RICE UNIVERSITY

Differentiated and Predictable Quality of Service in Web Server Systems

by

Mohit Aron

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

DOCTOR OF PHILOSOPHY

Approved, Thesis Committee:

Dr. Peter Druschel (Chair),
Associate Professor,
Computer Science

Dr. Willy Zwaenepoel,
Professor,
Computer Science

Dr. Edward W. Knightly,
Assistant Professor,
Electrical and Computer Engineering

Houston, Texas

October, 2000

Differentiated and Predictable Quality of Service in Web Server Systems

Mohit Aron

Abstract

As the World Wide Web experiences increasing commercial and mission-critical use, server systems are expected to deliver high and predictable performance. The phenomenal improvement in microprocessor speeds, coupled with the deployment of clusters of commodity workstations has enabled server systems to meet the continually increasing performance demands in a cost-effective and scalable manner. However, as the volume, variety and sophistication of services offered by server systems increase, effective support for providing differentiated and predictable quality of service has also become important. For example, it is often desirable to differentiate between the resources allocated to virtual web sites hosted on a server system so as to provide predictable performance to individual sites, regardless of the load imposed upon others.

Server systems lack adequate support for providing predictable performance to hosted services in terms of metrics that are meaningful to server applications, such as average throughput, response time etc. This is because conventional systems multiplex resources by dealing with system level metrics such as CPU/disk bandwidth, memory pages etc. Maintaining predictable levels of performance in application level metrics, therefore, requires a corresponding mapping to system level metrics that is not supported in conventional systems. High performance server systems based upon cluster-based architectures also lack adequate support for providing differentiated quality of service. This is because providing differentiated quality of service in a

cluster-based server system requires global resource management not found in current clusters.

This dissertation is concerned with support for both differentiated as well as predictable quality of service in server systems. The specific requirements of server applications and their interactions with the resource management facilities in the operating system software are studied. This leads to a concerted design of a resource management framework for providing effective quality of service in server systems.

To my grandmother

Acknowledgments

This dissertation is the culmination of many years of hard work. However, it wouldn't have been possible without the continuous support and guidance of many people, who I'd like to thank at this opportunity.

First and foremost, I'd like to thank my family for their continuous support throughout the period of my education. I thank my parents for making countless sacrifices in always providing me with the means for obtaining the best possible education. My grandparents hold a special place in my heart for all the wonderful times I spent with them throughout the course of my education. I thank my brother, Anirudh, for being such a wonderful companion and for the immeasurable amount of fun that we've had together. My dog, Jeffy, also deserves special mention for being my best toy, and for being such an immense source of joy to me all these years.

I'd like to thank my advisor, Peter Druschel, for his invaluable guidance, his patience and his tireless efforts in shaping my technical abilities. But most importantly, I'd like to thank him for being there when I needed him. I am indebted to him for the continuous technical and moral support that he made available to me without which this dissertation wouldn't have been possible. I would also like to thank the other members of my committee, Willy Zwaenepoel and Ed Knightly, for their comments and suggestions that helped improve the quality of this work. I have also immensely benefited from the many rules of thumb dropped by Willy in my many informal conversations with him.

The entire Computer Science department at Rice University also deserves credit for maintaining a healthy and invigorating technical environment. I am specially thankful to Gaurav Banga and Johnny Chen for being such great buddies as well as excellent technical colleagues. I am also thankful to Parthasarathy Ranganathan, Karthick Rajamani, Vikram Kanodia, Henrik Weimer and Geraldine Morin for their friendship and support.

Finally, I'd like to thank my writing instructor, Dr Janice Hewitt, for immensely improving my writing skills, and my colleague, Sitaram Iyer, for his help with several experiments in this dissertation.

Contents

Illustrations

Tables

Chapter 1

Introduction

In the past few years, Internet usage has been expanding at a phenomenal rate. This is mainly attributed to (1) the explosive growth in microprocessor performance and, (2) the deployment of World Wide Web (WWW) [105], where clients attached to the Internet use browsers to download content supplied by information servers. In the past, most computing involved local processing of data with individual computers being isolated entities. Nowadays, computers are networked together and the computation relies heavily upon data (or content) provided by servers.

A server is a computer system that provides specialized services for other computers. A server might be formed from either a single workstation or a cluster of workstations networked together with a fast LAN (local area network). Servers that are connected to the Internet and use HTTP [19, 44] (hypertext transfer protocol) to provide content in response to client requests are referred to as web servers. Examples of services provided by web servers are the retrieval of static and dynamic web pages, online databases for information retrieval and electronic commerce, and search engines.

The growing user community in the Internet has introduced two significant challenges for server systems. First, server systems are expected to keep up with the increasing performance demands that stress server resources. Examples of such resources are CPU cycles, disk and network bandwidth, physical memory etc. Second, servers are expected to manage their resources so as to provide differentiated and predictable quality of service (QoS) to their clients. With the increased reliance on remote services, the user-perceived speed of computing depends on the ability of servers to cope with these challenges.

Several technical advances have been instrumental in helping servers meet the continually increasing performance demands. These include significant increases in the raw speeds of both the server hardware and the physical network, the use of multiprocessors, and the construction of server systems with clusters of commodity workstations that provide a cost-effective and scalable solution to the problem.

However, the operating system software deployed in state-of-the-art server systems affords inadequate support for providing QoS to clients. In particular, server systems do not provide predictable levels of QoS in terms of metrics that are meaningful to server applications, such as average throughput, response time etc. Also, in cluster-based servers, the provision of differentiated QoS, based on client community, type of document or type of service requires global resource management not found in current clusters.

This dissertation is concerned with support for both predictable as well as differentiated QoS in server systems. It starts by evaluating support for application-level QoS metrics in server systems. This leads to a novel resource management framework capable of dynamically adjusting resource allocations to provide predictable QoS to clients in application-level QoS metrics. Then, operating system mechanisms required for supporting differentiated QoS in server systems are considered.

1.1 The problem of providing QoS in server systems

This section discusses the reasons why resource management facilities in classical systems are unsuitable in server environments. It also motivates the need for both differentiated as well as predictable QoS in server systems.

Current general purpose operating systems, e.g., Unix, were originally designed to support large expensive timesharing systems. Individual applications on such systems comprised a single process that spent most of its time in user mode performing some computation. Occasionally, the process would block in the kernel waiting for some slow I/O device.

The principal objective of resource management was to maximize the utilization of resources in the system while simultaneously providing some fairness between the contending processes. As the applications were mostly compute intensive, the CPU consumption was monitored closely so as to ensure fair distribution of CPU resources between the processes. However, owing to the high cost of I/O operations, fairness in allocation of I/O resources was of a lesser concern as compared to maximizing the utilization of I/O devices. For this purpose, I/O processing by any process was given strictly higher priority than all other processing in the system. Similarly, owing to the high cost of physical memory, effective utilization of memory was of a higher concern than fairness in allocation of physical memory pages. Therefore, page-replacement algorithms aimed at replacing the least useful page in all of the physical memory.

Modern server systems differ significantly than the timesharing systems of earlier times. These differences span both the architecture of server systems as well as the nature of server applications in their resource usage and service requirements.

In contrast to earlier timesharing systems, modern server systems are no longer confined to a single workstation. Servers based on clusters of commodity workstations are becoming increasingly popular. The relatively cheap cost of commodity hardware coupled with the incremental scalability afforded by adding additional workstations to the cluster provide a cost-effective and scalable solution to meet the increasing performance demands of the client community.

The nature of server applications also differs considerably from the applications that were run on classical timesharing systems. Server applications are expected to implement a variety of sophisticated services, such as the retrieval of static and dynamic Web pages, search engines and online databases. Such services typically handle requests from a large number of network streams, making them particularly I/O intensive. Coupled with the significant increases in the speed of I/O devices, the CPU processing associated with I/O in server systems forms a large fraction of the overall CPU usage. Further, the content served by such services is fetched from disk and cached in the physical memory of the system. Different services, therefore, contend with each other for physical memory pages on the system as well as disk bandwidth. Hence, effective resource management mechanisms in server systems must encompass all resource usage, e.g., CPU and disk bandwidth, physical memory utilization etc.

With the increasing deployment of server systems for commercial use, it is often desirable to differentiate between groups of services with respect to the resources allocated to them. Such groups of services for which the resources are accounted for and scheduled separately from resources in other groups are collectively referred to as a *service class*. The need for differentiated QoS between such service classes arises in situations such as:

- **Web Hosting:** The advances in computing have enabled server systems that are powerful enough to simultaneously host Web services for several organizations. This has resulted in some Web and proxy operators (e.g., Akamai [2]) to co-locate *virtual sites* from several different organizations on the same server platform. In such environments, it is desirable to differentiate between the services of various organizations, based perhaps on the amount of money paid.

- **Differing priorities between requests:** An organization involved in electronic commerce might wish to provide higher service guarantees to requests that involve monetary benefits (e.g., credit card transactions) as opposed to requests that simply browse the site.

- **Differing priorities between client communities:** The administrators of a corporate Web site might wish to provide higher priority to requests from inside the corporation than from outside. Similarly, organizations might wish to provide higher guarantees to paying clients (e.g., members) as opposed to those who browse for free.

- **Limiting usage by resource intensive services:** Administrators of a web site might wish to limit the collective system resources used by resource intensive requests such as CGI or database transactions. Otherwise, excessive consumption of resources by such requests might adversely affect the response times for other requests.

In such situations, it is desirable that individual service classes be *performance isolated* from each other. That is, a given proportion of server resources be reserved for a service class, such that it is guaranteed a minimal level of performance that remains unaffected by the behavior of other services. Guaranteeing a minimal level of performance in this fashion enables it to make progress and serve client requests even in the face of denial of service attacks on other services.

Another desirable feature is to ensure predictable performance to services within individual service classes. In situations such as web hosting, owners of virtual sites might wish to establish contracts with the web site or proxy operator so as to ensure predictable levels of performance in metrics such as average throughput in connections per second or average response time in milliseconds. The problem is complicated by the fact that in order to multiplex the resources between various service classes, operating system software usually deals with system level metrics such as CPU/disk bandwidth, memory pages etc. Maintaining predictable levels of performance in application level metrics (such as average throughput) therefore requires a corresponding mapping to system level metrics. The web site or proxy operator then becomes obligated to ensure that sufficient system resources are available for the corresponding virtual site in order to meet the contracted performance metric. Further, to maximize profits, it is desirable for the web/proxy operator to maximize resource utilization while still meeting service contracts. That is, the web/proxy operator should be able to co-locate as many virtual sites on a common hardware platform as possible without over-committing the system's resources.

1.2 Thesis statement and contributions

The hypothesis of this dissertation is that current server systems lack effective facilities for providing quality of service to clients. This support is imperative for resource management in server systems in order to meet service contracts specified in application-level metrics (such as average throughput, response time etc) and for providing performance isolation between service classes. The dissertation further hypothesizes that effective QoS support in a server system requires:

1. A facility that ensures that the collective resource requirements for meeting the desired service quality can be provided by the system.

2. A dynamic association between service quality specified in application-level metrics and resource allocations by the system software.

3. Accurate and fine-grained accounting of all resource usage in the system.

4. Scheduling of resources to match the desired allocations.

Without such support, the resource management in any server system would be limited in its effectiveness in providing desired levels of QoS to clients. It is therefore necessary to design a resource management framework that can meet all the aforementioned requirements.

The goal of this dissertation is to develop effective support in server systems to provide both predictable as well as differentiated quality of service to clients. The specific requirements of server applications and their interactions with the resource management facilities in the operating system software are studied. This leads to a concerted design of a resource management framework for providing effective quality of service to clients. Specifically, this dissertation presents:

- **An analysis of the limitations in server systems for supporting QoS effectively.** This analysis characterizes the poor association between application-level metrics (like average throughput, response time etc) for predictable QoS and the resource management facilities in the operating system. It also explores the system requirements for providing QoS between clients.

- **A novel measurement-based framework for supporting predictable QoS in server systems.** The framework allows webserver and proxy operators to provide a minimal quality of service, expressed in application-level metrics, to service classes hosted on a server system. The framework ensures that (1) services are admitted into the system only if the collective resource requirements of all the hosted services can be met with high probability, and (2) all service contracts are satisfied as long as the total resource requirements do not exceed the capacity of the hardware.

- **A new mechanism for global resource management in cluster-based server systems.** This mechanism is based on a new abstraction called a *cluster reserve* [12]. A cluster reserve serves as a cluster-wide resource principal in cluster-based server systems and encompasses all system resources used for a service class hosted on the cluster. All user and kernel processing performed on behalf of a service class are accounted to the corresponding cluster reserve. Resource allocation policies and load distribution remain independent of the cluster reserve mechanism enabling server systems to support a wide range of resource management scenarios.

- **An implementation and a rigorous quantitative evaluation.** This evaluation shows that the proposed facilities provide effective support for both differentiated as well as predictable QoS in server systems.

1.3 Dissertation overview

The rest of this dissertation is organized as follows. Chapter 2 presents background information necessary for understanding this dissertation. This chapter also provides an analysis of the limitations in server systems for supporting predictable and differentiated quality of service and motivates the work in this dissertation.

Chapter 3 describes the design of a measurement-based resource management framework that is capable of providing predictable quality of service in server systems. The framework allows Web servers and proxy operators to co-locate virtual sites and Web services, while simultaneously providing predictable quality of service, in terms of average request rate or average response time. An experimental evaluation with synthetic as well as trace-based workloads on a prototype implementation is also presented that shows that the framework is capable of effectively supporting predictable quality of service in server systems.

In Chapter 4, support for differentiated QoS in cluster-based server systems is considered. The basic design of the cluster-reserve abstraction is presented and a prototype implementation is evaluated for demonstrating its effectiveness in providing differentiated QoS in server systems. Several benchmark configurations are considered and results are provided for both synthetic as well as trace-based workloads.

Chapter 5 presents related work. It covers other resource management abstractions and past efforts to support quality of service in both server systems as well as in networks.

Finally, Chapter 6 summarizes the motivations behind the work in this dissertation and recounts its contributions. It also identifies future directions of research.

Chapter 2

Background and Analysis of QoS Support in Server Systems

This chapter presents background information necessary for the rest of this dissertation. It covers the operation and architecture of high performance World Wide Web (WWW) servers and the quality of service mechanisms deployed in contemporary commercial and research systems. Finally, it discusses the limitations of these mechanisms in their support for quality of service and motivates the goals of this dissertation.

Section 2.1 starts by discussing the basic operation of WWW servers as well as the architectural design of high performance servers deployed in the Internet. Both single-workstation as well as cluster-based servers are covered. Section 2.2 covers background regarding mechanisms in contemporary systems for supporting quality of service. Section 2.3 discusses limitations of these mechanisms and motivates the goals of this dissertation.

2.1 Web Servers

In the World Wide Web (WWW), the servers interact with clients through the Hypertext Transfer Protocol (HTTP) [44, 19]. This protocol follows a *request/response* paradigm where the server returns data (or content) to the client in response to a request. The HTTP protocol itself is layered over the Transmission Control Protocol (TCP) [96, 109] that ensures reliable delivery of network packets over the Internet. In addition, the TCP protocol also provides for congestion control thereby preventing network collapse due to excessive demand.

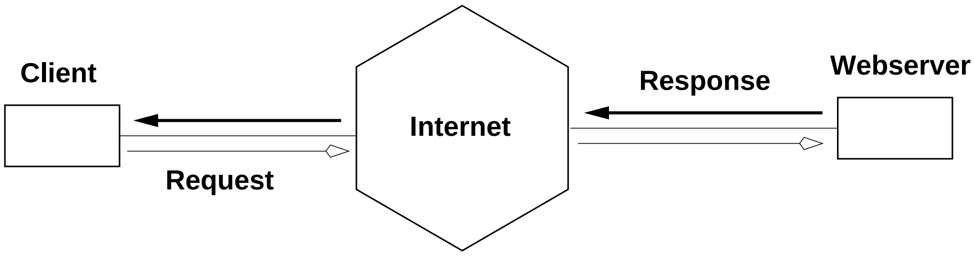

Figure 2.1 : **Webserver Operation**

Figure 2.1 shows the operation of a webserver. The workstations corresponding to the client and the webserver are networked to each other through the Internet. The browser application running on the client workstation sends an HTTP request to the webserver. The webserver application responds by sending the corresponding content back to the client. While the figure shows the webserver application to be running on a single workstation, in general the webserver might be deployed on a cluster of workstations.

The rest of this section is outlined as follows. Section 2.1.1 discusses some commonly used formats that are used to encode the content returned by webservers in response to client requests. Section 2.1.2 describes the methods of content generation at the webservers. Section 2.1.3 briefly describes the architectural configuration of typical webservers implemented on a single workstation. Following this, Section 2.1.4 discusses the HTTP/1.1 persistent connections. Finally Section 2.1.5 covers webservers implemented on a cluster of workstations.

2.1.1 Content encoding

While the content returned to clients by webservers can be in a wide variety of formats, typically it is structured using the the Hypertext Markup Language (HTML) [18]. HTML is a simple markup language used to create hypertext documents that are platform independent. HTML documents have generic semantics that are appropriate for representing information from a wide range of domains. HTML markup can represent hypertext news, mail, documentation, and hypermedia; menus of options; database query results; simple structured documents with in-lined graphics; and hypertext views of existing bodies of information. HTML documents support the embedding of images in commonly used formats like GIF and JPEG [75].

Recently, the Extensible Markup Language (XML) [58] is increasingly replacing HTML for encoding web documents. XML uses HTML-styled tags and attributes but in an unambiguous fashion for easy parsing. This facilitates the use of XML for structured data encoding.

2.1.2 Methods of generating content

The content sent by webservers in response to client requests can be classified into two principal categories. The first is *static content* where the data remains unchanged over long periods of time relative to the time taken to service the request. An example of such content are the files present on the system's filesystem. The second type of content is *dynamic content* where the response is generated dynamically upon receiving the client request. Such content is generated based on the data produced by running auxiliary third-party programs. Several web technologies are now available that support dynamic content generation. These are discussed in more detail in Section 2.1.3.

One class of servers known as *proxies* obtain their content from other webservers on the Internet. Proxies are located *close* to clients and intercept HTTP requests from clients to remote servers. By caching the content that they obtain from remote servers, proxies serve to reduce the response time for web requests.

2.1.3 Architecture of single-workstation servers

The simplest system architecture for running a webserver is one where a single workstation is employed as shown in Figure 2.1. As the HTTP protocol is layered above TCP, an HTTP client, e.g., a Web browser, needs to establish a TCP connection with the server's application software running on the workstation. For the purpose of receiving connection requests from clients, the server application *listens* on a well-known port (typically port 80). Once the connection with the client has been established, the operating system kernel delivers the connection to the server application via the **accept()** system call. The server application then waits for the client to send HTTP requests on this connection. Upon receiving the request, the server application parses the request and responds by sending the requested content back to the client on the same connection. HTTP/1.0 allows clients to only send a single request on the TCP connection; the connection is closed by the server once it has responded to the client's request. HTTP/1.1 allows several requests to be sent on a single TCP connection. This is discussed in greater detail in Section 2.1.4.

Webservers typically obtain the content for static requests from the local filesystem, while proxy servers obtain the content from other servers (if not found in the local filesystem cache). However, both kinds of servers may use a memory cache to speed retrieval. The basic operation of webservers is described in more detail by Stevens [97].

The design of the server application software has undergone several radical changes over the years. Following the classical UNIX model, early servers forked a new process to handle each HTTP connection. Forking processes has a large overhead associated with it. Subsequent servers (such as the NCSA **httpd** [82]) used a set of pre-forked processes. One process was designated as the master process and was made responsible for accepting new connections. After accepting a connection, it was passed over to one of the pre-forked worker processes that performed the rest of the HTTP processing. Each process, whether master or slave, has a single thread of control. This design is shown in Figure 2.2.

Subsequent improvements to the design eliminated the master process. Instead, each of the pre-forked worker processes calls **accept()** directly. The operating system kernel delivers a new connection to one of the worker processes that have issued an

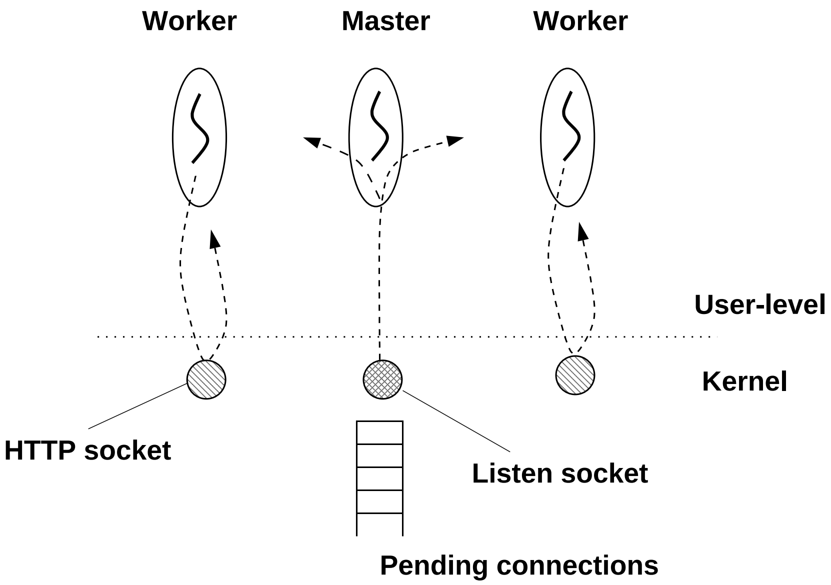

Figure 2.2 : **A process-per connection server with a master process**

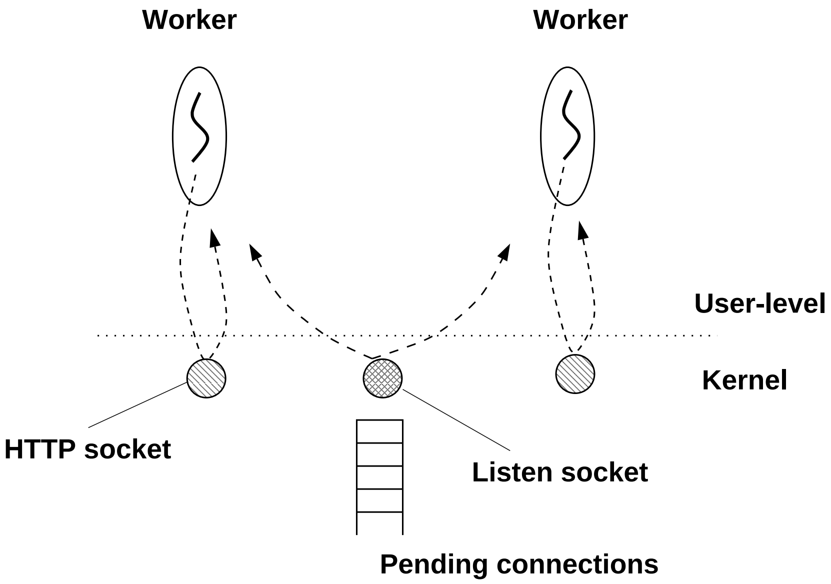

Figure 2.3 : **A process-per connection server with no master process**

accept on the listen socket. This design is shown in Figure 2.3 and is used by the Apache [8] webserver, which is the most commonly deployed webserver in the Internet.

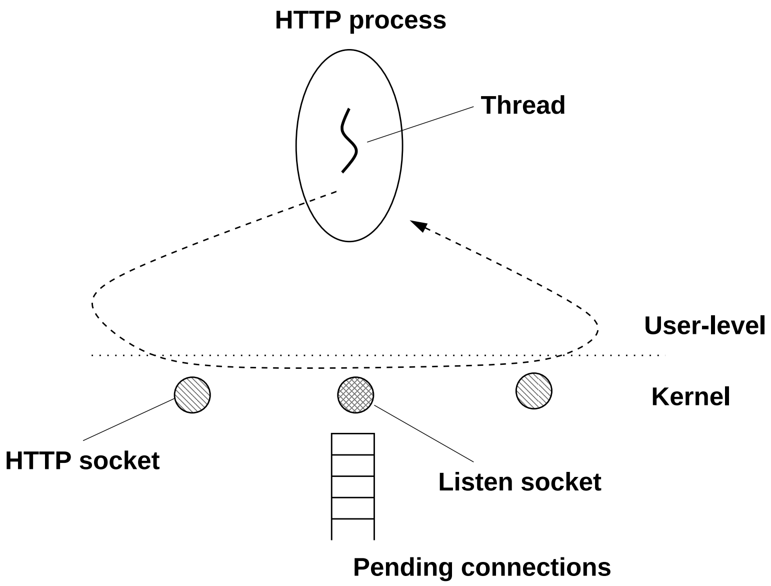

Figure 2.4 : **An event driven server**

Multi-process servers suffer from large context-switching and IPC overheads [30, 92]. To reduce these overheads, recent high performance servers use a single-process architecture. In an event driven model, a single thread is used within a single process for performing all HTTP processing. This architecture is shown in Figure 2.4 and is used by several modern webservers [30, 83, 100, 95, 113] .

The principal problem with the event driven model is that delays in handling requests for one connection can unnecessarily affect the response time of other connections. For example, a request that requires high disk activity might require a long time to service when other requests that could have been serviced from the memory cache are made to wait.

