employ kernel-based profiling mechanisms to empirically monitor an application’s resource usage and propose techniques to derive QoS requirements from this observed behavior. We then use these techniques to experimentally profile several server applications such as Web, streaming, game, and database servers. Our results show that the bursty resource usage of server applications makes it feasible to extract statistical multiplexing gains by under-provisioning the hosted applications.

Revenue maximization through under-provisioning: We discuss resource under-provisioning techniques and application placement strategies for shared hosting platforms. Provisioning cluster resources solely based on the worst-case needs of an application results in low average utilization, since the average resource requirements of an application are typically
smaller than its worst case (peak) requirements, and resources tend to idle when the application does not utilize its peak reserved share. In contrast, provisioning a cluster based on a high percentile of the application needs yields statistical multiplexing gains that significantly increase the average utilization of the cluster at the expense of a small amount of under-provisioning, and increases the number of applications that can be supported on a given hardware configuration.

A well-designed shared hosting platform should be able to provide performance guarantees to applications even when they are under-provisioned, with the proviso that this guarantee is now probabilistic (for instance, an application might be provided a 99% guarantee (0.99 probability) that its resource needs will be met). Since different applications have different tolerance to such under-provisioning (e.g., the latency requirements of a game server make it less tolerant to violations of performance guarantees than a Web server), an under-provisioning mechanism should take into account diverse application needs.

We demonstrate the feasibility and benefits of under-provisioning resources in shared hosting platforms, and propose techniques to under-provision resources in a controlled fashion based on application resource needs. Although such under-provisioning can result in transient overloads where the aggregate resource demand temporarily exceeds capacity, our techniques limit the chances of transient overload of resources to predictably rare occasions, and provide useful performance guarantees to applications in the presence of under-provisioning. The techniques we describe are general enough to work with many commonly used OS resource allocation mechanisms.

Placement and isolation of antagonistic applications: We describe an additional aspect of the resource management problem: placement and isolation of antagonistic applications. We assume that third-party applications may be antagonistic and cannot be trusted by the platform, due either to malice or bugs. Our work demonstrates how untrusted third-party applications can be isolated from one another in shared hosting platforms in two ways. Local to a machine, each processing node in the platform employs resource management
techniques that “sandbox” applications by restricting the resources consumed by an application to its reserved share. Globally, we present automated placement techniques that allow a platform provider to exert sufficient control over the placement of application components onto nodes in the cluster, since manual placement of applications is unfeasibly complex and error-prone in large clusters.

5.1.2 System Model

The shared hosting platform assumed in our research consists of a cluster of nodes, each of which consists of processor, memory, and storage resources as well as one or more network interfaces. Platform nodes are allowed to be heterogeneous with different amounts of these resources on each node. The nodes in the hosting platform are assumed to be interconnected by a high-speed LAN such as gigabit ethernet (see Figure 5.1). Each cluster node is assumed to run an operating system kernel that supports some notion of quality of service such as reservations or shares. Such mechanisms have been extensively studied over the past decade and many deployed commercial and open-source operating systems such as Solaris [106], IRIX [98], Linux [107], and FreeBSD [19] already support such features. In this chapter, we focus on managing two resources—CPU and network interface bandwidth—in shared hosting platforms. The challenges of managing other resources in hosting environments, such as memory and storage, are beyond the scope of this thesis. As Aron points out, management of other resources which are inherently temporal in nature, such as disk bandwidth, can be performed by similar mechanisms [8]. Spatial resources, in particular physical memory, present a different challenge. A straightforward approach is to use static partitioning as in [8], although recently more sophisticated approaches have been implemented [17, 121].

The rest of this chapter is structured as follows. Section 5.2 discusses techniques for empirically deriving an application’s resource needs, while Section 5.3 discusses our resource under-provisioning techniques and capsule placement strategies. We discuss imple-
5.2 Automatic Derivation of Application QoS Requirements

The first step in hosting a new application is to derive its resource requirements. While the problem of QoS-aware resource management has been studied extensively in the literature [14, 59, 68], the problem of how much resource to allocate to each application has received relatively little attention. In this section, we address this issue by proposing techniques to automatically derive the QoS requirements of an application (the terms resource requirements and QoS requirements are used interchangeably in this chapter.) Deriving the QoS requirements is a two step process: (i) we first use profiling techniques to monitor application behavior, and (ii) we then use our empirical measurements to derive QoS requirements that conform to the observed behavior.
5.2.1 Application QoS Requirements: Definitions

The QoS requirements of an application are defined on a per-capsule basis. For each capsule, the QoS requirements specify the intrinsic rate of resource usage, the variability in the resource usage, the time period over which the capsule desires resource guarantees, and the level of under-provisioning that the application (capsule) is willing to tolerate. As explained earlier, in this chapter, we are concerned with two key resources, namely CPU and network interface bandwidth. For each of these resources, we define the QoS requirements along the above dimensions in an OS-independent manner. In Section 5.4.1, we show how to map these requirements to various OS-specific resource management mechanisms that have been developed.

More formally, we represent the QoS requirements of an application capsule by a quintuple \((\sigma, \rho, \tau, U, V)\):

- **Token Bucket Parameters** \((\sigma, \rho)\): We capture the basic resource requirements of a capsule by modeling resource usage as a token bucket \((\sigma, \rho)\) [109]. The parameter \(\sigma\) denotes the intrinsic rate of resource consumption, while \(\rho\) denotes the variability in the resource consumption. More specifically, \(\sigma\) denotes the rate at which the capsule consumes CPU cycles or network interface bandwidth, while \(\rho\) captures the maximum burst size. By definition, a token bucket bounds the resource usage of the capsule to \(\sigma \cdot t + \rho\) over any interval \(t\).

- **Period** \(\tau\): The third parameter \(\tau\) denotes the time period over which the capsule desires guarantees on resource availability. Put another way, the system should strive to meet the QoS requirements of the capsule over each interval of length \(\tau\). The smaller the value of \(\tau\), the more stringent are the desired guarantees (since the capsule needs to be guaranteed resources over a finer time scale). In particular, for the above token bucket parameters, the capsule requires that it be allocated at least \(\sigma \cdot \tau + \rho\) resources every \(\tau\) time units.
• **Usage Distribution** $U$: While the token bucket parameters succinctly capture the capsule’s resource requirements, they are not sufficiently expressive by themselves to denote the QoS requirements in the presence of under-provisioning. Consequently, we use two additional parameters—$U$ and $V$—to specify resource requirements in the presence of under-provisioning. The first parameter $U$ denotes the probability distribution of resource usage. Note that $U$ is a more detailed specification of resource usage than the token bucket parameters $(\sigma, \rho)$, and indicates the probability with which the capsule is likely to use a certain fraction of the resource (i.e., $U(x)$ is the probability that the capsule uses a fraction $x$ of the resource, $0 \leq x \leq 1$). A probability distribution of resource usage is necessary so that the hosting platform can provide (quantifiable) probabilistic guarantees even in the presence of under-provisioning.

• **Violation Tolerance** $V$: The parameter $V$ is the violation tolerance of the capsule. It specifies the probability with which the capsule’s requirements may be violated in a period due to resource under-provisioning (by providing it with less resources than the required amount). Thus, the violation tolerance indicates the minimum level of service that is acceptable to the capsule. To illustrate, if $V = 0.01$, the capsule’s resource requirements should be met 99% of the time (or with a probability of 0.99 in each interval $\tau$).

In general, we assume that parameters $\tau$ and $V$ are specified by the application provider. This may be based on a contract between the platform provider and the application provider (e.g., the more the application provider is willing to pay for resources, the stronger are the provided guarantees), or on the particular characteristics of the application (e.g., a streaming media server requires more stringent guarantees and is less tolerant to violations of these guarantees). In the rest of this section, we show how to derive the remaining three parameters $\sigma$, $\rho$ and $U$ using profiling, given values of $\tau$ and $V$. 
5.2.2 Kernel-based Profiling of Resource Usage

Our techniques for empirically deriving the QoS requirements of an application rely on profiling mechanisms that monitor application behavior. Recently, a number of application profiling mechanisms ranging from OS-kernel-based profiling [4] to run-time profiling using specially linked libraries [99] have been proposed.

We use kernel-based profiling mechanisms in the context of shared hosting platforms, for a number of reasons. Firstly, being kernel-based, these mechanisms work with any application and require no changes to the application at the source or binary levels. This is especially important in hosting environments where the platform provider may have little or no access to third-party applications. Secondly, accurate estimation of an application’s resource needs requires detailed information about when and how much resources are used by the application at a fine time-scale. Whereas detailed resource allocation information is difficult to obtain using application-level techniques, kernel-based techniques can provide precise information about various kernel events such as CPU scheduling instances and network packet transmissions times.

The profiling process involves running the application on a set of isolated platform nodes (the number of nodes required for profiling depends on the number of capsules). By isolated, we mean that each node runs only the minimum number of system services necessary for executing the application and no other applications are run on these nodes during the profiling process—such isolation is necessary to minimize interference from unrelated tasks when determining the application’s resource usage. The application is then subjected to a realistic workload, and the kernel profiling mechanism is used to track its resource usage. It is important to emphasize that the workload used during profiling should be both realistic and representative of real-world workloads. While techniques for generating such realistic workloads are orthogonal to our current research, we note that a number of different workload-generation techniques exist, ranging from trace replay of actual workloads to running the application in a “live” setting, and from the use of synthetic workload gener-
ators to the use of well-known benchmarks. Any such technique suffices for our purpose as long as it realistically emulates real-world conditions, although we note that, from a business perspective, running the application “for real” on an isolated machine to obtain a profile may be preferable to other workload generation techniques.

We use the Linux trace toolkit as our kernel profiling mechanism [72]. The toolkit provides flexible, low-overhead mechanisms to trace a variety of kernel events such as system call invocations, process, memory, file system, and network operations. The user can specify the specific kernel events of interest as well as the processes that are being profiled to selectively log events. For our purposes, it is sufficient to monitor CPU and network activity of capsule processes—we monitor CPU scheduling instances (the time instants at which capsule processes get scheduled and the corresponding quantum durations) as well as network transmission times and packet sizes. Given such a trace of CPU and network activity, we now discuss the derivation of the capsule’s QoS requirements.

5.2.3 Empirical Derivation of the QoS Requirements

We use the trace of kernel events obtained from the profiling process to model CPU and network activity as a simple On-Off process. This is achieved by examining the time at which each event occurs and its duration and deriving a sequence of busy (On) and idle (Off) periods from this information (see Figure 5.2). This trace of busy and idle periods can

Figure 5.2. An example of an On-Off trace.
then be used to derive both the resource usage distribution $U$ as well as the token bucket parameters $(\sigma, \rho)$.

Determining the usage distribution $U$: Recall that, the usage distribution $U$ denotes the probability with which the capsule uses a certain fraction of the resource. To derive $U$, we simply partition the trace into measurement intervals of length $I$ and measure the fraction of time for which the capsule was busy in each such interval. This value, which represents the fractional resource usage in that interval, is converted into a histogram and then each bucket is normalized with respect to the number of measurement intervals $I$ in the trace to obtain the probability distribution $U$. Figure 5.3(a) illustrates this process.

Deriving token bucket parameters $(\sigma, \rho)$: Recall that a token bucket limits the resource usage of a capsule to $\sigma \cdot t + \rho$ over any interval $t$. A given On-Off trace can have, in general, many $(\sigma, \rho)$ pairs that satisfy this bound. To intuitively understand why, let us compute the cumulative resource usage for the capsule over time. The cumulative resource usage is simply the total resource consumption thus far and is computed by incrementing the cumulative usage after each ON period. Thus, the cumulative resource usage is a step...
function as depicted in Figure 5.3(b). Our objective is to find a line $\sigma \cdot t + \rho$ that bounds the cumulative resource usage; the slope of this line is the token bucket rate $\sigma$ and its Y-intercept is the burst size $\rho$. As shown in Figure 5.3(b), there are in general many such curves, all of which are valid descriptions of the observed resource usage.

Several algorithms that mechanically compute all valid $(\sigma, \rho)$ pairs for a given On-Off trace have been proposed recently. We use a variant of one such algorithm [109] in our research—for each On-Off trace, the algorithm produces a range of $\sigma$ values (i.e., $[\sigma_{\text{min}}, \sigma_{\text{max}}]$) that constitute valid token bucket rates for observed behavior. For each $\sigma$ within this range, the algorithm also computes the corresponding burst size $\rho$. Although any pair within this range conforms to the observed behavior, the choice of a particular $(\sigma, \rho)$ has important practical implications.

Since the violation tolerance $V$ for the capsule is given, we can use $V$ to choose a particular $(\sigma, \rho)$ pair. To illustrate, if $V = 0.05$, the capsule needs must be met 95% of the time, which can be achieved by reserving resources corresponding to the 95th percentile of the usage distribution. Consequently, a good policy for shared hosting platforms is to pick a $\sigma$ that corresponds to the $(1 - V) \times 100^{th}$ percentile of the resource usage distribution $U$, and to pick the corresponding $\rho$ as computed by the above algorithm. This ensures that we provision resources based on a high percentile of the capsule’s needs and that this percentile is chosen based on the specified violation tolerance $V$.

5.2.4 Profiling Server Applications: Experimental Results

In this section, we profile several commonly-used server applications to illustrate the process of deriving an application’s QoS requirements. Our experimentally derived profiles not only illustrate the inherent nature of various server applications but also demonstrate the utility and benefits of resource under-provisioning in shared hosting platforms.

The test bed for our profiling experiments consists of a cluster of five Dell PowerEdge 1550 servers, each with a 966 MHz Pentium III processor and 512 MB memory running
Red Hat Linux 7.0. All servers runs the 2.2.17 version of the Linux kernel patched with the Linux trace toolkit version 0.9.5, and are connected by 100Mbps Ethernet links to a Dell PowerConnect (model no. 5012) ethernet switch.

![Graphs](image1.png)  ![Graphs](image2.png)  ![Graphs](image3.png)

(a) PDF  (b) CDF  (c) Token bucket parameters

**Figure 5.4.** Profile of the Apache Web server using the default SPECWeb99 configuration.

To profile an application, we run it on one of our servers and use the remaining servers to generate the workload for profiling. We assume that all machines are lightly loaded and that all non-essential system services (e.g., mail services, X windows server) are turned off to prevent interference during profiling. The parameters \( \tau \) and \( \mathcal{I} \) were both set to 1 sec in all our experimentation. We profile the following server applications in our experiments:

- **Apache Web server:** We use the SPECWeb99 benchmark [104] to generate the workload for the Apache Web server (version 1.3.24) [5]. The SPECWeb benchmark allows control along two dimensions—the number of concurrent clients and the percentage of dynamic (cgi-bin) HTTP requests. We vary both parameters to study their impact on Apache’s resource needs.

- **MPEG streaming media server:** We use a home-grown streaming server to stream MPEG-1 video files to multiple concurrent clients over UDP. Each client in our experiment requests a 15 minute long variable bit rate MPEG-1 video with a mean bit rate of 1.5 Mb/s. We vary the number of concurrent clients and study its impact on the resource usage at the server.
- **Quake game server:** We use the publicly available Linux Quake server to understand the resource usage of a multi-player game server; our experiments use the standard version of Quake I [89]—a popular multi-player game on the Internet. The client workload is generated using a bot—an autonomous software program that emulates a human player. We use the publicly available “terminator” bot to emulate each player; we vary the number of concurrent players connected to the server and study its impact on the resource usage.

- **PostgreSQL database server:** We profile the PostgreSQL database server (version 7.2.1) [86] using the pgbench 1.2 benchmark. This benchmark is part of the PostgreSQL distribution and emulates the TPC-B transactional benchmark [85]. The benchmark provides control over the number of concurrent clients as well as the number of transactions performed by each client. We vary both parameters and study their impact on the resource usage of the database server.

We now present some results from our profiling study.

Figure 5.4(a) depicts the CPU usage distribution of the Apache Web server obtained using the default settings of the SPECWeb99 benchmark (50 concurrent clients, 30% dynamic cgi-bin requests). Figure 5.4(b) plots the corresponding cumulative distribution function (CDF) of the resource usage. As shown in the figure (and summarized in Table 5.1), the worst case CPU usage (100th percentile) is 25% of CPU capacity. Further, the 99th and the 95th percentiles of CPU usage are 10 and 4% of capacity, respectively. These results indicate that CPU usage is bursty in nature and that the worst-case requirements are significantly higher than a high percentile of the usage. Consequently, under-provisioning by a mere 1% reduces the CPU requirements of Apache by a factor of 2.5, while under-provisioning by 5% yields a factor of 6.25 reduction (implying that 2.5 and 6.25 times as many Web servers can be supported when provisioning based on the 99th and 95th percentiles, respectively, instead of the 100th profile). Thus, even small amounts of under-provisioning can potentially yield significant increases in platform capacity. Figure 5.4(c)
Figure 5.5. Profiles of Various Server Applications

depicts the possible valid \((\sigma, \rho)\) pairs for Apache’s CPU usage. Depending on the specified violation tolerance \(V\), we can set \(\sigma\) to an appropriate percentile of the usage distribution \(U\), and the corresponding \(\rho\) can then be chosen using this figure.

Figures 5.5(a)-(d) depict the CPU or network bandwidth distributions, as appropriate, for various server applications. Specifically, the figure shows the usage distribution for the Apache Web server with 50% dynamic SPECWeb requests, the streaming media server with 20 concurrent clients, the Quake game server with 4 clients and the postgreSQL server with 10 clients. Table 5.1 summarizes our results and also presents profiles for several additional scenarios (only a small subset of the three dozen profiles obtained from our
Table 5.1. Summary of profiles. Although we profiled both CPU and network usage for each application, we only present results for the more constraining resource. Abbreviations: WS=Apache, SMS=streaming media server, GS=Quake game server, DBS=database server, k=number of clients, dyn.=dynamic, Res.=Resource.

<table>
<thead>
<tr>
<th>Application</th>
<th>Res.</th>
<th>Res. usage at percentile</th>
<th>(σ, ρ) for O = 0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS,default</td>
<td>CPU</td>
<td>0.25 0.10 0.04</td>
<td>(0.10, 0.218)</td>
</tr>
<tr>
<td>WS, 50% dyn.</td>
<td>CPU</td>
<td>0.69 0.29 0.12</td>
<td>(0.29, 0.382)</td>
</tr>
<tr>
<td>SMS,k=4</td>
<td>Net</td>
<td>0.19 0.16 0.11</td>
<td>(0.16, 1.89)</td>
</tr>
<tr>
<td>SMS,k=20</td>
<td>Net</td>
<td>0.63 0.49 0.43</td>
<td>(0.49, 6.27)</td>
</tr>
<tr>
<td>GS,k=2</td>
<td>CPU</td>
<td>0.011 0.010 0.009</td>
<td>(0.010, 0.00099)</td>
</tr>
<tr>
<td>GS,k=4</td>
<td>CPU</td>
<td>0.018 0.016 0.014</td>
<td>(0.016, 0.00163)</td>
</tr>
<tr>
<td>DBS,k=1 (def)</td>
<td>CPU</td>
<td>0.33 0.27 0.20</td>
<td>(0.27, 0.184)</td>
</tr>
<tr>
<td>DBS,k=10</td>
<td>CPU</td>
<td>0.85 0.81 0.79</td>
<td>(0.81, 0.130)</td>
</tr>
</tbody>
</table>

Together, Figure 5.5 and Table 5.1 demonstrate that all server applications exhibit burstiness in their resource usage, albeit to different degrees. This burstiness causes the worst-case resource needs to be significantly higher than a high percentile of the usage distribution. Consequently, we find that the 99th percentile is smaller by a factor of 1.1-2.5, while the 95th percentile yields a factor of 1.3-6.25 reduction when compared to the 100th percentile. Together, these results illustrate the potential gains that can be realized by under-provisioning resources in shared hosting platforms.

