QOS ADAPTATION IN REAL-TIME SYSTEMS

by

Tarek F. Abdelzaher

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1999

Doctoral Committee:
Professor Kang G. Shin, Chair
Assistant Professor Peter Chen
Associate Professor Farnam Jahanian
Assistant Professor Kimberly M. Wasserman
Nina Bhatti, Research Scientist, Hewlett Packard Laboratories

ABSTRACT

QOS ADAPTATION IN REAL-TIME SYSTEMS

by

Tarek F. Abdelzaher

Chair: Kang G. Shin

We propose to design, implement, and evaluate a software framework, called the *Adaptware*, that consists of architectural support, resource-management mechanisms, and programming abstractions for adapting Quality-of-Service (QoS) to dynamically-fluctuating resource capacity and demands. This framework is to reduce the cost and time of real-time software development by providing the infrastructure necessary for building *reusable* multi-purpose real-time software components. In much the same way as today's consumers can buy software and hardware components from different vendors and construct a computing environment tailored to their needs, the proposed framework will provide the means of building and integrating real-time system components so as to preserve their temporal correctness while making it possible to dynamically compute predictable end-to-end temporal guarantees commensurate with available resources.

The benefits of our framework are demonstrated by building a testbed and executing representative applications, drawn from both soft real-time systems such as multimedia, and hard real-time systems such as automated flight and process control. Software development costs of these applications will be reduced significantly by separating real-time application design from resource-capacity assumptions on the target platform, hence facilitating changes in platform capacity, configuration or load. We also demonstrate how our abstraction of platform capacity (away from the programmer) can help build real-time systems that can tolerate a range of violations of their load and failure hypotheses by adapting application QoS dynamically to the resources available.

For my parents and my dear wife

ACKNOWLEDGEMENTS

I would like to thank all the people who offered me help, encouragement, and support during the period of my graduate study. I would especially like to thank

- Professor Kang Shin, my advisor, for being a great mentor to me both personally and professionally. I owe him my success in Michigan and I thank him earnestly for giving me patient guidance, exciting motivation, and continued support throughout my studies and letting me have the freedom to make and learn from my mistakes in the process of shaping my research identity.

- Professor Farnam Jahanian for his friendly support, and for sharing interesting intuition, ideas, and visions with with me, as well as collaborating on a few exciting research efforts.

- Nina Bhatti, Rich Friedrich, Gita Gopal, Tai Jin, David Mosberger, Anna Zara, Martin Arlitt, John Dilley and Stephane Perret for offering me a great research environment and giving me their advice and support during my summer internship at Hewlett Packard in 1998. Special thanks goes to Nina Bhatti for being a great friend and collaborator and for a wonderful research and learning experience.

- Jan Jansson for offering exciting research and collaboration opportunities and for stimulating discussions on issues of interest to my work.

- Ella Atkins and John Reumann for lots of brainstorming discussions and for collaborating with me on *Adaptware*. Special thanks goes to Ella for developing a flight control application to test parts of the *Adaptware* resource management framework.

- Sugih Yamin for his constructive criticism and for shedding light on my research from a different perspective.

- Kim Wasserman and Peter Chen for being on my committee and for their insightful comments and suggestions.
- Anees Shaikh, Ashish Mehra, Wu Chang, Jennifer Rexford, Scott Dawson, Scott Johnson, Todd Mitton, and Martin Bjorklund, for being great office mates, for helping me with technical issues, and for discussing my work at different points in time. Special thanks goes to Anees and Ashish for their repeated collaboration with me on research problems related to the *ARMADA* project.

Last but not least, I would like to thank my parents very much for their support and for making it possible for me to pursue my professional goals in life. I would like to thank my friends for being there for me when I needed help, and I would like to thank my wife for her understanding and support of my research endeavors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

APPENDIX

CHAPTER 1

INTRODUCTION

In recent years, the real-time systems community has been making significant effort in using commercial-off-the-shelf components to develop performance-assured systems. Several research initiatives have been undertaken to enable writing cost-effective software that provides Quality-of-Service (QoS) guarantees on multiple platforms whose resource capacity, speed and load profile are unknown at application design time. Ideally, performance-assured systems should provide QoS guarantees that are easily customizable to the size, speed, and capability of the target hardware and OS. Platform-specific code, on the other hand, is a main contributer to the cost and complexity of commercial software. In QoS-sensitive applications, resource-capacity-dependent code unduly complicates the software programming effort and maintainability. In order to reduce this complexity, new programming abstractions, operating system mechanisms and portable middleware are needed to facilitate next-generation platform independent real-time programming. They should eliminate resource capacity dependent code from real-time software while providing QoS guarantees commensurate with platform resource constraints.