The Flash [88] webserver addresses the above mentioned problem with event driven servers. While most HTTP processing is performed by a master process in an event driven manner, operations that require disk activity are delegated to worker processes. The worker processes simply read the document from disk and bring it into the memory cache. After that, the master process is notified and it takes over the connection to complete the rest of the processing.

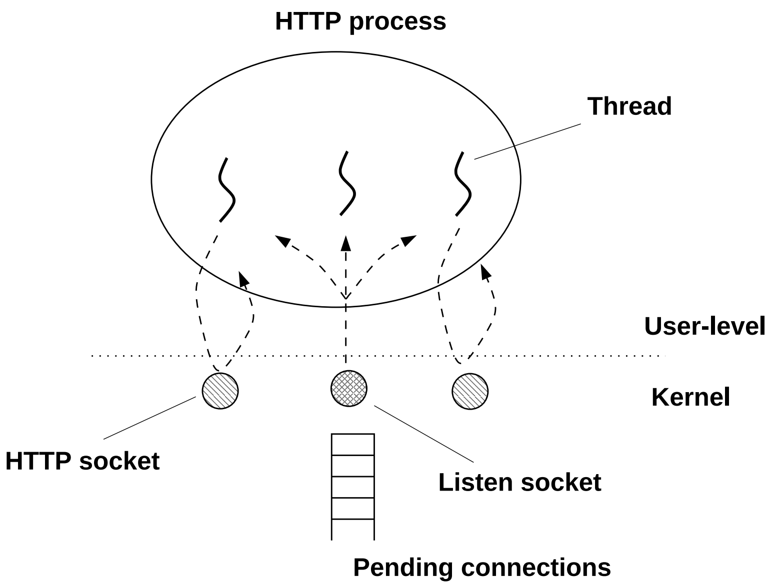

Figure 2.5 : **A single-process multi-threaded server**

Some webservers use multiple threads of control within a single process in order to efficiently utilize multiprocessor systems in addition to incurring low context switching and IPC overheads. In this model, each connection is assigned to a unique kernel thread operating within the address space corresponding to the server's application process. The thread scheduler is responsible for time-sharing the CPU between the various server threads. New connections from the listen socket are assigned to idle threads. The front-end of the *AltaVista* [5] search engine has this architecture and is depicted in Figure 2.5. Unfortunately, not all operating system kernels support multiple threads of activity within a single process. This limits the portability of multithreaded application server designs.

Dynamic content is generated by running auxiliary third-party programs at the webserver in response to client requests. Several technologies support dynamic content generation at webservers. The Common Gateway Interface (CGI) [29] provides an interface for running programs external to a webserver that generate the content returned to clients. The CGI interface is inefficient because it requires that a new program be started in a separate process for each request. The FastCGI [43] interface allows dynamic requests to be handled by persistently running programs. However, like CGI, FastCGI also requires that these programs be run in separate processes. While running programs in separate processes provides fault isolation, it also results in unnecessary overhead in situations where such fault isolation is not desired. To enable third-party modules to reside in the main server processes, some organizations have defined proprietary library based interfaces [61, 85]. More recently, dynamic content support has been facilitated by the development of PHP [90], which is a server-side HTML-embedded scripting language. PHP permits the embedding of code in HTML for generating content dynamically at the server upon receiving client requests. Similarly, the Java servlet [64] technology has enabled the execution of Java programs at the server that generate content in response to client requests.

2.1.4 Persistent HTTP connections

Obtaining an HTML document typically involves several HTTP requests to the Web server, to fetch embedded images, etc. Browsers using HTTP/1.0 [19] send each request on a separate TCP connection. This increases the server resource requirements per request, increases the number of network packets per request, and may increase the effective latency perceived by the client [86, 78].

HTTP/1.1 enables browsers to send several HTTP requests to the server on a single TCP connection. The Web server keeps the connection open after receiving a request for a configurable interval (typically 15 seconds), in anticipation of receiving further requests. This method amortizes the overhead of establishing a TCP connection (CPU, memory and network packets) over multiple HTTP requests and it allows for pipelining of requests [78]. Moreover, sending multiple server responses on a single TCP connection in short succession avoids multiple TCP *slow-starts* [62, 96], thus increasing network utilization and effective bandwidth perceived by the client. Slow-start is a congestion control mechanism in TCP used during startup phases to quickly (exponentially with time) increase TCP's packet window to take full advantage of the available network bandwidth.

2.1.5 Cluster-based servers

The increasing demands placed on popular Web sites and the growing importance of electronic commerce has precipitated the need for high performance, cost-effective, and highly available server systems. Clusters of commodity workstations are becoming increasingly popular for setting up server systems in such environments [50, 7, 38]. The principal reasons for this are:

- A cluster of cheap workstations tends to be more cost-effective than a single workstation that can afford performance comparable to the collective performance of the cluster.

- A cluster affords incremental scalability; that is, the capacity of a cluster can be increased incrementally by adding more workstations to the cluster. On the other hand, the use of a single fast workstation requires a complete replacement by a faster machine.

- A cluster provides better fault tolerance; the cluster remains functional even if some machines in the cluster crash. On the other hand, the use of a single workstation suffers from the drawback that there is a single point of failure. Cluster-based servers are, therefore, highly desirable in environments where high availability is important.

- The individual workstations in the cluster can be geographically distributed over the Internet. This enables strategic placement of workstations geographically closer to important client communities. Such clients incur reduced latencies when they fetch content from these servers.

The presence of multiple machines, however, adds additional complexity. Several challenges in designing cluster-based server systems stem from the distributed nature of the cluster. First, the HTTP clients still expect to be communicating with single-workstation servers. Therefore, the distributed nature of the server system should be as transparent to clients as possible. Second, the request processing might involve more than one workstation in the cluster. This requires efficient design of internal communication protocols within the cluster. Finally, the choice of the cluster nodes for handling incoming HTTP requests is important for obtaining high performance from the cluster. The policies involving request distribution decisions and mechanisms to support such policies have to be designed and implemented for efficient operation of the cluster.

The rest of this section discusses the various request distribution policies and mechanisms supported in cluster-based server systems used in the Internet.

Mirror sites

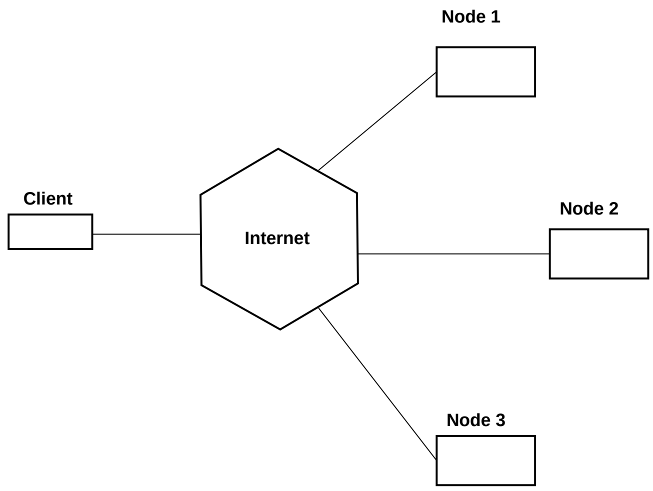

Figure 2.6 : **Geographically distributed cluster**

One of the ways of choosing the cluster node for handling a client's requests is to make the choice explicit to the client. Each cluster node is assigned a different hostname and each mirrors the content on other cluster nodes. As the hostnames are explicitly made available to the clients, any client can send HTTP requests to the cluster-node of its choice, which then responds by returning the requested content back to the client.

Such mirror sites are well suited for clusters where the individual nodes are geographically distributed. Such a configuration is shown in Figure 2.6. The client can make an educated choice of a cluster node that is in nearest geographical proximity. Choosing such a node reduces the response time seen by the client.

This scheme also has several disadvantages. First, the choice of the cluster node to serve the request depends upon the client's choice who usually does not know about load conditions on the cluster nodes. A heavily loaded node in close geographical proximity may be less suitable than one that is lightly loaded node but not located nearby. Second, as the individual cluster nodes are visible to the clients, non-operational cluster nodes affect the perceived availability of the cluster. Finally, the visibility of individual cluster nodes can also potentially raise security concerns.

DNS round-robin

DNS round-robin is a request distribution scheme that relies on the Domain Name System (DNS) [3, 76, 35] for distributing requests between server nodes. The Domain Name System is principally responsible for converting hostnames into IP addresses in the Internet. For convenience of use, clients usually address server systems by using convenient hostnames e.g., www.yahoo.com. In the DNS round-robin scheme of request distribution, a unique hostname is assigned to all the cluster nodes. However, as the TCP/IP protocol communication involves IP addresses, this hostname is converted into an IP address by the Domain Name System in the Internet before the communication can be established between the client and the server. Each cluster node is assigned a unique IP address different from all other cluster nodes. The DNS round-robin scheme affects request distribution by dynamically mapping the shared hostname into one of the IP addresses of the cluster nodes in a round-robin fashion.

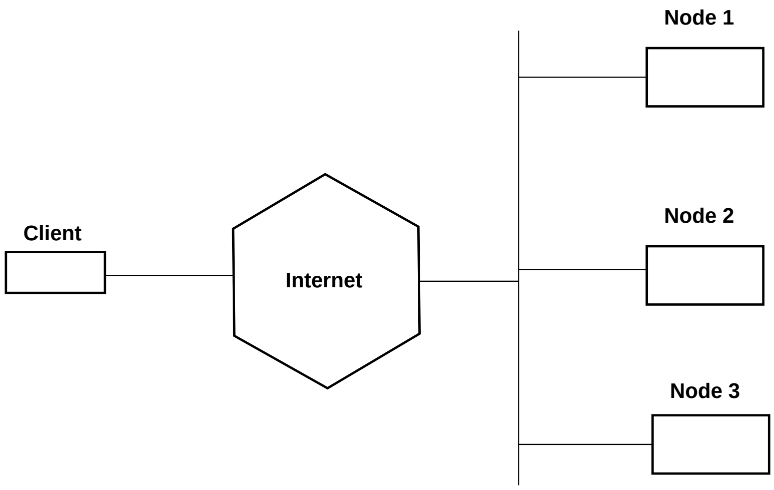

Figure 2.7 : **LAN-based cluster**

The DNS round-robin scheme has several advantages. It provides a method of distributing requests that is transparent to the clients; the clients deal with the single hostname that is shared by all the cluster nodes. Further, as cluster nodes are chosen in a round-robin fashion, DNS round-robin also provides for coarse-level load-balancing in the cluster. The DNS round-robin scheme can be used in both geographically distributed clusters (Figure 2.6) as well as in clusters where the nodes are networked to each other through a LAN (local area network) as depicted in Figure 2.7.

However, the DNS round-robin scheme also has disadvantages. While the hostname is shared by all the cluster nodes, they have different IP addresses that are visible to clients. Therefore, this scheme poses the security and availability concerns that were mentioned earlier in the context of mirror sites. Moreover, the mapping between hostname and IP address provided by the Domain Name System is usually cached by the clients for a limited period of time. If the cluster node with the cached IP address becomes non-operational, the corresponding client perceives the whole cluster to be non-operational. Another concern with the DNS round-robin scheme is that it only provides for coarse-level of load balance on a statistical basis. There is no guarantee that the load on individual servers would be actually balanced. Some cluster nodes might incur significantly more load than other nodes. This problem has been observed in a research study performed by Mogul [77].

Fine-grained load-balancing

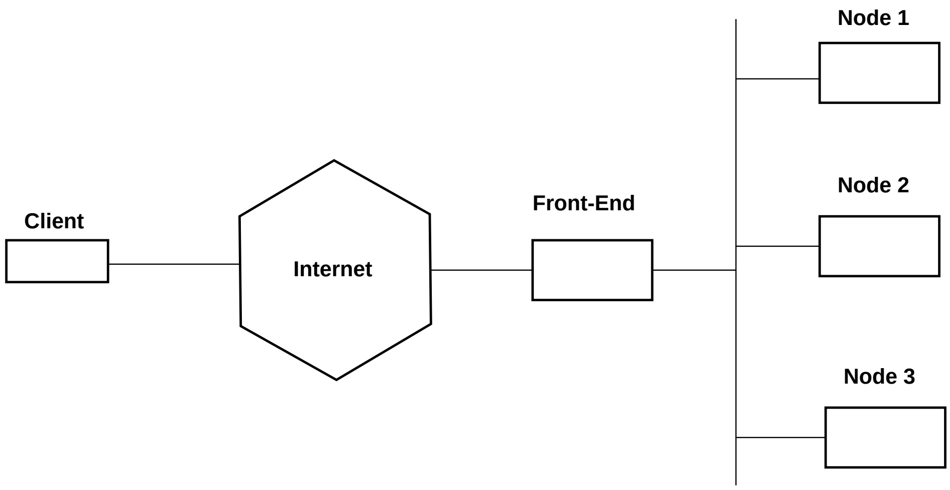

Figure 2.8 : **LAN-based cluster with Front-End**

High performance cluster-based server systems now use a dedicated front-end node for interfacing the rest of the cluster nodes (back-end nodes) with the Internet. The front-end receives connection requests from clients and chooses a back-end node that is made responsible for handling the connection. The front-end is used both for providing fine-grained load balancing, as well as for better security and availability by making the distributed nature of the cluster completely transparent to the clients. The latter is made possible because the client is only aware of the hostname and IP address of the front-end node [34].

Effective load balancing in the cluster can be achieved by using the Weighted round-robin (WRR) request distribution policy at the front-end node. This policy essentially assigns new connections to one of the least loaded nodes, chosen in a round-robin manner. The WRR strategy is described in detail in Section 7.1.1.

In order to assign connections to back-end nodes so as to balance load across the cluster, the front-end node needs to minimally inspect the packet headers corresponding to the transport layer in the network stack. In other words, the front-end has to minimally support layer-4 switching. Several commercial software and hardware-based layer-4 switches are now available. Among the software solutions, IBM's Network Dispatcher [55] and Cisco's LocalDirector [31] are well known. Efficient hardware implementations of layer-4 switches are commercially available from companies such as Alteon, Fore, and Foundry [6, 47, 48].

One of the main challenges in designing clusters with front-ends is to ensure that the front-end can support sufficient number of back-end cluster nodes before it becomes a bottleneck. One of the techniques employed to improve front-end scalability is to make the back-end nodes send all response data directly back to the clients, rather than relaying it through the front-end. This is achieved by putting the IP address of the front-end in the packet headers of all response data sent to the clients so as to maintain the illusion of a server based on a single workstation. However, all packets sent by clients necessarily have to be switched through the front-end.

Bestavros et al. [21] describe another cluster configuration that improves on the load-balancing provided by DNS round-robin and yet does not require a centralized front-end. Their technique, known as Distributed Packet Rewriting, uses a configuration similar to that shown in Figure 2.7. However, each node is capable of performing layer-4 switching, in addition to serving HTTP requests. A client makes an initial choice of a front-end cluster node using DNS round-robin strategy. The chosen cluster node hashes the client address to determine another cluster node that actually serves the request. The front-end node then performs layer-4 switching to enable communication between the client and the cluster node that sends the response data to the client. A good hash function used at front-ends is expected to uniformly spread the load across the cluster.

Content-aware Request Distribution

Content-aware request distribution strategies in clusters take into account the request content or type of service requested when deciding which cluster node should handle a given request. The potential advantages of content-aware request distribution are:

- Increased performance due to improved hit rates in the main memory caches of cluster nodes.
- Increased secondary storage scalability due to the ability to partition the server's database over the different cluster nodes.
- The ability to employ cluster nodes that are specialized for certain types of requests (e.g., audio and video).

In [87], Pai et al. explore the use of content-aware request distribution in a cluster Web server environment. The work presents an instance of a content-aware request distribution strategy, called LARD. The strategy achieves both locality, in order to increase hit rates in the Web servers' memory caches, and load balancing. Performance results with the LARD algorithm show substantial performance gains over WRR strategy. The work in [11] extends the LARD strategy to work efficiently with HTTP/1.1 persistent connections. Zhang et al. [115] explore another content-based request distribution algorithm that looks at static and dynamic content and also focuses on cache affinity. They confirmed the results of [87] by showing that focusing on locality can lead to significant improvements in cluster throughput.

The mechanisms for supporting content-aware request distribution policies are more complex than those for policies that do not take request content into account. This is because in content-aware request distribution, a client's request is first inspected before a decision is made about which cluster node should handle the request. The difficulty lies therein, that in order to inspect a request, the client must first establish a TCP connection with a node that will ultimately not handle the request. This requires layer-7 processing at the node that is more complex than the

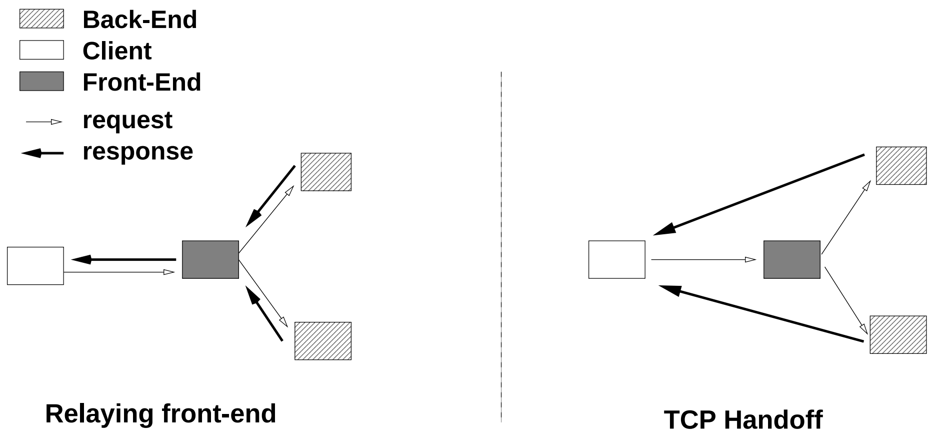

Figure 2.9 : **Mechanisms for content-aware request distribution**

layer-4 processing required if the request content is not necessary for request distribution. Several mechanisms have been proposed in past literature that are capable of supporting content-aware request distribution:

- *HTTP Redirects:* An HTTP redirect is a message sent by a server in response to a client request, directing the client to fetch the content using an alternate URL. A cluster front-end can use HTTP redirects to support content-aware request distribution by instructing the client to fetch the content directly from the chosen back-end cluster node. HTTP redirects are a standard part of the HTTP protocol and therefore require no extension to the protocol. The principal disadvantage is that HTTP redirects increase the client-perceived latency for fetching the request content as every HTTP redirect requires the client to initiate a new TCP connection with the server system.

- *Relaying front-end:* A simple client-transparent mechanism for supporting content-aware request distribution is a relaying front-end depicted in Figure 2.9. An HTTP proxy running on the front-end accepts client connections, and maintains persistent control TCP connections with all other back-end nodes. When a request arrives on a client connection, the proxy forwards the client's request message to the chosen back-end node through the corresponding control TCP connection. When the response arrives from the back-end node, the front-end forwards it to the client. The scheme is simple and can be implemented using standard operating systems. However, the disadvantage is that the front-end can become a bottleneck quickly as both request and response data goes through the front-end.

- *TCP splicing:* TCP splicing [42, 33, 111] is an optimization of the front-end relaying approach where the data forwarding at the front-end is done directly in the operating system. This eliminates the expensive copying and context switching operations that result from the use of a user-level application. While TCP splicing has lower overhead, the front-end still can become a bottleneck quickly as all request and response data passes through it. Additionally, splicing requires modifications to operating system kernel at the front-end node.

- *TCP handoff:* The TCP handoff [87, 11] was introduced to enable the forwarding of back-end responses directly to the clients without passing through the front-end as an intermediary. This is achieved by handing off the TCP connection established with the client at the front-end to the back-end where the request was assigned. The mechanism remains transparent to the clients in that the data sent by the back-ends appears to be coming from the front-end and any TCP acknowledgments sent by the client to the front-end are forwarded to the appropriate back-end. TCP handoff offers substantially higher scalability than TCP splicing. However, it requires modification to the operating system on both the front-end and the back-end operating systems.

- The work in [13] proposes a new scalable cluster architecture for supporting content-aware request distribution. The cluster front-end is a layer-4 switch that assigns a connection to one of the back-end nodes based only upon load information. This switch does not perform any content-aware request distribution. Rather, this function is performed by each of the back-end nodes that contact a centralized cluster node for determining the cluster node that is to send the requested content back to the client. The TCP handoff is used for handing the connection to this cluster node.

Layer-7 switches that support content-aware request distribution through TCP splicing are also commercially available from companies such as Alteon, Fore, and Foundry [6, 47, 48]. Resonate Inc. [91] offers a cluster-based server system that uses a mechanism similar to TCP Handoff at the front-end.

2.2 Overview of Quality of Service support in server systems

While the last section provided background on the operation and architecture of web servers, this section discusses the support for QoS in contemporary commercial and research server systems.

Most commonly used server applications do not make any distinction between different requests for the purpose of attributing system resources. At the best, each connection is handled by a separate process (or kernel thread) to ensure some fairness between the various active connections. As mentioned in Section 1.1, it is often desirable to distinguish the resources allocated to requests by grouping them into *service classes*. That is, the system resources need to be accounted for and scheduled separately for each service class in the system. Section 1.1 motivated the need for supporting both differentiated as well as predictable QoS by server systems. Unfortunately, classical operating systems like Unix and Windows NT lack system support for supporting either differentiated or predictable QoS.

To date, some support for differentiated QoS has been provided in research systems at both the level of the server application as well as at the level of operating system kernel. In order to support differentiated QoS, HTTP requests have to be classified so as to attribute the processing on their behalf to the corresponding service class. Section 2.2.1 covers various ways in which requests can be classified. Section 2.2.2

discusses the support added to server applications for the purpose of differentiating between service classes. One shortcoming of this approach is that solely distinguishing the service classes at the application layer is inherently limited in its effectiveness because a significant part of the HTTP processing is performed in the operating system kernel. Section 2.2.3 discusses the support added to conventional operating systems for supporting differentiated QoS in web servers and Section 2.2.4 covers the same in the context of cluster-based servers. Predictable performance in application level metrics is not yet supported by any commercial or research server system.

2.2.1 Request Classification

A key requirement to support multiple service classes is the ability to identify and classify incoming HTTP requests and attribute their processing to the corresponding class. The classification of requests in a webserver can be divided into two main categories:

1. *Client-based classification*: This form of classification is based on some client specific attribute. Some examples are the client IP address, HTTP cookies embedded in requests, special client identifiers embedded in HTTP requests by browser plugins etc. Classification based on client specific attributes usually serves to establish preferred client groups that need premium service as opposed to clients in other service classes.

2. *Target-based classification*: Classification can also be based on target specific attributes. Examples are URL in the HTTP request, IP address or port number at the server (used when multiple virtual sites are co-hosted on the same server) etc. This form of classification can be used to give better service to important requests (e.g., credit card payments) or to virtual sites that pay more to get premium service. It can also be used to control the resources spent on behalf of resource intensive requests such as requests for dynamic documents.

While classification based on the above categories is usually sufficient, some sites may need to classify requests based on a combination of both client and target specific attributes. Irrespective of the classification scheme used, it is the responsibility of the system software to control the QoS between various service classes. The rest of this section discusses techniques for supporting differentiated QoS in server applications and operating system kernels.

2.2.2 QoS by server applications

Traditional webserver applications do not discriminate between requests; the processing on every request is started as soon as it is received. The system resources attributed towards a service class are therefore proportional to the number of HTTP requests that are being handled for that service class. A service class receiving a high rate of requests can therefore receive a higher proportion of the system's resources than other service classes. This can adversely affect the perceived performance by clients belonging to preferred service classes.

Several ongoing research efforts attempt to enhance the server application software in order to provide differentiated QoS between the service classes. This is achieved primarily by affecting the scheduling and admission control at the level of HTTP requests received within the webserver. Little or no use is made of the resource management facilities provided by the operating system. Detailed discussions on some research efforts in supporting QoS in this manner are next presented.