5.3 Resource Under-provisioning and Capsule Placement in Hosting Platforms

Having derived the QoS requirements of each capsule, the next step is to determine which platform node will run each capsule. Several considerations arise when making such placement decisions. First, since the applications are being under-provisioned, the platform should ensure that the QoS requirements of a capsule will be met even in the presence of experiments are presented).
under-provisioning. Second, since multiple nodes may have the resources necessary to house each application capsule, the platform will need to pick a specific mapping from the set of feasible mappings. In this section, we present techniques for under-provisioning applications in a controlled manner. The aim is to ensure that: (i) the QoS requirements of the application are satisfied and (ii) violation tolerances as well as external policy constraints are taken into account while making placement decisions.

5.3.1 Resource Under-provisioning Techniques

A platform node can accept a new application capsule so long as the resource requirements of existing capsules are not violated, and sufficient unused resources exist to meet the requirements of the new capsule.

To verify that a node can meet the requirements of all capsules, we simply sum the requirements of individual capsules and ensure that the aggregate requirements do not exceed node capacity. For each capsule $i$ on the node, the QoS parameters $(\sigma_i, \rho_i)$ and $\tau_i$ require that the capsule be allocated $(\sigma_i \cdot \tau_i + \rho_i)$ resources in each interval of duration $\tau_i$. Further, since the capsule has a violation tolerance $V_i$, in the worst case, the node can allocate only $(\sigma_i \cdot \tau_i + \rho_i) \cdot (1 - V_i)$ resources and yet satisfy the capsule needs. Consequently, even in the worst case scenario, the resource requirements of all capsules can be met so long as the total resource requirements do not exceed the capacity:

$$\sum_{i=1}^{k+1}(\sigma_i \cdot \tau_{\text{min}} + \rho_i) \cdot (1 - V_i) \leq C \cdot \tau_{\text{min}}$$

(5.1)

where $C$ denotes the CPU or network interface capacity on the node, $k$ denotes the number of existing capsules on the node, $k + 1$ is the new capsule, and $\tau_{\text{min}} = \min(\tau_1, \tau_2, \ldots, \tau_{k+1})$ is the period $\tau$ for the capsule that desires the most stringent guarantees\(^1\).

---

\(^1\)Note that since the $\sigma_i$ for capsule $i$ was chosen based on the $(1 - V_i) \cdot 100^{th}$ percentile of the capsule’s resource usage distribution, this multiplication with $(1 - V_i)$ may seem like penalizing the capsule twice. However, this is not so because $\sigma_i$ in combination with the burst $\rho_i$ is an upper envelop of the requirements of capsule $i$. The multiplication with $(1 - V_i)$ allows us to under-provision the capsule in a controlled manner.
Equation (5.1) can easily handle heterogeneity in nodes by using appropriate $C$ values for the CPU and network capacities on each node.

A new capsule can be placed on a node if Equation (5.1) is satisfied for both the CPU and network interface. Since multiple nodes may satisfy a capsule’s CPU and network requirements, especially at low and moderate utilizations, we need to devise policies to choose a node from the set of all feasible nodes for the capsule. We discuss this issue next.

### 5.3.2 Capsule Placement Algorithms

We refer the reader to our discussion of online placement algorithms in Section 3.5.1. In addition to the *Best-fit* and *Worst-fit* based policies presented there we introduce a *Random* policy which picks a node uniformly at random from the set of nodes feasible for a capsule. We experimentally compare the effectiveness these three policies in Section 5.5.2.

### 5.3.3 Handling Dynamically Changing Resource Requirements

Our discussion thus far has assumed that the resource requirements of an application at run-time do not change after the initial profiling phase. In reality though, resource requirements change dynamically over time, in tandem with the workload seen by the application. In this section we outline our approach for dealing with dynamically changing application workloads.

---

**Figure 5.6.** Demonstration of how an application overload may be detected by comparing the latest resource usage profile with the original offline profile.
First, recall that we provision resources based on a high percentile of the application’s resource usage distribution. Consequently, variations in the application workload that affect only the average resource requirements of the capsules, *but not the tail of the resource usage distribution*, will not result in violations of the probabilistic guarantees provided by the hosting platform. In contrast, workload changes that cause an increase in the tail of the resource usage distribution will certainly affect application QoS guarantees.

How a platform should deal with such changes in resource requirements depends on several factors. Since we are interested in yield management, the platform should increase the resources allocated to an overload application only if it increases revenues for the platform provider. Thus, if an application provider only pays for a fixed amount of resources, there is no economic incentive for the platform provider to increase the resource allocation beyond this limit even if the application is overloaded. In contrast, if the contract between the application and platform provider permits usage-based charging (i.e., charging for resources based on the actual usage, or a high percentile of the usage\(^2\)), then allocating additional resources in response to increased demand is desirable for maximizing revenue. In such a scenario, handling dynamically changing requirements involves two steps: (i) detecting changes in the tail of the resource usage distribution, and (ii) reacting to these changes by varying the actual resources allocated to the application.

To detect such changes in the tail of an application’s resource usage distribution, we propose to conduct continuous, on-line profiling of the resource usage of all capsules using low-overhead profiling tools. This would be done by recording the CPU scheduling instants, network transmission times, and packet sizes for all processes over intervals of a suitable length. At the end of each interval, this data would be processed to construct the latest resource usage distributions for all capsules. An application overload would manifest...

\(^2\)ISPs charge for network bandwidth in this fashion—the customer pays for the 95\(^{th}\) percentile of its bandwidth usage over a certain period.
fest itself through an increased concentration in the high percentile buckets of the resource usage distributions of its capsules.

We present the results of a simple experiment to illustrate this. Figure 5.6(a) shows the CPU usage distribution of the Apache Web server obtained via offline profiling. The workload for the Web server was generated by using the SPECWeb99 benchmark emulating 50 concurrent clients with 50% dynamic cgi-bin requests. The offline profiling was done over a period of 30 minutes. Next, we assumed a violation tolerance of 1% for this Web server capsule. As described in Section 5.2.3, it was assigned a CPU rate of 0.29 (corresponding to the 99th percentile of its CPU usage distribution). The remaining capacity was assigned to a greedy dhrystone application (this application performs compute-intensive integer computations and greedily consumes all resources allocated to it). The Web server was then subjected to exactly the same workload (50 clients with 50% cgi-bin requests) for 25 minutes, followed by a heavier workload consisting of 70 concurrent clients with 70% dynamic cgi-bin requests for 5 minutes. The heavier workload during the last 5 minutes was to simulate an unexpected flash crowd. The Web server’s CPU usage distribution was recorded over periods of length 10 minute each. Figure 5.6(b) shows the CPU usage distribution observed for the Web server during a period of expected workload. We find that this profile is very similar to the profile obtained using offline measurements, except being upper-bounded by the CPU rate assigned to the capsule. Figure 5.6(c) plots the CPU usage distribution during the period when the Web server was overloaded. We find an increased concentration in the high percentile regions of this distribution compared to the original distribution.

The detection of application overload would trigger remedial actions that would proceed in two stages. First, new resource requirements would be computed for the affected capsules. Next, actions would be taken to provide the capsules the newly computed resource shares—this may involve increasing the resource allocations of the capsules, or moving the capsules to nodes with sufficient resources. Implementing and evaluating these
techniques for handling application overloads are part of our ongoing research on shared hosting platforms.

5.4 Implementation Considerations

In this section, we first discuss implementation issues in integrating our resource under-provisioning techniques with OS resource allocation mechanisms. We then present an overview of our prototype implementation.

5.4.1 Providing Application Isolation at Run Time

The techniques described in the previous section allow a platform provider to under-provision applications and yet provide guarantees that the QoS requirements of applications will be met. The task of enforcing these guarantees at run-time is the responsibility of the OS kernel. To meet these guarantees, we assume that the kernel employs resources allocation mechanisms that support some notion of quality of service. Numerous such mechanisms—such as reservations, shares, and token bucket regulators [14, 59, 68]—have been proposed recently. All of these mechanisms allow a certain fraction of each resource (CPU cycles, network interface bandwidth) to be reserved for each application and enforce these allocations on a fine time scale.

In addition to enforcing the QoS requirements of each application, these mechanisms also isolate applications from one another. By limiting the resources consumed by each application to its reserved amount, the mechanisms prevent a malicious or overloaded application from grabbing more than its allocated share of resources, thereby providing application isolation at run-time—an important requirement in shared hosting environments running untrusted applications.

Our under-provisioning techniques can exploit many commonly used QoS-aware resource allocation mechanisms. Since the QoS requirements of an application are defined in a OS- and mechanism-independent manner, we need to map these OS-independent QoS
requirements to mechanism-specific parameter values. We outline these mappings for three commonly-used QoS-aware mechanisms—reservations, proportional-share schedulers, and rate regulators.

**Reservations:** A reservation-based scheduler [59, 68] requires the resource requirements to be specified as a pair \((x, y)\) where the capsule desires \(x\) units of the resource every \(y\) time units (effectively, the capsule requests \(\frac{x}{y}\) fraction of the resource). For reasons of feasibility, the sum of the requests allocations should not exceed 1 (i.e., \(\sum_j \frac{x_j}{y_j} \leq 1\)). In such a scenario, the QoS requirements of a capsule with token bucket parameters \((\sigma_i, \rho_i)\) and a violation tolerance \(V_i\) can be translated to reservation by setting \((1 - V_i) \cdot \sigma_i = \frac{x_i}{y_i}\) and \((1 - V_i) \cdot \rho_i = x_i\). To see why, recall that \((1 - V_i) \cdot \sigma_i\) denotes the rate of resource consumption of the capsule in the presence of under-provisioning, which is same as \(\frac{x_i}{y_i}\). Further, since the capsule can request \(x_i\) units of the resource every \(y_i\) time units, and in the worst case, the entire \(x_i\) units may be requested continuously, we set the burst size to be \((1 - V_i) \cdot \rho_i = x_i\). These equations simplify to \(x_i = (1 - V_i) \cdot \rho_i\) and \(y_i = \rho_i / \sigma_i\).

**Proportional-share and lottery schedulers:** Proportional-share and lottery schedulers [49, 122] enable resources to be allocated in relative terms—in either case, a capsule is assigned a weight \(w_i\) (or \(w_i\) lottery tickets) causing the scheduler to allocate \(\frac{w_i}{\sum_j w_j}\) fraction of the resource. Further, two capsules with weights \(w_i\) and \(w_j\) are allocated resources in proportion to their weights \((w_i : w_j)\). For such schedulers, the QoS requirements of a capsule can be translated to a weight by setting \(w_i = (1 - V_i) \cdot \sigma_i\). By virtue of using a single parameter \(w_i\) to specify the resource requirements, such schedulers ignore the burstiness \(\rho\) in the resource requirements. Consequently, the underlying scheduler will only approximate the desired QoS requirements. The nature of approximation depends on the exact scheduling algorithm—the finer the time-scale of the allocation supported by the scheduler, the better will the actual allocation approximate the desired requirements.

**Rate regulators:** Rate regulators are commonly used to police the network interface bandwidth used by an application. Such regulators limit the sending rate of the application
based on a specified profile. A commonly used regulator is the token bucket regulator that limits the amount of bytes transmitted by an application to \( \sigma \cdot t + \rho \) over any interval \( t \). Since we model resource usage of a capsule as a token bucket, the QoS requirements of a capsule trivially map to an actual token bucket regulator and no special translation is necessary.

5.4.2 Prototype Implementation

We have implemented a Linux-based shared hosting platform that incorporates the techniques discussed in the previous sections. Our implementation consists of three key components: (i) a profiling module that allows us to profile applications and empirically derive their QoS requirements, (ii) a control plane that is responsible for resource under-provisioning and capsule placement, and (iii) a QoS-enhanced Linux kernel that is responsible for enforcing application QoS requirements.

The profiling module runs on a set of dedicated (and therefore isolated) platform nodes and consists of a vanilla Linux kernel enhanced with the Linux trace toolkit. As explained in Section 5.2, the profiling module gathers a kernel trace of CPU and network activities of each capsule. It then post-processes this information to derive an On-Off trace of resource usage and then derives the usage distribution \( U \) and the token bucket parameters for this usage.

The control plane is responsible for placing capsules of newly arriving applications onto nodes while under-provisioning them. The control plane also keeps state consisting of a list of all capsules residing on each node and their QoS requirements. It also maintains information about the hardware characteristics of each node. The requirements of a newly arriving application are specified to the control plane using a resource specification language. This specification includes the CPU and network bandwidth requirements of each capsule. The control plane uses this specification to derive a placement for each capsule as discussed in Section 5.3.2. In addition to assigning each capsule to a node, the control
plane also translates the QoS parameters of the capsules to parameters of commonly used resource allocation mechanisms (discussed in the previous section).

The third component, namely the QoS-enhanced Linux kernel [88], runs on each platform node and is responsible for enforcing the QoS requirements of capsules at run time. For the purposes of this thesis, we implement the H-SFQ proportional-share CPU scheduler [49]. H-SFQ is a hierarchical proportional-share scheduler that allows us to group resource principals (processes, lightweight processes) and assign an aggregate CPU share to the entire group. This functionality is essential since a capsule contains all processes of an application that are collocated on a node and the QoS requirements are specified for the capsule as a whole rather than for individual resource principals. To implement such an abstraction, we create a separate node in the H-SFQ scheduling hierarchy for each capsule, and attach all resource principals belonging to a capsule to this node. The node is then assigned a weight (determined using the capsule’s QoS requirements) and the CPU allocation of the capsule is shared by all resource principals of the capsule.³ We implement a token bucket regulator to provide QoS guarantees at the network interface card. Our rate regulator allows us to associate all network sockets belonging to a group of processes to a single token bucket. We instantiate a token bucket regulator for each capsule and regulate the network bandwidth usage of all resource principals contained in this capsule using the \((\sigma, \rho)\) parameters of the capsule’s network bandwidth usage. In Section 5.5.3, we experimentally demonstrate the efficacy of these mechanisms in enforcing the QoS requirements of capsules even in the presence of under-provisioning.

5.5 Experimental Evaluation

In this section, we present the results of our experimental evaluation. The setup used in our experiments is identical to that described in Section 5.2.4—we employ a cluster of

³The use of the scheduling hierarchy to further multiplex capsule resources among resource principals in a controlled way is clearly feasible but beyond the scope of this work.
Linux-based servers as our shared hosting platform. Each server runs a QoS-enhanced Linux kernel consisting of the H-SFQ CPU scheduler and a leaky bucket regulator for the network interface. The control plane for the shared platform implements the resource under-provisioning and capsule placement strategies discussed earlier in this chapter. For ease of comparison, we use the same set of applications discussed in 5.2.4 and their derived profiles (see Table 5.1) for our experimental study.

5.5.1 Efficacy of Resource Under-provisioning

Our first set of experiments examine the efficacy of under-provisioning applications in shared hosting platforms. We first consider shared Web hosting platforms—a type of shared hosting platform that runs only Web servers. Each Web server running on the platform is assumed to conform to one of the four Web server profiles gathered from our profiling study (two of these profiles are shown in Table 5.1; the other two employed varying mixes of static and dynamic SPECWeb99 requests). The objective of our experiment is to examine how many such Web servers can be supported by a given platform configuration for various violation tolerances. We vary the violation tolerance from 0% to 10%, and for each tolerance value, attempt to place as many Web servers as possible until the platform resources are exhausted. We first perform the experiment for a cluster of 5 nodes (identical to our hardware configuration) and then repeat it for cluster sizes ranging from 16 to 128 nodes (since we lack clusters of these sizes, for these experiments, we only examine how many applications can be accommodated on the platform and do not actually run these applications). Figure 5.7(a) depicts our results with 95% confidence intervals. The figure shows that, the larger the amount of under-provisioning, the larger is the number of Web servers that can be run on a given platform. Specifically, for a 128 node platform, the number of Web servers that can be supported increases from 307 when no under-provisioning is employed to over 1800 for 10% under-provisioning (a factor of 5.9 increase). Even for a modest 1% under-provisioning, we see a factor of 2 increase in the number of Web
servers that can be supported on platforms of various sizes. Thus, even modest amounts of under-provisioning can significantly enhance revenues for the platform provider.

Next, we examine the benefits of under-provisioning applications in a shared hosting platform that runs a mix of streaming servers, database servers, and Web servers. To demonstrate the impact of burstiness on under-provisioning, we first focus only on the streaming media server. As shown in Table 5.1, the streaming server (with 20 clients) exhibits less burstiness than a typical Web server, and consequently, we expect smaller gains due to resource under-provisioning. To quantify these gains, we vary the platform size from 5 to 128 nodes and determine the number of streaming servers that can be supported with 0%, 1%, and 5% under-provisioning. Figure 5.7(b) plots our results with 95% confidence intervals. As shown, the number of servers that can be supported increases by 30-40% with 1% under-provisioning when compared to the no under-provisioning case. Increasing the amount of under-provisioning from 1% to 5% yields only a marginal additional gain, consistent with the profile for this streaming server shown in Table 5.1 (and also indicative of the less-tolerant nature of this soft real-time application). Thus, less bursty applications yield smaller gains when under-provisioning resources.

Although the streaming server does not exhibit significant burstiness, large statistical multiplexing gains can still accrue by collocating bursty and non-bursty applications. Further, since the streaming server is heavily network-bound and uses a minimal amount of

**Figure 5.7.** Benefits of resource under-provisioning for a bursty Web server application, a less bursty streaming server application and for application mixes.
CPU, additional gains are possible by collocating applications with different bottleneck resources (e.g., CPU-bound and network-bound applications). To examine the validity of this assertion, we conduct an experiment where we attempt to place a mix of streaming, Web, and database servers—a mix of CPU-bound and network-bound as well as bursty and non-bursty applications. Figure 5.7(c) plots the number of applications supported by platforms of different sizes with 1% under-provisioning. As shown, an identical platform configuration is able to support a large number of applications than the scenario where only streaming servers are placed on the platform. Specifically, for a 32 node cluster, the platform supports 36 and 52 additional Web and database servers in addition to the approximately 80 streaming servers that were supported earlier. We note that our capsule placement algorithms are automatically able to extract these gains without any specific “tweaking” on our part. Thus, collocating applications with different bottleneck resources and different amounts of burstiness enhance additional statistical multiplexing benefits when under-provisioning applications.

5.5.2 Capsule Placement Algorithms

Our next experiment compares the effectiveness of the best-fit, worst-fit and, random placement algorithms discussed in Section 5.3.2. Using our profiles, we construct two types of applications: a replicated Web server and an e-commerce application consisting of a front-end Web server and a back-end database server. Each arriving application belongs to one of these two categories and is assumed to consist of 2-10 capsules, depending on the degree of replication. The violation tolerance is set to 5%. We then determine the number of applications that can be placed on a given platform by different placement strategies. Figure 5.8(a) depicts our results. As shown, best-fit and random placement yield similar performance, while worst-fit outperforms these two policies across a range of platform sizes. This is because best-fit places capsules onto nodes with smaller unused capacity, resulting in “fragmentation” of unused capacity on a node; the leftover capacity may be
wasted if no additional applications can be accommodated. Worst fit, on the other hand, reduces the chances of such fragmentation by placing capsules onto nodes with the larger unused capacity. While such effects become prominent when application capsules have widely varying requirements (as observed in this experiment), they become less noticeable when the application have similar resource requirements. To demonstrate this behavior, we attempted to place Quake game servers onto platforms of various sizes. Observe from Table 5.1 that the game server profiles exhibit less diversity than a mix of Web and database servers. Figure 5.8(b) shows that, due to the similarity in the application resource requirements, all policies are able to place a comparable number of game servers.

Finally, we examine the effectiveness of taking the violation tolerance into account when making placement decisions. We compare the worst-fit policy to a violation-conscious worst-fit policy. The latter policy chooses the three worst-fits among all feasible nodes and picks the node that best matches the violation tolerance of the capsule. Our experiment assumes a Web hosting platform with two types of applications: less-tolerant Web servers that permit 1% under-provisioning and more tolerant Web servers that permit 10% under-provisioning. We vary the platform size and examine the total number of applications placed by the two policies. As shown in Figure 5.8(c), taking violation tolerances into
account when making placement decisions can help increase the number of applications placed on the cluster. However, we find that the additional gains are small (< 6% in all cases), indicating that a simple worst-fit policy may suffice for most scenarios.