The thesis presents a resource-management framework, called *Adaptware*, that (i) enables application designers to express "flexible" performance requirements, (ii) enforces QoS guarantees, and (iii) adapts application timing behavior to platform capacity and load conditions. *Adaptware* is layered between the application and the OS kernel. It handles, on behalf of the application, the temporal correctness, adaptation, and resource allocation issues in a performance-assured system.

We focus on using *Adaptware* in conjunction with commercial servers. Services performed by the server, as well as service software itself, may have a substantial commercial value which generates interest in proper service design within the computing industry. The performance of the service affects the revenue it generates. With the advent of real-time and mission critical services such as online brokerage and multimedia, the problem of proper QoS-sensitive service design acquires added importance. *Adaptware* complements existing service architectures with QoS guarantees. In *Adaptware* we design and implement resource-monitoring mechanisms that automatically profile service execution times and monitor platform-load conditions. This resource monitoring will be fed back to, and integrated with, service scheduling. We develop scheduling mechanisms and policies that achieve maximum utility for a finite amount of resources. By abstracting away resource allocation and management mechanisms for QoS adaptation, *Adaptware* allows service developers to write distributed applications without regard to particular platform capacity or load. To demonstrate the independence between application code and QoS management in *Adaptware*, we demonstrate in this thesis the use of *Adaptware* to provide QoS guarantees on behalf of existing best-effort legacy servers, such as Apache, whose development predated our research. We show that when layered on top of Adaptware, the unmodified best-effort service achieves QoS-sensitive characteristics without code changes or recompilation.

1.1 Adaptation Support for Servers

A performance-assured server must be able to provide each client or class of clients their desired QoS irrespective of the behavior of other clients. This implies protection of well-behaving clients from those violating their QoS specification. Such protection may imply preempting, delaying or dropping the processing of non-conforming client requests or blocking mis-behaving connections, we call these mechanisms *QoS isolation*.

In order to achieve better QoS isolation between different clients, flows, traffic classes, or other abstract entities, we suggest the abstraction of QoS contracts. *Adaptware* allocates resources to QoS contracts which makes it easier to perform per-contract QoS control. Adaptware addresses and offloads the following concerns associated with QoS contracts:

- *QoS mapping:* Given contract parameters, the system should be able to map them into the corresponding resource requirements (e.g., CPU time and communication bandwidth). This mapping, however, depends on parameters of the underlying platform (e.g., processor speed). A system that adapts transparently to available platform resources should employ an adaptive QoS-mapping function. Some form of self-profiling capability is required to estimate service cost. *Adaptware* will be used to realize this self-profiling capability.

- *QoS optimization:* By virtue of the QoS contract between the server and clients, each client specifies a *range* of acceptable QoS. Given a variable amount of platform resources and a dynamically-changing number of clients, the system should adapt individual clients' QoS to achieve the maximum possible aggregate utility under given load conditions. We will derive (near) optimal policies to automatically select the level of QoS for each contract so as to maximize the aggregate utility for the community of clients.

- *QoS enforcement:* Enforcement mechanisms are required to achieve QoS-sensitive performance. Such mechanisms will monitor and police QoS contracts so that malicious clients cannot dominate the available resources. Enforcement mechanisms ensure that the clients abide by their part of the QoS contract. *Adaptware* investigates and implements enforcement mechanisms that do not require modification to the OS kernel, and do not require architectural changes to service code.

By shifting the authority in managing clients' QoS levels to *Adaptware*, the application code is decoupled from the assumptions on the underlying resource availability and capacity. Service utility is maximized over the community of clients for the given platform capacity and load.

1.2 Adaptation Middleware

We developed new middleware tools and APIs to embody our analytical and experimental results for future development of performance-assured QoS-adaptive systems. *Adaptware* tools reported in this thesis are summarized below:

- *qthreads:* A user-thread package that offloads QoS adaptation from the application programmer. Programming in *qthreads* enables expression of multiple application-QoS levels as well as QoS-degradation policies. Under an overload condition, the *qthreads* user-level scheduler transparently triggers QoS adaptation to select the appropriate QoS level for system load and maximize system utility under resource constraints. Load monitoring, thread execution time profiling, and QoS adaptation are performed internally by the scheduler.