In the context of the WebQoS [59] project at HP Labs, Bhatti and Friedrich [22] describe modifications to the Apache webserver in order to support differentiated QoS between service classes. The proposed server architecture is shown in Figure 2.10. A connection manager intercepts all requests, classifies them and places them into the queues for the corresponding service classes. The Apache worker processes receive requests from these service class specific queues, rather than directly from the HTTP sockets. The queues are shared between worker processes using shared memory fa-

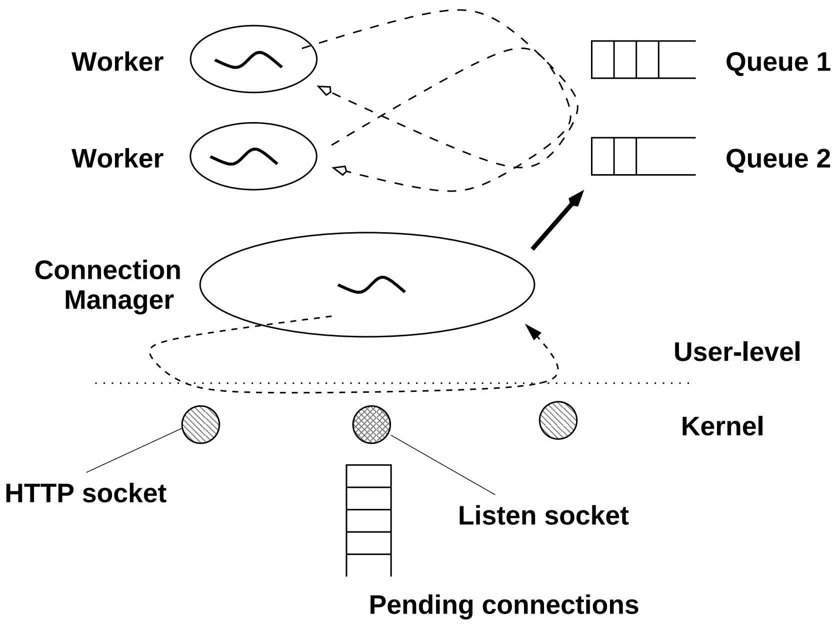

Figure 2.10 : **WebQoS server architecture**

cility in UNIX. Priorities assigned to service classes are observed by picking requests from high priority classes before processing those from classes with lower priority. A simple admission control scheme is employed to ensure that high request rate for lower priority classes does not adversely affect performance of higher priority classes. This is achieved by maintaining thresholds for the queued requests; requests are dropped once these thresholds are reached. The "nice" facility in UNIX is employed to lower the scheduling priority of worker processes that handle requests from lower priority classes.

Li and Jamin [72] propose a related model that attempts to provide proportional bandwidth allocation to web clients. The model uses a measurement-based approach to maintain bandwidth estimates for the various service classes in the system. When the bandwidth estimation for a service class falls below its target, the bandwidth attributed to over aggressive service classes is throttled by appropriately delaying the processing of their requests.

2.2.3 Operating system support for QoS

Performing service differentiation only at the level of the server application is inherently limited in its effectiveness. A significant part of the processing associated with an HTTP request is performed in the operating system kernel and the absence of service differentiation at this level results in inaccurate accounting for this processing. For example, the absence of service differentiation in the kernel might result in some service class getting more than its allocated share of the system's resources. Another problem is associated with scheduling resources at application level when the processing corresponding to a service class is performed by multiple processes or threads. Traditional operating systems treat either processes or threads as units of resource management (or resource principals) and multiplex the system's resources between these principals. When multiple processes/threads are associated with a service class, it becomes difficult to allocate resources among processes/threads so as to meet the overall desired allocation to the service class. For example, if some process/thread does not fully utilize its allocated resources, these are divided by the operating system among all the contending principals rather than only among processes/threads associated with the corresponding service class.

Operating systems researchers have realized the need for the active involvement of the operating system kernel in supporting service differentiation. Banga et al. [17] proposed the resource container abstraction that separates the notion of a resource principal from threads or processes and provides support for fine-grained resource management in the operating system. Coupled with a network subsystem architecture based on Lazy Receiver Processing (LRP) [39], resource containers are capable of supporting effective service differentiation in single-workstation server systems. The rest of this subsection describes resource containers and LRP in detail.

Resource Containers

As mentioned earlier, most traditional operating systems treat a process, or a thread within a process, as the schedulable entity. The process is also the "chargeable" entity for the allocation of resources, such as CPU time and memory. The basic design premise of such process-centric systems is that a process is the unit that constitutes a service class. Since a process also defines the virtual address space for a computation, the process abstraction serves a dual function: it serves both as a protection domain as well as a resource principal. As protection domains, processes provide isolation between applications. As resource principals, processes provide the operating system's resource management subsystem with accountable entities between which the system's resources are shared. This equivalence between protection domains and resource principals is not always appropriate, specially in server systems where service differentiation is desired. Some examples are:

- Multiple processes might be associated with a single service class. It becomes difficult to control the resources allocated to these processes such that desired allocation for the service class is achieved. This situation arises in multi-process webservers. This issue also becomes important for requests for dynamic content that involve CGI processes.

- A single process might be performing activities on behalf of several service classes. As the process acts as the resource principal the processing performed in the kernel does not distinguish between the various service classes. This situation arises when an event driven webserver handles connections for several service classes.

In some operating systems, e.g., Sun Microsystem's Solaris Operating Environment [98], threads assume some of the role of a resource principal. In these systems, CPU usage is charged to individual threads rather than to their parent processes. The process is still the resource principal for the allocation of memory and other kernel resources, such as network processing and protocol buffers. In situations where one thread performs processing for several service classes and in those where several threads are associated with a single service class, service differentiation again becomes difficult.

Resource containers are a first-class operating system abstraction for resource principals, and are independent of processes and threads. A resource container logically contains all the system resources consumed by a service class. For example, for a given HTTP connection associated with a service class and managed by a webserver, the resources include the CPU time devoted to the connection, the kernel objects such as sockets, protocol control blocks, and network buffers used by the connection. Using resource containers, a Web server can associate, for instance, a client network connection, a server thread, and a CGI process with a single resource principal that represents the service class being served. This principal competes with other principals representing other service classes for server resources. Resource containers allow accurate accounting and scheduling of resources consumed on behalf of a service class, both in the kernel as well as at user-level, and enable performance isolation and differentiated quality of service when combined with an appropriate resource scheduler.

With resource containers, the binding between a process/thread and a resource principal is dynamic, and under the explicit control of the application. The kernel charges the resource consumption of the process/thread to this container. One container might have several threads/processes associated with it. A process/thread that is time-multiplexed between several service classes changes its binding dynamically to the container corresponding to the service class being served. The system schedulers make resources available to resource containers rather than threads or processes. These resources are then further multiplexed among the associated threads/processes by a second-level container-specific scheduler.

Resource containers allow a Web server or proxy operator to associate a resource principal with each virtual site or client community hosted on the server/proxy. To be effective in providing differentiated services for different virtual sites or client communities, resource containers must be combined with an appropriate scheduling policy for each type of resource (CPU, memory, disk and network bandwidth).

Other related abstractions for resource principals have also been described in literature. Some examples are Activities [66] in the Rialto real-time operating system, software performance units (SPU) [104] proposed in the context of shared-memory multiprocessors, reservation domains in the Eclipse operating system [27, 26] and paths in the Scout [94] operating system. These are discussed in more detail in Section 5.

Lazy Receiver Processing (LRP)

The rest of this subsection describes the LRP network subsystem architecture that accurately accounts for network processing performed in the kernel. As this processing forms a significant fraction of the overall processing associated with HTTP requests, accurately accounting for the former is imperative in effectively supporting differentiated QoS in server systems.

Traditional operating systems provide little control over the system resources consumed by network-intensive applications. This is because of the following reasons:

- *Eager receiver processing:* Most of the network processing is performed in response to the arrival of network packets, with highest priority given to the capture and storage of packets in main memory; second highest priority given to the protocol processing of packets; and, lowest priority given to the user-level applications that consume the packets. A packet arrival always interrupts an executing application, irrespective of the scheduling priority of the application that is to receive the incoming packet.

- *Inaccurate accounting:* System resources consumed in processing network traffic are accounted to the application process that was running prior to the reception of the packet. If the system was idle before the reception of the packet, the system resources consumed during network processing are not accounted at all. These lead to inaccurate accounting and subsequently, inaccurate scheduling.

- *Lack of traffic separation:* Upon reception of a packet, a high priority hardware interrupt causes the packet to be removed from the network interface queues and placed in a single shared queue (called IP queue). Further processing of the packet is performed in the context of a lower priority software interrupt that processes packets in FIFO (first in first out) order. Due to the shared nature of the IP queue, excess incoming traffic destined for one application (or service class) can lead to delay and loss of packets for another application.

The goal of LRP is to extend the system's global resource accounting and management mechanisms to include resources consumed in network processing. To achieve this, LRP uses the following combination of techniques:

- The system's global IP queue is replaced with a per-socket queue. The network interrupt handler performs *early demultiplexing* [99] of incoming packets. In other words, packets are associated with their destined sockets as early as possible and are placed directly in the corresponding socket queues. A packet is silently discarded if the corresponding socket queue is full. This enables effective traffic separation between incoming network flows.

- Protocol processing is performed at the priority of the receiving principal. Additionally, this processing is also accounted to the receiving principal.

With the above techniques, network processing is correctly accounted to and performed at the scheduling priority of the receiving principal. The LRP architecture can be used with any reasonable notion of what constitutes a resource principal. That is, while LRP can be used in traditional systems that use processes as resource principals, it can also be used in systems that use abstractions like resource containers as their resource principals. A system that combines resource containers with LRP has been shown to provide effective support for differentiated QoS in single-workstation server systems [17, 15].

2.2.4 QoS support in Cluster-based Server Systems

In the absence of effective support for differentiated QoS in traditional operating systems, cluster-based server systems achieve performance isolation by providing separate cluster nodes for each service class. For example the various Yahoo! [110] services (search, email, etc) are hosted on separate and dedicated set of cluster nodes. Similarly, organizations often run separate server nodes for internal clients and for external clients. While this approach achieves performance isolation, it typically results in lower average utilization of cluster resources and higher average request latencies, because resources that are not currently utilized by one service class cannot be used by other service classes.

With mechanisms such as resource containers and LRP, operating systems can effectively provide support for differentiated QoS. Therefore, service classes can share cluster nodes rather than be hosted on disjoint nodes. However, supporting effective differentiated QoS in a cluster environment requires global coordination between the individual resource principals of the cluster nodes. This coordination is necessary for the purpose of allocating resources on a cluster-wide basis. This aspect is discussed in more detail in Chapter 4.

2.3 Discussion

This section discusses the limitations of QoS support in both commercial as well as prototype research server systems and motivates the goals of this dissertation. Section 2.3.1 summarizes support for differentiated QoS in server systems and motivates the need for predictable QoS. Section 2.3.2 compares QoS support in networks with that in server systems.

2.3.1 Differentiated vs Predictable QoS

The hosting of several virtual sites on a common server platform is becoming increasingly commonplace. The site operator usually charges a fee from the owners of the virtual sites in exchange for providing capacity on the server platform. In such environments, it is not only important to differentiate effectively between the resources consumed by the individual sites, but it is also becoming increasingly desirable to support predictable performance in application-level metrics such as average throughput, response time etc.

With recent technological enhancements, operating systems can provide effective support for differentiated QoS in server systems. This permits a proportional allocation of system's resources to the various service classes hosted on the system so as to provide performance isolation. Although necessary, support for differentiated QoS is not sufficient for efficiently supporting predictable QoS on server systems. For example, reserving 30% of a server's resources for a particular virtual site has little meaning for an organization that is more interested in application-level performance metrics. Such an organization might be willing to draw a contract only if the server system is capable of supporting an average throughput of say, 100 conn/s at peak loads.

At the very least, for supporting predictable QoS in application-level metrics, a site operator needs a mapping between application-level metrics and system level metrics like CPU/disk bandwidth, memory pages etc. One method of providing predictable performance then would be to reserve for a virtual site the maximal system resources that were ever needed to meet the desired performance in the application-level metric. However, a site operator is also interested in maximizing profits. For this purpose, it is desirable to host as many virtual sites on the server platform as are possible without danger of violating contracts. Statically reserving maximally needed system resources for virtual sites may not result in an efficient use of these resources. For example, consider two sites both of which need 60% of CPU bandwidth to meet their performance in the desired application-level metric. Further, assume that the peak load for the two sites occur at different times during the day and that at any given time of the day, the system is capable of meeting the loads for both sites simultaneously. However, a strategy that reserves 60% of the CPU resources for these sites would require that only one of these sites be hosted on the server system, thus resulting in inefficient use of server resources.

For efficient support of predictable QoS, a resource management framework is required that is capable of (1) dynamically mapping application-level performance into system level metrics, (2) dynamically allocating resources between the service classes so as to meet contracts in application-level metrics, and (3) detecting conditions when the system is in danger of violating contracts due to insufficient resources. Such a framework is presented in Chapter 3.

While effective support for differentiated QoS is now available in single-workstation server systems, cluster-based platforms still lack such support. Effective differentiated QoS support is a precursor to predictable QoS. Chapter 4 presents *cluster reserves* [12], a mechanism that enables effective support for differentiated QoS in cluster-based server systems.

2.3.2 Network vs Server QoS

Providing effective QoS to clients entails providing QoS both at the server systems as well as in the network. In other words, differentiation and predictability in service should be supported not only at the server system but also by the network infrastructure that connects the clients to the server system. The service quality perceived by clients is determined by the QoS support in the component that becomes the bottleneck – if the server resources are the bottleneck, then QoS support by server systems becomes important; if the network resources are the bottleneck, then perceived service quality depends upon network support for QoS. This dissertation considers QoS in server systems; QoS in the network is outside the scope of this thesis. However, solving the problem in the server/proxy is an important part of the overall solution, because user-perceived Web performance (i.e., response time) is increasingly dominated by server delays, especially when contacting busy servers.

For the purpose of this dissertation, it is assumed that sufficient network resources are available such that the service quality is dependent upon QoS support by the server system. A significant amount of research effort is being expended on developing effective QoS support in networks. This is complementary to the work in this dissertation and is discussed in detail in Chapter 5.

Chapter 3

Resource Management Framework for Predictable Quality of Service

As the World Wide Web experiences increasing commercial and mission-critical use, users and content providers expect high availability and predictable performance. Towards this end, work is being done in scalable Web infrastructure [112, 50, 7], Web content caching [87, 30, 108, 28], provision of differentiated services in the Internet, and differentiated services in Web servers and proxies [12, 17, 72, 26, 22, 69].

This chapter focuses on the provision of predictable quality of service in Web servers and proxies. Web site and proxy operators often wish to ensure that requests for certain Web pages or requests from certain clients receive a minimal level of quality of service, regardless of the load imposed by other requests. This chapter presents a feedback-based resource scheduling framework that is able to ensure a specified QoS level (e.g., average throughput or average response time) for a class of requests, despite varying resource demands (CPU, memory, disk and network bandwidth) and competing requests, as long as the collective resource requirements do not exceed the available resources in the system.

A closely related problem is that Web hosting sites and content delivery network operators (for instance, Akamai [2]) wish to provide predictable QoS levels to multiple *virtual sites* hosted on the same system. The problem for the hosting server or proxy operator is to co-locate virtual sites in such a way that (1) the contracted QoS levels are maintained with high probability, and (2) the average hardware utilization is high. To address this problem, this chapter presents a novel framework for measurement-based admission control of Web services that allows proxy and Web hosting servers to decide if a given virtual site can be co-located with other sites on the same machine or cluster, while simultaneously maintaining service contracts for each site and achieving high hardware utilization.

With the proposed approach, a new virtual site with unknown load and resource demand characteristics is first operated in "trial mode", i.e., without a guaranteed QoS level, and on a server node with plenty of available resources. During this phase, the site is exposed to its regular live workload, and both its load characteristics (average throughput and response time) as well as its demand for various resources (CPU, memory, disk and network bandwidth) are recorded by the system. Using this data, the system calculates a statistical envelope of load as well as demand for server resources of each type over a period of time, typically 24 hours.

Based on this statistical envelope, the hosting server or proxy operator can predict the resource demands (and thus the cost) needed to guarantee a certain QoS level, and can quote the content provider accordingly. Once the content provider has contracted for a given guaranteed QoS level (which can vary based on time of day, day of week, etc.), the hosting server/proxy operator can use the virtual site's measured statistical envelop to decide which virtual sites can be co-located on a given server node or cluster.

Once a set of virtual sites have been co-located on a given server node, the feedback-based resource scheduling framework guarantees that (1) all service contracts are satisfied with high probability, as long as the total resource requirements do not exceed the capacity of the hardware, and (2) unused resources are allocated to virtual sites whose current load exceeds the contracted service rate. All the while, load and resource demands of each virtual site are continually monitored to detect changes in a site's workload and resource demands, which may require relocation of the virtual site or a change in contract.

This chapter describes the design and implementation of a measurement-based admission control and resource scheduling framework that is capable of effectively supporting predictable quality of service in server systems. Results from a performance evaluation based on synthetic and trace-based workloads are presented. These results suggest that the framework is able to predict which sites can be co-located with high confidence, and can satisfy service contracts with high probability.

The rest of this chapter is organized as follows. Section 3.1 discusses the limitations of prior mechanisms that support differentiated QoS in also providing effective support for predictable QoS. The design of the proposed resource management framework is presented in Section 3.2, and a prototype implementation is described in Section 3.3. Section 3.4 presents experimental results obtained with the prototype and Section 3.5 summarizes the information presented in this chapter. Related work is covered in Chapter 5.

3.1 Limitations of prior work in supporting predictable QoS

This section discusses prior work for supporting differentiated QoS in server systems and describes its limitations in supporting predictable QoS.

In prior work on Web server QoS, proportional share scheduling policies are most often used to schedule the CPU. Examples of such policies are lottery scheduling [107, 106] and start-time fair queuing (SFQ) [53]. These policies allow a certain fraction of the system's CPU to be allocated to each resource principal. The principal is then guaranteed to be able to obtain minimally its fraction of the CPU resource whenever needed. Unused CPU cycles are allocated to resource principals that can use them.

Disk scheduling policies based on fair queuing (e.g., YFQ [25]) have been used to schedule the disk bandwidth for Web server QoS. As with proportional CPU scheduler, these policies allow a certain fraction of the disk bandwidth to be allocated to each resource principal.

Finally, policies based on fair queuing are typically used to allocate outgoing (transmit) network bandwidth among resource principals. Again, these policies allow a certain proportion of the available network bandwidth to be allocated to each resource principal.

All of these policies have been shown to be effective in guaranteeing that a predetermined fraction of each resource is available to each resource principal. The key difficulty with these approaches is that it is very difficult to relate the performance targets of a virtual site (in terms of request rate or average response time) to the allocations of each resource needed to achieve that target.

First, the incident request rate generally changes over time, leading to variable resource demands due to load variations. Second, it is not generally possible to predict the fraction of each resource needed to sustain a given request rate, since that depends on the performance of the hardware platform and the resource demands of the computation necessary to handle each request (note that handling a request may require arbitrary computations, as is the case for dynamically generated Web content). Finally, the fraction of resources needed to sustain a given request rate generally varies over time, depending on the types and sequence of requests received.

As a result, known approaches to Web server QoS are generally able to ensure that a certain fraction of Web resources is available to a given virtual site, but they are unable to guarantee that a specified request rate or average response time can be maintained. To provide such guarantees, it is first necessary to perform admission control to ensure that achieving the desired service rates for the set of virtual sites hosted on a given server/proxy is feasible, given the resources of the server/proxy machine or cluster. Second, scheduling policies must be in place for each resource that ensure that each virtual site receives the resources it needs to meet its service contract, despite variations in load and resource demands. Later sections will show that proportional share resource schedulers are not sufficient for this purpose.

The next section presents a measurement-based framework for admission control and resource scheduling that can provide differentiated services based on request rates and average response times for Web servers and proxies.

3.2 A framework for measurement-based QoS

This section presents the proposed mechanisms for providing measurement-based quality of service in server systems. These consist of (i) an *admission control* framework for admitting services into the system and determining their contracts, and (ii) a *resource scheduling framework* that ensures contracts are met despite varying resource demands.

This section begins by defining some terminology:

- *System:* A system consists of the hardware and software resources necessary for hosting services. The hardware might consist of either a single workstation or a cluster of workstations.

- *Service class:* A service class (or simply service) defines a set of requests (defined either by the URL they are requesting or the identity of the client issuing the request) such that the resources used in serving the requests are accounted for and scheduled separately from resources used to serve requests in different service classes. The notion of a service class, coincides with that of a resource principal in the system.

- *Resource class:* The resources available in a system are classified into resource classes. All resources in a resource class are scheduled independently of resources in other resource classes. Each of CPU time, memory pages, disk and network bandwidths define a resource class of their own.

- *Contractor and contractee:* A contractor is the operator of a system that is used for hosting services on behalf of one or more contractees.

- *QoS metric:* QoS metric is a service specific term that measures the service's performance and hence the quality of service it is providing to its users. Examples are average number of requests per second or average response time.

- *Contractual target:* Contractual target is a specific value of the QoS metric agreed upon by both the contractor and contractee of a service. Examples of such metrics include average throughput or response time. The contract between a contractor and the contractee is said to be in violation if, despite sufficient load, the system is unable to provide sufficient resources to the service in order for it to meet its contractual target.

- *Service type:* Service type determines the obligation of the contractor to meet the contractual target. For guaranteed services, a system must always provide sufficient resources to a service in order to meet its contract. For this purpose, the maximal resources that can ever be needed to meet the contractual target are reserved for the guaranteed services. Predictive services allow a weaker contract where occasional contract violations are acceptable. Best-effort services do not have contractual targets and are only provided with resources left over after meeting the contractual targets of guaranteed and predictive services.

The rest of this section describes the admission control and the feedback-based scheduling frameworks that form the basis of the approach for providing measurement-based quality of service in server systems.

3.2.1 Admission Control Framework

Prior to settling on a contract for a service, the contractor must determine the resource needs required in order to meet its contractual target. The goal is that even after the admission of the service, the system should remain capable of always meeting the contractual targets of guaranteed services, while those of predictive services should still be met with high probability. Services of type best-effort do not need to go through the admission control phase — the resource scheduling framework ensures that the contractual targets for guaranteed and predictive services are met before any resources are given to best-effort services.

The problem of determining resource needs is complicated by (i) varying client load (measured in requests per second) on the service over time, and (ii) varying resource demands to sustain a given load. For example, the load during a 24-hour period in a trace from IBM's main Web site (www.ibm.com) varies by a factor of five. Resource demands can vary for a given load when, for example, the average size of the content requested differs at various times during the day or when different amounts of processing are required for generating dynamic documents.

Before admitting a service into the system, the service goes through a *trial* phase where its resource requirements are determined. For this purpose, the service is hosted on a trial system, which has plenty of uncommitted resources. (The trial system may be a normal *live* system that hosts other services under contract, or it may be a special server node set aside for this purpose). The trial system should have the same hardware characteristics as the system in which the service is to be admitted. The resource usages and the QoS metric (e.g., throughput or response time) for the service are monitored for a period of time that is deemed long enough to determine the service's resource needs with high confidence. The required contractual target can either be already specified by the contractee, or the contractor can determine the feasible targets that can be supported.

The load on many Web sites follows a periodic pattern, with a fixed period (e.g., 24 hours). Therefore, the resource need of service i for resource class r is described by a family of random variables $U_{ir}(t)$, where t denotes one of k discrete, fixed length intervals in the period T. The length of the interval and the period are tunable depending upon the characteristics of the client load.

The QoS metric can exceed the contractual target when there are available resources in the system after the contractual targets of all services are met. However, the recorded resource need does not include excess resources consumed when the QoS metric exceeds the contractual target.

The random variables $U_{ir}(t)$ are continuously sampled based on usage reports by the resource scheduling framework. For this purpose, samples are collected for each interval. Let $usage_{ir}(t)$ denote the resource usage for service i for resource class r during interval t expressed as a percentage of the total resources in class r. Let $qos_i(t)$ denote the QoS metric during interval t and let $target_i$ denote the contractual target for service i. Without loss of generality, higher values of $qos_i(t)$ are assumed to be more desirable[1]. Then, at any time interval, the value of the random variable $U_{ir}(t)$ is sampled as follows:

$$U_{ir}(t) = \begin{cases} usage_{ir}(t) & qos_i(t) \ \leq \ target_i \\ usage_{ir}(t) * target_i/qos_i(t) & otherwise \end{cases}$$

Intuitively, the above estimates the resource need by scaling the reported usage when the value of the QoS metric exceeds the contractual target. Note that the framework makes the simplifying assumption that the QoS metric and resource consumption are approximately linearly related. Experimental results indicate that this is a perfectly reasonable assumption for CPU time, disk and network bandwidth.

An exception is the main memory resource. For memory, the working set required for the service to operate is determined by increasing the memory allocated to the service in a step-wise fashion until a knee is observed in the disk utilization when plotted as a function of the allocated memory. An equivalent amount of main memory is then reserved for the service when it is admitted into the live system. If no knee is observed before allocating all the available memory, the default amount of memory allocated at the start of admission control is reserved.

Estimating the disk resource needs during the trial phase also requires some care. Experimental results indicate that the combined disk needs of a set of services may exceed the sum of their individual disk needs, as measured by running them standalone on an idle system. This is because the interleaving of disk requests from different services increases the disk seek times and rotational latency overheads. To account

---

[1] For QoS metrics such as response time, $qos_i(t)$ can be computed by reciprocating the value of the QoS metric

for this, our disk scheduler implements a feature that does not permit trial services to occupy disk resources continuously for more than 50 milliseconds. If disk requests from other services under contract are not pending after 50 milliseconds of disk usage by the trial service, a disk request for a random disk block is inserted into the disk queue. This approximates the likely interference from other services when the trial service is admitted into the live system. Experimental results show that disk needs measured for the trial service using this method reflect those observed when hosted on the live system.

Let P be the set of predictive services and G be the set of guaranteed services currently admitted in the system. The admission control decision is made as follows:

- *Predictive service:* Admit the trial service S if for all intervals t the following holds:

  $$\forall_r \; P(\sum_{i \in P \cup G \cup \{S\}} U_{ir}(t) \; \leq \; thresh \;) \; \geq \; C \tag{3.1}$$

  That is, the probability that the sum of the resource needs during any interval t over all admitted services plus the trial service sums to less than a threshold $thresh$ remains high (higher than a confidence value C where $0 < C < 1$).