5.5.3 Effectiveness of Kernel Resource Allocation Mechanisms

While our experiments thus far have focused on the impact of under-provisioning on platform capacity, in our next experiment, we examine the impact of under-provisioning on application performance. We show that combining our under-provisioning techniques with kernel-based QoS resource allocation mechanisms can indeed provide application isolation and quantitative performance guarantees to applications (even in the presence of under-provisioning). We begin by running the Apache Web server on a dedicated (isolated) node and examine its performance (by measuring throughput in requests/s) for the default SPECWeb99 workload. We then run the Web server on a node running our QoS-enhanced Linux kernel. We first allocate resources based on the 100th percentile of its usage (no under-provisioning) and assign the remaining capacity to a greedy dhrystone application (this application performs compute-intensive integer computations and greedily consumes all resources allocated to it). We measure the throughput of the Web server in presence of this background dhrystone application. Next, we reserve resources for the Web server based on the 99th and the 95th percentiles, allocate the remaining capacity to the dhrystone application, and measure the server throughput. Table 5.2 depicts our results. As shown, provisioning based on the 100th percentile yields performance that is comparable to running the application on a dedicated node. Provisioning based on the 99th and 95th percentiles results in a small degradation in throughput, but well within the permissible limits of 1% and 5% degradation, respectively, due to under-provisioning. Table 5.2 also shows that provisioning based on the average resource requirements results in a substantial fall in throughput, indicating that reserving resources based on mean usage is not advisable for shared hosting platforms.
Table 5.2. Effectiveness of kernel resource allocation mechanisms. All results are shown with 95% confidence intervals.

<table>
<thead>
<tr>
<th>Application</th>
<th>Metric</th>
<th>Isolated Node</th>
<th>100&lt;sup&gt;th&lt;/sup&gt;</th>
<th>99&lt;sup&gt;th&lt;/sup&gt;</th>
<th>95&lt;sup&gt;th&lt;/sup&gt;</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>Throughput (req/s)</td>
<td>67.93 ± 2.08</td>
<td>67.51 ± 2.12</td>
<td>66.91 ± 2.76</td>
<td>64.81 ± 2.54</td>
<td>39.82 ± 5.26</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>Throughput (trans/s)</td>
<td>22.84 ± 0.54</td>
<td>22.46 ± 0.46</td>
<td>22.21 ± 0.63</td>
<td>21.78 ± 0.51</td>
<td>9.04 ± 0.85</td>
</tr>
<tr>
<td>Streaming</td>
<td>Length of viols (sec)</td>
<td>0</td>
<td>0</td>
<td>0.31 ± 0.04</td>
<td>0.59 ± 0.05</td>
<td>5.23 ± 0.22</td>
</tr>
</tbody>
</table>

We repeat the above experiment for the streaming server and the database server. The background load for the streaming server experiment is generated using a greedy UDP sender that transmits network packets as fast as possible, while that in case of the database server is generated using the dhrystone application. In both cases, we first run the application on an isolated node and then on our QoS-enhanced kernel with provisioning based on the 100<sup>th</sup>, 99<sup>th</sup>, and the 95<sup>th</sup> percentiles. We also run the application with provisioning based on the average of its resource usage distribution obtained via offline profiling. We measure the throughput in transaction/s for the database server and the mean length of a playback violation (in seconds) for the streaming media server. Table 5.2 plots our results. Like with the Web server, provisioning based on the 100<sup>th</sup> percentile yields performance comparable to running the application on an isolated node, while a small amount of under-provisioning results in a corresponding small amount of degradation in application performance. Again, we observe that provisioning based on the average resource usage results in significantly degraded performance.

For each of the above scenarios, we also computed the application profiles in the presence of background load and under-provisioning and compared these to the profiles gathered on the isolated node. Figure 5.9 shows one such set of profiles. It should be seen in combination with the second row in Table 5.2 that corresponds to the PostgreSQL application. Together, they depict the performance of the database server for different levels of CPU provisioning. Figures 5.9(b) and (c) show the CPU profiles of the database server when it is provisioned based on the 99<sup>th</sup> and the 95<sup>th</sup> percentiles respectively. As can be seen, the two profiles look similar to the original profile shown in Figure 5.9(a). Cor-
respondingly, Table 5.2 shows that for these levels of CPU provisioning, the throughput received by the database server is only slightly inferior to that on an isolated node. This indicates that upon provisioning resources based on a high percentile, the presence of background load interferes minimally with the application behavior. In Figure 5.9(d), we show the CPU profile when the database server was provisioned based on its average CPU requirement. This profile is drastically different from the original profile. We also present the corresponding low throughput in Table 5.2. This reinforces our earlier observation that provisioning resources based on the average requirements can result in significantly degraded performance.
Together, these results demonstrate that our kernel resource allocation mechanisms are able to provide quantitative performance guarantees even when applications are under-provisioned.

### 5.6 Related Work

Research on clustered environments over the past decade has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [75]. Fox et al. study the design of scalable, fault-tolerant network services running on server clusters [43]. Govil et al. study the use of virtual clusters to manage resources and contain faults in large multiprocessor systems [48]. Saito et al. investigate scalability, availability, and performance issues in dedicated clusters in the context of clustered mail servers [95] while Aron et al. address these issues in replicated Web servers [9]. Numerous middleware-based approaches for clustered environments have also been proposed [32, 35]. Ongoing efforts in the grid computing community have focused on developing standard interfaces for resource reservations in clustered environments [51]. Finally, efforts such as gang scheduling and co-scheduling have investigated the issue of coordinating the scheduling of tasks in distributed systems [13, 53]; however, neither of these techniques incorporates the issue of quality of service while making scheduling decisions. In the context of QoS-aware resource allocation, numerous efforts over the past decade have developed predictable resource allocation mechanisms for single machine environments [14, 59, 68]. Such techniques form the building block for resource allocation in clustered environments.

Statistical admission control techniques that under-provision (or overbook) resources have been studied in the context of video-on-demand servers [119] and ATM networks [20], but little work as been published to date in the context of shared, cluster-based hosting platforms.
Most closely related to our work is Aron [8, 10], who presents a comprehensive framework for resource management in Web servers, with the aim of delivering predictable QoS and differentiated services. New services are profiled by running on lightly-loaded machines, and contracts subsequently negotiated in terms of application level performance (connections per second), reported by the application to the system. CPU and disk bandwidth are scheduled by lottery scheduling [122] and SFQ [49] respectively, while physical memory is statically partitioned between services with free pages allocated temporarily to services that can make use of them. A resource monitor running over a longer timescale examines performance reported by the application and system performance information and flags conditions which might violate contracts, to allow extra resources to be provided by external means.

In Aron’s system, resource allocation is primarily driven by application feedback and the primary concern is allowing a principal to meet its contract. It is instructive to compare this with our own goal of maximizing the yield in the system, which amounts to maximizing the proportion of system resources used to satisfy contracts.

This subtle but important difference also motivates other differences in design choices between the two systems. In Aron’s work, application overload is detected when resource usage exceeds some predetermined threshold. We, on the other hand, detect overload by observing the tail of the recent resource usage distributions. On the other hand, several features of Aron’s work, such as the use of multiple random variables to capture the behavior of services over different time scales, are directly applicable to our system.

The specific problem of QoS-aware resource management for clustered environments has been investigated by Aron et al. [9]. This effort builds upon single node QoS-aware resource allocation mechanisms and proposes techniques to extend their benefits to clustered environments. Chase et al. propose a system called Muse for provisioning resources in hosting platforms based on energy considerations [26]. Muse is based on an economic approach to managing shared server resources in which services “bid” for resources as a
function of delivered performance. It also provides mechanisms to continuously monitor load and compute new resource allocations by estimating the value of their effects on service performance. A salient difference between Muse and our approach is that Muse provisions resources based on the average resource requirements whereas we provision based on the tail of the resource requirements. As shown in Section 5.5.3, provisioning resources based on average requirements can result in substantially degraded QoS and is therefore not advisable for shared hosting platforms.

5.7 Concluding Remarks

In this chapter, we presented techniques for provisioning CPU and network resources in shared hosting platforms running potentially antagonistic third-party applications. We argued that provisioning resources solely based on the worst-case needs of applications results in low average utilization, while provisioning based on a high percentile of the application needs can yield statistical multiplexing gains that significantly increase the utilization of the cluster. Since an accurate estimate of an application’s resource needs is necessary when provisioning resources, we presented techniques to profile applications on dedicated nodes, possibly while in service, and used these profiles to guide the placement of application components onto shared nodes. We then proposed techniques to under-provision hosted applications in a controlled fashion such that the platform can provide performance guarantees to applications even with this under-provisioning. Our techniques, in conjunction with commonly used OS resource allocation mechanisms, can provide application isolation and performance guarantees at run-time in the presence of under-provisioning. We implemented our techniques in a Linux cluster and evaluated them using common server applications. We found that the efficiency benefits from controlled under-provisioning of applications can be dramatic when compared to provisioning resources based on the worst-case requirements of applications. Specifically, under-provisioning applications by as little as 1% increases the utilization of the hosting platform by a factor of 2, while under-
provisioning by 5-10% results in gains of up to 500%. The more bursty the application resources needs, the higher are the benefits of resource under-provisioning. More gener-
ally, our results demonstrate the benefits and feasibility of under-provisioning resources for the platform provider.
CHAPTER 6
SHARC: MANAGING RESOURCES IN SHARED HOSTING PLATFORMS

6.1 Introduction

6.1.1 Motivation

In the last chapter we addressed issues that arise in shared hosting platforms when hosting an application. This chapter deals with dynamic resource management in a shared hosting platform hosting distributed applications. Whereas several techniques for predictable allocation of resources within a single machine have been developed over the past decade [14, 50, 59, 68, 116], relatively less work has been done on predictable resource allocation for distributed applications running on a shared cluster. There are a number of research issues that must be addressed to enable effective resource sharing in commodity clusters. Since lots of applications share a relatively small number of machines, the ability to reserve resources for individual applications (especially when application owners may be paying for these resources), the ability to isolate applications from one another, and the need to manage the heterogeneous performance requirements of applications are some challenges that must be addressed in shared environments. High availability and scalability are other important issues, although they are common to dedicated clusters as well.
6.1.2 Research Contributions

In this chapter we present Sharc: a system for managing resources in shared clusters\(^1\). Sharc extends the benefits of single node resource management mechanisms to clustered environments.

The primary advantage of Sharc is its simplicity. Sharc typically requires no changes to the operating system—so long as the operating system supports resource management mechanisms such as reservations or shares, Sharc can be built on top of commodity hardware and commodity operating systems. Sharc is not a cluster middleware; rather it operates in conjunction with the operating system to facilitate resource allocation on a cluster-wide basis. Applications continue to interact with the operating system and with one another using standard OS interfaces and libraries, while benefiting from the resource allocation features provided by Sharc. Sharc supports resource reservation both within a node and across nodes; the latter functionality enables aggregate reservations for distributed applications that span multiple nodes of the cluster (e.g., replicated Web servers). The resource management mechanisms employed by Sharc provide performance isolation to applications, and when desirable, allow distributed applications to dynamically share resources among resource principals based on their instantaneous needs. Finally, Sharc provides high availability of cluster resources by detecting and recovering from many types of failures.

In this chapter, we discuss the design requirements for resource management mechanisms in shared clusters and present techniques for managing two important cluster resources, namely CPU and network interface bandwidth. We discuss the implementation of our techniques on a cluster of Linux PCs and demonstrate its efficacy using an experimental evaluation. Our results show that Sharc can (i) provide predictable allocation of CPU and network interface bandwidth, (ii) isolate applications from one another, and (iii) handle a variety of failure scenarios. A key advantage of our approach is its efficiency—unlike

\(^1\)As an acronym, SHARC stands for Scalable Hierarchical Allocation of Resources in Clusters. As an abbreviation, Sharc is short for a shared cluster. We prefer the latter connotation.
previous approaches [9] that have polynomial time complexity, our techniques have complexity that is linear in the number of applications in the cluster. Our experiments show that this efficiency allows Sharc to easily scale to moderate size-clusters with 256 nodes running 100,000 applications.

The rest of this chapter is structured as follows. Section 6.2 lists the design requirements for resource management mechanisms in shared clusters. Section 6.3 presents an overview of the Sharc architecture, while Section 6.4 discusses the mechanisms and policies employed by Sharc. Section 6.6 describes our prototype implementation, while Section 6.7 presents our experimental results. We present related work in Section 6.8. Finally Section 6.9 presents our conclusions.

6.2 Resource Management in Shared Clusters: Requirements

Consider a shared cluster built using commodity hardware and software. Applications running on such a cluster could be centralized or distributed and could span multiple nodes in the cluster. Recall from Chapter 1 that the component of the cluster that manages resources (and capsules) on each individual node is referred to as the nucleus. The component of the cluster that coordinates various nuclei and manages resources on a cluster-wide basis is referred to as the control plane. Together, the control plane and the nuclei enable the cluster to share resources among multiple applications. The control plane and the nuclei should address the following requirements.

**Application Heterogeneity.** Applications running on a shared cluster will have diverse performance requirements. To illustrate, a third-party hosting platform can be expected to run a mix of applications such as game servers (e.g., Quake), vanilla Web servers, streaming media servers, e-commerce, and peer-to-peer applications. Similarly, shared clusters in workgroup environments will run a mix of scientific applications, simulations, and batch jobs. Observe that these applications have heterogeneous performance requirements. For instance, game servers need good interactive performance and thus low average response time.
times, scientific applications need high aggregate throughput, and streaming media servers require real-time performance guarantees.

In addition to heterogeneity across applications, there could be heterogeneity within each application. For instance, an e-commerce application might consist of capsules to service HTTP requests, to handle electronic payments, and to manage product catalogs. Each such capsule imposes a different performance requirement. Consequently, the resource management mechanisms in a shared cluster will need to handle the diverse performance requirements of capsules within and across applications.

**Resource Reservation.** Since the number of applications exceeds the number of nodes in a shared cluster, applications in this environment compete for resources. In such a scenario, soft real-time applications such as streaming media servers need to be guaranteed a certain level of service in order to meet timeliness requirements of streaming media. Resource guarantees may be necessary even for non-real-time applications, especially in environments where application owners are paying for resources. Consequently, a shared cluster should provide the ability to reserve resources for each application and enforce these allocations on a sufficiently fine time-scale.

Resources could be reserved either based on the aggregate needs of the application or based on the needs of individual capsules. In the former case, applications specify their aggregate resource needs but do not specify how these resources are to be partitioned among individual capsules. An example of such an application is a replicated Web server that runs on multiple cluster nodes—the aggregate throughput achieved by such an application is of greater concern than the throughput of any individual replica. At the other end of the spectrum are applications that need fine-grain control over the allocation to each individual capsule. An e-commerce application exemplifies this scenario, since each individual capsule (e.g., catalog database, payment handler) performs a different task and has different resource requirements. For such applications, the cluster should provide the flexibility of resource reservation on a per-capsule basis.
Finally, the ability of a capsule to trade resources with other peer capsules is also important. For instance, application capsules that are not utilizing their allocations should be able to temporarily lend resources, such as CPU cycles, to other needy capsules of that application. Since resource trading is not suitable for all applications, the cluster should allow applications to refrain from trading resources when undesirable.

**Capsule Placement and Admission Control.** A shared cluster that supports resource reservation for applications should ensure that sufficient resources exist on the cluster before admitting each new application. In addition to determining resource availability, the cluster also needs to determine *where* to place each application capsule—due to the large number of application capsules in shared environments, manual mapping of capsules to nodes may be infeasible. Admission control and capsule placement are interrelated tasks—both need to identify cluster nodes with sufficient unused resources to achieve their goals. Consequently, a shared cluster can employ a unified technique that integrates both tasks. Further, due to the potential lack of trust among applications in shared clusters, especially in third-party hosting environments, such a technique will also need to consider trust (or lack thereof) among applications, in addition to resource availability, while admitting applications and determining their placement onto nodes. We addressed these issues in Chapters 3 and 5.

**Application Isolation.** Third party applications running on a shared cluster could be untrusted or mutually antagonistic. Even in workgroup environments where there is more trust between users (and applications), applications could misbehave or get overloaded and affect the performance of other applications. Consequently, a shared cluster should isolate applications from one another and prevent untrusted or misbehaving applications from affecting the performance of other applications.

**Scalability and Availability.** Most commonly used clusters have sizes ranging from a few nodes to a few hundred nodes; each such node runs tens or hundreds of application capsules. Consequently, resource management mechanisms employed by a shared cluster
should scale to several hundred nodes running tens of thousands of applications (techniques that scale to very large clusters consisting of thousands or tens of thousands of nodes are beyond the scope of this thesis). A typical cluster with several hundred nodes will experience a number of hardware and software failures. Consequently, to ensure high availability, such a cluster should detect common types of failures and recover from them with minimal or no human intervention.

**Compatibility with Existing OS Interfaces.** Whereas the use of a middleware is one approach for managing resources in clustered environments [32, 35], this approach typically requires applications to use the interface exported by the middleware to realize its benefits. Sharc employs a different design philosophy. We are interested in exploring techniques that allow applications to use standard operating system interfaces and yet benefit from cluster-wide resource allocation mechanisms. Compatibility with existing OS interfaces and libraries is especially important in commercial environments such as hosting platforms where it is infeasible to require third-party applications to use proprietary or non-standard APIs. Such an approach also allows existing and legacy applications to benefit from these resource allocation mechanisms without any modifications. Our goal is to use commodity PCs running commodity operating systems as the building block for designing shared clusters. The only requirement we impose on the underlying operating system is that it support some notion of quality of service such as reservations [59, 68] or shares [50]. Many commercial and open-source operating systems such as Solaris [106], IRIX [98] and FreeBSD [19] already support such features.

Next we present the architecture, mechanisms, and policies employed by Sharc to address these requirements.

### 6.3 Sharc Architecture Overview

Figure 6.1(a) shows the Sharc architecture. In general, applications are oblivious of the nucleus and the control plane except at application startup time where they interact
with these components to reserve resources. Once resources are reserved, applications interact solely with the OS kernel and with one another, with no further interactions with Sharc. The control plane and the nucleus act transparently on the behalf of applications to determine allocations for individual capsules. To ensure compatibility with different OS platforms, these allocations are determined using OS-independent QoS parameters that are then mapped to OS-specific QoS parameters on each node. The task of enforcing these QoS requirements is left to the operating system kernel. This provides a clean separation of functionality between resource reservation and resource scheduling, with Sharc responsible for the former and the OS kernel for the latter.

In this chapter, we show how Sharc manages two important cluster resources, namely CPU and network interface bandwidth. As already mentioned, techniques for managing other resources such as memory and disk bandwidth are beyond the scope of this thesis.

\[\text{Figure 6.1.} \text{ Sharc architecture and abstractions. Figure (a) shows the overall Sharc architecture. Figure (b) shows a sample cluster-wide virtual hierarchy, a physical hierarchy on a node and the relationship between the two.}\]
6.3.1 The Control Plane

As shown in Figure 6.1(a), the Sharc control plane consists of a resource manager, an admission control and capsule placement module, and a fault-tolerance module. The admission control and capsule placement module performs two tasks: (i) it ensures that sufficient resources exist for each new application, and (ii) it determines the placement of capsules onto nodes in the cluster. We discussed these issues in detail in Chapters 3 and 5. Once an application is admitted into the system, the resource manager is responsible for ensuring that the aggregate allocation of each application and those of individual capsules are met. For those applications where trading of resources across capsules is permitted, the resource manager periodically determines how to reallocate resources unused by underutilized capsules to other needy capsules of that application. The fault-tolerance module is responsible for detecting and recovering from node and nucleus failures.