- *CLIPS:* A *C*ommunication *L*ibrary for *I*mplementing *P*erformance-assured *S*ervices. Communication is an essential element of any distributed application. We build an end-host communication subsystem that facilitates QoS-adaptive communication. It enables the expression of multiple QoS levels on a per-connection, per-client, or per-service-class basis associated with each application. CLIPS implements the necessary monitoring and load-profiling support to perform QoS mapping and manage QoS-adaptive end-point communications.

- *qContracts:* A portable middleware layer that provides the approximate capabilities of *qthreads* and *CLIPS* on regular UNIX.

- *RTPOOL:* A distributed programming environment that augments *qthreads* and *CLIPS* with support for load sharing and invocation migration in distributed real-time applications. The environment implements protocols to migrate threads between machines, and assign their QoS levels such that the aggregate distributed system utility is maximized.

The above tools are the research vehicle we use to test ideas, models and algorithms for supporting next-generation platform-independent composable, adaptive real-time software.

1.3 Experimental Evaluation

A goal of our design and experimental study is to shed light on architectural challenges, resource-management solutions, and implementation difficulties of adaptive real-time systems. Thus, our evaluation of *Adaptware*'s efficacy includes the following broad metrics.

- *Performance Predictability:* Our resource-management mechanisms must make it possible for applications to give performance assurances. Our implementation must ensure that on-line mechanisms indeed meet the expectations set off-line.

- *Adaptation:* We will quantify the efficacy of our adaptation mechanisms. In case of resource shortage, application performance using *Adaptware*'s support will be compared to that of uncontrolled degradation of traditional applications at overload. System's reactions to overload are of particular interest to our research.

- *Deployment effort:* The advantages of *Adaptware* do not come for free. The cost of utilizing the new flexibility should be weighted against the benefits in terms of adaptability. We investigate this tradeoff in the context of different application scenarios. We show that in many cases the new QoS APIs can be hidden from existing application code thereby significantly reducing the *Adaptware* deployment effort, and that adaptation software imposes no significant performance penalty.

In order to experiment with platform-dependencies we need at least two different implementation platforms. Hence, we implement our abstractions both inside an experimental operating system and in middleware on top of regular UNIX. Insights are gained from comparing the two implementations.

1.4 Thesis Outline

In the rest of this thesis, we describe elements of the *Adaptware* framework. We begin in Chapter 2 by motivating the framework using web servers as a preferred example. Web servers were chosen due to the ubiquity of the Web and the proliferation of emerging web-based applications such as e-shopping, banking, brokerage, call routing (e.g., for 800 numbers), and multimedia access. The chapter demonstrates the need for QoS adaptation in web servers and illustrate its practicality from three different view points: (i) feasibility to apply adaptation technology (on the web) with no modification to existing servers, clients or service protocols, (ii) ability to achieve sufficient savings in resource requirements by adaptation, and (iii) non-intrusiveness of QoS adaptation in terms of performance overhead.

We then embark on developing a QoS adaptation framework that consists of architectural support, resource-management mechanisms, programming abstractions, and theory for adapting QoS to dynamically-fluctuating resource capacity and demands. Chapter 3 describes the main architecture of *Adaptware*, and explores utility-optimizing resource allocation policies.

A major focus of this work lies in QoS-enforcement mechanisms. Chapter 4 describes basic enforcement mechanisms on the end-system. It describes and evaluates a QoS-adaptive communication subsystem running on top of a microkernel-based operating system. It describes how to provide QoS guarantees for single and aggregate flows. It also describes a middleware framework that implements QoS adaptation on top of UNIX. While the middleware solution offers coarser-grained guarantees, the advantage of using this middleware is increased portability, since it may be used with little or no modifications on any standard variant of UNIX. An approach based on dynamic shared libraries is suggested for augmenting legacy code with QoS adaptation capabilities without recompilation.

Chapter 5 extends the basic adaptation mechanisms of Chapter 4 to distributed systems. It proposes a server architecture for distributed compute-intensive real-time applications. The server offers an abstraction of a single computing resource, while transparently performing QoS adaptation and load sharing that maximizes distributed system utility. We describe and evaluate the performance of a control application that uses the distributed server.