  The above equation ensures that the contracts of all predictive services shall be met with high probability after the service under trial is admitted into the system.

- *Guaranteed service:* Admit the trial service S if the following equation holds for all intervals t in addition to equation 3.1.

  $$\forall_r \; \max_t \; (max \; (U_{Sr}(t))) \; + \; R_r \; \leq \; 100 \tag{3.2}$$

  where

  $$R_r \; = \sum_{i \in G} \; \max_t \; (max \; (U_{ir}(t))) \tag{3.3}$$

  In addition to equation 3.1, Equation 3.2 ensures that sufficient resources are available in the system to meet contracts of existing guaranteed services even after the worst case resource needs are met for the service under trial. The admission of a predictive service cannot affect the contracts of guaranteed services because worst case resource needs for guaranteed services are reserved.

If the above conditions are satisfied, then resources corresponding to $\max_t$ $(max\ (U_{Sr}(t)))$ are reserved for the service for every resource class r and the service is admitted into the live system under contract.

The threshold *thresh* used in equation 3.1 is a fraction of the total resources available in the system, and is intended to provide a *safety margin* of available resources. It allows the resource scheduling framework to (i) to discover and measure minor increases in resource needs of any service under contract, and (ii) to reduce contract violations of predictive services. The choice of the value of *thresh* involves a tradeoff; a high value is desirable for attaining high utilization, while a low value is desirable for providing a large safety margin. Experimental results with a prototype implementation indicate that a value of 90% for *thresh* works well in practice.

The probability in equation 3.1 is estimated by approximating the probability distribution of the random variable $U_r(t) = \sum_i U_{ir}(t)$ by the distribution of the observed samples of $U_r(t)$. For example, the probability $P(U_r(t) \leq 90\%)$ is estimated by sampling $U_r(t)$ a number of times and then computing the fraction of samples that are all less than 90%.

The resource scheduling framework is described next.

3.2.2 Resource Scheduling Framework

The framework for resource scheduling ensures that contracts for services are met despite varying resource demands and client loads. The framework consists of *resource schedulers* for each resource class and a *resource monitor*. The former are operating system schedulers that multiplex resources among the services based on both their resource reservations as well as application level feedback regarding the current value of the QoS metric. The resource monitor records the resource usages for services over time and determines any contract violations.

Feedback-based resource schedulers

Existing proportional schedulers (like SFQ [53], Lottery [107], SMART [84], BVT [40] etc) are capable of proportional allocation of resources among resource principals, but they do not consider application level progress when scheduling resources.

Proportional schedulers assign each resource principal a fraction of the total available resource (e.g., a fraction of the available CPU cycles, or disk bandwidth). These schedulers allocate resources to a demanding principal as long as that principal's resource usage is at or below its assigned share. In addition, a demanding principal is allocated resources that are unused because other principals currently demand less than their share.

The feedback-based resource schedulers extend the proportional allocation model by taking application feedback into account. Each principal is assigned a target QoS metric that is reported by the application. In addition, a fraction of the total resources can also optionally be reserved for any principal (as with proportional schedulers). For each resource class r, resources are scheduled between principals based on the following rules given in order of decreasing priority:

1. All principals i such that $(qos_i(t) < target_i \ and \ usage_{ir}(t) \ < \ reservation_{ir})$
2. All principals i such that $(qos_i(t) < target_i)$
3. All remaining principals

The schedulers allocate resources to a demanding principal as long as that principal's reported QoS metric is below target. Therefore, resource allocation is primarily driven by application feedback and allowing a principal to meet its contract is the primary concern. However, priority is given to those principals whose resource usage falls below its reservation. This ensures principals with reservations a minimum system capacity on occasions when the system is incapable of meeting the target of every principal (such a condition would ultimately be detected by the resource monitor). On the other hand, reserved but unneeded resources are made available to meet contracts of other demanding principals.

A simple example serves to distinguish the scheduling framework from proportional schedulers. Let three services A, B and C be hosted on a system with resource reservations of 10%, 40% and 50% for CPU respectively. Let the contractual targets for A and B be 8 and 2 units respectively and let us assume without loss of generality that 10% of the resources are needed for every 1 unit of the QoS metric. Finally, let the load on services A and B be sufficient to consume as many resources as are provided, while there is no load on service C.

As per the first rule above, the resource scheduler would assign 10% CPU resources to service A (up to its reservation) and 20% to service B (sufficient to meet target). The 50% of the resources reserved for C and 20% of the resources reserved for B are tagged unneeded and are additionally made available to service A in order to meet its contractual target (rule 2). A traditional CPU scheduler would instead distribute all CPU resources among A and B in the ratio 1:4 (in proportion of their reservations). As a result, the QoS metrics for A and B shall be 2 and 8 and A would incur a contract violation.

Resource Monitor

The resource requirements of services hosted on a live system can change over time. This can result from either changing load patterns on services, or from changing resource requirements to maintain the same contractual target. The former can cause contract violations only for predictive services while the latter can cause contract violations even for guaranteed services. The resource monitor serves to warn the contractor to take corrective action when the system is in danger of violating contracts. Corrective actions can be either adding more resources to the system (e.g., by upgrading the hardware or by adding more nodes to a cluster), by moving some services to other systems that might be owned by the contractor, or by changing the contract for some services.

The resource monitor detects when the system is in danger of violating contracts by checking equation 3.1 for every time interval t in the period T. That is:

$$\forall_r \ P(\sum_{i \in P \cup G} U_{ir}(t) \ \leq \ thresh) \ \geq \ C \tag{3.4}$$

The probability in equation 3.4 is estimated from the relative frequency of the event when the sum total of the resource needs are less than *thresh*. If during any interval t, equation 3.4 is violated, the system is considered to be in danger of violating the contracts. The contractor of the system can choose to take corrective action after one or more violations of equation 3.4.

The resource scheduling framework trusts the services hosted on the system to report the correct value of the QoS metric. This is reasonable in environments such as Web hosting where the servers for the individual services hosted on the system are provided by the contractor. Even for environments where this is not the case, the resource monitor would flag a service for removal if it detects that it requires more resources (because of false QoS reports) for meeting the contractual target than are available in the system.

3.3 Prototype Implementation

This section briefly describes a prototype implementation of the proposed measurement-based framework for Web server QoS. Experimental results with the prototype are presented in Section 3.4.

I implemented the prototype by modifying the FreeBSD-4.0 operating system. Resource containers [17] were added to the kernel, and used to associate separate resource principals with the various services hosted on the system. The prototype supports resource management for the CPU, disk and memory resource classes.

The resource management framework is not directly applicable to the memory resource class as it is generally not linearly related to application-level QoS metrics like average throughput and response time. Instead, the prototype supports memory isolation through partitioning – a fixed number of memory pages can be assigned to a principal. Page replacements for a principal only affect the memory pages assigned to it. However, unused memory pages are made available to demanding principals until needed by principals to which they are assigned.

Scheduling of network bandwidth is not supported by the prototype. In the experimental setup used for this chapter, the network resources were not the bottleneck. However, support for scheduling network bandwidth on the server's interfaces is straightforward and is similar to the scheduling of CPU and disk bandwidth.

The Apache-1.3.12 [8] webserver was used to host the Web services. I modified Apache slightly to report its QoS metric to the kernel periodically using a new system call. The QoS metric used was average throughput measured in requests per second.

The implementation of the resource scheduling framework described in Section 3.2.2 is presented next.

3.3.1 Implementation of feed-back based schedulers

For the CPU scheduler, I extended the lottery scheduling [107] policy to implement the feedback-based resource scheduler, as described in Section 3.2.2. The scheduler was used to schedule the CPU among the resource containers, which represent the different services hosted on the system.

Lottery scheduling implements proportional share CPU scheduling by managing resources using tickets and currencies, where each principal gets resources proportional to the number of tickets it possesses. Thus, to allocate a fixed percentage of a machine's CPU to a resource principal, a proportional number of tickets are associated with it.

I modified my implementation of the lottery scheduler to periodically compute the CPU resource requirements in order to meet the contractual target for every resource principal (resource container). These are translated into an equivalent number of *needed* tickets for each principal. The total number of *available* tickets are also computed, corresponding to CPU cycles that are not currently claimed by any principals. The available tickets are distributed among principals in proportion of their needed tickets. For the purpose of scheduling, the CPU quanta are multiplexed among the principals in proportion to the *effective* number of tickets associated with that principal.

In implementing the disk scheduler, the start-time fair queuing (SFQ) [53] policy was similarly extended. This scheduler was used to schedule disk requests from various resource containers in the system. SFQ belongs to a class of scheduling algorithms derived from the weighted fair queuing [36] algorithm developed in the context of packet scheduling in routers. Weighted fair queuing maintains the notion of a virtual time line for every competing principal. The virtual time for each principal is advanced by the amount of resource consumed, weighted in inverse proportion to its reservation. Resources are assigned to the principal with the minimum virtual time.

The implementation of the CPU and disk schedulers supports hosting services under *trial* on the live system with other services already under contract. The service being considered for admission control is specially marked and resources are only allocated to it from available resources left after contractual targets of all other services have been met. This eliminates the need for a separate system for the purpose of performing admission control. The application of the admission control framework, however, remains unchanged.

3.3.2 Implementation of the Resource Monitor

The resource monitor is implemented as a Unix application process. It periodically uses a system call to sample the average resource consumption of each service for every resource class. A 24-hour period is divided into fixed length intervals (1 minute in our prototype) and samples are maintained for each interval. During each interval, the average resource needs per service for every resource class are recorded. The sampled resource needs for the past 5 days are maintained for each interval. Thus, about 58 KB of memory space is needed for a service to store samples for the CPU and disk resource classes (the implementation does not sample other resource classes). The memory requirements for storing the samples are, therefore, reasonably small.

Based upon the stored samples, for each interval the monitor determines the danger of contractual violations using equation 3.4. As discussed in Section 3.2.2, the probability is estimated from the relative frequency of the event when the sum total of the resource needs are less than 90%.

While I implemented this prototype on a single-node server for simplicity, the concepts presented in Section 3.2 are applicable to any system that supports the notion of a resource principal. For example, the concepts can be deployed on system consisting of a cluster of workstations where the resource principals are cluster reserves [12].

3.4 Experimental Results

This section presents performance results obtained with the prototype implementation. The results are based on both synthetic as well as real traces derived from Web server logs. In all experiments, throughput as measured in connections per second (same as requests per second for HTTP/1.0) was considered as the QoS metric.

As mentioned in Section 3.3, a slightly modified Apache-1.3.12 Web server was used, running on a FreeBSD-4.0 kernel, extended with resource containers and with the resource management framework presented in this chapter. All experiments were performed on a 500MHz Pentium III machine configured with 1 GB of main memory. The Web requests were generated by a HTTP client program designed for Web server benchmarking [16]. The program can generate HTTP requests from synthetic or real logs either as fast as the Web server can handle them or at a rate dictated by timestamps in the log. Seven 166 MHz Pentium Pro machines were used to run the client program.

The server machine and the seven client machines were networked via a 100Mbps Ethernet switch. Available network bandwidth was not a limiting factor in any of the experiments reported in this section.

3.4.1 Feedback-based CPU scheduler

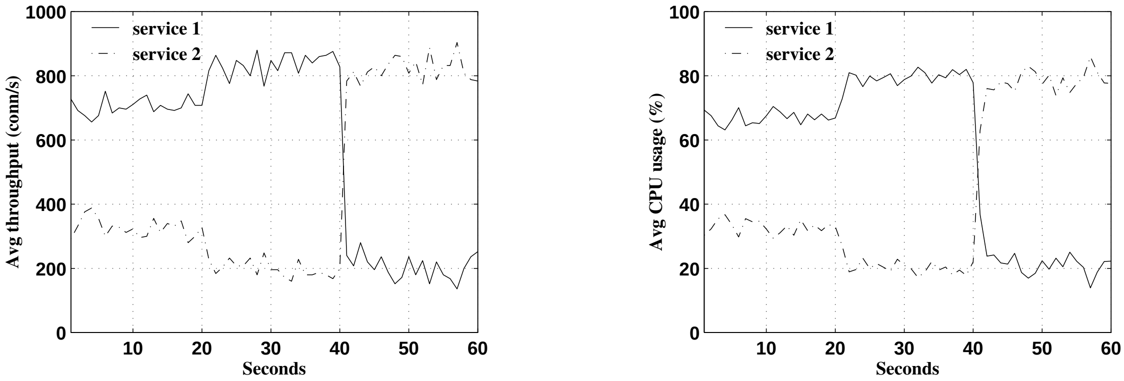

Figure 3.1 : **Feedback-based CPU scheduling**

The first experiment demonstrates the operation of the feedback-based CPU scheduler. Three services were hosted on the prototype and CPU reservations of 40%, 20% and 40% were made for them, respectively. The client programs generated requests for the first two services, while service 3 did not have any load. A synthetic workload was used where all requests are for a single file of size 6KB. The rate of request generation matched the capacity of the server.

Figure 3.1 shows the throughput and CPU usage of both services over time. For the first 20 seconds of the experiment, no contractual targets were set for the services. In this phase, the scheduler behaves like a conventional proportional scheduler and distributes CPU resources between service 1 and service 2 in proportion to their reservations, i.e. 2:1.

After 20 seconds, contractual targets of 1000 conn/s and 200 conn/s were set for service 1 and service 2, respectively. The figure shows that this results in an allocation of 80% and 20% to the respective services. The reason is that service 2 can meet its contractual target with its reservation of 20%, while service 1 needs additional resources. The scheduler realizes that the 40% resources reserved for service 3 are available, and provides them to service 1. Even at a 80% CPU allocation, service 1 remains unable to meet its contractual target of 1000 conn/s. However, as no more resources are available, the throughput of service 1 is limited to 800 conn/s.

At 40 seconds into the experiment, the contractual targets of services 1 and 2 are reversed – i.e., they are assigned values of 200 conn/s and 1000 conn/s, respectively. The scheduler reacts accordingly and as a result, the CPU allocations of services 1 and 2 become 20% and 80%, respectively. This is because service 1 can meet its contractual target of 200 conn/s with 20% of the CPU resources. The remainder of the 40% reserved for service 1 and the 40% reserved for service 3 are considered available. Therefore, service 2 receives these 60% of the CPU resources and its effective allocation becomes 80%.

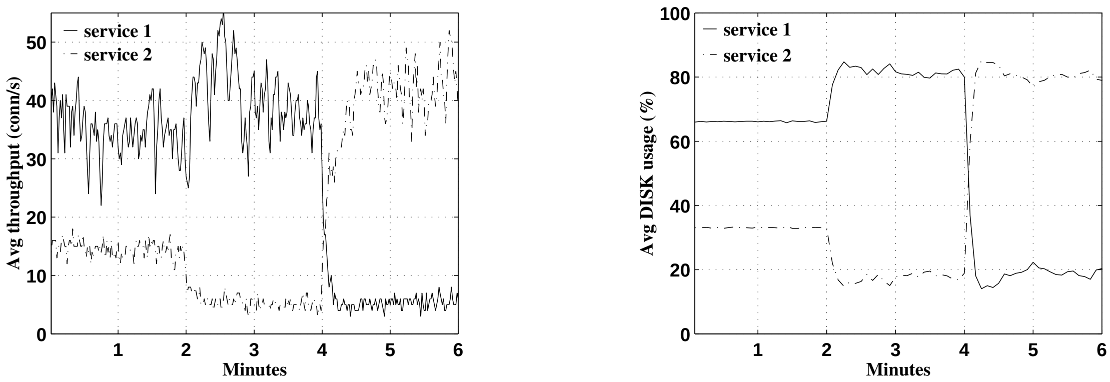

Figure 3.2 : **Feedback-based disk scheduling**

Figure 3.2 presents results from a similar experiment designed to demonstrate the operation of the disk scheduler. Again, reservations of 40% and 20% were made for two active services that both generate a significant disk load. Targets of 40 conn/s and 5 conn/s were assigned after 2 minutes of experimental time and reversed after 4 minutes of experimental time. Like the CPU scheduler in the previous experiment, the disk scheduler adjusts the disk allocation in order to meet the respective targets.

3.4.2 Resource Monitor

An experiment to demonstrate the operation of the resource monitor (Section 3.2.2) is next presented. With the help of artificially varying load patterns introduced in synthetic traces, I show the detection of situations that lead to contract violations.

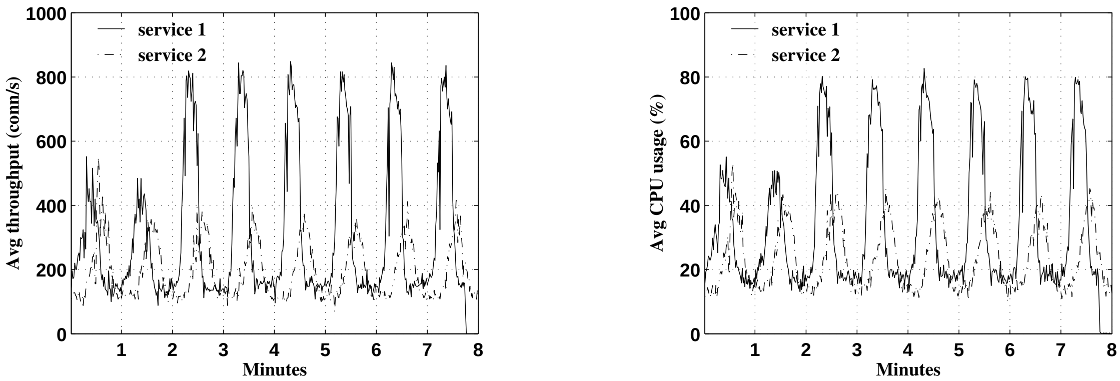

Figure 3.3 : **CPU:** $target_1$ **= 1000,** $target_2$ **= 500**

Two services were hosted on the server. The first service is predictive while the second is guaranteed. The traces containing the requests for the services were produced by synthetically generated timestamps to resemble the load characteristics typical of real Web servers. Typically, loads on Web servers reach their peak at certain hours of day while remaining low at other times. For experimental convenience, rather than reflecting this variation across a day, the synthetic traces reflect this variation over a period of 60 seconds. The client programs issued requests for the services based on timestamps in the synthetic traces. All requests were for a cached 6 KB file. The experiment was conducted for a period of 8 minutes, which reflects an equivalent of 8 days of simulated periodic load variation on a real Web server. The trace for service

1 was such that after two minutes of experimental time (i.e., two days), the peak load increases unexpectedly and significantly (enough to consume all available CPU resources). The contractual target for the first service (predictive) was set to 1000 conn/s. The contractual target for the second service (guaranteed) was 500 conn/s and it was given a CPU reservation of 55%, which was sufficient to meet its peak demand.

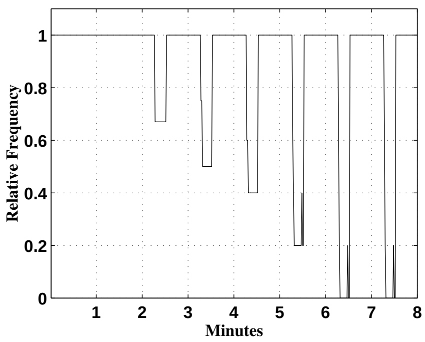

Figure 3.4 : **Confidence Plot**

Figure 3.3 shows the variation of throughput and CPU usage with time. Figure 3.4 plots the relative frequency of the event when the collective resource needs of all services are less than *thresh* (as computed by equation equation 3.4). This value is computed from samples of resource needs taken every second in the experiment. The values for *thresh* and C were chosen to be 90% and 0.75, respectively.

The results from Figure 3.4 indicate that for the first two minutes of the experiment, the relative frequency of the event when the collective resource needs are less than 90% of the total CPU resources remains high. However, once the load on service 1 increases, the relative frequency drops down in the corresponding time intervals. Figure 3.3 also shows that despite the increased load on service 1, the performance of the guaranteed service remains unaffected. This shows that the contracts of guaranteed services cannot be affected by load variations of other services. On the other hand, short term contract violations of predictive services can occur, but the resource monitor is capable of reporting this early, so that corrective action can be taken.

3.4.3 Real Traces

Results with workloads obtained from real Web servers are presented next. The traces were derived from logs of four Web servers: (i) the Rice University Computer Science departmental server; (ii) the server for the 1998 Soccer World Cup; (iii) IBM corporation's main server (www.ibm.com), and (iv) Google's main server (www.google.com). The Rice trace spans a period of 15 days in March 2000 and contains requests for 15000 distinct files with a dataset of 1.13 GB and an average request size of 34 KB. The trace from the Soccer World Cup covers 6 days and contains requests for 5163 distinct files with a dataset of 89 MB and an average request size of 6 KB. The IBM trace spans a period of 6 days and contains requests for 38500 distinct files with an average request size of 3 KB. While the above traces consist mainly of requests for static documents, the trace from Google consists solely of CGI requests for dynamic content. This trace spans a period of 6 days with an average request size of 12 KB and an average response time of 0.721 seconds.

To generate suitable values of load on the server by these traces, and also to reduce experimental runtime, I modified the timestamps in the original traces in such a way that the load was scaled in the various traces by different factors, while simultaneously preserving the synchronization between the traces with respect to the daily load variations. The client program played requests from the processed traces based upon these modified timestamps.

To achieve this compression of the workload in time, one can take several approaches, such as taking future requests in the original trace or repeating requests in the same period to supplement the load in the processed trace. All of these approaches change the characteristics of the original trace in subtle ways. However, I explored both approaches and found that the results of the experiments were virtually unaffected by this choice.

Experiments that show the throughput and resource usage trends for the various traces mentioned above are now presented. Figures 3.5, 3.6, 3.7 and 3.8, respectively, show these trends for the Rice Computer Science trace, the 1998 Soccer World Cup trace, the trace from IBM's main webserver[60], and the trace from Google's main webserver[52]. The first three traces consist mainly of static document requests while the trace from Google consists exclusively of CGI requests. The experiments were conducted when the services were run in trial mode. To observe disk utilization, memory partitioning was deployed for each experiment such that the Rice trace was assigned 30 MB, the World Cup trace 20 MB, the IBM trace 40 MB, and the Google trace 60 MB.

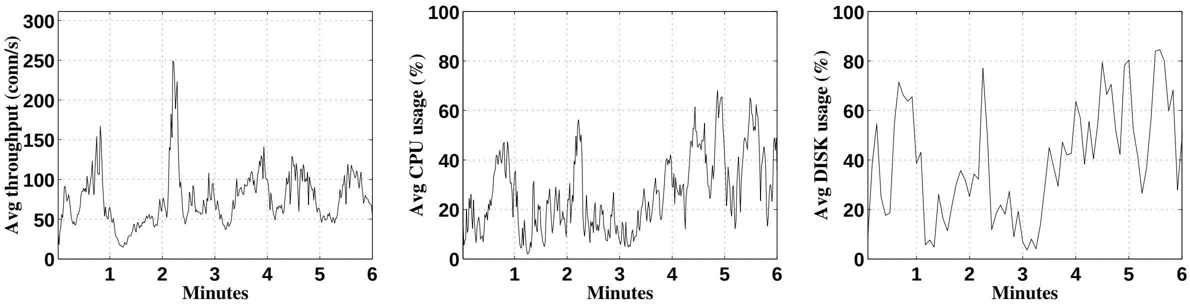

Figure 3.5 : **Rice trace**

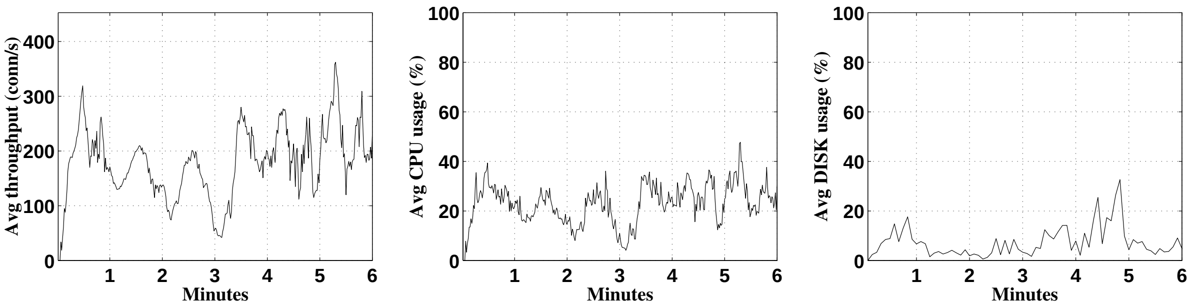

Figure 3.6 : **1998 Soccer World Cup trace**

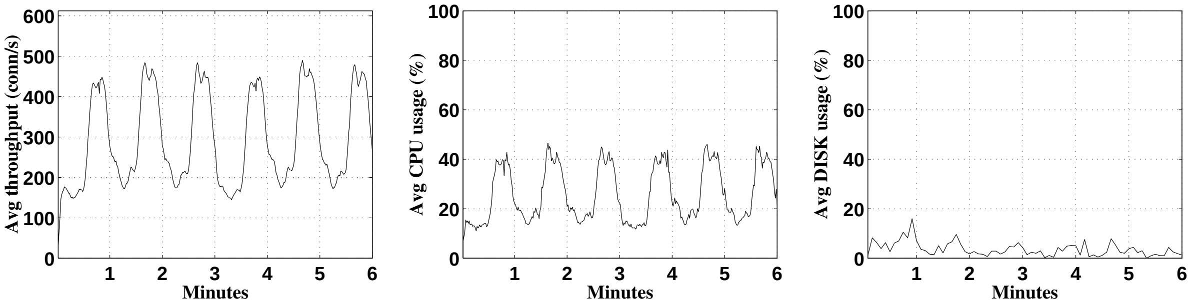

Figure 3.7 : **IBM trace**

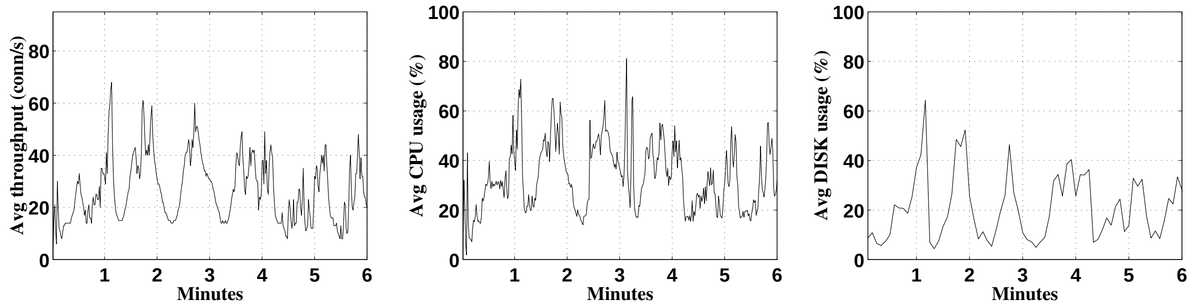

Figure 3.8 : **Google trace**

Using these traces, I performed experiments to demonstrate the admission control and resource scheduling frameworks presented in Section 3.2. I next present experimental results under three different conditions, namely (1) the CPU was the bottleneck resource, (2) the disk was the bottleneck resource, and (3) both CPU as well as the disk were the bottleneck.