The key abstraction employed by the control plane to achieve these tasks is that of a cluster-wide virtual hierarchy (see Figure 6.1(b)). The virtual hierarchy maintains information about what resources are currently in use in the cluster and by whom. This information is represented hierarchically in the form of a tree. The root of the tree represents all the resources in the cluster. Each child represents an application in the cluster. Information about the number of capsules and the aggregate reservation for that application is maintained in each application node. Each child of an application node represents a capsule. A capsule node maintains information about the location of that capsule (i.e., the node on which the capsule resides), its reservation on that node, its current CPU and network usage, and the current allocation (the terms reservation and allocation are used interchangeably in this chapter). Note that the current allocation may be different from the initial reservation if the capsule borrows (or lends) resources from another capsule.
6.4 Sharc Mechanisms and Policies

In this section, we describe how Sharc enables capsules to trade resources with one another based on their current usage.

6.4.1 Resource Requirement Inference

The Sharc control plane employs the offline profiling technique described in Chapter 5 to determine the resource requirements of an application at application startup time. The control plane then determines whether sufficient resources exist in the cluster to service the new application and the placement of capsules onto nodes.

Sharc allows resources to be traded among the capsules of an application but not among the applications themselves. The reason for prohibiting inter-application resource trading is that Sharc has been developed primarily for environments such as commercial hosting platforms where applications negotiate contracts with the cluster seeking guarantees on resource availability. The application providers pay the cluster in return for these resource guarantees. Failure to meet these resource guarantees may imply loss in revenue for the cluster. By not allowing inter-application resource trading, we ensure that when an application needs to utilize all the resources that it had reserved (e.g., when a high number of requests arrive at a news site), it gets them. Systems that allow inter-application resource trading may fail to ensure this. We will see in Section 6.7 that although we do not allow inter-application resource trading explicitly, the use of work conserving resource schedulers on the nodes in the Sharc cluster ensures that any idle resources are automatically given to applications that need them in addition to their own shares (from this perspective, inter-application trading is “automatic” and implicit in Sharc).

Next, we describe how the Sharc control plane adjusts the resources allocated to capsules based on their usages.
6.4.2 Trading Resources based on Capsule Needs

Consider a shared cluster with \( n \) nodes that runs \( m \) applications. Let \( A_{ij} \) and \( U_{ij} \) denote the current allocation and current resource usage of the \( j^{th} \) capsule of application \( i \). \( A_{ij} \) and \( U_{ij} \) are defined to be the fraction of resource allocated and used, respectively, over a given time interval; \( 0 \leq U_{ij} \leq 1 \) and \( 0 < A_{ij} \leq 1 \). Recall also that \( R_{ij} \) is the fraction of the resource requested by the capsule at application startup time. The techniques presented in this section apply to both CPU and network bandwidth—the same technique can be used to adjust CPU and network bandwidth allocations of capsules based on their usages.

The nucleus on each node tracks the resource usage of all capsules over an interval \( I \) and periodically reports the corresponding usage vector \(< U_{i1j1}, U_{i2j2}, \ldots >\) to the control plane. Nuclei on different nodes are assumed to be unsynchronized, and hence, usage statistics from nodes arrive at the control plane at arbitrary instants (but approximately every \( I \) time units). Resource trading is the problem of temporarily increasing or decreasing the reservation of a capsule to match its usage, subject to aggregate reservation constraints for that application. Intuitively, the allocation of a capsule is increased if its past usage indicates it could use additional resources; the allocation of the capsule is decreased if it is not utilizing its reserved share and this unused allocation is then lent to other needy capsules.

To enable such resource trading, the control plane recomputes the instantaneous allocation of all capsules every \( I \) time units. To do so, it first computes the resource usage of a capsule using an exponential smoothing function.

\[
U_{ij} = \alpha \cdot U_{ij}^{new} + (1 - \alpha) \cdot U_{ij}
\]  
(6.1)

where \( U_{ij}^{new} \) is the usage reported by the nuclei and \( \alpha \) is a tunable smoothing parameter; \( 0 \leq \alpha \leq 1 \). Use of an exponentially smoothed moving average ensures that small transient changes in usages do not result in corresponding fluctuations in allocations, yielding a more stable system behavior. In the event a nucleus fails to report its usage vector (due to clock drift, failures or overload problems, all of which delay updates from the node), the control
plane conservatively sets the usages on that node to the initial reservations (i.e., $U_{ij}^{new} = R_{ij}$ for all capsules on that node). As explained in Section 6.5, this assumption also helps deal with possible failures on that node.

Our algorithm to recompute capsule allocations is based on three key principles: (1) Trading of resources among capsules should never violate the invariant $\sum_j A_{ij} = \sum_j R_{ij} = R_i$. That is, redistribution of resources among capsules should never cause the aggregate reservation of the application to be exceeded. (2) A capsule can borrow resources only if there is another capsule of that application that is under-utilizing its allocation (i.e., there exists a capsule $j$ such that $U_{ij} < A_{ij}$). Further there should be sufficient spare capacity on the node to permit borrowing of resources. (3) A capsule that lends its resources to a peer capsule is guaranteed to get it back at any time; moreover the capsule does not accumulate credit for the period of time it lends these resources. Resource trading is only permitted between capsules of the same application, never across applications.

Our re-computation algorithm proceeds in three steps. First, capsules that lent resources to other peer capsules but need them back reclaim their allocations. Second, allocations of under-utilized capsules are reduced appropriately. Third, any unutilized bandwidth is

---

3Accumulating credit for unused resources can cause starvation. For example, a capsule could sleep for an extended duration of time and use its accumulated credit to continuously run on the CPU, thereby starving other applications. Resource schedulers that allow accumulation of credit need to employ techniques to explicitly avoid this problem [12].
distributed (lent) to any capsules that could benefit from additional resources. Thus, the algorithm proceeds as follows.

**Step 0: Determine capsule allocations when resource trading is prohibited.** If resource trading is prohibited, then the allocations of all capsules of that application are simply set to their reservations ($\forall j, A_{ij} = R_{ij}$) and the algorithm moves on to the next application.

**Step 1: Needy capsules reclaim lent resources.** A capsule is said to have lent bandwidth if its current allocation is smaller than its reservation (i.e., allocation $A_{ij} <$ reservation $R_{ij}$). Each such capsule signals its desire to reclaim its due share if its resource usage equals or exceeds its allocation (i.e., usage $U_{ij} \geq$ allocation $A_{ij}$). Figure 6.2, Case 1 pictorially depicts this scenario.

For each such capsule, the resource manager returns lent bandwidth by setting

\[
A_{ij} = \min(R_{ij}, (1 + \epsilon_{ij}) \cdot U_{ij})
\]

(6.2)

where $\epsilon_{ij}$, $0 < \epsilon_{ij} < 1$, is a per-capsule positive constant that may be specified in the RSL and takes a default value if unspecified. In our experiments, we use a value of 0.1 for this parameter.

Rather than resetting the allocation of the capsule to its reservation, the capsule is allocated the smaller of its reservation and the current usage. This ensures that the capsule is returned only as much bandwidth as it needs (see Figure 6.2). The parameter $\epsilon_{ij}$ ensures that the new allocation is slightly larger than the current usage, enabling the capsule to (gradually) reclaim lent resources.

**Step 2: Underutilized capsules give up resources.** A capsule is said to be underutilizing resources if its current usage is strictly smaller than its allocation (i.e., usage $U_{ij} <$ allocation $A_{ij}$). Figure 6.2, Case 2 depicts this scenario.

Since the allocated resources are under-utilized, the resource manager should reduce the new allocation of the capsule. The exact reduction in allocation depends on the relationship of the current allocation and the reservation. If the current allocation is greater
reservation (Case 2(a) in Figure 6.2), then the new allocation is set to the usage (i.e., the allocation of a capsule that borrowed bandwidth but didn’t use it is reduced to its actual usage). On the other hand, if the current allocation is smaller the reservation (implying that the capsule is lending bandwidth), then any further reductions in the allocations are made gradually (case 2(b) in Figure 6.2). Thus,

$$A_{ij} = \begin{cases} U_{ij} & \text{if } A_{ij} \geq R_{ij} \\ (1 - \epsilon_{ij}) \cdot A_{ij} & \text{if } A_{ij} < R_{ij} \end{cases}$$

(6.3)

where $\epsilon_{ij}$ is a small positive constant, $0 < \epsilon_{ij} < 1$.

After examining capsules of all applications in Steps 1 and 2, the resource manager can then determine the unused resources for each application and the spare capacity on each node; the unused resources can then be lent to the remaining (needy) capsules of these applications.

It is possible to have two different values for the parameter $\epsilon_{ij}$ in Steps 2 and 3. Also, capsules that need to reclaim lent resources fast may be assigned a large $\epsilon_{ij}$ value. In this chapter, we report results from experiments in which all the capsules had the same $\epsilon_{ij}$ value of 0.1.

**Step 3: Needy capsules are lent additional (unused) bandwidth.** A capsule signals its need to borrow additional bandwidth if its usage exceeds its allocation (i.e., usage $U_{ij} \geq$ allocation $A_{ij}$). An additional requirement is that the capsule shouldn’t already be lending bandwidth to other capsules ($A_{ij} \geq R_{ij}$), else it would have been considered in Step 1. Figure 6.2, Case 3 depicts this scenario.

The resource manager lends additional bandwidth to such a capsule. The additional bandwidth allocated to the capsule is smaller of the spare capacity on that node and the unallocated bandwidth for that application. That is,

$$A_{ij} = A_{ij} + \min\left(\frac{1 - \sum_{j \in \text{node}} A_{ij}}{N_1}, \frac{R_i - \sum_{j=1}^{C_j} A_{ij}}{N_2}\right)$$

(6.4)
where $1 - \sum A_{ij}$ is the spare capacity on a node, $R_i - \sum_{j=1}^{C_i} A_{ij}$ is the unallocated bandwidth for the application, and $N_1$ and $N_2$ are the number of needy capsules on the node and for the application, respectively, all of whom desire additional bandwidth. Thus, the resource manager distributes unused bandwidth equally among all needy capsules.

An important point to note is that the spare capacity on a node or the unallocated bandwidth for the application could be negative quantities. This scenario occurs when the amount of resource reclaimed in Step 1 is greater than the unutilized bandwidth recouped in Step 2. In such a scenario, the net effect of Equation (6.4) is to reduce the total allocation of the capsule; this is permissible since the capsule was already borrowing bandwidth which is returned back.\textsuperscript{4} Thus, Equation (6.4) accounts for both positive and negative spare bandwidth in one unified step.

**Step 4:** *Ensure the invariant for the application.* After performing the above steps for all capsules of the application, the resource manager checks to ensure that the invariant

\[ \sum_j A_{ij} = \sum_j R_{ij} = R_i \]

holds. Additionally, $\sum_{j \in \text{node}} A_{ij} \leq 1$ should hold for each node. Under certain circumstances, it is possible that the total allocation may be slightly larger or smaller than the aggregate reservation for the application after the above three steps, or an increase in capsule allocation in Step 1 may cause the capacity of the node to be exceeded. These scenarios occur when capacity constraints on a node prevent redistribution of all unused bandwidth or the total reclaimed bandwidth is larger than the total unutilized bandwidth. In either case, the resource manager needs to adjust the new allocations to ensure these invariants. It can be shown that the bin-packing problem, which is NP-hard [46], reduces to the resource allocation problem that the resource manager solves using steps 3 and 4. Consequently, the resource manager has to resort to the following heuristic: it

\[ A_{ij} = \max(A_{ij}, R_{ij}) \]

\textsuperscript{4}For simplicity of exposition, we have omitted a couple of details in Eqs. (6.2), (6.3), and (6.4). First, these steps also involve ensuring that any lower bounds specified by the application on the capsules’ CPU and network bandwidth allocations are maintained. Second, after computing $A_{ij}$ in Eq. (6.4), the allocation is constrained as $A_{ij} = \max(A_{ij}, R_{ij})$ to prevent it from becoming smaller than $R_{ij}$ when the spare capacity is negative.
performs a small, constant number of additional scans of all capsules to increase or decrease their allocations slightly. This heuristic has been found to perform well in practice, yielding total allocations within 5% of the aggregate reservations for applications throughout our experimental study.

The newly computed allocations are then conveyed to each nucleus. The nucleus then maps these new allocations to OS-specific QoS parameters as discussed in Section 5.4.1 and conveys them to the OS scheduler.

A salient feature of the above algorithm is that it has two tunable parameters—the interval length $I$ and the smoothing parameter $\alpha$. As will be shown experimentally in Section 6.7, use of a small re-computation interval $I$ enables fine-grain resource trading based on small changes in resource usage, whereas a large interval focuses the algorithm on long-term changes in resource usage of capsules. Similarly, a large $\alpha$ causes the resource manager to focus on immediate past usages while computing allocations, while a small $\alpha$ smooths out the contribution of recent usage measurements. Thus, $I$ and $\alpha$ can be chosen appropriately to control the sensitivity of the algorithm to small, short-term changes in resource usage.

### 6.5 Failure Handling in Sharc

In this section, we describe the failure recovery techniques employed by Sharc. We consider three types of failures—nucleus failure, control plane failure, and node and link failures. The key principle employed by Sharc to recover from these failures is replication of state information—the virtual and the physical hierarchies replicate state information maintained by Sharc (see Figure 6.1(b)); this replication is intentional and enables reconstruction of state lost due to a failure.
6.5.1 Nucleus Failure

A nucleus failure occurs when the nucleus on a node fails but the node itself remains operational. It is the responsibility of the control plane to detect a nucleus failure. If a nucleus fails to report usage statistics for two consecutive intervals of duration $I$, then the fault tolerance module on the control plane is invoked to diagnose the problem. The fault-tolerance module first checks if the node is alive by sending echo messages to the node and then executing a remote script that examines the health of various operating system services. If the node is found to be healthy, the module then attempts to contact the nucleus. If the nucleus fails to respond, a nucleus failure is flagged.

The fault tolerance module then attempts to recover from the failure by starting a new nucleus (using a remote script that first cleans up the remnants of the previous nucleus and then starts up a new one). The control plane then synchronizes its state with the nucleus by (i) examining the virtual hierarchy to determine all capsules residing on that node, and (ii) reconstructing the physical hierarchy using this information. Since the kernel is unaffected by the nucleus failure, the QoS parameters maintained by the CPU scheduler for individual capsules are also unaffected. Note that the control plane disables resource trading for capsules on that node until failure recovery is complete; this is done by setting $A_{ij} = U_{ij} = R_{ij}$ for all resident capsules in the absence of usage reports from the node.

6.5.2 Control Plane Failure

A control plane failure is caused by the failure of the node running the control plane or the failure of the control plane itself. In either case, the control plane becomes unreachable from the nuclei.

In the event of a control plane failure, all nuclei run a leader election algorithm [108] to elect a new node to host the control plane. This is achieved as follows. Upon detecting an unreachable control plane, the fault tolerance module on the nucleus invites all other nuclei, using a broadcast message, to participate in a voting process to elect a new control
plane. Using a variant of the election algorithm described in [108], the nuclei then elect the node with the largest ID that has sufficient resources to run the control plane (the amount of resources required to run the control place is known a priori, since this is configured statically at system startup time based on the number of nodes and applications in the cluster). Each nucleus that receives this broadcast either agrees or declines to participate in the election—if the nucleus finds that the control plane is indeed unreachable, it agrees to participate, else it declines. If the election fails due to the lack of sufficient resources on nodes to run the control plane, then the need for human intervention is signaled. If the election succeeds, then the nucleus on the elected node starts up a new control plane with the appropriate reservation. The control plane then tries to recover the state of the virtual hierarchy—this is achieved by polling each nucleus for the physical hierarchy and creating a union of the physical hierarchies.

Under rare circumstances, the cluster might have two concurrent control planes running. This happens if the node running the control plane experiences a transient link failure but the node itself remains operational during the failure. Before the restoration of the link, the other nuclei could vote and start up a new control plane. Each control plane broadcasts a periodic heartbeat message and listens for similar messages from other control planes. If a second control plane is detected then a simple election algorithm is run to choose between the two—typically a younger control plane (i.e., one that was started later) is always given preference and the older control plane terminates itself.

6.5.3 Node and Link Failures

A node failure occurs when the operating system on a node crashes due to a software or hardware fault. A link failure occurs when the link connecting the node to the cluster interconnect fails. From the perspective of the control plane, both kinds of failures have the same effect—the node becomes unreachable. Whereas recovering from a node or link failure requires human intervention (to reboot the system or to repair faults), the control
plane can aid the recovery process. Upon detecting an unreachable node, the control plane can examine the virtual hierarchy and automatically reassign any capsule running on that node to other nodes in the cluster. The reassignment process involves admission control and capsule placement for the affected capsules. After determining the new mappings, the corresponding nuclei are notified and their physical hierarchies are updated. The affected application capsules can then be restarted on that node. Note that this process only helps determine a new set of nodes to run the capsules residing on the failed node; it does not help in recovering the state of the failed capsules—recovery of lost capsule state, if desirable, is left to the application and requires application-specific mechanisms such as check-pointing or logging [108].

6.5.4 Application Failure

Applications can fail in many ways. Whereas certain types of application failures are detectable (e.g., software crashes) to the cluster, many types of failures, such as deadlocks, are not. Consequently, our current design of Sharc does not deal with application failures; handling application failures is left to the owners of applications. In the future, we plan to examine how certain classes of application failures can be automatically detected and handled by the cluster.

6.6 Implementation Considerations and Complexity

The complexity of the mechanisms employed by the control plane and the nuclei is as follows.

**Resource trading.** The resource trading algorithm described in Section 6.4.2 proceeds one application at a time; capsules of an application need to be scanned a constant number of times to determine their new allocations (once for the first three steps and a constant number of times in Step 4). Thus, the overall complexity is linear in the number of capsules and takes $O(mk)$ time in a system with $m$ applications, each with $k$ capsules (total of $mk$
capsules). Each nucleus on a node participates in this process by determining resource usages of capsules and setting new allocations; the overhead of these tasks is two system calls every $I$ time units. Thus, the overall complexity of resource trading is linear in the number of capsules, which is more efficient than the polynomial time complexity of prior approaches [9].

**Communication overheads.** The number of bytes exchanged between the control plane and the various nuclei is a function of the total number of capsules in the system and the number of nodes. Although the precise overhead is $\beta \cdot n + \beta' \cdot mk$, it reduces to $O(mk)$ bytes in practice, since $mk \gg n$ in shared clusters ($\beta, \beta'$ are constants).

**Implementation details.** We have implemented a prototype of Sharc on a cluster on Linux PCs. We chose Linux as the underlying operating system since implementations of several commonly used QoS scheduling algorithms are available for Linux, allowing us to experiment with how capsule reservations in Sharc map onto different QoS parameters supported by these schedulers. Briefly, our Sharc prototype consists of two components—the control plane and the nucleus—that run as privileged processes in user space and communicate with one another on well-known port numbers. The implementation is multi-threaded and is based on Posix threads. The control plane consists of threads for (i) admission control and capsule placement, (ii) resource management and trading, (iii) communication with the nuclei on various nodes, and (iv) for handling nucleus and node failures. The resource specification language described in Section 6.4 is used to allocate resources to new applications, to modify resources allocated to existing applications, or to terminate applications and free up allocated resources. Each nucleus consists of threads that track resource usage, communicate with the control plane, and handle control plane failures. For the purposes of this chapter, we chose a Linux kernel that implements the H-SFQ proportional-share scheduler [88] and the leaky bucket rate regulator for allocating CPU and network interface bandwidth, respectively. This allows us to demonstrate that Sharc can indeed inter-operate with different kinds of kernel resource management mechanisms.
Next, we discuss our experimental results.

6.7 Experimental Evaluation

In this section, we experimentally evaluate our Sharc prototype using two types of workloads—a commercial third-party hosting platform workload and a research workgroup environment workload. Using these workloads and micro-benchmarks, we demonstrate that Sharc: (i) provides predictable allocation of CPU based on the specified resource requirements, (ii) can isolate applications from one another, (iii) can scale to clusters with a few hundred nodes running 100,000 capsules, and (iv) can handle a variety of failure scenarios. In what follows, we first describe the test-bed for our experiments and then describe our experimental results.