Finally, Chapter 6 presents elements of the theoretical framework required for *Adaptware* deployment in distributed *hard* real-time systems, such as the application presented in Chapter 5. Unlike soft real-time applications such as web hosting and multimedia, hard real-time applications require absolute guarantees on the satisfaction of individual timing constraints during system operation. An important problem is to demonstrate *a priori* that a system of tasks is able to meet its timing constraints under the given scheduling and resource assumptions (and assuming the correctness of the implementation of resource management). This NP-complete problem is attacked in Chapter 6 under assumptions of increasing complexity regarding task models, intertask communication, task precedence constraints, and task resource requirements. Both optimal and heuristic solutions are developed.

CHAPTER 2

ADAPTATION IN WEB SERVERS

2.1 Introduction

We investigate performance-assured, adaptive web servers, a motivating application for the *Adaptware* framework. The case study illustrates the need for adaptation in contemporary web servers, demonstrates how adaptation can be achieved without modifying existing client and server software, and evaluates the impact of the approach on service performance, overload behavior, predictability, and user-perceived utility. The goal of the study is to provide an "existential proof" of the feasibility and performance benefits of *Adaptware* in the context of a chosen application of interest. Such proof establishes a case for embarking on developing a generalized QoS architecture in the remainder of this thesis. The work presented in this chapter has been carried out at Hewlett-Packard Laboratories.[1]

We have chosen web servers as our example application for several important reasons. The Internet is presently undergoing substantial changes from a communication and browsing infrastructure to a medium for conducting business and selling a myriad of emerging services. The World Wide Web provides a uniform and widely-accepted application interface used by these services to reach multitudes of clients. These changes place the web server at the center of a gradually emerging e-service infrastructure with increasing requirements for service quality, reliability, and security guarantees in an unpredictable and highly dynamic environment. Yet, web servers today offer poor performance under overload, no means for client prioritization, no means for service-level adaptation, and no means for performance isolation among different classes of connections. As the request rate increases beyond server capacity, server response-time and connection error rate

[1] All software and ideas described in this Chapter were developed by the author during his internship at HP (Patent pending).

deteriorate dramatically, potentially causing client-perceived service outage. Since clients typically issue several sequential requests during a session with the web server, the entailing accumulation of errors results in severe, indiscriminate, performance deterioration. Due to the increasing scope and importance of web-based applications, and the inadequacy of QoS support in present web servers, we choose them for our case study.

Web QoS extensions can take several forms. In the simplest form, as an alternative to connection failure or rejection, clients may be willing to receive a degraded, less resource-intensive version of the requested content. Web content is adapted by the server in accordance with load conditions. The scheme not only allows more clients to access the server concurrently at peak load, but also reduces the amount of wasted resources when server load exceeds capacity. Resources at overload are wasted as the OS, communication subsystem, and server are tied up by an increasing number of client connections that eventually fail (e.g., time out) and are aborted by the client. In our measurements, for example, when the offered load is 3 times the server capacity, more than 50% of resources are wasted on eventually-aborted connections. The remaining capacity allows less than 1 in 6 connections to succeed, instead of 1 in 3. We present a content adaptation mechanism that allows the server to control its load by adapting delivered content while optimally utilizing its available resources. The mechanism virtually eliminates connection errors over a significant range of overload conditions and improves server resource utilization.

In dealing with the overload problem we recognize today's changing nature of the WWW. While initially, the WWW may have been a uniform, distributed hypertext browsing infrastructure, today both the published content and the category of applications accessing it are becoming increasingly diversified. A web server today might host several sites on behalf of parties with potentially conflicting interests, or serve clients of different importance to the content provider. Web servers, therefore, must support performance isolation between multiple hosted services,[2] and performance differentiation between classes of clients, e.g., based on the identity of the client, or the nature of accessed content. For example, a web server which hosts multiple sites should not allow any one site to dominate all resources forcing other sites to starve. Similarly, a web server at overload may need to give preferential treatment to more important clients (as defined by some configurable metric) as opposed to treating all requests alike. A main goal of our content adaptation mechanisms, therefore, is to provide performance isolation and QoS differentiation within the web server.