CPU as bottleneck resource

I performed an experiment where the services corresponding the the World Cup trace and the IBM trace are hosted on the live system as predictive services with contractual targets of 200 conn/s and 450 conn/s, respectively. The service corresponding to the Rice trace is considered for admission in this system and is made to operate in trial mode for this purpose as a prospective predictive service. The resources on the server were configured so as to make CPU the bottleneck resource. That is, the memory was partitioned so as to comfortably fit the working sets of each of the services in memory.

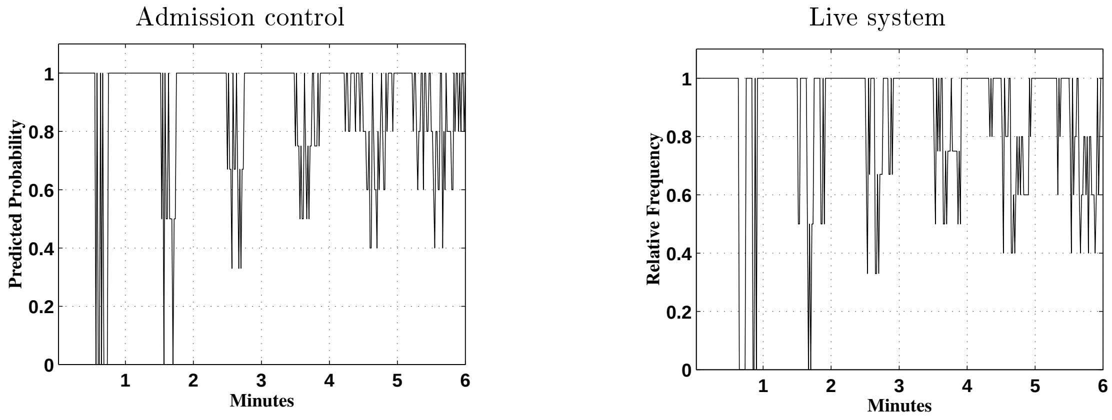

Figure 3.9 : **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Rice}$ **= 100**

A contractual target of 100 conn/s was first considered for the Rice trace. Based on the resource usage on the idle machine, and on the collective usage of services on the live system, the admission control framework computes the probability from equation 3.1 at one second time intervals (the value of $thresh$ is 90% and the value of C is chosen to be 0.75). The left plot in Figure 3.9 shows results from the admission control framework indicating that the service cannot be admitted into the live system with a contractual target of 100 conn/s (as there are times when the probability is less than $C = 0.75$). The right plot in Figure 3.9 shows results from the resource monitor when that service was admitted despite the rejection of the admission control framework. It plots the relative frequency of the event when the collective resource needs of all services are less than 90%. The close agreement between the plots shows the predictive power and effectiveness of the admission control framework.

Admitting the service corresponding to the Rice trace with a contractual target of 24 conn/s was considered next. The left plot in Figure 3.10 shows the probability as computed by the admission control framework and indicates that the service can

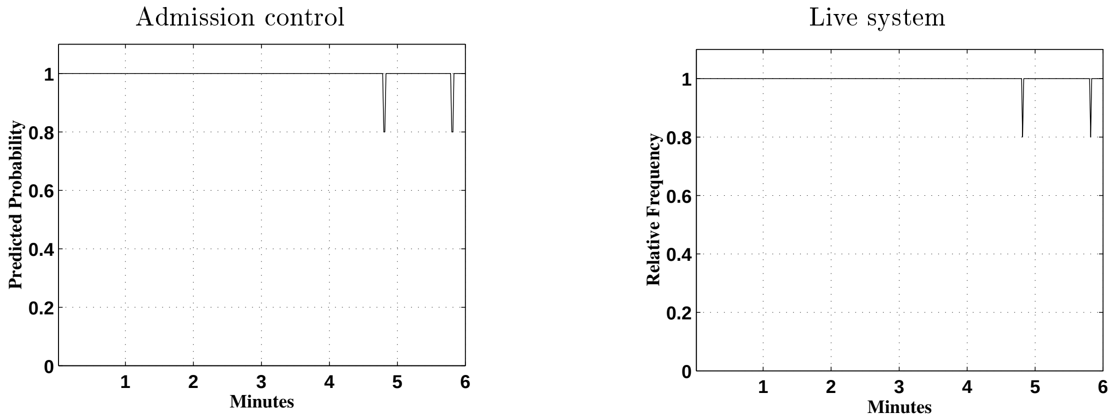

Figure 3.10 : **CPU:** $target_{WC}$ **= 200,** $target_{IBM}$ **= 450,** $target_{Rice}$ **= 24**

be admitted into the live system. The right plot in Figure 3.10 shows shows results produced by the resource monitor after actually hosting the service on the system. The two plots closely agree with each other as before.

Next I considered whether the three services could be hosted jointly on the server as guaranteed services. From the CPU usage and throughput on the idle prototype, I computed the maximum CPU resources required for meeting the contractual targets for each of the three services. These were 66.50%, 42.68% and 55.66%, respectively, for the Rice, World Cup and IBM traces. As these total up to more than 100%, this implies that not all three services can be hosted on the prototype as guaranteed services. This demonstrates that the weaker contracts of predictive services allow higher system utilization, while still maintaining contracts with high probability.

Serving dynamic content requests involves the execution of arbitrary server applications and often involves database queries and updates. Therefore, both the amount and the variance of resource consumption per request tends to be much larger for dynamic than for static content. To ensure that the proposed resource management framework is effective under such conditions, I performed experiments with workloads that contain CGI requests.

I repeated the experiments above, with the Rice trace and the World Cup trace

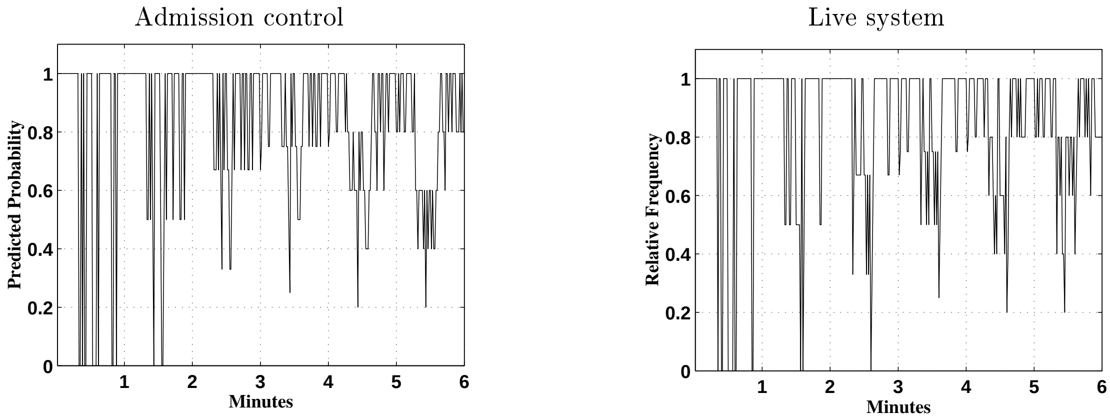

Figure 3.11 : **CPU:** $target_{Rice}$ **= 70,** $target_{WC}$ **= 350,** $target_{Google}$ **= 50**

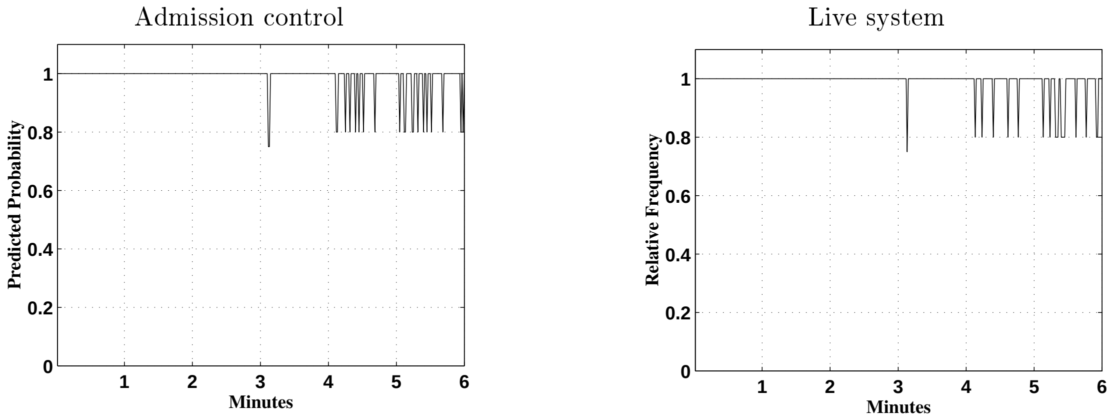

Figure 3.12 : **CPU:** $target_{Rice}$ **= 70,** $target_{WC}$ **= 350,** $target_{Google}$ **= 12**

hosted on the live system with targets of 70 conn/s and 350 conn/s, respectively, while the service corresponding to the Google trace was considered for admission control in the system. All requests for the service corresponding to the Google trace were handled by a CGI program. For each request, the CGI program responds with content of a size corresponding to the size specified in the trace. Moreover, for each request, the program consumes CPU time corresponding to the response time specified in the Google trace.

Figure 3.11 show the results from the admission control framework and the resource monitoring system when the service was considered for admission with a target of 50 conn/s. Similarly, Figure 3.12 shows the corresponding results when the service is considered with a target of 12 conn/s. The results show that the resource management framework is also applicable to services that consist mainly of requests for dynamic content.

Disk as bottleneck resource

An experiment is now presented where the server was configured so as to make the disk subsystem the bottleneck resource. To achieve this, the memory was partitioned so as to allocate an amount of memory that is only slightly larger than the working set of each service. In particular, 20 MB, 40 MB and 30 MB were assigned to the World Cup, IBM and Rice traces, respectively.

The services corresponding the the World Cup trace and the IBM trace are hosted on the live system as predictive services with contractual targets of 250 conn/s and 350 conn/s, respectively. As before, the service corresponding to the Rice trace is operating in trial mode on the live system.

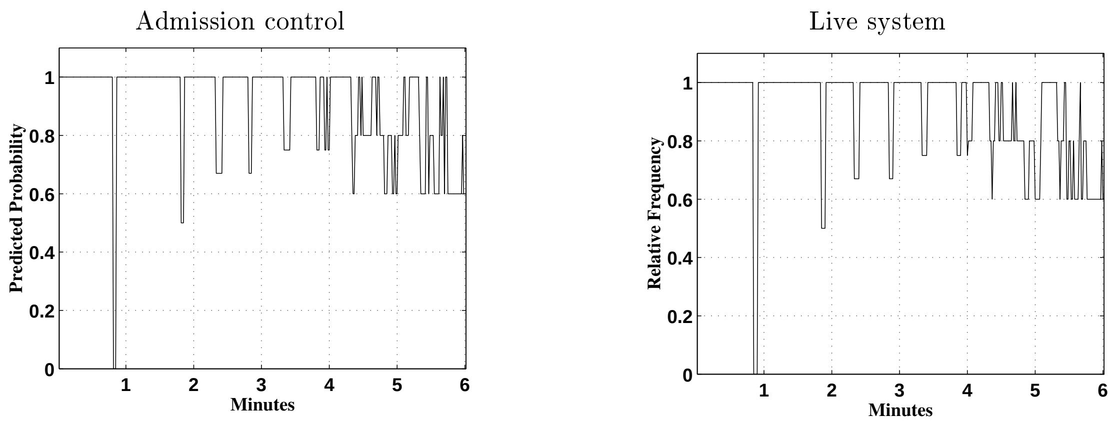

Figure 3.13 : **Disk:** $target_{WC}$ **= 250,** $target_{IBM}$ **= 350,** $target_{Rice}$ **= 50**

A contractual target of 50 conn/s was first considered for the Rice trace. The left plot in Figure 3.13 shows the probability versus time as computed by the admission control framework. As the probability falls below $C = 0.75$ at several times, the service cannot be admitted into the live system with a target of 50 conn/s. The right plot in Figure 3.13 shows results from the resource monitor when that service was admitted despite the rejection of the admission control framework. The plot depicts the relative frequency of the event when the collective resource needs are less than 90%. The close agreement between the two plots in Figure 3.13 shows that the admission control framework is capable of accurately characterizing the live system in the presence of a disk-bound set of services.

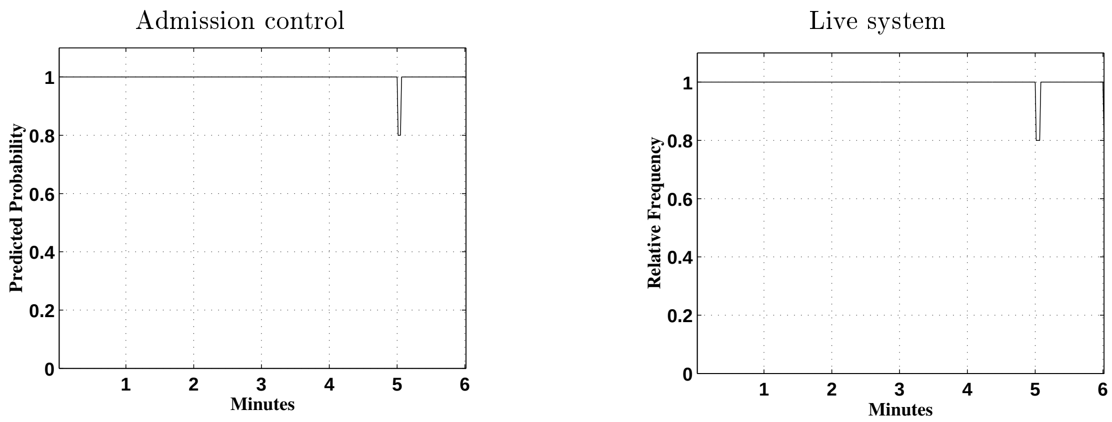

Figure 3.14 : **Disk:** $target_{WC}$ $=$ **250,** $target_{IBM}$ $=$ **350,** $target_{Rice}$ $=$ **20**

The service corresponding to the Rice trace was reconsidered for admission with a target of 20 conn/s. The left plot in Figure 3.14 shows results from the admission control framework and indicates that the service can be admitted into the live system (as at all times the value of the probability remains above $C = 0.75$). The right plot in Figure 3.14 is similar and shows results from the resource monitor after actually hosting the services on the system.

CPU and Disk as bottleneck resources

An experiment is now described where the server was configured so as to make both the CPU as well as disk resources to be a bottleneck. The services corresponding to the Rice trace and the World Cup trace were hosted on the live system as pre-

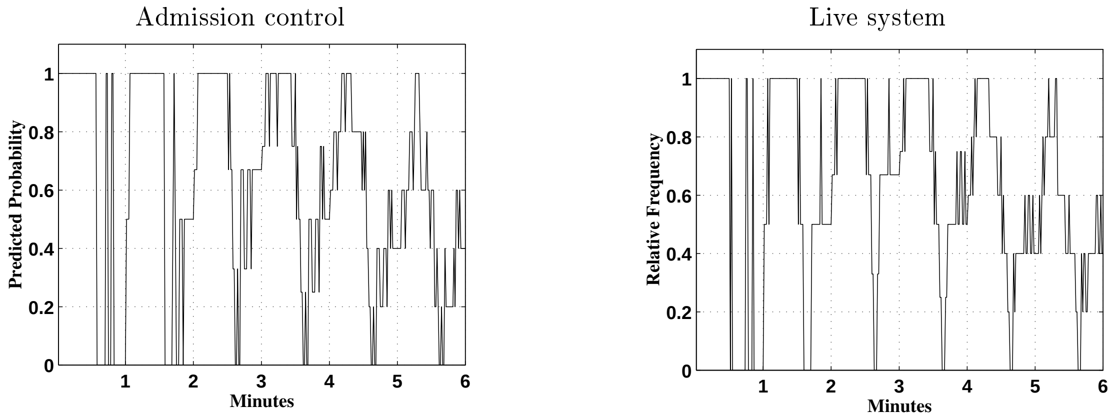

Figure 3.15 : **CPU:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

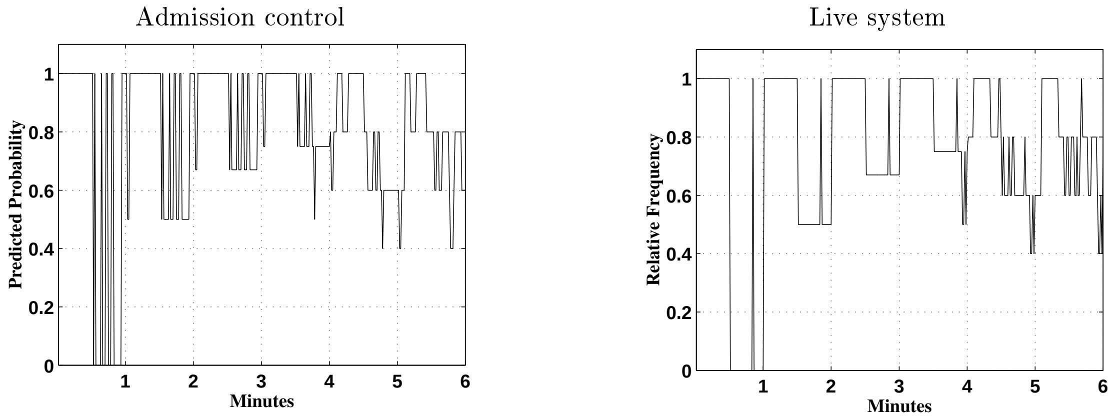

Figure 3.16 : **Disk:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 50**

dictive services with contractual targets of 100 conn/s and 250 conn/. The service corresponding to the Google trace is considered for admission into this system. In order to stress the disk subsystem, the memory was partitioned so as to allocate an amount of memory that is only slightly larger than the working set of each of the static services. Thus, 30 MB and 20 MB were assigned to the Rice and World Cup services, respectively. Further, the CGI script that handles requests for the Google trace was modified to access an uncached 8 KB document for requests with response time larger than 10 milliseconds. 60 MB of memory were assigned to the service corresponding to the Google trace.

A contractual target of 50 conn/s was first considered for the Google trace. The probability plots produced by the admission control framework are shown in Figures 3.15 and 3.16 for the CPU and the disk resource class, respectively. As the probability falls below $C = 0.75$ several times in each case, the service cannot be admitted into the live system with a target of 50 conn/s. Figures 3.15 and 3.16 also contain results from the resource monitor when the service is admitted despite rejection by the admission control framework. These depict the relative frequency of the event when the collective resource needs are less than 90%. Again, the close agreement between the plots from the admission control and the resource monitor shows that the admission control framework is capable of accurately characterizing the live system.

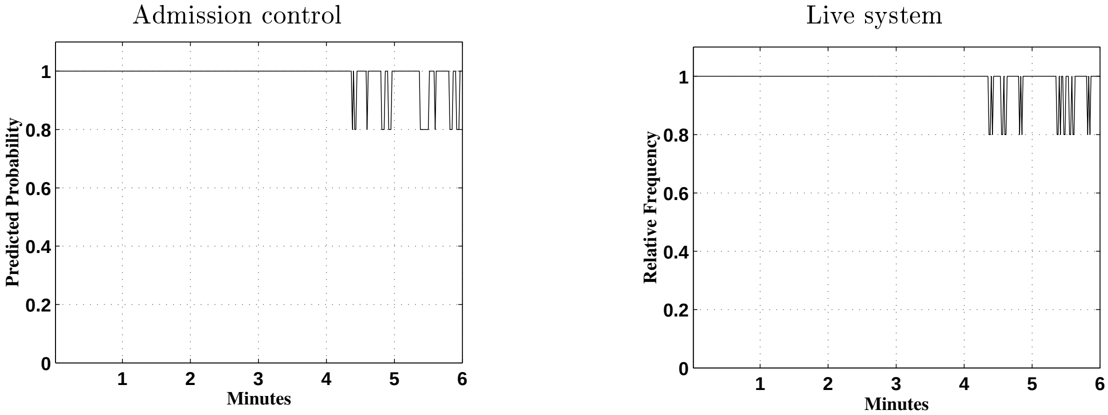

Figure 3.17 : **CPU:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

The service corresponding to the Google trace was reconsidered for admission with a target of 5 conn/s. Figures 3.17 and 3.18 show the results for the CPU and the disk resources respectively. The plots from the admission control framework indicate that the service can be admitted into the live system (as at all times the value of the probability remains above $C = 0.75$). The plots produced by the resource monitor after hosting the services on the live system are similar to those from the admission control framework.

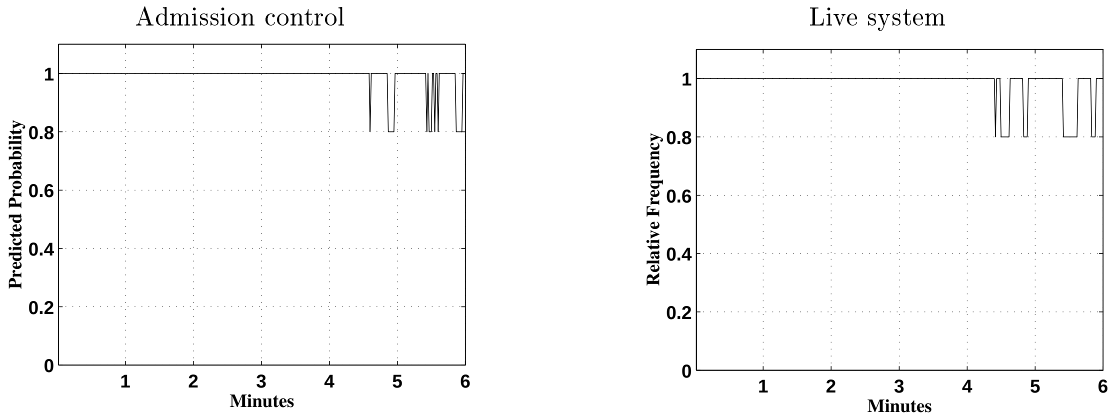

Figure 3.18 : **Disk:** $target_{Rice}$ **= 100,** $target_{WC}$ **= 250,** $target_{Google}$ **= 5**

3.5 Summary

This chapter presented a measurement-based resource management framework for providing differentiated and predictable quality of service in Web servers. The framework allows Web servers and proxy operators to co-locate virtual sites and Web services, while simultaneously providing predictable quality of service, in terms of average request rate or average response time.

The framework consists of a measurement-based admission control process that allows operators to determine whether a set of sites can be co-located on the same server system, based on the measured statistics of the sites' resource consumption under its live workload, and its desired quality of service and service class (guaranteed, predictive, or best effort). Once a set of services has been admitted, feedback-based resource schedulers ensure that all sites achieve their QoS targets with high probability, while being allowed to use excess resources not currently claimed by other sites.

An empirical evaluation of a prototype implementation shows that the system is able to predict with high confidence if sites can be co-located on a system, that it is able to maintain the target QoS levels of admitted sites with high probability, and that it is able to achieve high average hardware utilization on the server system.

Chapter 4

Cluster Reserves

Web servers based on clusters of commodity PCs or workstations offer a cost-effective and scalable solution to the increasing performance demands placed on popular Web sites. As the volume, variety and sophistication of services offered on the World Wide Web (WWW) increases, such servers must host a rich set of services on a common hardware platform. Examples of such services are the retrieval of static and dynamic Web pages, online databases for information retrieval and electronic commerce, and search engines.