6.7.1 Experimental Setup

The testbed for our experiments consists of a cluster of Linux-based workstations interconnected by a 100 Mb/s switched ethernet. Our experiments assume that all machines are lightly loaded and so is the network. Unless specified otherwise, the Sharc control plane is assumed to run on a dedicated cluster node, as would be typical on a third-party hosting platform.

Our experiments involved two types of workloads. Our first workload is representative of a third-party hosting platform and consists of the following applications: (i) an e-commerce application consisting of a front-end Web server and a back-end relational database, (ii) a replicated Web server that uses Apache version 1.3.9, (iii) a file download server that supports download of large audio files, and (iv) a home-grown streaming media server that steams 1.5 Mb/s MPEG-1 files. Our second workload is representative of a research workgroup environment and consists of (i) Scientific, a compute-intensive scientific application that involved matrix manipulations, (ii) Summarize, an information retrieval
application, (iii) Disksim, a publicly-available compute-intensive disk simulator, and (iv) Make, an application build job that compiles the Linux 2.2.0 kernel using GNU make.

In all the experiments, the best-fit based placement algorithm described in Section 3.5.1 was used for placing the applications. A value of 0.1 was used for the parameter $\epsilon$ in all the experiments.

Next, we present the results of our experimental evaluation using these applications.

### 6.7.2 Predictable Resource Allocation and Application Isolation

Our first experiment demonstrates the efficacy of CPU and network interface bandwidth allocation in Sharc. We emulate a shared hosting platform environment with six applications. The placement of various application capsules and their CPU and network reservations are depicted in Table 6.1. Our first two applications are e-commerce applications with two capsules each—a front-end Web server and a back-end database server. For both applications, a fraction of requests received by the front-end Web server are assumed to trigger (compute-intensive) transactions in the database server (to simulate customer purchases on the ecommerce site). Our file download application emulates a music download site that supports audio file downloads; its workload is predominantly I/O intensive. Each streaming server application streams 1.5 Mb/s MPEG-1 files to multiple clients, while the Web server application services dynamic HTTP requests (which involves dynamic HTML generation via Apache’s PHP3 scripting). For the purposes of this experiment, we focus on the behavior of the first three applications, namely the two e-commerce applications and the file download server. The other three applications serve as the background load for our experiments.

To demonstrate the efficacy of CPU allocation in Sharc, we introduced identical, periodic bursts of requests in the two e-commerce applications. Resource trading was turned off for the first application and was permitted for the other. Observe that each burst triggers compute-intensive transactions in the database capsules. Since resource trading is permit-
<table>
<thead>
<tr>
<th>Applications</th>
<th>Capsules &amp; % (cpu,net) Reservations</th>
<th>Workload</th>
<th>Resource trading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecommerce 1 (EC1)</td>
<td>(10,5) (10,5) – –</td>
<td>Mixed</td>
<td>no</td>
</tr>
<tr>
<td>Ecommerce 2 (EC2)</td>
<td>– (10,5) (10,5) –</td>
<td>Mixed</td>
<td>yes</td>
</tr>
<tr>
<td>File download (FD)</td>
<td>(5,10) – (5,10) (5,10)</td>
<td>I/O intensive</td>
<td>yes</td>
</tr>
<tr>
<td>Streaming (S1)</td>
<td>– – (5,5) (5,5)</td>
<td>I/O intensive</td>
<td>no</td>
</tr>
<tr>
<td>Streaming (S2)</td>
<td>(5,5) (5,5) – –</td>
<td>I/O intensive</td>
<td>no</td>
</tr>
<tr>
<td>Dynamic HTTP server (WS)</td>
<td>– (20,5) – (20,5)</td>
<td>CPU intensive</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 6.1. Capsule Placement and Reservations

For EC2, the database capsule can borrow CPU cycles from the Web server capsule (which is I/O intensive) and use these borrowed cycles to improve transaction throughput. Since resource trading is prohibited in EC1, the corresponding database capsule is unable to borrow additional resources, which affects its throughput. Figure 6.3 plots the CPU allocations of the various capsules for the two applications and the throughput of both applications. The figure shows that trading CPU resources in EC2 allows it to process each burst faster than EC1. Specifically, trading CPU bandwidth among its capsules enables the database capsule of EC2 to finish the two bursts 85 sec and 25 sec faster, respectively, than the database capsule of EC1.

![Allocations for database servers](image1)
(a) Database capsules

![Allocations for web servers](image2)
(b) Web server capsules

![Progress of Database Transaction](image3)
(c) Progress of DB trans.

Figure 6.3. Predictable CPU allocation and trading. Figures (a) and (b) show the CPU allocation for the database server and the Web server capsules, Figure (c) shows the progress of the two bursts processed by these database servers.
Next we demonstrate the efficacy of network bandwidth allocation in Sharc. We consider the file download application that has three replicated capsules. To demonstrate the efficacy of resource trading, we send a burst of requests at $t = 70$ sec to the application; the majority of these requests go to the first capsule and the other two capsules remain underloaded. To cope with the increased load, Sharc reassigns unused bandwidth from the two under-loaded capsules to the overloaded capsule. We then send a second similar burst at $t = 160$ sec and observe a similar behavior. We send a third burst at $t = 300$ sec that is skewed towards the latter two capsules, leaving the first capsule with unused bandwidth. In this case, both overloaded capsules borrow bandwidth from the underutilized capsule; the borrowed bandwidth is shared equally among the two overloaded capsule. Finally, at $t = 500$ sec, a similar simultaneous burst is sent to the two capsules again with similar results. Figure 6.4 plots the network allocations of the three capsules and demonstrates the above behavior.

![Network Allocation Graphs](image)

(a) File download 1  (b) File download 2  (c) File download 3

**Figure 6.4.** Predictable network allocation and trading. Figure (a), (b) and (c) depict network allocations of capsules of the File download application.

An interesting feature exhibited by these experiments is related to the exponential smoothing parameter $\alpha$ mentioned in Section 6.4.2. For CPU bandwidth allocation, $\alpha$ was chosen to be 1.0 (no history), causing Sharc to reallocate bandwidth to the database capsule of EC2 very quickly. For network bandwidth allocation, $\alpha$ was chosen to be 0.5 resulting in a more gradual trading of network bandwidth among the capsules of the file download
Table 6.2. Capsule Placement and Reservations

<table>
<thead>
<tr>
<th>Applications</th>
<th>Arrival (min)</th>
<th>Capsules &amp; their Reservations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summarizer</td>
<td>1</td>
<td>N1 20% N2 30% N3 20% N4 —</td>
</tr>
<tr>
<td>Scientific</td>
<td>2.5</td>
<td>— 20% N2 30% N3 20% —</td>
</tr>
<tr>
<td>Disksim</td>
<td>36</td>
<td>N1 50% — N2 50% —</td>
</tr>
<tr>
<td>Make</td>
<td>37</td>
<td>— N1 50% — N2 50% —</td>
</tr>
</tbody>
</table>

application. Figures 6.3 and 6.4 depict this behavior. Thus, the value of $\alpha$ can be used to control the sensitivity of resource trading. One additional aspect of the above experiments is that Sharc isolates the remaining three applications, namely S1, S2, and WS, from the bursty workloads seen by the first three applications. This is achieved by providing each of these applications with a guaranteed resource share, which is unaffected by the bursty workloads of the ecommerce and file download applications.

6.7.3 Performance of a Scientific Application Workload

We conducted an experiment to demonstrate resource sharing among four applications representing a research workgroup environment. The placement of various capsules and their CPU reservations are listed in Table 6.2 (since these applications are compute-intensive, we focus only on CPU allocations in this experiment). As shown in the table, the first two applications arrive in the first few minutes and are allocated their reserved shares by Sharc. The capsule of the scientific application running on node 2 is put to sleep at $t = 25$ min, until $t = 38$ min. This allows the other capsules of that application on nodes 3 and 4 to borrow bandwidth unused by the sleeping capsule. The DiskSim application arrives at $t = 36$ min and the bandwidth borrowed on node 3 by the scientific application has to be returned (since the total allocation on the node reaches 100%, there is no longer any spare capacity on the node, preventing any further borrowing). Finally, two kernel builds startup at $t = 37$ min and are allocated their reserved shares. We measured the CPU allocations and the actual CPU usages of each capsule. Since there are ten capsule in this
experiment, for the sake of clarity, we only present results for the three capsules on node 3. As shown in Figure 6.5, the allocations of the three capsules closely match the above scenario. The actual CPU usages are initially larger than the allocations, since SFQ is a fair-share CPU scheduler and fairly redistributes unused CPU bandwidth on that node to runnable capsules (regardless of their allocations). Note that, at $t = 36$ min, the total allocation reaches 100%; at this point, there in no longer any unused CPU bandwidth that can be redistributed and the CPU usages closely match their allocations as expected. Thus, a proportional-share scheduler behaves exactly like a reservation-based scheduler at full capacity, while redistributing unused bandwidth in presence of spare capacity; this behavior is independent of Sharc, which continues to allocate bandwidth to capsules based on their initial reservations and instantaneous needs.

![Predictable allocation and resource trading](image)

(a) Summarizer (b) Scientific (c) Disksim

**Figure 6.5.** Predictable allocation and resource trading. Figure (a), (b) and (c) depict CPU usages and allocations of capsules residing on node 3.

### 6.7.4 Application Isolation in Sharc

We demonstrate application isolation in Sharc using a workload representative of a shared hosting platform. We use the following setup: (i) Node 1: mySQL server [79] (50% reservation), (ii) Node 2: Quake server (15% reservation), and (iii) Node 3: streaming media server (15% reservation). We used the benchmark suite distributed with the mySQL server to emulate a heavy database workload. The Quake and streaming media servers are lightly loaded at all times. We ran a replicated Web server (server A) with capsules on
nodes 1 and 2; the aggregate reservation was set to 80% (40% per capsule) and resource trading was permitted. A second replicated Web server (server $B$) was run on nodes 2 and 3 with a reservation of 20% per capsule; resource trading was turned off for this application. The following experiment demonstrates that Sharc can effectively isolate applications from one another in the presence of bursty Web workloads. We used the `httperf` tool [77] to send a burst of Web requests to server $A$ on node 1. The burst consists of requests for static Web pages as well as dynamically generated Web pages (Apache’s PHP3 scripting language is used for dynamic Web page generation). The burst causes the capsule on node 1 to borrow bandwidth from its peer on node 2, but does not affect the database server (see Figure 6.6(a)). Next we send a simultaneous burst to both capsules of server $A$; this causes the bandwidth borrowed on node 1 to be returned to node 2, but other applications are unaffected (see Figures 6.6(a) and (b)). Finally, we send a burst of Web requests to the capsule of server $B$ on node 3 (while maintaining a bursty workload on server $A$). Since resource trading is prohibited for server $B$, the capsule is unable to borrow bandwidth from its peer, even though the latter has bandwidth to spare. Again, the bursts do not affect other applications on the cluster (see Figure 6.6). This demonstrates that Sharc can effectively isolate applications from one another.

![Allocations on Node 1](image1.png)
![Allocations on Node 2](image2.png)
![Allocations on Node 3](image3.png)

(a) Node 1 (b) Node 2 (c) Node 3

Figure 6.6. Application Isolation in Sharc. The allocations of all capsules on the three nodes are shown (due to space constraints, CPU usages of these capsules have been omitted).
6.7.5 Impact of Resource Trading

To show that resource trading can help applications provide better quality of service to end-users, we conducted an experiment with a streaming video server. The server has two capsules, each of which streams MPEG-1 video to clients. We configure the server with a total network reservation of 8 Mb/s (4 Mb/s per capsule). At t=0, each capsule receives two requests each for a 15 minute long 1.5 Mb/s video and starts streaming the requested files to clients. At t=5 min, a fifth request for the video arrives and the first capsule is entrusted with the task of servicing the request. Observe that the capsule has a network bandwidth reservation of 4 Mb/s, whereas the cumulative requirements of the three requests is 4.5 Mb/s. We run the server with resource trading turned on, and then repeat the entire experiment with resource trading turned off. When resource trading is permitted, the first capsule is able to borrow unused bandwidth from the second capsule and service its two clients at their required data rates. In the absence of resource trading, the token bucket regulator restricts the total bandwidth usage to 4 Mb/s, resulting in late packet arrivals at the three clients. To measure the impact of these late arrivals on video playback, we assume that each client can buffer 4 seconds of video and that video playback is initiated only after this buffer is full. We then measure the number of playback discontinuities that occur due to a buffer underflow (after each such glitch, the client is assumed to pause until the buffer fills up again). Figure 6.7(a) plots the number of discontinuities observed by the clients of the first capsule in the two scenarios. The figure shows that when resource trading is permitted, there are very few playback discontinuities (the two observed discontinuities are due to the time lag in lending bandwidth to the first capsule—the control plane can react only at the granularity of the re-computation period $T$, which was set to 5 seconds in our experiment). In contrast, lack of resource trading causes a significant degradation in performance. Figures 6.7(b) and (c) show a 150 second long snapshot of the reception and playback of one of the streams provided by the first capsule (stream 2) for the two cases. Observe that the client is receiving data at nearly 1.5 Mbps when trading is allowed, but
only at about 1.4 Mbps in the absence of trading. As shown in Figure 6.7(b), there are repeated buffer underflows (represented by the horizontal portions of the plot) due to the bandwidth restrictions imposed by the rate regulator. Thus, the experiment demonstrates the utility of resource trading in improving application performance.

Figure 6.7. Impact of resource trading. Figure (a) shows the number of playback discontinuities seen by the three clients of the overloaded video server with and without the trading of network bandwidth. Figures (b) and (c) show a portion of the reception and playback of the second stream for the two cases.

6.7.6 Scalability of Sharc

To demonstrate the scalability of Sharc, we conducted experiments to measure the CPU and communication overheads imposed by the control plane and the nucleus. Observe that these overheads depend solely on the number of capsules and nodes in the system and are relatively independent of the characteristic of each capsule. The experiments reported in this section were conducted by running the control plane and the nuclei on 1 GHz Pentium III workstations with 256MB memory running RedHat Linux version 6.2.

6.7.6.1 Overheads Imposed by the Nucleus

We first measured the CPU overheads of the nucleus for varying loads; the usages were computed using the `times` system call and profiling tools such as `gprof`. We varied the number of capsules on a node from 10 to 10,000 and measured the CPU usage of the nucleus for different interval lengths. Figure 6.8(a) plots our results. As shown, the CPU
overheads decrease with increasing interval lengths. This is because the nucleus needs to
the query the kernel for CPU and network bandwidth usages and notify it of new alloca-
tions once in each interval $I$. The larger the interval duration, the less frequent are these
operations, and consequently, the smaller is the resulting CPU overhead. As shown in the
figure, the CPU overheads for 1000 capsules was less than 2% when $I = 5s$. Even with
10,000 capsules, the CPU usage was less than 4% when $I = 20s$ and less than 3% when
$I = 30s$.

Figure 6.8. Overheads imposed by the nucleus.

Figure 6.8(b) plots the system call overhead incurred by the nucleus for querying CPU
and network bandwidth usages and for notifying new allocations. As shown, the overhead
increases linearly with increasing number of capsules; the average overhead of these system
calls for 500 capsules was only 497μs and 297μs, respectively

Figure 6.8(c) plots the communication overhead incurred by the nucleus for varying
number of capsules. The communication overhead is defined to be the total number of
bytes required to report the usage vector to the control plane and receive new allocations
for capsules. As shown in the Figure, when $I = 30s$, the overhead is around 1300KB
for 10,000 capsules (43.3 KB/s) and is around 130KB per interval (4.3 KB/s) for 1000
capsules. Together these results show that the overheads imposed by the nucleus for most
realistic workloads is small in practice.
6.7.6.2 Control Plane Overheads

Next we conducted experiments to examine the scalability of the control plane. Since we were restricted by a five PC cluster, we emulated larger clusters by starting up multiple nuclei on each node and having each nucleus emulate all operations as if it controlled the entire node. Due to memory constraints on our machines, we didn’t actually start up a large number of applications but simulated them by having the nuclei manage the corresponding physical hierarchies and report varying CPU and network bandwidth usages. The nuclei on each node were unsynchronized and reported usages to the control plane every $I$ time units. From the perspective of the control plane, such a setup was no different from an actual cluster with a large number of nodes.

![Graphs showing CPU overhead, total busy time, and communication overhead for varying cluster sizes and interval lengths.](image)

**Figure 6.9.** Overheads imposed by the control plane.

Figure 6.9(a) plots the CPU overhead of the control plane for varying cluster sizes and interval lengths. The figure shows that a control plane running on a dedicated node can easily handle the load imposed by a 256 node cluster with 10,000 capsules (the CPU overhead was less than 16% when $I = 30s$). Figure 6.9(b) plots the total busy time for a 256 node cluster. The busy time is defined to the total CPU overhead plus the total time to send and receive messages to all the nuclei. As shown in the figure, the control plane can handle up to 100,000 capsules before reaching saturation when $I = 30s$. Furthermore, smaller interval lengths increase these overheads, since all control plane operations occur more frequently. This indicates that a larger interval length should be chosen to scale to
larger cluster sizes. Finally, Figure 6.9(c) plots the total communication overhead incurred by the control plane. Assuming $\mathcal{I} = 30$s, the figure shows that a cluster of 256 nodes running 100,000 capsules imposes an overhead of 3.46Mb/s, which is less than 4% of the available bandwidth on a FastEthernet LAN. The figure also shows that the communication overhead is largely dominated by the number of capsules in the system and is relatively independent on the number of nodes in the cluster.

### 6.7.7 Effect of Tunable Parameters

To demonstrate the effect of tunable parameters $\mathcal{I}$ and $\alpha$, we used the same set of workgroup applications described in Table 6.2. We put a capsule of the scientific application to sleep for a short duration. We varied the interval length $\mathcal{I}$ and measured its impact on the allocation of the capsule. As shown in Figure 6.10(a), increasing the interval length causes the CPU usage to be averaged over a larger measurement interval and diminishes the impact of the transient sleep on the allocation of the capsule (with a large $\mathcal{I}$ of 5min the effect of the sleep was negligibly small on the allocation). Next we put a capsule of Disksim to sleep for a few minutes and measured the effect of varying $\alpha$ on the allocations. As shown in Figure 6.10(b), use of a large $\alpha$ makes the allocation more sensitive to such transient changes, while a small $\alpha$ diminishes the contribution of transient changes in usage on the allocations. This demonstrates that an appropriate choice of $\mathcal{I}$ and $\alpha$ can be used to control the sensitivity of the allocations to short-term changes in usage.

### 6.7.8 Handling Failures

We used fault injection to study the effect on failures in Sharc. We ran 100 capsules of our workgroup applications on each of the four nodes and ran the control plane on a dedicated node and set $\mathcal{I} = 30$s. We killed the nucleus on various nodes at random time instants and measured the times to detect and recover from the failure. As shown in Table 6.3, the control plane was able to detect the failure in 80.7s (around $2.5 \times \mathcal{I}$). Once detected, starting up a new nucleus remotely took around 11.13 sec, while reconstructing the 100
node physical hierarchy and resynchronizing state with the nucleus took an additional 54ms (total recovery time was 11.18s). Next we studied the effect of node failures by halting the OS on nodes at arbitrary time instants. Detecting a node failure took around 79.27s; the control plane then attempted to reassign the 100 capsules on the failed node to other nodes. The resulting admission control, capsule placement and sending updates to nuclei took 55.1ms. In one case, we used a heavily loaded system, and as expected, the control plane signaled its inability to reassign capsules to other nodes due to lack of sufficient resources. Finally, we studied the impact of control plane failures. The control plane was run on a dedicated cluster node and was killed at random instants. The nuclei were able to detect the failure in 19.8s; running the election algorithm took 16.63s, starting up a new control plane took 9.45ms, while reconstruction of the 400 capsule virtual hierarchy took another 294.9ms (total recovery time was 17.41s). Our current prototype can only handle the case where a control plane running on a dedicated node fails; handling the failure of a control plane that runs on a node with active capsules is more complex and is not currently handled.