[2]The term *service* is used here to denote, for example, web hosting services (e.g., hosting a particular web site), e-commerce services (e.g., running a shopping mall's on-line cash register), or search services.

As the WWW extends to encompass QoS-sensitive content, such as sound and movie clips, as well as mission-critical applications, such as e-commerce, it becomes important to design web servers capable of providing QoS guarantees. For example, the web server should be able to guarantee for a hosted site a maximum throughput in terms of request rate and bandwidth delivered to its clients. We investigate content adaptation mechanisms that allow for such QoS provisioning.

In addition to dealing with server overload in a QoS-sensitive fashion, content adaptation technology, such as the mechanisms we describe in this chapter, has other important benefits. For example, it may be used to adapt server output to client-side resource limitations. The processing power, connection bandwidth and display resolution may vary significantly from one client to another. Content adaptation can provide the most appropriate version of content to each client in accordance with their resource constraints. Currently, content providers must fine-tune a compromise version of content that hopefully will not encumber slower clients, yet remain satisfactory to higher-end clients. The existence of adaptation technology, such as that described in this chapter, will allow content providers to deliver higher-quality content whenever possible thus enhancing their clients' browsing experience. While acknowledging this additional benefit of adaptation technology, we focus on adaptation as a means to control server overload.

The rest of this chapter is organized as follows. Our adaptation model and assumptions are presented in Section 2.2, followed by a feasibility study in Section 2.3 that demonstrates the potential benefits of deploying the proposed web technology. Section 2.4 describes our basic architecture, refined in Section 2.5 to provide QoS management capabilities such as performance isolation, QoS differentiation and QoS guarantees. Implementation details are discussed in Section 2.6. The presented web content adaptation architecture is evaluated in Section 2.7 by testing the performance of an implemented software prototype. The chapter concludes in Section 2.8 with a summary of contributions and implications on the *Adaptware* framework.

2.2 Content Adaptation Model

Content adaptation is viable if there were meaningful ways to adapt content on the web in a way that preserves adequate information while decreasing resource requirements. For example:

- *Image degradation by lossy compression:* In a survey of 80 shopping web sites (listed in Appendix A), conducted in July 1998, we found that GIF and JPG images alone constitute, on average, more than 65% of the total bytes of a site. In many cases, these images can be significantly compressed without an appreciable decrease in quality. To help appreciate the potential bandwidth savings, Figure 2.1 compares a 74KB GIF to a 8.4KB JPG image of the same object. Although the difference in byte size is roughly an order of magnitude, the difference in quality is insignificant on most clients' displays. This demonstrates a potential to conserve resources by degrading image quality (e.g., via JPG lossy compression).

- *Reduction of embedded objects per page:* From the server-side perspective, document size is not as important as the number of embedded objects per page. Upon retrieving a URL the client application sends independent requests to fetch its embedded objects. Each request to the server consumes a relatively large fixed overhead in addition to a variable document-size-dependent overhead. On an Apache server running on HP-UX we found the processing time per request to be 1.6 ms independently of the retrieved URL size, plus an additional 65 μs per each KB of delivered data.[3] Retrieving an 8 KB file will thus consume only 50% more processing resources at the server than retrieving a 1 byte file. Therefore, at overload, processing-time savings arising from reducing the number of expendable or cosmetic items per page (such as little icons, bullets, bars, separators, and backgrounds) can be significant.

- *Reduction in local links:* Another way of adapting content is to reduce local links. This reduction will affect user browsing behavior in a way that tends to decrease the load on the server as users access less content.

Having illustrated several ways to degrade web content, a designer must decide whether content should be pre-processed *a priori* and stored in multiple copies of different qualities, or be compressed on the fly as need arises. Previous work considered adapting multimedia content to match client bandwidth limitations assuming sufficient bandwidth on the server. Such efforts utilize on-the-fly compression techniques which introduce extra processing overhead but reduce the size of delivered data. In contrast to these methods, we focus on content adaptation as a means to alleviate server overload. When the server is overloaded introducing an extra data compression stage, for example, will only further impair performance. Thus, we require that content be pre-processed *a priori* and stored in multiple copies that differ in quality and size. The approach is quite affordable for web servers given the low prices of secondary storage. One may imagine appropriate authoring tools that allow web content developers to annotate parts of the content as expendable, degrad-

[3]These are HTTP 1.0 measurements taken for Apache 1.3.0 on a single-processor K460 (PA-8200 CPU) running HP-UX 10.20, with 512 MB main memory, and GSC 100-BaseT network connection.