It is often desirable to support differentiated QoS in servers so that individual services hosted by a Web site be *performance isolated* from each other. That is, a minimal proportion of server resources is reserved for a service, independent of the present demand for other services. Similarly, sites often wish to ensure that a minimal fraction of server resources be available to serve requests from a certain client community, independent of load generated by other clients. The need for differentiated QoS arises, for instance, when different services are being paid for by different content providers (e.g., Web hosting), when services have different priorities (e.g., purchasing versus browsing in an e-commerce site), or when an organization wishes to preferentially serve certain client communities (e.g., internal versus external clients).

As in the rest of this dissertation, I use the term "service class" to refer to a set of requests for which the server wishes to reserve a certain minimal amount of resources. Each incoming request can be classified as belonging to exactly one service class. Requests can be classified based on the requested resource (i.e., the requested URI), the originator of the request (as indicated by the client network address or some stronger form of client authentication), or both.

State of the art Web sites typically achieve performance isolation by providing separate server nodes for each service class. For example, the various Yahoo! services (search, email, etc.) are hosted on dedicated and separate sets of server cluster nodes. Similarly, organizations often run separate server nodes for internal clients and for external clients. While this approach achieves performance isolation, it typically results in lower average utilization of cluster resources and higher average request latencies, because resources that are not currently utilized by one service class cannot be used by other service classes.

Recent advances in operating systems research [66, 27, 104, 17, 26] allow effective differentiated QoS in single node Web servers. In this chapter, I address the problem of providing differentiated QoS in cluster-based network servers. I propose a new cluster-wide abstraction called *cluster reserve* that is capable of achieving performance isolation between service classes that share the cluster nodes. Cluster reserves act as cluster-wide resource principals by extending the resource management facilities in individual nodes to the cluster. A cluster resource manager is responsible for mapping the resources assigned to a cluster reserve to individual nodes in the cluster. The mapping is dynamically adjusted based on prevailing load conditions and is independent of the request distribution strategy employed in the cluster.

I use a set of benchmarking scenarios to evaluate performance isolation in cluster-based Web servers and present performance results based on a prototype implementation of cluster reserves under synthetic and trace-based workloads. The results show that (1) hosting multiple service classes on a joint set of server nodes affords higher average resource utilization and higher average throughput when compared to the state-of-the-art approach of dedicating cluster nodes to service classes; and, (2) cluster reserves are capable of achieving effective performance isolation when different service classes share a common set of cluster nodes.

The rest of the chapter is organized as follows. Section 4.1 describes the basic design of the cluster reserves abstraction. Section 4.2 discusses the prototype implementation and Section 4.3 presents performance results obtained with the prototype in a number of benchmark scenarios. Section 4.4 summarizes this chapter.

4.1 Basic Design

This section presents cluster reserves, a facility for achieving differentiated QoS between service classes hosted on a cluster. Cluster reserves are cluster-wide resource principals obtained by logically combining the resource principals on individual cluster nodes. Resources can then be allocated to this cluster-wide resource principal and get translated into allocations for the resource principals (e.g., resource containers) on the individual cluster nodes. Support for differentiated QoS on individual cluster nodes, thus, is extended to achieve differentiated QoS for the cluster.

Support for differentiated QoS in a system implies that individual services hosted on the system can be performance isolated from each other. A system is said to be capable of affording performance isolation between service classes hosted on it if (1) the system permits its resources to be proportioned between the service classes hosted on it, and (2) given sufficient request load, a service class receives at least as much resources as were assigned to it irrespective of the load on other service classes. A desirable third property is that resources not used by some service class may be distributed amongst other services classes.

As I will show, cluster reserves afford effective performance isolation for multiple service classes hosted on a joint set of cluster server nodes. Furthermore, this approach allows unused resources from one service class to be utilized by other service classes. This sharing yields increased utilization and increased throughput when compared to the current approach of hosting different service classes on separate server nodes. This chapter is primarily concerned with the CPU time resource. However, the proposed techniques can be applied to manage other resources as well.

Figure 4.1 depicts the hierarchical relationship between cluster reserves (cluster

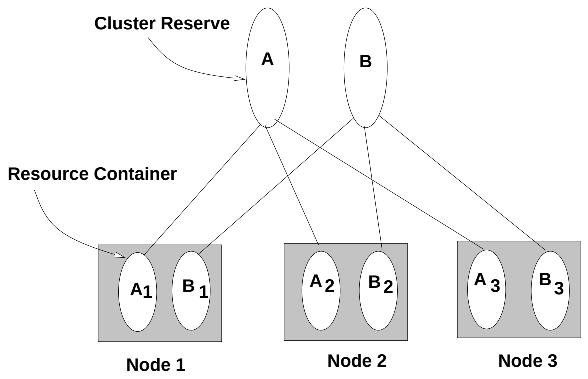

Figure 4.1 : **Cluster Reserves**

resource principals) and resource containers (node resource principals). The figure shows a cluster with three nodes and two cluster reserves A and B. Resources assigned to A and B are dynamically split into resource assignments for the corresponding resource containers on each node. Resource allocations to corresponding resource containers on each node add up to the desired allocation for the associated cluster reserve. For example, resources assigned to A_1, A_2 and A_3 add up to the desired allocation for cluster reserve A.

The partitioning of the resources allocated to a cluster reserve amongst the corresponding resource containers is determined by a cluster resource manager. In general, such a partitioning can be made in an infinite number of ways. For example, an allocation of 50% to cluster reserves A and B in Figure 4.1 can be achieved with the partitioning $(A_1 = B_1 = A_2 = B_2 = A_3 = B_3 = 50\%)$ or with $(A_1 = B_2 = 100\%, B_1 = A_2 = 0\%, A_3 = B_3 = 50\%)$. A good partitioning depends upon the resource usage on the individual nodes. For example, the first partitioning is suitable in situations when all nodes are equally loaded with requests from service classes associated with reserves A and B. However, the second partitioning is more suitable when *Node 1* and *Node 2* only get requests associated with service classes A and B, respectively.

To compute the partitioning, the cluster resource manager collects resource usage statistics from the cluster nodes and maps the allocation problem to an equivalent constrained optimization problem. The resource usage statistics and the target cluster allocations form the constraints for the problem and the solution yields the individual per-node resource allocations. This method is independent of the request distribution strategy deployed in the cluster.

I now formally show how to map the cluster resource management problem to a constrained optimization problem. First, I describe the inputs and outputs of the problem and state the goal. Then I outline the steps involved in formulating and solving the problem.

I first define the notion of a *resource sink*. A node is considered a resource sink with respect to a particular service class if all resources allocated to the corresponding resource container at that node are being fully utilized by the service class. Intuitively, a resource sink for a particular service class can potentially make use of an increase in resource allocation to the corresponding container.

Goal: Let the cluster consist of N nodes and S service classes. Each service class is associated with a distinct cluster reserve and a distinct resource container at each cluster node. Let r and u be NxS matrices such that r_{ij} and u_{ij} denote the percentage resource allocation and resource usage, respectively, at node i for service class j. Let D be a vector composed of S elements such that D_j gives the desired percentage resource allocation for the cluster reserve corresponding to service class j. Given input matrices r and u and the vector D, the resource manager computes a NxS matrix R such that R_{ij} gives the new percentage resource allocation for service class j on node i.

The constraints for the problem are formulated using two distinct steps, each of which solves a constrained optimization problem. The first step computes the least feasible deviation between the desired and actual allocations. The second step computes the new resource allocations such that (1) the deviation computed in the first step is achieved, and (2) the computed resource allocations are close to the service class usage on each node. Finally, a third step is used to distribute unassigned cluster resources to idle service classes whose allocations fall below their desired cluster-wide allocation.

Step 1: The objective in this step is to choose the matrix R such that the cluster-wide allocation deviates the least from its desired allocation. This can be stated formally as:

$$Minimize \sum_{j=1}^{S} |\sum_{i=1}^{N} R_{ij} - N * D_j| \tag{4.1}$$

Additionally, the problem is constrained as follows:

- The resource allocations on any cluster-node should sum to no more than 100. That is,

$$\forall_{i=1}^{N} \sum_{j=1}^{S} R_{ij} \leq 100$$

- On any node, the new allocation should be no more than the usage if the node is not a resource sink i.e. if the previous allocation exceeds the usage.

$$\forall_{i,j}\ R_{ij} \leq u_{ij}\ if\ r_{ij} > u_{ij}$$

- A minimum allocation of 1% is imposed so as to allow some progress to requests whose service classes happen to have the minimal allocation.

$$\forall_{i,j}\ R_{ij} \geq 1$$

The above problem can have infinitely many solutions (i.e. many different matrices R) each of which, however, yields the same value V for equation 4.1. In this step, I am primarily interested in the value V. Intuitively, V reflects the minimum feasible deviation of the new allocations from the desired ones. The purpose of Step 2 is to choose a solution R, that is the most desirable while still yielding the value V for equation 4.1.

The value of V computed from this step shall be zero if the resources can be assigned such that the desired cluster-wide allocation is met for all service classes.

V can be greater than zero due to the presence of nodes that are not resource sinks with respect to some service classes. A simple example where V will be greater than zero is when no node is a resource sink with respect to a particular service class.

Step 2: A desirable solution matrix R is such that it deviates minimally from the reported resource usage values, while still yielding the minimal feasible value V for equation 4.1 in Step 1. For this reason, I formulate another constrained optimization problem that adds the following constraint to the constraints for the problem in Step 1:

$$\sum_{j=1}^{S} |\sum_{i=1}^{N} R_{ij} - N * D_j| = V$$

The objective can be stated as:

$$Minimize \sum_{i=1}^{N}\sum_{j=1}^{S}(R_{ij} - (u_{ij} + k_{ij}))^2 \tag{4.2}$$

The term k_{ij} is a small offset (not more than 5 in absolute value) that is intended to bias the solution towards equal allocation to any service class across all the nodes. The small offset, if positive, serves to probe whether a particular service can make use of more resources on a node that is a resource sink with respect to that service[1].

$$\begin{aligned} k_{ij} &\equiv min(5, 500 * (D_j - u_{ij})/D_j) \quad && if\ u_{ij} < D_j \\ k_{ij} &\equiv max(-5, 500 * (D_j - u_{ij})/D_j) \quad && otherwise \end{aligned}$$

The solution to this problem yields the matrix R whose elements R_{ij} are further processed in Step 3.

Step 3: The solution from Step 2 might still result in some service classes whose allocations do not add up to their desired cluster-wide allocation (the value V from Step 1 shall be non-negative). This is possible only if some of the nodes are not resource sinks with respect to such service classes and hence would have unallocated resources in the solution matrix R. These unassigned resources are allocated to

[1]A resource sink can potentially make use of more resources; however, the absolute resource demand is not known a priori.

such service classes so as to bring up their aggregate cluster-wide allocation to the desired allocation level. This makes these resources immediately available to these service classes if needed. Any unused resources not used are proportioned amongst other service classes dynamically by the resource container mechanism. The resulting matrix R, therefore, is such that all cluster resources are fully assigned (all allocations on any node sum to 100). The values of R_{ij} are then used to set the allocation of service class j on node i.

Constrained optimization problems are well understood and commercial tools are available for solving problems with thousands of variables and constraints. However, due to the relatively small problem size that needs to be solved by the resource manager, I used a freely available software tool called **LOQO** [102, 103]. It is also possible to write a hand-tuned solver for this specific problem, but owing to the satisfactory speed afforded by LOQO, I did not adopt this approach.

In the following subsection I show a simple example that demonstrates the resource allocations computed by the resource manager after collecting usage statistics for the services from the various cluster nodes.

4.1.1 Dynamics of the resource manager

The example consists of a cluster with two nodes that each host two service classes (Svc 1 and Svc 2). Each service class reserves 50% of the cluster capacity.

Table 4.1 shows a series of successive calls invoking the optimizer. Every invocation uses the current resource usages for the two services on each node as input. Also used as input is the information whether a node is a sink with respect to a service (this is given in parenthesis along with the usage). Each call computes the corresponding allocations for each service on each node.

The resource manager is assumed to start at a time when the allocations for each service on all the nodes are 50% each. The reported usage in Call 1 is consistent with the previous allocations and the new allocations are computed to have the same

Call #		% usage (sink)		% allocation	
		Svc 1	Svc 2	Svc 1	Svc 2
1	Node 1	50 (1)	50 (1)	50	50
	Node 2	50 (1)	50 (1)	50	50
2	Node 1	40 (0)	60 (1)	40	60
	Node 2	50 (1)	50 (1)	60	40
3	Node 1	40 (1)	60 (1)	41	59
	Node 2	60 (1)	40 (1)	59	41
4	Node 1	41 (1)	59 (1)	42	58
	Node 2	59 (1)	41 (1)	58	42
5	Node 1	40 (0)	60 (1)	40	60
	Node 2	58 (1)	42 (1)	60	40
6	Node 1	40 (1)	60 (1)	41	59
	Node 2	60 (1)	40 (1)	59	41
27	Node 1	48.9 (1)	51.1 (1)	49	51
	Node 2	51.1 (1)	48.9 (1)	51	49

Table 4.1 : **Dynamics of the Resource Manager**

values.

Call 2 reports that node 1 is no longer a resource sink with respect to service 1 and that the usage for service 1 has fallen to 40% on node 1 while it has increased to 60% for service 2. To equalize cluster-wide usage for the two services, the resource manager computes allocations such that the share for service 1 is raised to 60% on Node 2 while the share for service 2 is lowered to 40%.

Calls 3 and 4 report usages that are consistent with the previous allocations. Therefore, the resource manager computes the new allocations by offsetting the usages by a small amount in hope of ultimately equalizing the allocations for each service across all nodes.

Call 5, however, is similar to call 2 and reports node 1 to be no longer a resource sink with respect to service 1. The new allocations are similar to those for call 2. Call 6 again is similar to call 3.

From call 6 onwards, all nodes are reported to be resource sinks with respect to each service. The results are shown for call 27 – the allocations for each service have nearly equalized across all the nodes.

4.2 Prototype Implementation

In this section, I describe my prototype implementation of cluster reserves. Section 4.2.1 presents the cluster configurations used for obtaining the experimental results presented in Section 4.3. Section 4.2.2 describes the support added to the operating system on individual cluster nodes for the purpose of effectively supporting differentiated QoS. In Section 4.2.3, the implementation of cluster reserves is presented, which extends the mechanisms for supporting differentiated QoS on individual cluster nodes. Finally, Section 4.2.4 discusses issues related to fault tolerance of the cluster.

4.2.1 Cluster Configuration

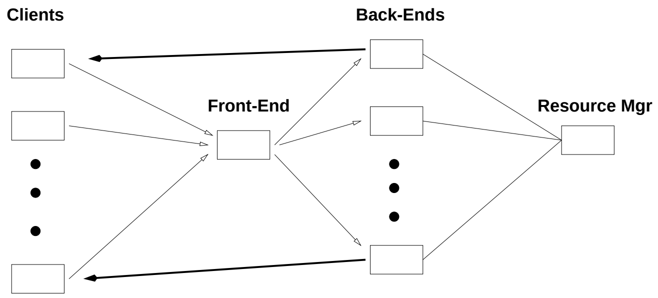

Figure 4.2 : **LAN configuration**

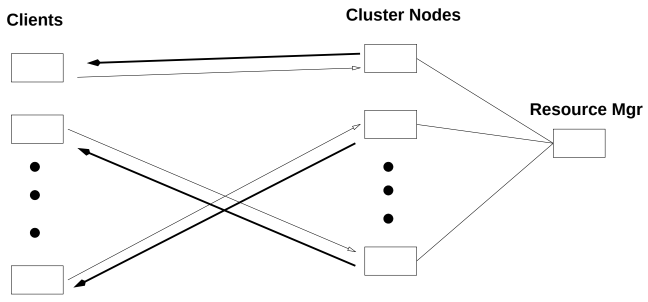

Figure 4.3 : **Geographically Distributed Nodes**

The prototype cluster nodes consist of 300MHz Pentium II machines configured with 128 MB of RAM and run the FreeBSD-2.2.6 operating system. For the experiments presented in Section 4.3, two different cluster configurations were employed.

Figure 4.2 shows the first configuration where the cluster consists of a front-end machine that receives client requests and distributes them to the back-end nodes in the cluster. The TCP handoff protocol [87, 11, 13] is employed by the front-end that enables response data from servers at the back-end nodes to be directly sent to the clients.

The second configuration shown in Figure 4.3 emulates a geographically distributed cluster where clients directly send requests to specific nodes. It is also possible to emulate WAN delays between various cluster nodes and the clients by employing a *delayrouter* [16] between the nodes. However, as WAN delays do not affect the results presented in Section 4.3, I did not emulate these delays.

The requests were generated by a client program based on the S-client architecture [16]. The program generates HTTP requests as fast as the Web server can handle them. Seven 166 MHz Pentium Pro machines were used as client machines. The client machines and all cluster nodes are connected via switched 100Mbps Ethernet. The services were hosted on the cluster nodes using the Apache-1.3.9 [8] Web server.

4.2.2 Support for differentiated QoS in individual cluster nodes

For supporting differentiated QoS on individual cluster nodes, I implemented resource containers [17] in the FreeBSD-2.2.6 operating system configured on each of the cluster nodes. For accurate accounting of network processing, the LRP [39] network subsystem was used in conjunction with resource containers [17].

The lottery scheduling [107] policy was employed for implementing proportional share CPU scheduling among resource containers. Lottery scheduling manages resources using tickets and currencies where each principal gets resources proportional to the number of tickets it possesses. Thus, to allocate a fixed percentage of a machine's CPU to a resource container, a proportional number of tickets are associated with it.

As mentioned in Section 4.2.1, the Apache Web server was used for hosting services on the cluster nodes. In order to correctly account the request processing for a service to the corresponding resource container, the server code needed to be modified appropriately. This could be achieved in two different ways:

1. The server code could be modified to dynamically change the association to the resource container corresponding to the service that is being processed. This has the advantage that a single web server program can be used for servicing all requests. However, the disadvantages are: (1) the code modification required to the Apache server is complex, (2) This approach only associates the request processing to the correct container after a request is received; kernel processing required before the server application receives the request is misaccounted.

2. A different copy of the server program can be used for processing requests for different services. The disadvantage of this approach is higher costs because of larger context switching overhead. However, the advantages are: (1) code modifications to Apache are simple – only at program startup the program has to be associated with the corresponding service's resource container, (2) all

kernel processing on behalf of the service is correctly accounted to its container through early demultiplexing.

Owing to the significant advantages afforded by the second approach, I chose that for my prototype implementation. Multiple instances of the Apache server were run, one each for each of the services hosted on the cluster node. Each instance of Apache *listened* at a separate service port for connection requests for that service. Alternatively, each cluster node could have been configured with several interface addresses and each Apache instance could have listened on a separate interface address.

4.2.3 Implementation of Cluster Reserves

Cluster reserves are implemented by a resource manager that runs as a user process on a separate cluster node. For the purpose of communicating with the resource manager, each node in the cluster runs a user process called *tracker* that is capable of (1) collecting usage statistics from the kernel and sending them to the resource manager, (2) setting resource allocations once they are computed by the resource manager, and (3) sending requests to the resource manager for recomputing resource allocations. The resource manager communicates with the tracker on each back-end node through a persistent TCP connection.

As mentioned in Section 4.1, I used the freely available LOQO tool for solving the constrained optimization problem in the resource manager. The resource manager formulated the constrained optimization problems in the AMPL [49] modeling language commonly used for mathematical programming. The freely available AMPL tool was used for processing inputs in this language and presenting them to LOQO, which then computed the solution.

Table 4.2 shows the computation time on some platforms for typical problems. As can be observed, the computation time needed for typical problems is well within 1 second. For all the experimental results reported in Section 4.3, a 300 MHz Pentium II machine was used to run the resource manager.

	Time (ms)	
	4 nodes 5 services	32 nodes 8 services
300 MHz PII	160	510
500 MHz PIII	85	290

Table 4.2 : **Computation time**

4.2.4 Fault Tolerance

The continuous availability of web servers is extremely important for business success. Fault tolerance in web servers is, therefore, highly desirable. An advantage of a cluster-based configuration is that failure of some cluster nodes does not affect server availability as long as other nodes in the clusters are functional.

However, the failure of some key cluster nodes that perform important functions can still affect server operation. For example, the failure of the front-end node in Figure 4.2 affects server availability. Such nodes are typically backed up using a hot swappable spare for such occasions.

Another centralized point of failure in our cluster configuration that supports cluster reserves is the node that runs the resource manager. However, in the event of a failure of the cluster resource manager, the cluster continues to serve requests, albeit with potentially suboptimal resource allocations (graceful degradation). As soon as a new resource manager is started, the resource allocations are adjusted. Therefore, the presence of a centralized resource manager does not affect the fault tolerance in the cluster.

4.3 Resource management in clusters

In this section, I use a set of benchmarking scenarios to (1) demonstrate the need for performance isolation in Web servers, (2) demonstrate cases where cluster-wide resource management (as opposed to per-node resource management) is needed to achieve effective performance isolation, and (3) show that cluster reserves are an effective solution for providing performance isolation in cluster-based Web servers that run multiple service classes on a common set of cluster nodes.

4.3.1 Performance isolation via node separation

As mentioned earlier, the inadequate resource management facilities available in general-purpose operating systems render performance isolation between different service classes ineffective, even in single node Web servers. For this reason, state-of-the-art servers that host multiple service classes on individual nodes are not capable of affording performance isolation between the service classes.

For example, Internet Service Providers (ISPs) tend to host small Webs from different organizations on a common hardware platform. In the absence of performance isolation between service classes (i.e., requests for content from different organizations in this example) higher request loads for any one organization's web pages can unfairly steal server resources paid for by other organizations. That is, high load for one organization's Web can cause high latency for other customer's Webs.

Many state-of-the-art cluster-based servers achieve performance isolation by reserving a disjoint subset of the cluster nodes for each service class. While this approach is capable of achieving performance isolation, it can result in poor resource utilization and lower performance. First, every service requires a set of distinct cluster nodes specifically reserved for the service and enough hardware must be provided to cover the peak load expected for the service class. This can increase the capital and maintenance costs of the cluster. Second, this approach does not permit resource sharing — requests for an overloaded service cannot utilize idle cluster resources reserved for other service classes. For example, the nodes dedicated for running a database server might be idle even though the nodes serving static content might be fully saturated at a given time.

My first experiment evaluates the performance advantages of using a shared cluster as opposed to using disjoint cluster nodes for different service classes. The setup consists of a front-end node that distributes incoming requests to four back-end nodes. Three front-end request distribution strategies are considered:

1. The front-end assigns requests for any resource class to a specific back-end node that is reserved for serving requests only for this class.

2. The WRR strategy (see Appendix 7.1.1) is used for request distribution and requests for a service can be given to any cluster node. The strategy is unaware of service classes and resource management.

3. The LARD [87, 11] strategy (see Appendix 7.1.2) is used for request distribution and requests for a service can be given to any cluster node. The strategy is unaware of service classes and resource management.

A trace obtained by merging logs of four different Rice University departmental Web servers was used to generate requests for the cluster. This trace spans a two-month period and its dataset consists of about 31,000 different documents covering 1.015 GB of space. The results show that this trace needs 526/619/745 MB of memory cache to cover 97/98/99% of all requests, respectively.

Four service classes were hosted on the cluster. Each service class provides resources for requests from one of the four original Web server logs. With shared use of the cluster (i.e., with WRR and LARD strategies), cluster reserves were employed and each service class was allotted a cluster-wide allocation of 25%.

	Disjoint	Shared	
		WRR	LARD
Xput (conn/s)	252 (1.0)	517 (2.0)	1214 (4.8)
CPU util. (%)	15	35	60

Table 4.3 : **Disjoint vs shared cluster use**

Table 4.3 shows the results for the three different front-end request distribution strategies employed. The results show that disjoint use of the cluster nodes results in relatively low performance because of load imbalance when requests for some services exceed those of others. Substantially higher CPU utilization and throughput is achieved when the service classes are allowed to share the cluster nodes (WRR and LARD). The use of LARD yields an additional improvement in utilization and throughput, because it also aggregates the total cluster memory available for caching the documents [87, 11].

In summary, clusters that permit different service classes to share cluster nodes running general-purpose operating systems are incapable of achieving performance isolation. On the other hand, statically dedicating distinct sets of server nodes for different services can result in low utilization of server resources and lower throughput.

4.3.2 Performance isolation via per-node resource allocations

I next consider a scenario where static per-node resource assignments are used. This can be accomplished with an operating system mechanism for fine-grained resource management, like resource containers.

Resource containers afford performance isolation between service classes hosted on a single cluster node. I will show that statically assigning equal resources to each cluster node for every service class is sufficient for achieving performance isolation between services on a cluster-wide basis if, and only if, requests for all service classes are load balanced across the cluster nodes. More generally, static resource assignments to services on each node is sufficient for cluster-wide performance isolation if the load for each service class is always perfectly distributed in proportion to the resources assigned to it on each cluster node.