Figure 6.10. Impact of tunable parameters on capsule allocations.
<table>
<thead>
<tr>
<th>Failure type</th>
<th>Time to detect</th>
<th>Time to recover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nucleus</td>
<td>80.7s ± 5.91</td>
<td>11.18s ± 0.45</td>
</tr>
<tr>
<td>Node</td>
<td>79.27s ± 5.79</td>
<td>55.1ms ± 3.89</td>
</tr>
<tr>
<td>Control plane</td>
<td>19.85s ± 5.89</td>
<td>17.41s ± 1.99</td>
</tr>
</tbody>
</table>

Table 6.3. Failure Handling Times (with 95% Confidence Intervals)

6.8 Related Work

Resource management in shared platforms. Research on clustered environments has spanned a number of issues. Systems such as Condor have investigated techniques for harvesting idle CPU cycles on a cluster of workstations to run batch jobs [75]. Numerous middleware-based approaches for clustered environments have also been proposed [32, 35]. Finally, gang scheduling and co-scheduling efforts have investigated the issue of coordinating the scheduling of tasks in distributed systems [53]; however, this approach does not support resource reservation, which is a particular focus of our work.

Some recent efforts have focused on the specific issue of resource management in shared commodity clusters.

A proportional-share scheduling technique for a network of workstations was proposed in [12]. Whereas there are some similarities between their approach and Sharc, there are some notable differences. The primary difference is that their approach is based on fair relative allocation of cluster resources using proportional-share scheduling, whereas we focus on absolute allocation of resources using reservations (reservations and shares are fundamentally different resource allocation mechanisms). Even with an underlying proportional-share scheduler, Sharc can provide absolute bounds on allocations using admission control—the admission controller guarantees resources to applications and constrains the underlying proportional-share scheduler to fair redistribution of unused bandwidth (instead of fair allocation of the total bandwidth as in [12]). A second difference is that lending resources in [12] results in accumulation of credit that can be used by the task...
at a later time; the notion of lending resources in Sharc is inherently different—no credit is ever accumulated and trading is constrained by the aggregate reservation for an application.

Chase et al. present the design and implementation of Muse, an architecture for resource management in a hosting platform [26]. Muse uses an economic model for dynamic provisioning of resources to multiple applications. In the model, each application has a utility function which is a function of its throughput and reflects the revenue generated by the application. There is also a penalty that the application charges the system when its goals are not met. The system computes resource allocations by solving an optimization problem that maximizes the overall profit. Muse puts emphasis on energy as a driving resource management issue in server clusters. Like Sharc, Muse uses an exponential smoothing based predictor of future resource requirement. There are some important differences between Muse and Sharc. Muse allows resources to be traded between applications whereas Sharc does not. Sharc manages both CPU and network bandwidth. Muse manages only CPU, although we note that its resource management mechanism can be easily extended to manage network bandwidth.

The Cluster Reserves work at Rice University has also investigated resource allocation in server clusters [9]. The work assumes a large application running on a cluster, where the aim is to provide differential service to clients based on some notion of service class. This is achieved by providing fixed resource shares to application spanning multiple nodes, and dynamically adjusting the shares on each server based on the local resource usage. The approach uses resource containers [14] and employs a linear programming formulation for allocating resources, resulting in polynomial time complexity. In contrast, techniques employed by Sharc have complexity that is linear in the number of capsules. Further, Sharc can manage both CPU and network interface bandwidth, whereas Cluster Reserves only support CPU allocation (the technique can, however, be extended to manage network interface bandwidth as well).
**Resource management in highly distributed clusters.** Zhao and Karmacheti consider a model of hosting platforms different from that considered in our work [127]. They visualize future applications executing on platforms constructed by clustering multiple, autonomous distributed servers, with resource access governed by agreements between the owners and the users of these servers. They present an architecture for distributed, coordinated enforcement of resource sharing agreements based on an application-independent way to represent resources and agreements. In this work we have looked at hosting platforms consisting of servers in one location and connected by a fast network. However we also believe that distributed hosting platforms will become more popular and resource management in such systems will pose several challenging research problems.

### 6.9 Concluding Remarks

In this chapter, we argued the need for effective resource control mechanisms for sharing resources in commodity clusters. To address this issue, we presented the design of Sharc—a system that enables resource sharing in such clusters. Sharc depends on resource control mechanisms such as reservations or shares in the underlying OS and extends the benefits of such mechanisms to clustered environments. The control plane and the nuclei in Sharc achieve this goal by (i) supporting resource reservation for applications, (iii) providing performance isolation and dynamic resource allocation to application capsules, and (iv) providing high availability of cluster resources. Our evaluation of the Sharc prototype showed that Sharc can scale to 256 node clusters running 100,000 capsules. Our results demonstrated that a system such as Sharc can be an effective approach for sharing resources among competing applications in moderate size clusters.
CHAPTER 7
OVERLOAD MANAGEMENT

7.1 Introduction

7.1.1 Motivation

We have already discussed that the workload seen by Internet applications varies over multiple time-scales and often in an unpredictable fashion [1]. As argued in Chapter 4, certain workload variations such as time-of-day effects are easy to predict and handle by appropriate capacity provisioning. Other variations such as flash crowds are often unpredictable. On September 11th 2001, for instance, the workload on a popular news Web site increased by an order of magnitude in thirty minutes, with the workload doubling every seven minutes in that period [1]. The load on e-commerce retail Web sites can increase dramatically during the final days of the popular holiday season. Similarly, the load on online brokerage Web sites can be several times greater than the average load during an unexpected market crash.

In this chapter, we focus on handling extreme overloads seen by Internet applications. Informally, an extreme overload is a scenario where the workload unexpectedly increases by up to an order of magnitude in a few tens of minutes. Our goals are (i) to design a system that remains operational even in the presence of an extreme overload and even when the incoming request rate is several times greater than system capacity, and (ii) to maximize the revenue due to the requests serviced by the application during such an overload. A hosting platform can take one or more of three actions during an overload: (i) add capacity to the application by allocating idle or under-used servers, (ii) turn away excess requests...
and preferentially service only “important” requests, and (iii) degrade the performance of admitted requests in order to service a larger number of aggregate requests.

The first two approaches have been studied in the literature. The first approach involves dynamic provisioning to match application capacity to the workload demand and has been addressed by us (Chapter 4) and others [26, 76, 91]. The second approach involves policing in the form of admission control, which limits the number of admitted requests so that the contracted performance guarantees are met [29, 42, 117, 123]. The notion of providing preferential treatment to “important” requests has also been studied (e.g., by giving higher priority to certain requests, such as those involving financial transactions [18]). Last, the notion of gracefully degrading application performance with increasing loads, while intuitively appealing, has not been studied from the perspective of extreme overloads.

We argue that a comprehensive approach for handling extreme overloads should involve a combination of all of the above techniques. A hosting platform should, whenever possible, allocate additional capacity to an application in order to handle increased demands. The platform should degrade performance in order to temporarily increase effective capacity during overloads. When no capacity addition is possible or when the SLA does not permit any further performance degradation, the platform should turn away excess requests. While doing so, the platform should preferentially admit important requests and turn away less important requests to maximize overall revenue. For instance, small requests may be preferred over large requests, or financial transactions may be preferred over casual browsing requests.

It is important to note that such a comprehensive approach to handling severe overloads involves more than the implementation of separate mechanisms to achieve each of the above goals. Mechanisms such as dynamic provisioning and admission control can be coupled in useful and non-trivial ways to further improve the handling of extreme overloads. For instance, the admission controller can pro-actively invoke the dynamic provisioning mechanism when the request drop rate exceeds a certain threshold. The dynamic
provisioning mechanism in turn can provide useful information to the admission controller regarding the provisioned capacity so that the latter can set appropriate performance thresholds for admitted requests. Such an integration of mechanisms can enhance the ability of the platform to handle overloads.

An orthogonal goal for the hosting platform is robustness under severe overloads. Robustness—the ability to remain operational under overloads—requires the hosting platform to be both extremely agile and efficient. Agility requires a quick response in the face of a sudden workload spike. Efficiency requires the above-mentioned mechanisms, and in particular the admission controller, to have very low overheads. Since an extreme overload may involve request rates that are up to an order of magnitude greater than the currently allocated capacity, the admission controller must be able to quickly examine requests and discard a large fraction of these requests, when necessary, with minimal overheads.

Whereas prior approaches for handling overloads have considered individual mechanisms such as provisioning and admission control, in this thesis, we focus on an integrated approach, with a particular emphasis on handling extreme overloads.

7.1.2 Research Contributions of this Chapter

We describe the aspects of our hosting platform concerned with handling extreme overloads in Internet applications. Our approach differs from past work in three significant respects.

First, since an extreme overload may involve request rates that are an order of magnitude greater than the currently allocated capacity, the admission controller must be able to quickly examine requests and discard a large fraction of these requests, when necessary, with minimal overheads. Thus, the efficiency of the admission controller is important during heavy overloads. To address this issue, we propose very low overhead admission control mechanisms that can scale to very high request rates under overloads. Past work on admission control [29, 42, 117, 123] has focused on the mechanics of policing and did not
specifically consider the scalability of these mechanisms. In addition to imposing very low overheads, our mechanisms can preferentially admit important requests during an overload and transparently trade-off the accuracy of their decision making with the intensity of the workload. The trade-off between accuracy and efficiency is another contribution of our work and enables our implementation to scale to incoming rates of up to a few tens of thousands of requests/s. (not all of these requests are necessarily admitted and serviced; the admitted fraction depends on the available capacity).

Second, our platform has the ability to not only vary the number of servers allocated to an application but also other components such as the admission controller and the load balancing switches. Dynamic provisioning of the latter components has not been considered in prior work.

Last, our work demonstrates that dynamic provisioning and admission control can be coupled in useful ways to enhance the ability of the platform in handling extreme overloads. For instance, the admission controller can pro-actively invoke dynamic provisioning when the request drop rate exceeds a certain threshold, and the provisioning mechanisms can provide useful information to the admission controller for policing requests. Past work on admission control [29, 42, 117, 123] and dynamic provisioning [26, 91] considered each technique in isolation and did not study the impact of such couplings.

We have implemented our overload control mechanisms in our prototype Linux hosting. We demonstrate the effectiveness of our integrated overload control approach via an experimental evaluation. Our results show that (i) preferentially admitting requests based on importance and size can increase the utility and effective capacity of an application, (ii) our admission control is highly scalable and remains functional even for arrival rates of a few thousand requests/s, and (iii) our solution based on a combination of admission control and dynamic provisioning is effective in meeting response time targets and improving platform revenue.
Figure 7.1. The Hosting Platform Architecture.

7.1.3 Organization

The rest of this chapter is organized as follows. Section 7.2 provides an overview of the proposed system. Sections 7.3 and 7.4 describe the mechanisms that constitute our overload management solution. Section 7.5 describes the implementation of our prototype. In Section 7.6 we present the results of our experimental evaluation. Section 7.7 presents related work and Section 7.8 concludes this chapter.

7.2 System Overview

In this section, we present the system model for our hosting platform and the service-level agreement assumed in our work.

7.2.1 Hosting Platform Architecture

We show the hosting platform architecture in Figure 7.1. We assume a dedicated hosting model in this chapter.

Each application running on the platform is assigned one or more sentries. A sentry guards the servers assigned to an application and is responsible for two tasks. First, the sentry polices all requests to an application’s server pool—incoming requests are subjected to admission control at the sentry to ensure that the contracted performance guarantees are
met; excess requests are turned away during overloads. Second, each sentry implements a layer-7 switch that performs load balancing across servers allocated to an application. Since there has been substantial research on load balancing techniques for clustered Internet applications [83], we do not consider load balancing techniques in this work.

Whereas a single sentry suffices for small applications, large applications require multiple sentries, since a single sentry server will become a bottleneck when guarding a large number of servers. Just as the number of servers allocated to an application vary with the load, our hosting platform can dynamically vary the number of sentries depending on the incoming request rate (and the corresponding load on the sentries). When a sentry is assigned or deallocated, the application’s server pool is repartitioned and each remaining sentry is assigned responsibility for a mutually exclusive subset of nodes. Each sentry then independently performs admission control and load balancing on arriving requests, thereby collectively maintaining the SLA for the application as a whole. A round-robin DNS scheme is used to partition (and loosely balance) the incoming requests across multiple sentries.

As before, the control plane is responsible for dynamic provisioning of servers and sentries in individual applications. It tracks the resource usages on servers, as reported by the nuclei, and determines the resources (in terms of the number of servers and sentries) to be allocated to each application. The control plane runs on a dedicated server and its scalability is not of concern in the design of our platform.

7.2.2 Service-level Agreement

Given an Internet application, we assume that the application specifies the desired performance guarantees in the form of a service level agreement (SLA). An SLA provides a description of the QoS guarantees that the platform will provide to the application. The SLA we consider in our work is defined as follows:
### Table 7.1. A sample service-level agreement.

<table>
<thead>
<tr>
<th>Arrival rate</th>
<th>Avg. resp. time for admitted requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1000</td>
<td>1 sec</td>
</tr>
<tr>
<td>1000-10000</td>
<td>2 sec</td>
</tr>
<tr>
<td>&gt; 10000</td>
<td>3 sec</td>
</tr>
</tbody>
</table>

The SLA specifies the revenue that is generated by each request that meets its response time target. Table 7.1 illustrates an example SLA.

Each Internet application consists of \( L (L \geq 1) \) request classes: \( C_1, \ldots, C_L \). Each class has an associated revenue that an admitted request yields—requests of class \( C_1 \) are assumed to yield the highest revenue and those of \( C_L \) the least. The number of request classes \( L \) and the function that maps requests to classes is application-dependent. To illustrate, a vanilla Web server may define two classes and may map all requests smaller than a certain size \( s \) to class \( C_1 \) and larger requests to \( C_2 \). In contrast, an online brokerage Web site may define three classes and may map financial transactions to \( C_1 \), other types of requests such as balance inquiries to \( C_2 \), and casual browsing requests from non-customers to \( C_3 \). An application’s SLA may also specify lower bounds on the request arrival rates that its classes should always be able to sustain.

### 7.3 Sentry Design

In this section, we describe the design of a sentry. The sentry is responsible for two tasks—request policing and load balancing. As indicated earlier, the load balancing tech-
nique used in the sentry is not a focus of this work, and we assume the sentry employs a layer-7 load balancing algorithm such as the one proposed by Pai et al. [83]. The first key issue that drives the design of the request policer is to maximize the revenue yielded by the admitted requests while providing the following notion of class-based differentiation to the application: each class should be able to sustain the minimum request rate specified for it in the SLA. Given our focus on extreme overloads, the design of the policer is also influenced by the second key issue of scalability—ensuring very low overhead admission control tests in order to scale to very high request arrival rates seen during overloads. This section elaborates on these two issues.

7.3.1 Request Policing Basics

The sentry maps each incoming request to one of the classes \(C_1, \ldots, C_L\). The policer needs to guarantee to each class an admission rate equal to the minimum sustainable rate desired by it (recall our SLA from Section 7.2). It does so by implementing leaky buckets, one for each class, that admit requests confirming to these rates. Requests confirming to these leaky buckets are forwarded to the application. Leaky buckets can be implemented very efficiently, so determining if an incoming request confirms to a leaky bucket is an inexpensive operation. Requests in excess of these rates undergo further processing as follows. Each class has a queue associated with it (see Figure 7.2); incoming requests are appended to the corresponding class-specific queue. Requests within each class can be processed either in FIFO order or in order of their service times. In the former case, all requests within a class are assumed to be equally important, whereas in the latter case smaller requests are given priority over larger requests within each class. Admitted requests are handed to the load balancer, which then forwards them to one of the servers in the application’s server pool.

The policer incorporates the following two features in its processing of the requests that are in excess of the guaranteed rates to maximize revenue.
(1) The policer introduces different amounts of delay in the processing of newly arrived requests belonging to different classes. Specifically, requests of class $C_i$ are processed by the policer once every $d_i$ time units ($d_1 = 0 \leq d_2 \leq \ldots \leq d_L$); requests arriving during successive processing instants wait for their turn in their class-specific queues. These delay values, determined periodically, are chosen to reduce the chance of admitting less important requests into the system when they are likely to deny service to more important requests that arrive shortly thereafter. In Section Appendix B we show how to pick these delay values such that the probability of a less important request being admitted into the system and denying service to a more important request that arrives later remains sufficiently small.

(2) The policer processes queued requests in the decreasing order of importance—requests in $C_1$ are subjected to the admission control test first, and then those in $C_2$ and so on. Doing so ensures that requests in class $C_i$ are given higher priority than those in class $C_j$, $j > i$. The admission control test—which is described in detail in the next section—admits requests so long as the system has sufficient capacity to meet the contracted SLA. Note that, if requests in a certain class $C_i$ fail the admission control test, all queued requests in less important classes can be rejected without any further tests.

Observe that the above admission control strategy meets one of our two goals—it preferentially admits only important requests during an overload and turns away less important requests. However, the strategy needs to invoke the admission control test on each individual request, resulting in a complexity of $O(r)$, where $r$ is the number of queued up requests. Further, when requests within a class are examined in order of service times instead of FIFO, the complexity increases to $O(r \cdot \log(r))$ due to the need to sort requests. Since the incoming request rate can be substantially higher than capacity during an extreme overload, running the admission control test on every request or sorting requests prior to
admission control may be simply infeasible. Consequently, in what follows, we present two strategies for very low overhead admission control that scale well during overloads.

We note that a newly arriving request imposes two types of computational overheads on the policer—(i) protocol processing and (ii) the admission control test itself. Clearly, both these components need to scale for effective handling of overloads. When protocol processing starts becoming bottleneck, we respond by increasing the number of sentries guarding the overloaded application—a technique that we describe in detail in Section 7.4.2. In this section we present techniques to deal with the scalability of the admission control test.

7.3.2 Efficient Batch Processing

One possible approach for reducing the policing overhead is to process requests in batches. Request arrivals tend to be very bursty during severe overloads, with a large number of requests arriving in a short duration of time. These requests are queued up in the appropriate class-specific queues at the sentry. Our technique exploits this feature by conducting a single admission control test on an entire batch of requests within a class, instead of doing so for each individual request. Such batch processing can amortize the admission control overhead over a larger number of requests, especially during overloads.

To perform efficient batch-based admission control, we define $b$ buckets within each request class. Each bucket has a range of request service times associated with it. The sentry estimates the service time of a request and then hashes it into the bucket corresponding to that service time. To illustrate, a request with an estimated service time in the range $(0, s_1]$ is hashed to bucket 1, that with service time in the range $(s_1, s_2]$ to bucket 2, and so on. By defining an appropriate hashing function, hashing a request to a bucket can be implemented efficiently as a constant time operation.

Bucket-based hashing is motivated by two reasons. First, it groups requests with similar service times and enables the policer to conduct a single admission control test by assum-
ing that all requests in a bucket impose similar service demands. Second, since successive buckets contain requests with progressively larger service times, the technique implicitly gives priority to smaller requests. Moreover, no sorting of requests is necessary—the hashing implicitly “sorts” requests when mapping them into buckets.

![Diagram](https://via.placeholder.com/150)

**Figure 7.2.** Working of the sentry. First, the class a request belongs to is determined. If the request confirms to the leaky bucket for its class, it is admitted to the application without any further processing. Otherwise, it is put into its class-specific queue. The admission control processes the requests in various queues at frequencies given by the class-specific delays. A request is admitted to the application if there is enough capacity, else it is dropped.

When the admission control is invoked on a request class, it considers each non-empty bucket in that class and conducts a single admission control test on all requests in that bucket (i.e., all requests in a bucket are treated as a batch). Consequently, no more than $b$ admission control tests are needed within each class, one for each bucket. Since there are $L$ request classes, this reduces the admission control overhead to $O(b \cdot L)$, which is substantially smaller than the $O(r)$ overhead for admitting individual requests.