Figure 2.1: Comparing a 74KB GIF and an 8.4KB JPG

able, or important. This can be done by using an extended form of HTML that is preprocessed by content management tools to create and maintain separate standard-HTML versions of the site. Pre-processing will generate automatically multiple copies of the content that differ in quality and resource requirement. Such tools will offload the responsibility of generating and maintaining multiple content versions from the content provider.

From the server's perspective, multiple content trees can be made available with different versions of the same URLs. To identify the different versions, the path to a particular URL in a given content tree may be the concatenation of the content tree name and the URL name, prefixed by the name of the root service directory of the web server. For example, in the root service directory "/root" one may create two content trees, "/full_content" and "/degraded_content". A URL, such as,

"/my_picture.jpg" will be served from the directory "/root/full_content/my_picture.jpg" if appropriate for the particular client class and if server load permits. Otherwise the URL will be served from "/root/degraded_content/my_picture.jpg" thus supplying a more economic version.

The scheme applies to dynamic content as well, e.g., that generated by CGI scripts. Multiple content trees may contain different versions of the named CGI script (e.g., "/cgi-bin/my_script.cgi"). The script URL is prepended by the right tree name (e.g., "/full_content" or "/degraded_content") to determine which version of the script to execute under given load conditions. For example, the web server can invoke a less resource-intensive search script that looks for only the first 5 matches under overload, instead of one that looks for 25 matches under normal load conditions.

Our experience indicates that serving dynamic content is much more resource-consuming than serving static content. Our adaptation mechanism allows some dynamic content to be replaced by static content by switching to a different content tree. For example, an on-line vendor can use a dynamically-generated version of their product catalog that interacts with a stock database to display the items currently in stock. Content adaptation mechanisms allow the server to switch automatically, without system administrator assistance, to a statically pre-stored catalog version whenever bandwidth becomes scarce thereby saving resources.

We focus on static content, and assume that multiple copies of this content are available to the server. The development of authoring tools for content preprocessing is out of the scope of this thesis.

2.3 Feasibility of Adaptation

A first step towards motivating adaptation technology is to quantify its efficacy in terms of the achieved potential performance benefits. For that purpose we performed a quick feasibility study analyzing the nature of content of 80 selected web sites (listed in Appendix A). The goal of this initial study is to demonstrate a credible range of web applications for which benefits of content adaptation can be easily applied. We decided to analyze on-line shopping sites. On-line shopping is a growing practice. According to data obtained by Hewlett-Packard Labs in 1998, more than 98% of large businesses and more than 33% of small businesses advertise their products on-line. Although some e-shoppers still prefer to make the final purchase by phone, it is becoming more common to browse available alternatives electronically before embarking on a purchase. Shopping sites, in addition to their growing popularity, have several other favorable attributes that make them a good candidate for our analysis. For example:

- As more users find it easier to shop online, a shopping site may get a large number of hits, and thus (unlike personal web pages, for example) is susceptible to overload. Shopping sites can therefore credibly benefit from our adaptation techniques.

- Since businesses are often not web experts they outsource site management to professional web-hosting service providers who might be interested, for economical considerations, in cohosting several sites per machine yet providing performance guarantees to each. Since we develop the technology for performance isolation and QoS guarantees we find it natural to analyze the type of sites to which this technology is most likely to be applied.

- Shopping sites are visually intensive. They should attract e-shoppers attention, and depict products in a way that encourages a purchase. These sites can greatly benefit from adaptation technology that allows providing a richer content whenever possible, yet handles overload by switching to a less resource-intensive version when needed.

- We found that e-shopping sites are generally more moderate in size than other sites we considered, such as sports sites, or news sites, and hence are easier to analyze.

In our study, we downloaded site content, degraded its quality, and estimated the expected performance improvement. Performance improvement was estimated in terms of the reduction, upon content degradation, in consumed server utilization arising from client access. The consumed server utilization was derived from server request rate and bandwidth by profiling a real web server (Apache) to relate utilization, service request rate, and requested URL size as will be discussed later in this chapter. Performance improvement by content adaptation depends on the particular way content is adapted, which may be arbitrary and subjective. We therefore found it informative to determine an upper bound on such performance improvement achievable by any content adaptation heuristic.