My next experiment emulates conditions where the request load is balanced across all the cluster nodes for all service classes. For this purpose, I use a single front-end node in the cluster that employs the WRR request distribution strategy. The WRR strategy statistically balances the load for each service class on every cluster node. The cluster hosts five service classes configured on four back-end nodes. Each service is assigned a CPU allocation of 20% on each of the four back-end nodes. Seven client machines issue requests to the front-end node from a synthetic trace that contains a repeated set of five requests, one for each service class. The content size requested by each request is 6 KBytes. This value was chosen because the typical size of HTTP transfers has been observed to range between 5–13 KBytes [9, 78].

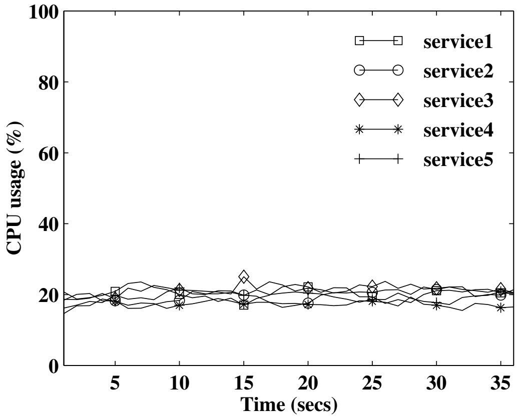

Figure 4.4 : **Typical node usage**

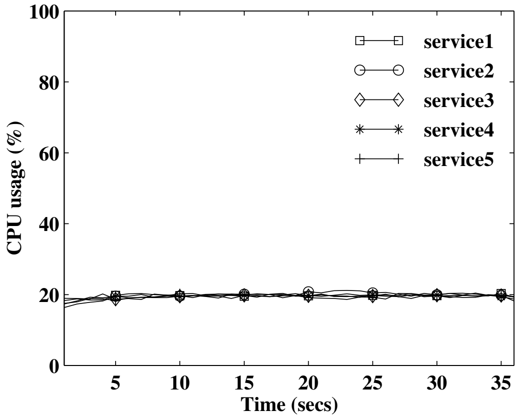

Figure 4.5 : **Cluster-wide usage**

Figure 4.4 shows the resource usage for each of the five services on a typical back-end node. Figure 4.5 shows the cluster-wide resource usage for each service class. The results show that with a balanced request load, static allocation of per-node resources is sufficient to ensure cluster-wide performance isolation.

Unfortunately, a static assignment of per-node resources is not sufficient in situations where perfect load balance among each server node cannot be achieved for each service class. In the following subsections, I cover different scenarios where perfect load balance is not attainable and, therefore, cluster-wide resource management is needed.

4.3.3 Geographically distributed clusters

One case where it is not generally possible to balance request loads for each service class arises when the cluster nodes are geographically distributed. To emulate this scenario, I used a configuration consisting of four cluster nodes hosting five service classes. Only the first two nodes receive requests for the first service class, while all nodes receive requests for all other service classes. The CPU allocation for each service class at any cluster node is set to 20%.

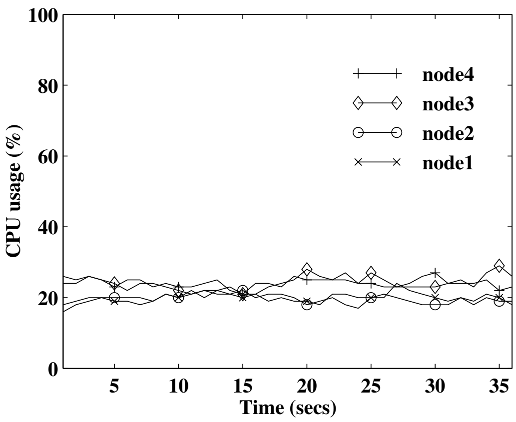

Figure 4.6 : **Typical service usage**

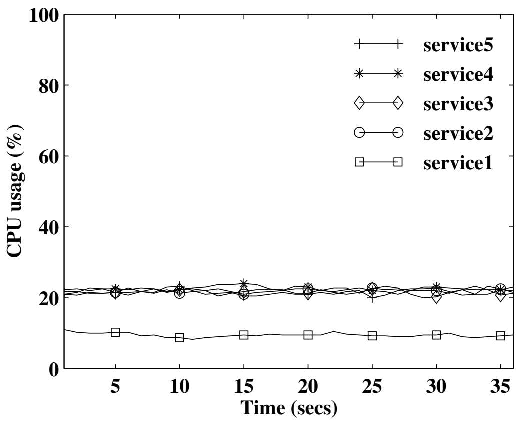

Figure 4.7 : **Cluster-wide usage**

Figure 4.6 shows the resource usage of service class 2 on all the four nodes. The resource utilization on nodes 3 and 4 is more than that on nodes 1 and 2 because this service class is able to *steal* the un-utilized CPU allocated to service class 1. Figure 4.7 shows the cluster-wide resource usage for all the services classes. While service class 1 only gets 10% of the cluster CPU, all other service classes are able to get nearly 23% by utilizing unused cycles from service class 1 on nodes 3 and 4.

I next repeated the above experiment with cluster reserves for attaining performance isolation. To observe the effect the resource manager has on resource utilization, it was started after eight seconds from the beginning of the experiment. Further, the resource manager was made to recompute the resource allocations on the cluster nodes every five seconds. An alternative strategy of recomputing resource allocations upon demand is considered in experiments in the following subsections.

Figure 4.8 again shows the resource usage of service class 2. As can be observed, the resource manager decreases the CPU allocation on nodes 1 and 2 and increases it on nodes 3 and 4 so as to allow service class 1 to effectively utilize its cluster-wide resources. Figure 4.9 shows that after the resource manager is started, service class 1 is able to utilize its cluster-wide share of CPU allocation (20%).

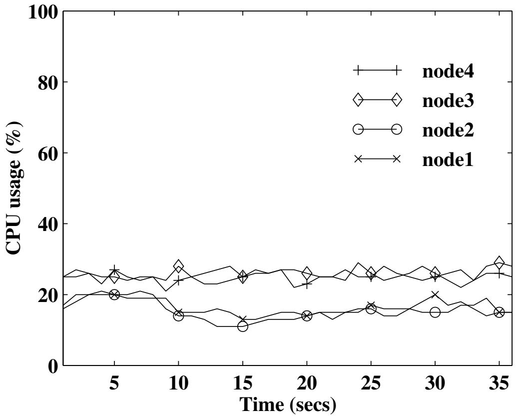

Figure 4.8 : **Typical service usage**

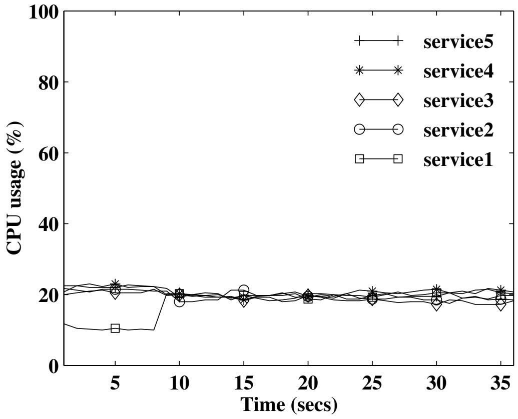

Figure 4.9 : **Cluster-wide usage**

This experiment demonstrates that geographically distributed clusters require dynamic, global resource allocation for performance isolation and that static per-node allocation is insufficient in achieving effective performance isolation. The results also indicate that cluster reserves are effective in providing performance isolation and can redistribute resources to meet the desired allocations within 1 second.

4.3.4 Sparse, resource intensive requests

My next experiment emulates a situation where a service class hosted on the cluster has a sparse incidence of compute intensive requests. Examples of services with this behavior include document translation services and rendering of maps in services that provide driving directions. These services are demanding of the resource manager, because the sparseness of requests prevents load balancing among nodes, and minimizing response time requires that reserved cluster resources are shifted quickly to a node that receives a request.

The experimental setup consists of five cluster nodes in a LAN environment. One of the nodes acts as a front-end, distributing requests to the four other back-end nodes. The request strategy employed at the front-end is WRR. Five services are hosted on the cluster and the desired resource allocation for each of them is 20%.

All requests for service class 1 access a CGI script that runs for 10 seconds before returning a 140 byte result to the client. Requests for all other service classes access a 6 KByte static document and are balanced across the cluster by the WRR policy employed at the front-end[2]. At any time, there is only one outstanding request for service class 1 and a new request is initiated as soon as the previous one finishes.

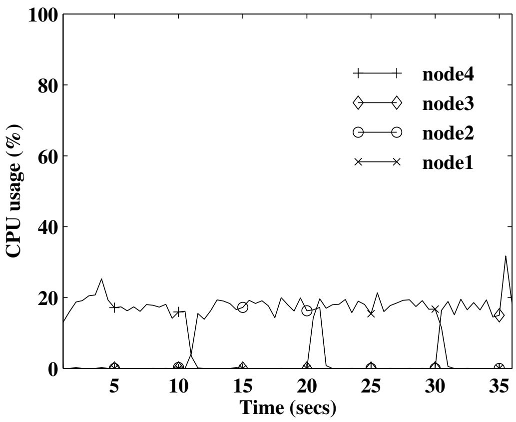

Figure 4.10 : **Usage for service 1**

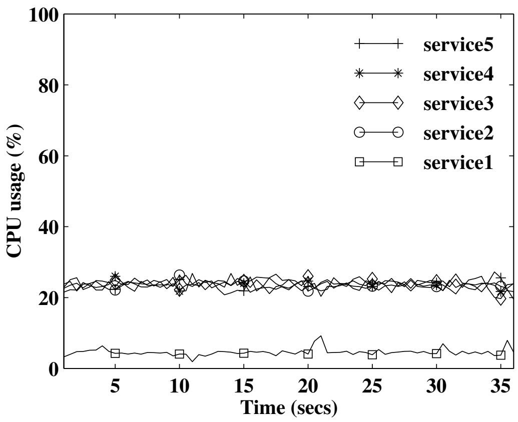

Figure 4.11 : **Cluster-wide usage**

Figure 4.11 shows the cluster-wide resource usage of the five service classes with a static per-node resource allocation of 20% to each service class. Service class 1

[2]A small modification was made to the WRR policy that ensured that every request for service class 1 is assigned to a different node than the last one.

gets only 5% of the cluster resources while all other service classes get nearly 24%. This is because the CGI script is only able to use the fractional capacity allocated to its service class on the node that serves the request; it cannot utilize the capacity allocated to its service class at other nodes, which is then used by other service classes. This is depicted in Figure 4.10.

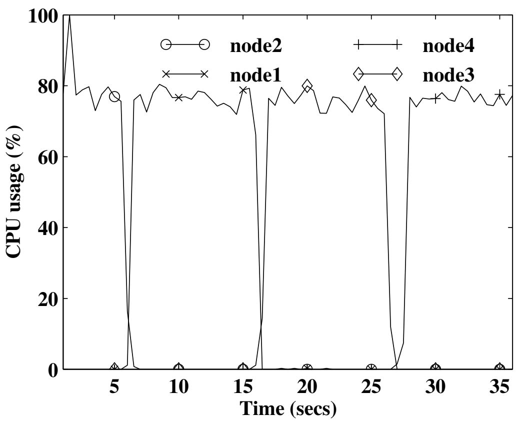

Figure 4.12 : **Usage for service 1**

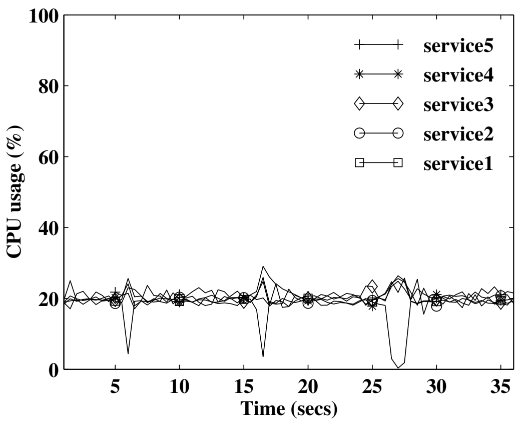

Figure 4.13 : **Cluster-wide usage**

I next repeated the experiment with cluster reserves. Resource allocation decisions by the resource manager are made on an on-demand basis. A back-end requests a reallocation of cluster resources when (1) the node ceases to be a sink for any service, or conversely, (2) the CPU usage of a service falls significantly (more than 20% of its last measured usage) on that node.

Figure 4.12 shows that the resource manager is capable of dynamically assigning resources to service class 1 such that its cluster-wide usage meets its allocation. The dips in the graph correspond to the intervals between the instant when a CGI request finishes on one node, and the time when the resource manager reallocates resources to the node that just received the next CGI request.

The results show that the time for re-allocation of resources ranges from 300ms to 1 second. This variation is the result of three factors: (1) the usage statistics at each node are computed every 250ms as a weighted mean of both the past and present usage; (2) the computation time taken by the resource manager ranges from 100–200ms; and, (3) new requests for resource allocation are sent to the resource manager only after the usage transients resulting from the last reallocation have stabilized (i.e., after 250ms).

Figure 4.13 shows the cluster-wide usage of all service classes with cluster reserves. The results show that resource usage corresponds to the allocation, except for the short intervals when a new request arrives at a different node and resources need to be reallocated.

4.3.5 Content-based request distribution

My final experiment demonstrates cluster resource management with content-based request distribution schemes. The experimental setup consists of a front-end node distributing requests to four back-end nodes. The LARD [11, 87] request distribution strategy was employed at the front-end to distribute requests. The cluster hosts three service classes, each of which are allocated 33% of the cluster's CPU resources. In order to realistically reflect the resource usage across the cluster nodes with content-based request distribution, the requests for each service classes are played from actual web logs from three different departmental web servers. The logs corresponding to service classes 1, 2 and 3 consist of datasets of 358 MB, 24 MB and 193 MB respectively and need 248 MB, 16 MB and 67 MB respectively of memory cache to satisfy 98% of their requests from main memory.

Figure 4.14 shows the cluster wide CPU usage of the three service classes when cluster reserves are not used. The results show that service class 1 is able to obtain more than its fair share (33%) of cluster CPU time while service class 2 gets less than its share. The high variation in the usage is mainly due to disk activity that is needed when requested documents are not found in the main memory cache. Figure 4.15 shows the results when cluster reserves are being used. With cluster reserves, all service classes achieve their allocated resource usage.

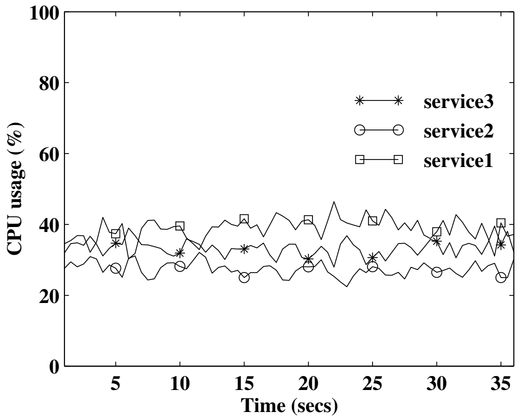

Figure 4.14 : **No Cluster Reserves**

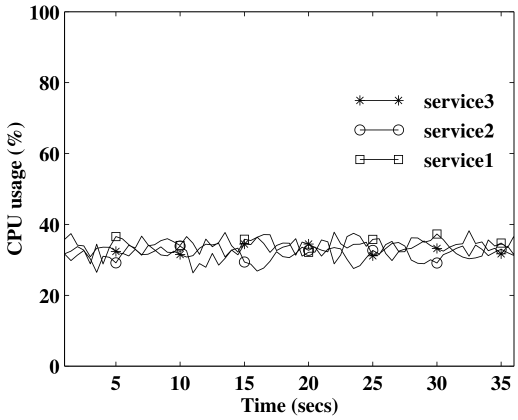

Figure 4.15 : **With Cluster Reserves**

4.4 Summary

This chapter presented and evaluated *cluster reserves*, a resource management facility for cluster based Web servers that affords effective performance isolation in the presence of multiple Web services that share a server cluster. Cluster reserves extend existing mechanisms for performance isolation in single-node servers to a cluster environment.

Using a set of benchmark scenarios and both synthetic and trace based workloads, I evaluated a prototype implementation of cluster reserves. The results show that hosting multiple Web services on a joint set of cluster nodes can result in higher resource utilization and improved performance than the state-of-the-art approach of hosting different services on dedicated server nodes. The results also demonstrate that cluster-wide (as opposed to per-node) resource management is needed to achieve effective performance isolation among services that share a set of clusters.

Finally, the results show that cluster reserves are an effective solution for providing performance isolation in cluster-based Web servers. Cluster reserves afford performance isolation among multiple service classes, which can be defined based on the requested content, the client who issues the request, or both.

The prototype implementation of cluster reserves presented in this dissertation uses resource containers to achieve performance isolation on individual cluster nodes. However, in principal any other abstraction capable of affording performance isolation on a single node can be used. While the prototype implementation was evaluated for the CPU time resource, cluster reserves can be extended to also provide performance isolation for other operating system resources like memory, disk and network bandwidth etc.

Chapter 5

Related Work

Much effort has been devoted in recent years to improving the performance and functionality provided by server systems. This chapter presents an overview of some of this work that is related to the work in this dissertation.

Section 5.1 discusses past work on developing fine-grained resource management mechanisms in the context of experimental real-time and multimedia operating systems. Section 5.2 describes scheduling policies that support proportional allocation of resources and can be used for providing differentiated QoS among resource principals. Section 5.3 presents related work for supporting quality of service in server systems while Section 5.4 covers related work for supporting quality of service in networks.

5.1 Operating system mechanisms for resource management

Fine-grained resource management mechanisms are now supported by many experimental systems. This section discusses some of these mechanisms and relates them to the work in this dissertation.

5.1.1 Resource Containers

Banga et al. [17, 15] proposed the resource container abstraction that separates the notion of a resource principal from threads or processes and provides support for fine-grained resource management in the operating system. Coupled with Lazy Receiver Processing (LRP) [39], resource containers are capable of providing effective differentiated QoS on single node server systems. Resource containers and LRP are described in more detail in Chapter 2. Resource containers enable effective differentiated quality of service when combined with an appropriate resource scheduler.

This dissertation proposes the cluster reserves mechanism that provides effective differentiated quality of service in cluster-based server systems. The prototype implementation of cluster reserves described in Chapter 4 uses resource containers to provide differentiated quality of service on individual nodes in a cluster.

While resource containers are able to ensure that a certain fraction of Web resources is available to a given virtual site, they are unable to guarantee that a specified request rate or average response time can be maintained. This dissertation proposes a measurement-based resource management framework that provides predictable QoS by dynamically translating application progress into system-level resource needs.

5.1.2 Activities of Rialto

The *activity* abstraction [66] in Microsoft's experimental Rialto real-time operating system [65] is similar to a resource container in that it serves as a first-class operating system abstraction for a resource principal. However, it differs from resource containers in that while an activity in Rialto can be associated with multiple threads, a single thread cannot accurately account for work done on behalf several activities.

As for resource containers, activities by themselves can neither provide global resource management in a cluster environment, nor can they support predictable quality of service in application-level metrics in server systems.

5.1.3 Software Performance Units (SPU)

Software Performance Units (SPU) [104] were proposed as resource principals in the context of shared memory multiprocessors and are similar to resource containers in many aspects. The implementation of SPUs described in [104] considers controlled allocation of CPU, memory and disk bandwidth in a large symmetric multiprocessor based compute server environment. However, neither was the work evaluated in the context of Web server systems, nor was it targeted towards providing predictable performance in application-level metrics.

Resource management in a symmetric multiprocessor is significantly different than that in a cluster-based system. In the former, it is cheap to move workloads between processors; all it takes is to assign a thread executing on one processor to another processor. However, in a cluster environment, shifting workloads between cluster nodes is significantly more complex as it involves affecting the workload distribution policy and can adversely affect memory cache behavior and load balance. The cluster reserves abstraction proposed in this dissertation assigns cluster wide resources by affecting resource allocations on individual cluster nodes. The workload distribution remains independent of the cluster reserve mechanism.

5.1.4 Reservation Domains

The *reservation domains* [27, 26] of the Eclipse operating system allow the system to control the total resource consumption of a set of processes together. A reservation domain provides resource guarantees to a group of processes allowing a variety of QoS requirements to be met. The current implementation of reservation domains focuses on real-time issues.

Like resource containers, reservation domains can also be used as the underlying single-node resource principals that are extended to provide cluster-wide resource management using the cluster-reserves mechanism. Similarly, the resource management framework proposed in Chapter 3 can use reservation domains as the system's resource principals and provide predictable quality of service in server systems.

5.1.5 Paths in Scout

The Scout operating system [81, 79, 94] has explicit support for a *path* abstraction, representing an I/O channel (such as a TCP connection) through a multi-layered system. A path encapsulates the specific attributes of an I/O channel, and allows access to these attributes across layers. Paths have been used to improve execution speed in network subsystems by enabling cross-layer optimizations [80]. Paths have also been used to implement fine-grained resource management in network appliances, such as network-attached TVs [79] and Web server appliances [94].

Paths can be used as resource principals in appliance operating systems, and have been demonstrated to provide predictable performance through reservation of system's resources. However, they do not provide (1) a mechanism that maps application-level metrics to resource requirements in system-level metrics, (2) effective differentiated QoS for cluster-based server systems.

5.1.6 Processor capacity reserves

Mercer et al. [74, 73] designed an operating system abstraction for resource principals called a *processor capacity reserve* in the context of the Real-Time Mach operating system [101]. Reserves insulate programs from the timing and execution characteristics of other programs. An application can reserve system resources, and the operating system ensures that these resources will be available to threads associated with the reserve when needed.

Processor capacity reserves do not address the problem of controlling resources consumed by the kernel in performing processing on behalf of applications. This dissertation employs resource containers coupled with LRP for accurate accounting of all processing in the operating system, thus providing effective differentiated QoS. Additionally, processor capacity reserves do not aim at providing differentiated QoS in clusters, nor do they provide predictable QoS in application-level metrics in server systems.

5.1.7 Migrating threads in Mach and Shuttles in Spring

The *migrating threads* [46] of Mach [1, 23] and AlphaOS [32], and the *shuttles* of Spring [56] allow the resource consumption of a thread (or a shuttle) to be accounted towards the correct resource principal, when the thread (or shuttle) moves across protection domains. However, a single thread cannot perform work on behalf of several resource principals. As with processor capacity reserves, migrating threads and shuttles also do not correctly account for kernel processing.

Migrating threads and shuttles again serve as resource principals for operating systems. Like the other abstractions discussed above, they do not aim at providing differentiated QoS in clusters, nor do they provide predictable QoS in application-level metrics in server systems.

5.1.8 Nemesis

The Nemesis operating system [71, 57] follows a radically different approach for providing fine-grained resource management. The application and most of the associated kernel processing is performed in the address space of the application. While this approach accurately accounts for all kernel processing to the corresponding application, it does not decouple the notion of a protection domain from a resource principal. Consequently, it does not provide effective support for differentiated QoS in server environments where significant amounts of processing associated with a service class might be done in a separate address space (e.g., for CGI programs).

5.1.9 Exokernels

The exokernel approach [67, 41, 68] advocates moving bulk of the operating system software into libraries that can be customized specifically according to application needs. The actual kernel is kept as small as possible and principally multiplexes access to the hardware safely between the hosted applications. Applications can exert tight control over resource management as most of the operating system code runs in the application's address space and is modifiable by the application writer.

The work performed in the context of the exokernel approach has not aimed at providing predictable QoS to applications, nor has support for differentiated QoS in cluster-based designs been considered.

5.2 Scheduling policies for differentiated QoS

This section describes resource scheduling policies reported in past literature that support proportional allocation of resources and can be used for providing differentiated QoS among resource principals. The main limitation of all system schedulers is that they do not relate application-level performance targets (in terms of metrics like request rate or average response time) to the allocations of the system resources needed to achieve the target. Predictable performance in application-level metrics, therefore, is not supported.

5.2.1 Lottery scheduling

Lottery scheduling [107] manages system's resources using tickets and currencies. Each principal is assigned a number of tickets proportional to the desired resources to be allocated to it. The allocation of each resource is determined by holding a lottery; the resource is granted to the principal with the winning ticket. Lottery scheduling, thus, probabilistically allocates resources to competing principals in proportion to the number of tickets that they hold.

5.2.2 Start-time fair queuing (SFQ)

Start-time fair queuing (SFQ) [53, 54] belongs to a class of scheduling algorithms derived from the weighted fair queuing [36] algorithm developed in the context of packet scheduling in routers. Weighted fair queuing maintains the notion of a virtual time line for every competing principal. The virtual time for each principal is advanced by the amount of resource consumed, weighted in inverse proportion to its reservation. Resources are assigned to the principal with the minimum virtual time.

Weighted fair queuing updates virtual times by the amount of resource needed before actually scheduling the corresponding principal. In packet scheduling, the amount of network resources required are known beforehand as they can be determined from packet sizes. However, resource scheduling in operating systems does not have such prior information about the applications' resource requirements. Start-time fair queuing modifies the weighted fair queuing by maintaining virtual times as the start times of the competing principles, and allocating resources to the principal with the earliest start time. With this modification, *a priori* knowledge about a principal's resource requirements is no longer needed.