Having provided the intuition behind batch-based admission control, we discuss the hashing process and the admission control test in detail. In order to hash a request into a bucket, the sentry must first estimate the *inherent service time* of that request. The inherent service time of a request is the time needed to service the request on a lightly loaded server.
(i.e., when the request does not see any queuing delays). The inherent service time of a request $R$ is defined to be

$$S_{\text{inherent}} = R_{\text{cpu}} + \alpha \cdot R_{\text{data}}$$  \hspace{1cm} (7.2)$$

where $R_{\text{cpu}}$ is the total CPU time needed to service $R$, $R_{\text{data}}$ is the IO time of the request (which includes the time to fetch data from disk, the time the request is blocked on a database query, the network transfer time, etc.), and $\alpha$ is an empirically determined constant. The inherent service time is then used to hash the request into an appropriate bucket—the request maps to a bucket $i$ such that $s_i \leq S_{\text{inherent}} \leq s_{i+1}$.

The specific admission control test for each batch of requests within a bucket is as follows. Let $\beta$ denote the batch size (i.e., the number of requests) in a bucket. Let $Q$ denote the estimated queuing delay seen by each request in the batch. The queuing delay is the time the request has to wait at a server before it receives service; the queuing delay is a function of the current load on the server and its estimation is discussed in Section 7.3.5. Let $\eta$ denote the average number of requests (connections) that are currently being serviced by a server in the application’s server pool. Then the $\beta$ requests within a batch are admitted if and only if the sum of the queuing delay seen by a request and its actual service time does not exceed the contracted SLA. That is,

$$Q + \left( \eta + \left\lfloor \frac{\beta}{n} \right\rfloor \right) \cdot S \leq R_{\text{sla}}$$  \hspace{1cm} (7.3)$$

where $S$ is the average inherent service time of a request in the batch, $n$ is the number of servers allocated to the application, and $R_{\text{sla}}$ is the desired response time. The term $\left( \eta + \left\lfloor \frac{\beta}{n} \right\rfloor \right) \cdot S$ is an estimate of the actual service time of the last request in the batch, and is determined by scaling the inherent service time $S$ by the server load—which is the number of the requests currently in service, i.e., $\eta$, plus the number of requests from the batch
that might be assigned to the server i.e., $\left[^{\beta}_{n}\right]$.\(^1\) Rather than actually computing the mean inherent service time of the request in a batch, it is approximated as $S = (s_i + s_{i+1})/2$, where $(s_i, s_{i+1}]$ is the service time range associated with the bucket.

As indicated above, the admission control is invoked for each class periodically—once every $d_i$ time units for newly arrived requests of class $C_i$. The invocation is more frequent for important classes and less frequent for less important classes, that is, $d_1 = 0 \leq d_2 \leq \ldots \leq d_L$. Since a request may wait in a bucket for up to $d_i$ time units before admission control is invoked for its batch, the above test is modified as

$$Q + \left(n + \left[^{\beta}_{n}\right]\right) \cdot S \leq R_{sla} - d_i \tag{7.4}$$

In the event this condition is satisfied, all requests in the batch are admitted into the system. Otherwise requests in the batch are dropped.

Observe that introducing these delays into the processing of certain requests does not cause a degradation in the response time of the admitted requests because they now undergo a more stringent admission control test as given by (7.4). However, these delays would have the effect of reducing the application’s throughput when it is not overloaded. Therefore, these delays should be changed dynamically as workloads of various classes change. In particular, they should tend to 0 when the application has sufficient capacity to handle all the incoming traffic. We discuss in Appendix B how these delay values are dynamically updated. Techniques for estimating parameters such as the queuing delay, inherent service time, and the number of existing connections are discussed in Section 7.3.5.

\(^1\)Note that we have made the assumption of perfect load balancing in the admission control test (7.3). One approach for capturing load imbalances can be to scale $\eta$ and $n$ by suitably chosen skew factors. These skew factors can be based on measurements of the load imbalance among the replicas of the application.
7.3.3 Scalable Threshold-based Policing

We now present a second approach to further reduce the policing overhead. Our technique trades efficiency of the policer for accuracy and reduces the overhead to a few arithmetic operations per request. The key idea behind this technique is to periodically precompute the fraction of arriving requests that should be admitted in each class and then simply enforce these limits without conducting any additional per-request tests. Again, incoming requests are first classified and undergo an inexpensive test to determine if they confirm to the leaky buckets for their classes. Confirming requests are admitted to the application without any further tests. Other requests undergo a more lightweight admission control test that we describe next.

Our technique uses estimates of future arrival rates and service demands in each class to compute a threshold, which is defined to be a pair (class $i$, fraction $p_{\text{admit}}$). The threshold indicates that all requests in classes more important than $i$ should be admitted ($p_{\text{admit}} = 1$), requests in class $i$ should be admitted with probability $p_{\text{admit}}$, and all requests in classes less important than $i$ should be dropped ($p_{\text{admit}} = 0$). We determine these parameters based on observations of arrival rates and service times in each classes over periods of moderate length (we use periods of length 15 sec). Denoting the arrival rates to classes $1, \ldots, L$ by $\lambda_1, \ldots, \lambda_L$ and the observed average service times by $s_1, \ldots, s_L$, the threshold $(i, p_{\text{admit}})$ is computed such that

\begin{equation}
\sum_{j=1}^{i} \lambda_j s_j \geq 1 - \sum_{j=1}^{L} \lambda_j^{\min} s_j \tag{7.5}
\end{equation}

and

\begin{equation}
p_{\text{admit}} \cdot \lambda_i s_i + \sum_{j=1}^{i-1} \lambda_j s_j < 1 - \sum_{j=1}^{L} \lambda_j^{\min} s_j \tag{7.6}
\end{equation}

where $\lambda_j^{\min}$ denotes the minimum guaranteed rate for class $j$.
Thus, admission control now merely involves applying the inexpensive classification function on a new request to determine its class, determining if it confirms to the leaky bucket for that class (also a lightweight operation), and then using the equally lightweight thresholding function (if it does not confirm to the leaky bucket) to decide if it should be admitted. Observe that this admission control requires estimates of per-class arrival rates. These rates are clearly difficult to predict during unexpected overloads. However, it is possible to react fast by updating our estimates of the arrival rates frequently. Our implementation of threshold-based policing estimates arrival rates by computing exponentially smoothed averages of arrivals over 15 sec periods. We will demonstrate the efficacy of this policer in an experiment in Section 7.6.3.

The threshold-based and the batch-based policing strategies need not be mutually exclusive. The sentry can employ the more accurate batch-based policing so long as the incoming request rate permits one admission control test per batch. If the incoming rate increases significantly, the processing demands of the batch-based policing may saturate the sentry. In such an event, when the load at the sentry exceeds a threshold, the sentry can trade accuracy for efficiency by dynamically switching to a threshold-based policing strategy. This ensures greater scalability and robustness during overloads. The sentry reverts to the batch-based admission control when the load decreases and stays below the threshold for a sufficiently long duration. We would like to note that several existing admission control algorithms such as [42, 61, 123] (discussed in Section 7.7) are based on dynamically set thresholds such as admission rates and can be implemented as efficiently as our threshold-based admission control. The novel feature in our approach is the flexibility to trade-off the accuracy of admission control for its computational overhead depending on the load on the sentry.
7.3.4 Analysis of the Policer

In Appendix B we show how the sentry can, under certain assumptions, compute the delay values for various classes based on online observations.

7.3.5 Online Parameter Estimation

The batch-based and threshold-based policing algorithms require estimates of a number of system parameters. These parameters are estimated using online measurements. The nuclei running on the servers and sentries collectively gather and maintain various statistics needed by the policer. The following statistics are maintained:

- *Arrival rate* \( \lambda_i \): Since each request is mapped onto a class at the sentry, it is trivial to use this information to measure the incoming arrival rates in each class.

- *Queuing delay* \( Q \): The queuing delay incurred by a request is measured at the server. The queuing delay is estimated as the difference between the time the request arrives at the server and the time it is accepted by the HTTP server for service (we assume that the delay incurred at the sentry is negligible). The nuclei can measure these values by appropriately instrumenting the operating system kernel. The nuclei periodically report the observed queuing delays to the sentry, which then computes the mean delays across all servers in the application’s pool.

- *Number of requests in service* \( \eta \): This parameter is measured at the server. The nuclei track the number of active connections serviced by the application and periodically report the measured values to the sentry. The sentry then computes the mean of the reported values across all servers for the application.

- *Request service time* \( s \): This parameter is also measured at the server. The actual service time of a request is measured as the difference between the arrival time at the server and the time at which the last byte of the response is sent. The measurement of the inherent service time is more complex. Doing so requires instrumentation of
the OS kernel and some instrumentation of the application itself. This instrumentation the nucleus to compute the CPU processing time for a request as well as the duration for which the requested is blocked on I/O. Together, these values determine the inherent service time (see Equation (7.2)).

- **Constant $\alpha$:** The constant $\alpha$ in Equation (7.2) is measured using offline measurements on the servers. We execute several requests with different CPU demands and different-sized responses under light load conditions and measure their execution times. We also compute the CPU demands and the I/O times as indicated above. The constant $\alpha$ is then estimated as the value that minimizes the difference between the actual execution time and the inherent service time in Eq. (7.2).

The sentry uses past statistics to estimate the inherent service time of an incoming request in order to map it onto a bucket. To do so, the sentry uses a hash table for maintaining the usage statistics for the requests it has admitted so far. Each entry in this table consists of the requested URL (which is used to compute the index of the entry in the table) and a vector of the resource usages for this request as reported by the various servers. Requests for static content possess the same URL every time and so always map to the same entry in the hash table. The URL for requests for dynamic content, on the other hand, may change (e.g. the arguments to a script may be specified as part of the URL). For such requests, we get rid of the arguments and hash based on the name of the script invoked. The resource usages for requests that invoke these scripts may change depending on the arguments. We maintain exponentially decayed averages of their usages.

### 7.4 Capacity Provisioning

Policing mechanisms may turn away a significant fraction of the requests during overloads. In such a scenario, an increase in the effective application capacity is necessary to reduce the request drop rate. The control plane implements dynamic provisioning to vary
the number of allocated servers based on application workloads. The application’s server pool is increased during overloads by allocating servers from the free pool or by reassigning under-used servers from other applications. The control plane can also dynamically provision sentry servers when the incoming request rate imposes significant processing demands on the existing sentries. The rest of this section discusses techniques for dynamically provisioning servers and sentries.

7.4.1 Model-based Provisioning for Applications

We employ the provisioning technique described in Chapter 4 in our hosting platform. Recall that this technique is based on a combination of a predictive provisioning technique based on the queuing-theoretic model presented in Chapter 2 and a reactive provisioning technique to handle errors in prediction and flash crowds.

Recall that our SLA permits degraded response time targets for higher arrival rates. The provisioning mechanism may degrade the response time to the extent permitted by the SLA, add more capacity, or a bit of both. The optimization drives these decisions, and the resulting target response times are conveyed to the request policers. Thus, these interactions enable coupling of policing, provisioning, and adaptive performance degradation.

7.4.2 Sentry Provisioning

In general, allocation and deallocation of sentries is significantly less frequent than that of servers. Further, the number of sentries needed by an application is much smaller than the number of servers running it. Consequently, a simple provisioning scheme suffices for dynamically varying the number of sentries assigned to an application. Our scheme uses the CPU utilization of the existing sentry servers as the basis for allocating additional sentries (or deallocating active sentries). If the utilization of a sentry stays in excess of a pre-defined threshold $high_{cpu}$ for a certain period of time, it requests the control plane for an additional sentry server. Upon receiving such requests from one or more sentries of an application, the control plane assigns each an additional sentry. Similarly, if the utilization
of a sentry stays below a threshold \( low_{cpu} \), it is returned to the free pool while ensuring that the application has at least one sentry remaining. Whenever the control plane assigns (or removes) a sentry server to an application, it repartitions the application’s servers pool equally among the various sentries. The DNS entry for the application is also updated upon each allocation or deallocation; a round-robin DNS scheme is used to loosely partition incoming requests among sentries. Since each sentry manages a mutually exclusive pool of servers, it can independently perform admission control and load balancing on arriving requests; the SLA is collectively maintained by virtue of maintaining it at each sentry.

7.5 Implementation Considerations

We implemented a prototype hosting platform on a cluster of 40 Pentium machines connected via a 1Gbps ethernet switch and running Linux 2.4.20. Each machine in the cluster runs one of the following entities: (1) an application replica, (2) a sentry, (3) the control plane, (4) a workload generator for an application.

**Sentry:** We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [66] to implement the policing mechanisms described in Section 7.3. ktcpvs is an open-source, Layer-7 load balancer implemented as a Linux module. It accepts TCP connections from clients, opens separate connections with servers (one for each client), and transparently relays data between these. We modified ktcpvs to implement all the sentry mechanisms described in Sections 7.3 and 7.4.

7.6 Experimental Evaluation

In this section we present the experimental setup followed by the results of our experimental evaluation.
7.6.1 Experimental Setup

The sentries were run on dual-processor 1GHz machines with 1GB RAM. The control plane (responsible for provisioning) was run on a dual-processor 450MHz machine with 1GB RAM. The machines used as servers had 2.8GHz processors and 512MB RAM. Finally, the workload generators were run on machines with processor speeds varying from 450MHz to 1GHz and with RAM sizes in the range 128MB-512MB. All machines ran Linux 2.4.20. In our experiments we constructed replicable applications using the Apache 1.3.28 Web server with PHP support enabled. The file set serviced by these Web servers comprised files of size varying from 1kB to 256kB to represent the range from small text files to large image files. In addition, the Web servers hosted PHP scripts with different computational overheads. The dynamic component of our workload consisted of requests for these scripts. In all the experiments, the SLA presented in Figure 7.1 was used for the applications. Application requests were generated using httpperf [77], an open-source Web workload generator.

7.6.2 Revenue Maximization and Class-based Differentiation

Our first experiment investigates the efficacy of the mechanisms employed by the sentry for revenue maximization and to provide class-based differentiation to requests during overloads. The provisioning was kept turned off in this experiment. We constructed a replicated Web server consisting of three Apache servers. This application supported three classes of requests—Gold, Silver and Bronze in decreasing order of revenue. The class of a request could be uniquely determined from its URL. The delay values for the three classes were fixed at 0, 50, and 100 msec, respectively. The minimum sustainable requests rates desired by all three classes were chosen to be 0.

The workload consisted of requests for a set of PHP scripts. The capacity of each Apache server for this workload (i.e., the request arrival rate for which the 95\textsuperscript{th} percentile response time of the requests was below the response time target) was determined offline.
and was found to be nearly 60 requests/sec. Figure 7.3(a) shows the workload used in this experiment. Nearly all the requests arriving till $t=130$ sec were admitted by the sentry. Between $t=130$ sec and $t=195$ sec, the Bronze requests were dropped almost exclusively. At $t=195$ sec the arrival rate of Silver requests shot up and reached nearly 120 requests/sec. The admission rate of Bronze requests dropped to almost zero to admit as many Silver requests as possible. At $t=210$ sec, the arrival rate of Gold requests shot up to 200 requests/sec. The sentry totally suppressed all arriving Bronze and Silver requests now and let in only Gold requests as long as the increased arrival rate of Gold requests persisted. Figure 7.3(c) is an alternate representation of the system behavior in this experiment and
Figure 7.4. Scalability of the admission control.

depicts the variation of the fraction of requests of the three classes that were admitted.
Figure 7.3(d) depicts the performance of admitted requests. We find that the sentry is successful in maintaining the response time below 1000 ms.

7.6.3 Scalable Admission Control

We measured the CPU utilization at the sentry server for different request arrival rates for both the batch-based and the threshold-based admission control. Figure 7.4 shows our observations of CPU utilization with 95% confidence intervals. Since we were interested only in the overheads of the admission control and not in the data copying overheads inherent in the design of the ktcpvs switch, we forced the sentry to drop all requests after conducting the admission control test. We increased the request arrival rates till the CPU at the sentry server became saturated (nearly 90% utilization). We observe more than a four-fold improvement in the sentry’s scalability—whereas the sentry CPU saturated at 4000 requests/sec with the batch-based admission control, it was able to handle almost 19000 requests/sec with the threshold-based admission control.

A second experiment was conducted to investigate the degradation in the decision making due to the threshold-based admission controller. We repeated the experiment reported
in Section 7.6.2 (Figure 7.3) but forced the sentry to employ the threshold-based admission controller. The thresholds used by the admission control were computed once every 15 sec.

Figure 7.5(a) shows changes in the admission rates for requests of the three classes. The impact of the inaccuracies inherent in the threshold-based admission controller resulted in degraded performance during periods when the threshold chosen was incorrect. We observe two such periods (120-135 sec during which all Bronze requests were dropped and 190-210 sec during which all Bronze and Silver requests were dropped while Gold requests were admitted with probability of 0.5) during which the $95^{th}$ percentile of the response time deteriorated compared to the target of 1000 msec. The response times during the rest of the experiment were kept under control due to the threshold getting updated to a strict enough value.

7.6.4 Sentry Provisioning

We conducted an experiment to demonstrate the ability of the system to dynamically provision additional sentries to a heavily overloaded service. Figure 7.6 shows the outcome
of our experiment. The workload consisted of requests for small static files sent to the sentry starting at 4000 requests/sec and increasing by 4000 requests/sec every minute and is shown in Figure 7.6(a). If the CPU utilization of the sentry server remained above 80% for more than 30 sec, a request was issued to the control plane for an additional sentry. Figure 7.6(b) shows the variation of the CPU utilization at the first sentry. At t=210 sec, a second sentry was added to the service. Subsequent requests were distributed equally between the two sentries causing the arrival rate and the CPU utilization at the first sentry to drop. A third sentry was added at t=420 sec, when the total arrival rate to the service had reached 32000 requests/sec overwhelming both the existing sentries.

7.6.5 Provisioning

We conducted an experiment with two Web applications hosted on our platform. The total number of servers available in this experiment was 11. The SLAs for both the applications were identical and are described in Figure 7.1. Further, the SLAs imposed a lower bound of 3 on the number of servers that each application could be assigned. The default provisioning duration used by the control plane was 30 min.
The workloads for the two applications consisted of requests for an assortment of PHP scripts and files in the size range 1kB-128kB. Requests were sent at a sustainable base rate to the two applications throughout the experiment. Overloads were created by sending increased number of requests for a small subset of the scripts and static files (to simulate a subset of the content becoming popular). The experiment began with the two applications running on 3 servers each. Sentries invoked the provisioning algorithm when more than 50% of the requests were dropped over a 5 min interval. Figures 7.7(a) and 7.7(c) depict the arrival rates to the two applications. The arrival rate for Application 1 was made to increase in a step-like fashion starting from 100 requests/sec, doubling roughly once every 5 min till it reached a peak value of 1600 requests/sec. At this point Application 1 was heavily overloaded with the arrival rate several times higher than system capacity (which was roughly 60 request/sec per server assigned to the service as determined by offline measurements). At t=910 sec the sentry, having observed more than 50% of the request being dropped, triggered the provisioning algorithm as described in Section 7.4. The provisioning algorithm responded by pulling one server from the free pool and adding it to Application 1. At t=1210 sec, another server was added to Application 1 from the free pool. Observe in Figure 7.7(a) the increases in the admission rates corresponding to these additional servers being made available to Application 1. The next interesting event was the default invocation of provisioning at t=1800 sec. The provisioning algorithm added all the 3 servers remaining in the free pool to the heavily overloaded Application 1. Also, based on recent observation of arrival rates, it predicted an arrival rate in the range 1000-10000 requests/sec and degraded the response time target for Application 1 to 2000 msec based on its QoS table (see Figure 7.1). In the latter part of the experiment, the overload of Application 1 subsided and Application 2 got overloaded. The functioning of the provisioning was qualitatively similar to when Service 1 was overloaded. Figures 7.7(b) and 7.7(d) show the 95th percentile response times for the two services during the experiment. The control plane was able to predict changes to arrival rates and degrade the response time target according to the
Figure 7.7. Dynamic provisioning and admission control: Performance of Applications 1 and 2. D: Default invocation of provisioning, T: Provisioning triggered by excessive drops, [N=n]: size of the server set is n now. Only selected provisioning events are shown.