Intuitively, the best performance improvement is achieved if all content is reduced to plain text. Being too severe a degradation, it serves as an upper bound on the actual performance improvement one may reasonably expect from using a milder degradation approach.

Performance improvement due to adaptation further depends on client access patterns and caching. More popular pages tend to be cached by proxies thereby reducing further requests for these pages on the server. As a result, page access distribution tends to be more uniform on the server than on the proxy. In the absence of detailed statistics on client access patterns for the surveyed sites we will ignore page popularity and assume that page access is uniform. We believe that this simplifying assumption will err on the safe side making our performance-improvement estimates more conservative than they actually are. This is because visually intensive pages attract more clients thus creating a positive correlation between access frequency and page complexity. This correlation increases the performance gains attained by adapting popular page content over what we account for when we ignore page popularity.

Client caching affects the efficacy of adaptation by reducing the number of accesses on the server. In the absence of client caching each HTML page access will entail downloading all its embedded objects. Server access rate and bandwidth delivered to the client will therefore depend on the total number of embedded objects per page. On the other hand, if the client has a large cache, once an embedded object is downloaded it will no longer be requested again by the same client. Thus, performance is affected on average by the incremental number of new objects per page (those not already in the client's cache). We computed two estimates of performance improvement, one that corresponds to no client caching, and one that corresponds to an infinite cache size (and long sessions). We call the former estimate optimistic, since it provides the maximum performance improvement. The latter estimate is pessimistic. The actual performance improvement is somewhere in between. In the following we present these estimates taking into account the aforementioned considerations.

2.3.1 The Optimistic Estimate

In this case we assume that the client accesses at random one page of the site.[4] No parts of that page are cached by the client *a priori*. Let the number of embedded objects per HTML page be Emb in the site's original content tree. Retrieving exactly one web page will therefore impose, on average, $1 + Emb$ accesses on the server (1 access to retrieve the HTML file and Emb accesses to retrieve its embedded objects). If content is degraded to pure text, retrieval of the same page will impose only 1 access.

To compute the savings in bandwidth resulting from content degradation, let the average HTML file size be H bytes, and let the average embedded object size be I bytes. The bandwidth delivered per page is therefore $H + IEmb$ if content is not degraded, and H if content is degraded to text.

We have experimentally verified, as discussed later in the chapter, that server utilization, U,

[4]This is only an approximation since some pages are more popular than others. Note that more popular pages, however, will tend to be cached by proxies making server accesses more uniformly distributed.

consumed by a given request rate R and delivered bandwidth BW is accurately approximated by the linear function $U = aR + bBW$, where a and b are measurable constants that depend on the server software and platform. The ratio of server utilization values consumed by retrieving an average web page before and after degradation is thus:

$$Ratio_{optimistic} = (a(1 + Emb) + b(H + IEmb))/(a + bH) \tag{2.1}$$

This ratio reflects the performance improvement achieved due to degradation. We computed parameters a and b by profiling our Apache server as will be discussed later. The parameters I, H and Emb were computed from the actual statistics of the downloaded sites. The top curve in Figure 2.2 plots the resulting performance improvement ratio. The curve is discussed in more detail in the following subsection after we compute the pessimistic estimate as well.

2.3.2 The Pessimistic Estimate

To compute the pessimistic estimate of performance improvement, we assume the client has an infinite cache. The longer the client's session is, the more pages are cached and the less is the load imposed by the server. An infinite session will eventually cache all pages of the site, downloading each exactly once. Let h be the number of HTML files on the site, d be the number of unique embedded objects. The performance improvement achieved by content adaptation is the ratio of utilization values consumed to download the site before and after degradation. The ratio is:

$$Ratio_{pessimistic} = (a(h + d) + b(Hh + Id))/(ah + bhH) \tag{2.2}$$

Using simple algebraic manipulation, the above equation can be rewritten as:

$$Ratio_{pessimistic} = (a(1 + Emb_{min}) + b(H + IEmb_{min}))/(a + bH) \tag{2.3}$$

where $Emb_{min} = d/h$ is the ratio of the number of embedded objects to HTML files on the site. Note that the number of embedded objects per page, is $Emb = Emb_{min}$ if every embedded object is referenced in exactly one HTML file. Since some embedded objects (e.g., common icons, or backgrounds) may be referenced in more than one HTML file, in general $Emb \geq Emb_{min}$. Substituting this inequality in Equation (2.1) and comparing with Equation (2.3) we can see that $Ratio_{optimistic} \geq Ratio_{pessimistic}$. For an arbitrary user session length and cache size, we expect the performance improvement to lie generally between $Ratio_{optimistic}$ and $Ratio_{pessimistic}$.