5.2.3 YFQ disk scheduler

YFQ [25] is a proportional-share scheduler designed for efficiently multiplexing disk resources between competing principals. Classical disk scheduling algorithms such as SSTF (shortest seek time first) and C-SCAN are designed to minimize disk head movement so as to obtain high disk bandwidth [51]. However, this can result in an unfair distribution of disk bandwidth among the competing principals.

YFQ extends the weighted fair queuing algorithm so as to attain fairness as well as high disk bandwidth. The disk requests are chosen for scheduling according to the weighted fair queuing algorithm; however, instead of choosing only one request, YFQ chooses a set of N requests. These chosen requests are then further sorted using classical disk scheduling algorithms so as to achieve high disk bandwidth.

5.2.4 Cello disk scheduling framework

The Cello [93] framework employs a two-level disk scheduling architecture. The first level provides proportional allocation of resources to competing principals and determines the number of disk requests from each principal that are put into a centralized disk queue for scheduling. The second level is application specific and determines the placement of requests within the centralized queue. Exercising control over placement enables the second level to both minimize seek time and rotational latency overheads as well as to support application requirements such as deadlines for real-time tasks, short response times for interactive tasks, etc.

The Cello disk scheduling approach can be used by the resource management framework proposed in Chapter 3 to provide predictability on a per-request basis as well as to inrease disk throughput by minimizing seek times and rotational latency overheads.

5.2.5 SMART multimedia scheduler

The SMART [84] multimedia scheduler integrates priorities and weighted fair queuing to meet real-time constraints while simultaneously supporting non real-time applications. SMART maintains a value tuple consisting of two components – priority and virtual time – with each principal. A value tuple is considered higher than another such tuple if either it has higher priority or if priority is the same and the virtual time is earlier than for the other. Principals are scheduled based on their value tuples in the following manner:

- If the highest value tuple belongs to a task that does not have real-time deadlines, then it is scheduled first.
- If the highest value tuple belongs to a real-time task, then a list of all real-time tasks is made such that scheduling any task in the list does not cause deadlines of higher value tuple tasks to be missed. Additionally, any real-time tasks in the list are required to have a higher value tuple than the highest value tuple non-real-time task. The task with the earliest deadline is scheduled from this list.

In this manner, the SMART scheduler supports both real-time tasks as well as provides proportional share scheduling for conventional tasks that don't have deadlines.

5.2.6 Borrowed Virtual Time (BVT) scheduler

The BVT [40] scheduling is similar to SFQ scheduling in that proportional resource shares are maintained using the concept of a virtual time kept on behalf of each principal. However, BVT is aimed at providing low-latency to real time tasks and interactive applications through the concept of principals borrowing virtual time from their own future resource allocation. In this fashion, principals are made to execute sooner in exchange for reduced resources in the future.

The measurement-based resource management framework proposed in this dissertation provides predictability in average metrics, e.g., average request rate, average response time etc. By combining with scheduling policies such as SMART and BVT, this framework might be used for providing predictability for individual Web requests. However, this also requires a coordination between all resource classes in the system (e.g., CPU, disk, memory etc). Application of BVT to the proposed resource management framework forms an interesting area for future research.

5.3 Support for QoS in Server Systems

This section describes some past work for supporting quality of service in web servers. Most of the prior effort in this direction has attempted to provide differentiated quality of service in servers by providing support in the webserver application. However, effective provision of differentiated QoS requires the involvement of the operating system as kernel processing forms a significant fraction of the processing required for web requests.

In the context of the WebQoS [59] project at HP Labs, Bhatti and Friedrich [22] describe modifications to the Apache webserver in order to support differentiated QoS between service classes. The HTTP requests are intercepted and classified by a connection manager. Priorities between service classes are then observed by servicing requests from higher priority classes earlier than those from lower priority classes. This server architecture does not provide for proportional resource allocation between service classes; a continuous stream of high priority requests can indefinitely delay processing of low priority requests.

Almeida et al. [4] provide QoS to various classes of requests in the Apache webserver by limiting the number of runnable processes that concurrently handle requests for individual classes. This scheme ensures that a service class getting a large number of requests does not affect the performance of other classes.

Li and Jamin [72] use a measurement-based approach to provide proportional bandwidth allocation to web clients by scheduling requests within a webserver. The model uses a measurement-based approach to maintain bandwidth estimates for the various service classes in the system. When the bandwidth estimation for a service class falls below its target, the bandwidth attributed to over aggressive service classes is throttled by appropriately delaying the processing of their requests. While this scheme can proportionally distribute network bandwidth between service classes, it is limited in its effectiveness when the server resources are the bottleneck as opposed to network resources. For example, consider a service class that requires a considerably larger amount of CPU resources to sustain the same network bandwidth as another service class. Servicing requests so as to equally distribute network bandwidth between the two service classes would result in the the former monopolizing the CPU resources along with an inefficient utilization of network resources.

OS mechanisms such as the resource containers [17] address the problem of service differentiation more generally by fully accounting for all kernel processing. The work in this dissertation extends and complements this prior work. It seeks to provide both differentiated as well as predictable quality of service in terms of application-level metrics such as average request rate and response time.

5.4 Support for QoS in Networks

A complete solution for providing QoS requires a combination of both server QoS as well as QoS support in the network infrastructure. This dissertation is concerned with effective support for server QoS; it assumes that either the network is not the bottleneck or else QoS support already exists in the network. Providing QoS in networks is a field of active research. This section discusses some related work from this field.

ATM (Asynchronous Transfer Mode) [37, 14] network technology was designed to efficiently support both high-speed digital voice as well as data communications. ATM provides rate guarantees to network flows by supporting bandwidth reservation at the network switches at the time of connection initiation. The use of small constant sized packets or cells enables ATM to minimize jitter in guaranteed flows. These features make ATM well suited for supporting traffic like voice and video that require quality of service.

Rate guarantees can also be provided in integrated services networks with schemes that support traffic shaping. Traffic shaping is a general term given to a broad range of techniques designed to make network traffic conform to some specified behavior. The leaky bucket scheme [20] regulates the burstiness of transmitted traffic using a bucket of finite capacity that generates permits at a fixed rate. Network packets are transmitted only after a permit is available for the packet. Therefore, packet bursts are limited by the capacity of the bucket, while the average transmission rate is governed by the rate at which permits are generated. Efficient mechanisms [10] have been designed in operating systems to support traffic shaping at Gigabit packet transmission rates.

The Virtual Clock [114] is a queuing discipline for network switches that is capable of affording rate guarantees to flows. It involves an explicit flow setup phase where each source indicates its needs to the network. Once a flow is admitted, the network switches monitor the average rate of the flow and drop packets upon queue overflows from flows that exceed the rate agreed upon flow setup. The Virtual Clock mechanism differs from ATM in that it can be deployed in conventional packet switched networks with large variable sized packets. However, for the same reason, virtual clock cannot guarantee low jitter to the flows managed by it.

The Fair Queuing (FQ) [36] queuing discipline was designed to fairly distribute the network bandwidth at a switch between competing flows. It involves maintaining separate queues for each flow and servicing them in a round-robin manner. If any queue overflows, additional packets for the corresponding flow are discarded and other flows remain unaffected. An advantage of the fair queuing algorithm is its simplicity; it does not monitor the burstiness of the flows, nor does it involve sending any feedback to the traffic sources. A disadvantage is that it does not provide any specific rate guarantees; although bandwidth is distributed fairly, the bandwidth received by any specific flow depends upon the number of flows sharing the network link. Weighted Fair Queuing (WFQ) is a variation of fair queuing where weights are assigned to each flow. This permits a bandwidth distribution that is proportional to the weights assigned to the flows.

Keshav [70] argues that while quality of service guarantees can be provided by making bandwidth reservations at the network switches, such reservations can reduce the statistical multiplexing in the network. He describes a provably stable flow-control system where senders adapt to changing network conditions in order to satisfy a flow's rate and delay guarantees. In this system, the network switches are assumed to be Rate Allocating Servers, i.e., they are assumed to deploy a queuing discipline such as fair queuing. In addition, the traffic sources are assumed to use the packet-pair algorithm to probe network state. Using packet-pair, a source sends out two back-to-back packets; the spacing between the corresponding acknowledgments is used to estimate the bottleneck bandwidth available to the source. The round-trip time is estimated by the difference in time between a packet transmission and the arrival of the corresponding acknowledgment. The control mechanism uses this information to maintain the number of packets in the queue at the bottleneck switch at a desired value.

The Random Early Detection (RED) [45] queuing discipline maintains a single shared queue for all flows. However, RED drops network packets from flows before the queue overflows so as to give an implicit notification to the source about impending congestion. A packet is dropped with a drop probability once the queue length grows beyond a certain threshold. This drop probability is proportional to the bandwidth share received by the corresponding flow at that time. Therefore, packets from aggressive flows are more likely to be dropped than from other flows.

RSVP (Resource Reservation Protocol) [24, 89] proposes to extend the Internet architecture and protocols for the purpose of supporting integrated services. RSVP follows a receiver-oriented approach where a flow's receiving host is made responsible for communicating resource reservation information to the network switches. RSVP aims to provide robustness by maintaining flow-specific information in switches as soft state – this state does not need to be explicitly deleted but is timed out if not periodically refreshed. This permits end-to-end connectivity to be maintained despite network links and switches going up and down. RSVP was designed to support multicast as that benefits many real-time services.

Jamin et al. [63] describe a measurement-based admission control algorithm for predictive[1] services in packet networks. Rather than characterizing the flows within a service class with their worst case behavior, they use a measurement-based characterization of these flows and use that to make admission control decisions. They show that this approach provides high network utilization through statistical multiplexing as opposed to those achievable with guaranteed services. The work in this dissertation applies similar concepts for providing predictable QoS to services in server systems. A measurement-based translation from application-level metrics to resource allocations in the system is used for the purpose of characterizing the resource needs of service classes. This is used in performing admission control, in monitoring the system's capability to meet resource needs and in multiplexing resources between service classes for the purpose of meeting target performance in application-level metrics.

[1] Predictive services consist of flows that can tolerate occasional delay bound violations.

Chapter 6

Conclusions and Future Work

The objective of this dissertation is to address the inadequate support in server systems for providing predictable as well as differentiated quality of service to clients. This dissertation introduces a set of new resource management facilities that enable effective support for quality of service in server systems.

6.1 Limitations of resource management in traditional server systems

This dissertation is motivated by the following limitations of resource management mechanisms found in server systems:

- Web site and proxy operators are often interested in differentiating the system resources multiplexed between various virtual sites hosted on a server system so as to provide predictable performance to sites in application-level metrics. Predictable performance in metrics such as average throughput, response time, etc, is not supported in traditional server systems.

- High performance server systems based upon cluster-based architectures lack adequate support for service differentiation. This is because providing differentiated quality of service in a cluster-based server system requires global coordination between resource management of individual cluster nodes. This global resource resource management is not found in current clusters.

The resource management facilities introduced in this dissertation address these limitations.

6.2 Contributions

This section summarizes the contributions made in this dissertation. These are:

- **A measurement-based resource management framework for supporting predictable QoS in server systems.** The framework allows Web servers and proxy operators to co-locate virtual sites and Web services, while simultaneously providing predictable quality of service, in terms of average request rate or average response time.

  The framework consists of a measurement-based admission control process that allows operators to determine whether a set of sites can be co-located on the same server system. This is based on the measured statistics of the sites' resource consumption under its live workload, and its desired quality of service and service class (guaranteed, predictive, or best effort). Once a set of services has been admitted, feedback-based resource schedulers ensure that all sites achieve their QoS targets with high probability, while being allowed to use excess resources not currently claimed by other sites.

  An empirical evaluation of a prototype implementation shows that the system is able to predict with high confidence if sites can be co-located on a system, that it is able to maintain the target QoS levels of admitted sites with high probability, and that it is able to achieve high average hardware utilization on the server system.

- **Cluster Reserves: A new mechanism for global resource management in cluster-based server systems.** Cluster reserves are a resource management facility for cluster-based server systems for supporting effective differentiated quality of service in the presence of multiple Web services that share a server cluster. Cluster reserves extend existing mechanisms for service differentiation in single-node servers to a cluster environment.

  Experimental results with a prototype implementation of cluster reserves were presented. The results show that cluster reserves are capable of affording effective performance isolation among multiple service classes. They also demonstrate that cluster-wide (as opposed to per-node) resource management is needed to achieve effective performance isolation among services that share a set of clusters. Finally, as opposed to the state-of-the-art approach of hosting different services on dedicated server nodes, hosting multiple Web services on a joint set of cluster nodes results in higher resource utilization and improved performance.

6.3 Future Work

This section discusses future research directions as continuations of the work described in this thesis.

The measurement-based resource management framework for providing predictable quality of service was evaluated in this dissertation on a server system supported on a single workstation. A logical extension would be to evaluate it for a server system comprising a cluster-based architecture. The measurement-based framework relies on the presence of resource principals on a system that provide effective support for differentiated quality of service. The cluster reserve mechanism proposed in this dissertation provides such a resource principal for a cluster. The measurement-based resource management framework can thus be directly layered on top of the cluster reserve mechanism to provide predictable quality of service in cluster-based server systems. One issue that would require further research in such an environment would be the granularity at which resource allocations are made to cluster reserves so as to meet application-level contracts. On a single workstation, resource allocations to principals can be made at fine granularities (e.g., a scheduling quantum). However, due to the loosely coupled nature of a cluster, such allocations have to be made at a coarser granularity to reduce communication overheads.

The proposed resource management framework needs to be evaluated on a larger number of workloads. Also, the use of more sophisticated admission control and resource monitoring algorithms need to be explored. The management of physical memory pages, so as to provide predictable quality of service to memory intensive services, also needs further attention.

The application of the proposed measurement-based resource management framework needs to be explored in the realm of real-time constraints. Specifically, the work in this dissertation provides predictability in average metrics, e.g., average request rate, average response time etc. Ensuring a maximal response time or a minimal data rate for *individual* Web requests is an interesting issue that requires further work.

A significant amount of engineering effort can be employed for tuning the resource allocation to the cluster reserve mechanism. Specifically, faster algorithms to compute the constrained optimization problem required for allocating resources to cluster reserves are desirable.

Another area that needs further research is support for quality of service in networking. Specifically, research is needed for integrating support for quality of service in TCP, the Internet transport protocol. Currently, TCP increases or decreases a connection's sending rate based upon its internal congestion control algorithms and is not very receptive to quality of service support provided in Internet routers.

Chapter 7

Appendix

7.1 Request Distribution Strategies

This section briefly describes the weighted round-robin (WRR) and the locality-aware request distribution (LARD) strategies employed by a front-end node to distribute requests to back-end nodes in a cluster.

7.1.1 The WRR strategy

The WRR request distribution strategy aims to efficiently utilize cluster resources by balancing the load across all the back-end cluster nodes. The pseudo-code for this strategy is given below:

```
while (true) {
  fetch next request r;
  chosen_backend ← (chosen_backend+1) mod num_backends;
  min_load ← Load(chosen_backend);
  cmp_backend ← (chosen_backend+1) mod num_backends;
  for (i←1; i < num_backends ;i++) {
    if (min_load > Load(cmp_backend)) {
      chosen_backend ← cmp_backend;
      min_load ← Load(chosen_backend);
    }
    cmp_backend ← (cmp_backend + 1) mod num_backends;
  }
```

$$cost\_balancing(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ Infinity & Load(server) > L_{overload} \\ Load(server) - L_{idle} & otherwise \end{cases}$$

$$cost\_locality(target, server) = \begin{cases} 1 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

$$cost\_replacement(target, server) = \begin{cases} 0 & Load(server) < L_{idle} \\ 0 & target\ is\ mapped\ to\ server \\ Miss\ Cost & otherwise \end{cases}$$

Figure 7.1 : LARD Cost Metrics

```
  send r to chosen_backend;
}
```

7.1.2 The LARD strategy

The LARD [87, 11] strategy yields scalable performance by achieving both load balancing and cache locality at the back-end servers. For the purpose of achieving cache locality, LARD maintains mappings between targets and back-end nodes, such that a target is considered to be cached on its associated back-end nodes. To achieve a balance between load distribution and locality, LARD uses three cost metrics: *cost_balancing*, *cost_locality* and *cost_replacement*.

The unit of cost (and also of load) is defined to be the delay experienced by a request for a cached target at an otherwise unloaded server. The load point L_{idle} defines a value below which a back-end node is potentially underutilized. $L_{overload}$ is defined such that the difference in delay between a back-end node operating at or above this load, compared to a back-end node operating at the point L_{idle}, becomes unacceptable.

The aggregate cost for sending the request to a particular server is defined as the sum of the values returned by the above three cost metrics. When a request arrives at the front-end, the LARD policy assigns the request to the back-end node that yields the minimum aggregate cost among all nodes, and updates the mappings to reflect that the requested target will be cached at that back-end node.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Conference*, July 1986.

[2] Akamai. http://www.akamai.com/.

[3] P. Albitz and C. Liu. *DNS and BIND in a nutshell.* O'Reilly & Associates, Inc., 1994.

[4] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[5] Compaq Equipment Corporation, *AltaVista.* http://www.altavista.com/.

[6] Alteon WebSystems. ACEdirector. http://www.alteonwebsystems.com.

[7] D. Andresen et al. SWEB: Towards a Scalable WWW Server on MultiComputers. In *Proccedings of the 10th International Parallel Processing Symposium*, Apr. 1996.

[8] Apache. http://www.apache.org/.

[9] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[10] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

[11] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient Support for P-HTTP in Cluster-based Web Servers. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[12] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[13] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable Content-aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, CA, June 2000.

[14] ATM Forum. http://www.atmforum.com/.

[15] G. Banga. *Operating system support for server applications*. PhD thesis, Rice University, May 1999.

[16] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. *World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation)*, 1999.

[17] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[18] T. Berners-Lee and D. Connolly. RFC 1866: Hypertext Markup Language — 2.0, Nov. 1995.

[19] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext transfer protocol – HTTP/1.0, May 1996.

[20] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[21] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *Proceedings of the 6th International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[22] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):64–71, Sept. 1999.

[23] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and Mach. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Seattle, WA, Apr. 1992. USENIX.

[24] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated Services in the Internet Architecture: an Overview, June 1994.

[25] J. Bruno, J. Brustoloni, E. Gabber, M. McShea, B. Özden, and A. Silberschatz. Disk Scheduling with Quality of Service Guarantees. In *In Proceedings of the IEEE ICMCS Conference*, Florence, Italy, June 1999.

[26] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[27] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proceedings of the USENIX 1998 Annual Technical Conference*, Berkeley, CA, June 1998.

[28] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, Dec. 1997.

[29] The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/.

[30] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[31] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[32] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 127–146, Seattle, WA, Apr. 1992. USENIX.

[33] A. Cohen, S. Rangarajan, and H. Slye. On the Performance of TCP Splicing for URL-Aware Redirection. In *Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[34] O. P. Damani, P.-Y. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[35] P. B. Danzig, K. Obraczka, and A. Kumar. An Analysis of Wide-Area Name Server Traffic: A study of the Domain Name System. In *Proceedings of the SIGCOMM '92 Conference*, pages 281–292, Aug. 1992.

[36] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the SIGCOMM '89 Symposium*, Austin, Sept. 1989.

[37] M. DePrycker. *Asynchronous Transfer Mode.* Ellis Horwood, London, 1993.

[38] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable Highly Available Web Server. In *Proceedings of the IEEE International Computer Conference*, San Jose, CA, Feb. 1996.

[39] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[40] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island, SC, Dec. 1999.

[41] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.

[42] K. Fall and J. Pasquale. Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability. In *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, Jan. 1993.

[43] Open Market. FastCGI Specification. http://www.fastcgi.com/.

[44] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999.

[45] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[46] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX 1994 Winter Conference*, San Francisco, CA, Jan. 1994.

[47] FORE Systems, Inc. ESX-2400/4800. http://www.fore.com.

[48] Foundry Networks. ServerIron. http://www.foundrynet.com.

[49] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[50] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[51] P. Galvin and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, Reading, MA, USA, 5th edition, 1998.

[52] Google. http://www.google.com/.

[53] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[54] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of the SIGCOMM '96 Conference*, New York, Aug. 1996.

[55] R. P. K. Guerney D. H. Hunt, German S. Goldszmidt and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th Internation World Wide Web Conference*, Brisbane, Australia, Apr. 1998.

[56] G. Hamilton and P. Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the 1993 Summer USENIX Conference*, pages 147–159, Cincinatti, OH, June 1993.

[57] S. M. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[58] S. Holzner. *XML Complete.* McGraw Hill, New York, NY, 1998.

[59] HP Labs. WebQoS. http://www.internetsolutions.enterprise.hp.com/webqos/.

[60] IBM. http://www.ibm.com/.

[61] Microsoft Corporation ISAPI Overview. http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.htm.

[62] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM '88 Symposium*, pages 314–32, Aug. 1988.

[63] S. Jamin, P. B. Danzig, S. J. Shenker, and L. Zhang. A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks. In *Proceedings of the SIGCOMM '95 Conference*, Boston, MA, Aug. 1995.

[64] Java Servlet Technology. http://java.sun.com/products/servlet/.

[65] M. B. Jones, J. S. B. III, A. Forin, P. J. Leach, D. Rosu, and M.-C. Rosu. An overview of the rialto real-time architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[66] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the Rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[67] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. B. no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[68] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server Operating Systems. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[69] V. Kanodia and E. W. Knightly. Multi-Class Latency-Bounded Web Services. In *Proceedings of the 8th International Workshop on Quality of Service*, Pittsburgh, PA, June 2000.

[70] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of the ACM SIGCOMM '91 Symposium*, pages 3–15, Sept. 1991.

[71] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Braham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communication*, 14(7):1280–1297, Sept. 1996.

[72] K. Li and S. Jamin. A Measurement-Based Admission Controlled Web Server. In *Proceedings of the IEEE Infocom Conference*, Tel-Aviv, Israel, Mar. 2000.

[73] C. W. Mercer. *Operating System Resource Reservation for Real-Time and Multimedia Applications*. PhD thesis, Carnegie Mellon University, June 1997.

[74] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[75] J. Miano. *The programmer's guide to compressed image files: JPEG, PNG, GIF, XBM, BMP*. Addison-Wesley Longman, 1999.

[76] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of the SIGCOMM '88 Symposium*, pages 11–21, Stanford, CA, Aug. 1988.

[77] J. C. Mogul. Operating system support for busy Internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[78] J. C. Mogul. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Symposium*, 1995.

[79] D. Mosberger. *Scout: A Path-based Operating System.* PhD thesis, University of Arizona, Tucson, July 1997.

[80] D. Mosberger, L. Peterson, P. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Latency. In *Proceedings of the SIGCOMM '96 Conference*, Palo Alto, CA, Aug. 1996.

[81] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

[82] NCSA httpd. http://hoohoo.ncsa.uiuc.edu/.

[83] Network Appliance, Inc., *NetCache.* http://www.netapp.com/level3/netcache/datasheet.html.

[84] J. Nieh and M. S. Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, New York, Oct. 1997.

[85] Netscape Server API. http://www.netscape.com/newsref/std/server_api.html.

[86] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[87] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[88] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[89] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 1996.

[90] PHP: Hypertext Preprocessor. http://www.php.net/.

[91] Resonate Inc. Resonate dispatch. http://www.resonateinc.com/.

[92] S. E. Schechter and J. Sutaria. A Study of the Effects of Context Switching and Caching on HTTP Server Performance. http://www.eecs.harvard.edu/~stuart/Tarantula/FirstPaper.html.

[93] P. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[94] O. Spatscheck and L. L. Peterson. Defending Against Denial of Service Attacks in Scout. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[95] Squid. http://squid.nlanr.net/Squid/.

[96] W. R. Stevens. *TCP/IP Illustrated Volume 1 : The Protocols.* Addison-Wesley, Reading, MA, 1994.

[97] W. R. Stevens. *TCP/IP Illustrated Volume 3 : TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols.* Addison-Wesley, Reading, MA, 1996.

[98] Sun Microsystems Inc. Solaris 7 operating environment. http://www.sun.com/solaris/.

[99] D. L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[100] thttpd. http://www.acme.com/software/thttpd/.

[101] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of the USENIX Mach Symposium*, Burlington, VT, Oct. 1990.

[102] R. Vanderbei. LOQO: An interior point code for quadratic programming. *Optimization Methods and Software*, 1999.

[103] R. Vanderbei and D. Shanno. An interior-point algorithm for nonconvex nonlinear programming. *Computational Optimization and Applications*, 13:231–252, 1999.

[104] B. Verghese, A. Gupta, and M. Rosenblum. Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[105] The World Wide Web Consortium. http://www.w3.org/.

[106] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Managament.* PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[107] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, Nov. 1994.

[108] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM SIGCOMM '96 Symposium*, Palo Alto, CA, Aug. 1996.

[109] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated Volume 2 : The Implementation.* Addison-Wesley, Reading, MA, 1995.

[110] Yahoo! Inc. http://www.yahoo.com/.

[111] C.-S. Yang and M.-Y. Luo. Efficient Support for Content-Based Routing in Web Server Clusters. In *Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[112] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Technical Conference*, Berkeley, CA, Jan. 1997.

[113] Zeus. http://www.zeus.co.uk/.

[114] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM Transactions on Computer Systems*, 9(2):101, May 1991.

[115] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An Architecture for Cluster-Based Web Servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.