SLA resulting in an increased number of requests being admitted. Moreover, the sentries were able to keep the admission rates well below system capacity to achieve response times within the appropriate target with only sporadic violations (which were on fewer than 4% of the occasions).
7.7 Related Work

Previous literature on issues related to overload management in platforms hosting Internet services spans several areas. In this section we describe the important pieces of work on these topics.

Admission Control for Internet Services: Many papers have developed overload management solutions based on doing admission control. Several admission controllers operate by controlling the rate of admission but without distinguishing requests based on their sizes imposing fixed, statically-determined limits on one or more service parameters. The simplest example of such admission control is the upper limit on the number of simultaneous processes or threads in commonly used servers such as Apache [5]. Voigt et al. present kernel-based admission control mechanisms to protect Web servers against overloads—SYN policing controls the rate and burst at which new connections are accepted, prioritized listen queue reorders the listen queue based on pre-defined connection priorities, HTTP header-based control enables rate policing based on URL names [120]. Welsh and Culler propose an overload management solution for Internet services built using the SEDA architecture [123]. A salient feature of their solution is feedback-based admission controllers embedded into individual stages of the service. The admission controllers work by gradually increasing admission rate when performance is satisfactory and decreasing it multiplicatively upon observing QoS violations. The QGuard system proposes an adaptive mechanism that exploits rate controls for inbound to fend off overload and provide QoS differentiation between traffic classes [56]. The determination of these rate limits, however, is not dynamic but is delegated to the administrator. Iyer et al. propose a system based on two mechanisms—using thresholds on the connection queue length to decide when to start dropping new connection requests and sending feedback to the proxy during overloads which would cause it to restrict the traffic being forwarded to the server [54]. However, they do not address how these thresholds may be determined online. Cherkasova and Phaal propose an admission control scheme that works at the granularity of sessions rather than
individual requests and evaluate it using a simple simulation study [29]. This was based on a simple model to characterize sessions. The admission controller was based on rejecting all sessions for a small duration if the server utilization exceeded a pre-specified threshold and has some similarity to our approximate admission control, except we use information about the sizes of requests in various classes to determine the drop threshold.

Several efforts have proposed solutions based on analytical characterization of the workloads of Internet services and modeling of the servers. Kanodia and Knightly utilize a modeling technique called service envelops to devise an admission control for web services that attempts to different response time targets for multiple classes of requests [62]. Li and Jamin present a measurement-based admission control to distribute bandwidth across clients of unequal requirement [70]. A key distinguishing feature of their algorithm is the introduction of controlled amounts of delay in the processing of certain requests during overloads to ensure different classes of requests are receiving the appropriate share of the bandwidth. Knightly and Shroff describe and classify a broad class of admission control algorithms and evaluate the accuracy of these algorithms via experiments [65]. They identify key aspects of admission control that enable it to achieve high statistical multiplexing gains.

Two admission control algorithms have been proposed recently that utilize measurements of request sizes to guide their decision making. Verma and Ghosal propose a service time based admission control that uses predictions of arrivals and service times in the short-term future to admit a subset of requests that would maximize the profit of the service provider [117]. Elnikety et al. [42] present an admission control for multi-tier e-commerce sites that externally observes execution costs of requests, distinguishing different requests types [42]. Our measurement-based admission control is based on similar ideas, although the techniques differ in the details.

**Dynamic Provisioning and Managing Resources in Clusters:**
We refer the reader to Chapters 4 and 6 for discussions on state-of-the-art dynamic capacity provisioning for clusters of servers.

An alternate approach for improving performance of overloaded Web servers is based on re-designing the scheduling policy employed by the servers. Schroeder and Harchol-Balter propose to employ the SRPT algorithm based on scheduling the connection with the shortest remaining time and demonstrate that it leads to improved average response time [97]. While scheduling can improve response times, under extreme overloads admission control and the ability to add extra capacity are indispensable. Better scheduling algorithms are complementary to our solutions for handling overloads.

**Design of Efficient Load Balancers:** Our admission control scheme is necessarily based on the use of a Layer-7 switch and hence the scalable design of such switches is important to our implementation. Pai et al. design locality-aware request distribution (LARD), a strategy for content-based request distribution that can be employed by front servers in network servers to achieve high locality in the back end servers and good load balancing [83]. They introduce a *TCP hand-off protocol* that can hand off an established TCP connection in a client-transparent manner. A load balancer based on TCP hand-off has been shown to be more scalable than the *ktcpvs* load balancer we have used. Aron et al. present a highly scalable architecture for content-aware request distribution in Web server clusters [11]. The front switch is a Layer-4 switch that distributed requests to a number of back-end nodes. Content-based distribution is performed by these back-end servers. Cardellini et al. provide a comprehensive survey of the main mechanisms to split traffic among the servers in a cluster, discussing both the various architectures and the load sharing policies [21]. Our admission control and load balancing schemes are independent of the actual switch implementation so long as it is Layer-7 and hence may be implemented in any of the aforementioned scalable switches.

**SLAs and Adaptive QoS Degradation:** The WSLA project at IBM addresses service level management issues and challenges in designing an unambiguous and clear specifi-
cation of SLAs that can be monitored by the service provider, customer and even by a third-party [125]. Abdelzaher and Bhatti propose to deal with server overloads by adapting delivered content to load conditions [2]. This is a different kind of QoS degradation than what we have proposed in our work, but it can be integrated into a platform by defining appropriate SLAs based on it.

In this chapter we showed the utility of coupling policing and provisioning, in contrast to prior approaches that considered these techniques in isolation.

### 7.8 Conclusions

In this chapter we presented a comprehensive approach for handling extreme overloads in a hosting platform running multiple Internet services. The primary contribution of our work was to develop a low overhead, highly scalable admission control technique for Internet applications. It provides several desirable features, such as guarantees on response time by conducting accurate size-based admission control, revenue maximization at multiple time-scales via preferential admission of important requests and dynamic capacity provisioning, and the ability to be operational even under extreme overloads. The sentry can transparently trade-off the accuracy of its decision making with the intensity of the workload allowing it to handle incoming rates of up to 19000 requests/second. We implemented a prototype hosting platform on a Linux cluster and demonstrated its benefits using a variety of workloads.
CHAPTER 8
SUMMARY AND FUTURE WORK

Hosting platforms for Internet applications have emerged as an important business during the past few years. These platforms typically employ large clusters of servers to host multiple applications. Hosting platforms provide performance guarantees to the hosted applications (such as guarantees on response time or throughput) in return for revenue.

We classified hosting platforms into two categories—dedicated and shared—and identified the shortcomings of existing resource management techniques in both these hosting models. We identified two key features of Internet applications that make the design of hosting platforms challenging. First, modern Internet applications are extremely complex—existing resource management solutions rely upon very simple abstractions of these applications and are therefore inadequate in several respects. Second, these applications exhibit highly dynamic workloads with multi-time-scale variations. Managing the resources in a hosting platform to realize the often opposing goals of meeting application performance targets and achieving high resource utilization is therefore a difficult endeavor. In this thesis, we developed resource management mechanisms that an Internet hosting platform can employ to address these challenges. In this chapter, we summarize our contributions and discuss directions for future work that our work in this thesis has spawned.

8.1 Summary of Research Contributions

In this dissertation, we made the following main contributions.
• **Analytical models for Internet applications:** Modern Internet applications are complex, distributed software systems designed using multiple tiers. They are built using diverse software components. They see dynamically changing workloads that contain long-term variations such as time-of-day effects as well as short-term fluctuations such as transient overloads. Additionally, these applications may employ replication and caching at one or more tiers. Existing models employ very simple abstractions of these applications (such as modeling only one tier) and are therefore inadequate in several respects. In this thesis, we proposed analytical models of multi-tier Internet applications running on a dedicated hosting platform. Our models can handle applications with an arbitrary number of tiers and tiers with significantly different performance characteristics. Our models are designed to handle session-based workloads and can account for application idiosyncrasies such as replication at tiers, load imbalances across replicas, caching effects, and concurrency limits at each tier.

• **Requirement inference and application placement:** We studied the problem of the placement of distributed applications on a shared hosting platform. We presented a technique to infer the resource requirements of such applications using offline kernel-based profiling. We presented automated placement techniques that allow a platform provider to exert sufficient control over the placement of application components onto nodes in the cluster, since manual placement of applications is unfeasibly complex and error-prone in large clusters. We studied theoretical properties of the application placement problem and developed online algorithms. Our approach attempted to increase the revenue of a shared hosting platform in two complementary ways—(i) we studied approximation algorithms for application placement to understand how many applications they are able to place on the platform and (ii) we showed how controlled under-provisioning of resources can be used to improve the platform’s revenue.
• Dynamic resource provisioning techniques: Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. Dynamic provisioning of resources—allocation and deallocation of servers to replicated applications—has been studied in the context of single-tier applications, of which clustered HTTP servers are the most common example. However, it is non-trivial to extend provisioning mechanisms designed for single-tier applications to multi-tier scenarios. We proposed a novel dynamic provisioning technique for multi-tier Internet applications that employs (i) a flexible queuing model to determine how much resources to allocate to each tier of the application, and (ii) a combination of predictive and reactive methods that determine when to provision these resources, both at large and small time scales. We proposed a novel hosting platform architecture based on virtual machine monitors to reduce provisioning overheads.

• Design, implementation, and evaluation: We implemented all our resource management algorithms in a prototype hosting platform based on a cluster of forty Linux machines and evaluated them using realistic applications and workloads. Experiments on our prototype hosting platform demonstrated the responsiveness of our techniques in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes while maintaining response time targets. Our technique also reduced the overhead of switching servers across applications from several minutes or more to less than a second, while meeting the performance targets of residual sessions.

8.2 Future Work

In this section, we discuss some future research directions based on ideas that have emerged from our work in this dissertation.
- **Virtual machine based architecture**: In Chapter 4, we introduced the use of the virtual machine technology to enable fast reactive provisioning in dedicated hosting platforms. Recent research has demonstrated the feasibility of the migration of entire virtual machine instances on a commodity cluster, recording service downtimes as low as 60ms; this makes live migration a practical tool even for servers running interactive Internet workloads [30]. This migration technology adds a new dimension to the design of a dynamic provisioning technique—in addition to changing resource allocations, now we can also move application capsules in a seamless fashion across the nodes in the cluster. We plan to investigate the benefits that this new capability might offer in managing resources in hosting platforms.

- **Automated determination of provisioning parameters**: In our existing dynamic provisioning technique, the frequencies at which the predictive and reactive algorithms are invoked are fixed. We plan to automate the computation of these values in a dynamic manner. This automation would relieve the system administrators of tuning these knobs, a difficult task.

- **Enhancements to the application model**: Our application model can be further enhanced to incorporate additional complexities of modern Internet applications. For large applications, various tiers are often located in separate data centers. For instance, the database servers of an online banking service may be located in a separate physical location and connected to the rest of the application via a dedicated high-speed network. We plan to investigate modeling and provisioning issues for such applications that are less tightly coupled than the applications that we have studied in this thesis.

- **Understanding and reproducing workloads**: Although a lot of research has been done on characterizing the load imposed on standalone Web servers by modern workloads [7, 34] (e.g., the distribution of the size of files served by many Web servers),
similar characterization lacks for components like application servers and databases.

As examples, what are typical service times of requests issued to EJB tiers, or of queries issued to databases of an online retail application? We would like to conduct research to answer such questions. A natural offshoot of this study would be the development of workload generators for multi-tier Internet applications as exist for single-tier Web servers [15, 77, 104].
We show that the application placement problem is NP-complete.

**Definition 10** Single-Capsule Application Placement Problem (DEC\_MAX\_CAP): Given \( n \) empty nodes \( N_1, \ldots, N_n \), a set of \( m \) single-capsule applications \( C_1, \ldots, C_m \), and an integer \( k \), determine if a placement of size \( k \) exists.

**Lemma 10** \( DEC\_MAX\_CAP \) is NP-complete.

**Proof:** The proof consists of two parts.

- **DEC\_MAX\_CAP** is in NP: Given an instance of **DEC\_MAX\_CAP** and a placement, we can in polynomial time verify — (a) if this is a valid placement — this involves checking for each node that the sum of the requirements of all the capsules placed on it does not exceed the node capacity, and (b) if the size of the placement is \( k \), i.e., could \( k \) capsules be placed. Thus, we have shown that **DEC\_MAX\_CAP** is in NP.

- **BIN-PACKING** reduces to **DEC\_MAX\_CAP**: Let us first state the decision version of the bin-packing problem which is known to be NP-complete [46].

**BIN-PACKING:** Given a set of \( m \) objects \( O_1, \ldots, O_m \) of sizes \( s_1, \ldots, s_m \) respectively, and an integer \( k \), determine if all the objects can be placed into \( k \) bins, where each bin has unit capacity.

Consider the following polynomial-time reduction from **BIN-PACKING** to **DEC\_MAX\_CAP**. Given an input to **BIN-PACKING**, we construct an input to **DEC\_MAX\_CAP** as follows. Corresponding to each object in the input to **BIN-PACKING**, we construct a
capsule whose requirement is equal to the size of the object. Next, we construct $k$ nodes, each with unit capacity. These node- and capsule-sets along with the integer $m$ comprise the input to DEC_MAX_CAP.

It is easy to see that the above is a reduction. Assume the input to BIN-PACKING had $m$ objects and the integer $k$. The input to DEC_MAX_CAP that we construct would have $k$ nodes, $m$ capsules and the integer $m$. If the $m$ objects can fit into $k$ bins, then clearly we can place the $m$ capsules in $k$ nodes. On the other hand, if the $m$ objects cannot fit into $k$ bins, then the $m$ capsules cannot all be placed into the $k$ nodes.

This completes the proof.

**Definition 11 General Application Placement Problem (DEC_MAX_APP):** Given $n$ empty nodes $N_1, \ldots, N_n$, a set of $m$ applications $A_1, \ldots, A_m$, and an integer $k$, determine if a placement of size $k$ exists.

**Lemma 11** DEC_MAX_APP is NP-complete.

**Proof:** Restrict DEC_MAX_APP to DEC_MAX_CAP by allowing only applications with one capsule.

---

Finally, we can show the NP-hardness of APP.
**Theorem 2** *APP is NP-hard.*

**Proof:** DEC\_MAX\_APP\_RES is the decision version of APP. Therefore, the NP-hardness of DEC\_MAX\_APP\_RES shown in Lemma 12 proves the NP-hardness of APP. ■
APPENDIX B

ANALYSIS OF THE POLICER

We show how the sentry can, under certain assumptions, compute the delay values for various classes based on online observations. The goal is to pick delay values such that the probability of a newly arrived request being denied service due to an already admitted less important request is smaller than a desired threshold.

Consider the following simplified version of the admission control algorithm presented in Section 7.3.2: Assume that the application runs on only one server—it is easy to extend the analysis to the case of multiple servers. The admission controller lets in a new request if and only if the total number of requests that have been admitted and are being processed by the application does not exceed a threshold $N$. Assume the application consists of $L$ request classes $C_1, \ldots, C_L$ in decreasing order of importance. We make the simplifying assumption of Poisson arrivals with rates $\lambda_1, \ldots, \lambda_L$, and service times with known CDFs $F_{s_1}(.), \ldots, F_{s_L}(.)$ respectively. As before, $d_1 = 0$. For simplicity of exposition we assume that the delay for class $C_2$ is $d$, and $\forall i > 2, d_{i+1} = k_i \cdot d_i, (k_i \geq 1)$. Denote by $A_i$ the event that a request of class $C_i$ has to be dropped at the processing instant $m \cdot d_i, (m > 0)$ and there is at least one request of a less important class $C_j, (j > i)$ still in service. Clearly,

$$Pr(A_1) = 0 \text{ and } Pr(A_L) = 0.$$

We are interested in ensuring

$$\forall i > 1, Pr(A_i) < \epsilon, 0 < \epsilon < 1.$$ (B.1)
Consider $1 < i < L$. For $A_i$ to occur, all of the following must hold: (1) $X_i$: at least one request of class $C_i$ arrives during the period $[(m - 1) \cdot d_i, m \cdot d_i]$, (2) $Y_i$: the number of requests in service at time $m \cdot d_i$ is $N$, (3) $Z_i$: at least one of the requests being serviced belongs to one of the classes $C_{i+1}, \ldots, C_L$. We have,

$$Pr(A_i) = Pr(X_i \land Y_i \land Z_i)$$

During overloads, we can assume that the number of requests in service would be $N$ with a high probability $p_{\text{drop}}$. The policer will record $p_{\text{drop}}$ over short periods of time. Also, $X_i$ and $Z_i$ are independent. This lets us have

$$Pr(A_i) \approx Pr(X_i) \cdot p_{\text{drop}} \cdot Pr(Z_i) \quad (B.2)$$

$$Pr(X_i) = 1 - e^{-\lambda_i \cdot d_i} \quad (B.3)$$

Denote by $Z^j_i$, $(i < j \leq L)$ the event that at least one of the requests being serviced at time $m \cdot d_i$ belongs to the class $j$. Clearly,

$$Pr(Z_i) = \sum_{j=i+1}^{j=L} Pr(Z^j_i) \quad (B.4)$$

Let us now focus on the term $Pr(Z^j_i)$. The event $Z^j_i$ is the disjunction of the following events, one for each $l, (l > 0)$: $P^j_l$: at least one request of class $j$ arrives during the period $[m \cdot d_i - (l + 1) \cdot d_j, m \cdot d_i - l \cdot d_j]$ and $Q^j_l$: at least one request of class $j$ is admitted at the processing instant $m \cdot d_i - l \cdot d_j$ and $R^j_l$: the service time of at least one admitted request is long enough so that it is still in service at time $m \cdot d_i$. As in Equation (B.3),

$$Pr(P^j_l) = 1 - e^{-\lambda_j \cdot d_j} \quad (B.5)$$
Consider $R^l_j$. During an overload each admitted request competes at the server with $(N-1)$ other requests during most of its lifetime. A fair approximation then is to assume that a request takes $N$ times its service time to finish. Therefore, we have,

$$Pr(R^l_j) = 1 - F_{s_j} \left( \frac{l \cdot d_j}{N} \right) \quad \text{(B.6)}$$

We approximate $Q^l_j$ using the following reasoning. During overloads, a request of class $C_j$ will be admitted at processing instant $t$ only if the number of requests in service at time $t$ is less than $N$ (the probability of this is approximated as $(1 - p_{\text{drop}})$) and no request of a more important class $C_h$ arrived during $[t - d_h, t]$. That is,

$$Pr(Q^l_j) \approx (1 - p_{\text{drop}}) \prod_{h=1}^{h=j-1} e^{-\lambda_h d_h} \quad \text{(B.7)}$$

Combining Equations (B.2)-(B.7), we get the following approximation. $Pr(A_i)$,

$$Pr(A_i) \approx p_{\text{drop}} (1 - p_{\text{drop}}) (1 - e^{-\lambda_i d_i}) \sum_{j=L}^{j=i+1} (1 - e^{-\lambda_j d_j}) \prod_{h=1}^{h=j-1} e^{-\lambda_h d_h} \sum_{l=1}^{\infty} \left( 1 - F_{s_j} \left( \frac{l d_j}{N} \right) \right)$$

The above approximation of $Pr(A_i)$ provides a procedure for iteratively computing the $d_i$ values using numerical methods. We pick delay values that make the term on the right hand side smaller than the desired bound $\epsilon$ for all $i$. This in turn guarantees that the inequalities in (B.1) are satisfied.
BIBLIOGRAPHY