The values d, h, Emb, H, I above were computed from the data of each surveyed site. The constants a and b were measured for our server (Apache 1.3 running on an HP-UX platform). The resulting estimates are shown in Figure 2.2. The figure depicts the percentage of surveyed sites, $P(x)$ (on the vertical axis), whose performance will improve by *at least* $x\%$ (the horizontal axis) upon adapting their content to pure text. Both the optimistic and pessimistic percentage estimates of are shown. Thus, for example, consider the point $x = 400$. The estimates indicate that 30% to 90% of all sites will improve performance by at least 400%. Another way of interpreting the graph is to consider a particular percentage of sites $P(x)$, and observe the corresponding performance improvement range. For example, consider the point $P(x) = 60$. The estimates indicate that *at least* 60% of all sites will see a performance improvement of 200% to 700%. Clearly, these are very encouraging results.

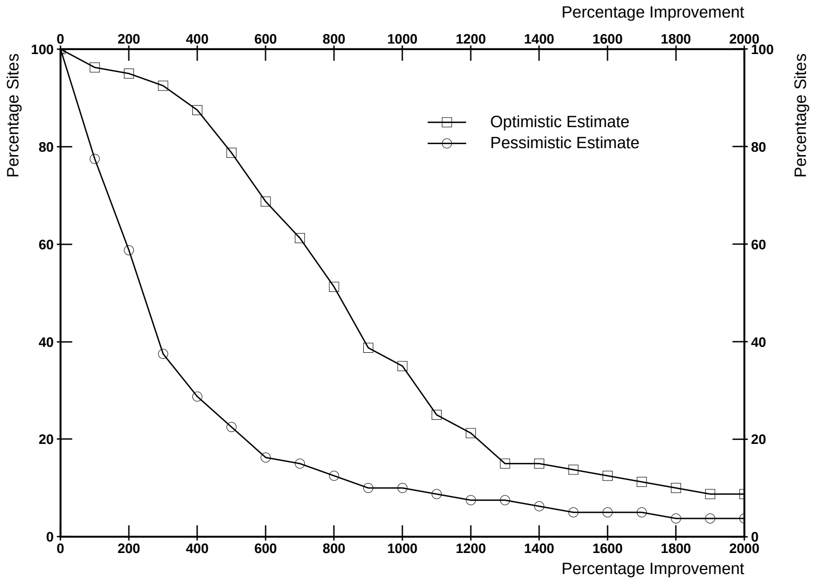


Figure 2.2: Performance Improvement Expected from Degrading to Text

2.3.3 A Degradation Heuristic

In order to access performance improvement for less severe degradation methods (as opposed to degrading to pure text) we repeated the derivation of optimistic and pessimistic performance improvement bounds when content of surveyed sites is degraded according to the following heuristic rules:

Rule 1: Remove all GIF images smaller than 1KB. Such images almost always constitute disposable cosmetic icons and clipart items.

Rule 2: Degrade all images that are longer than 32KB by a factor of 8. As demonstrated in Figure 2.1 large images can generally be compressed by an order of magnitude with no significant effect on quality.

Rule 3: Eliminate redundant items. Let each embedded object be referred to from exactly one HTML page.

We skip the derivation of performance improvement expressions used, since it is very similar to the derivations presented in Section 2.3.1 and Section 2.3.2. Figure 2.3 compares the optimistic and pessimistic performance improvement bounds computed in this case. While the optimistic estimate shows a large performance improvement, the pessimistic estimate reveals an important observation. Namely, the performance improvement, from the server's perspective, comes mostly from reducing the number of hits on the server (reducing the embedded objects per page), and not from image size reduction in the surveyed sites. To explain this observation, note that in the pessimistic case, the degradation heuristic at hand conserves bandwidth but does not significantly reduce the number of hits on the server. This is because in the pessimistic case we assume that clients have an infinite cache (i.e., download each embedded object only once), meaning that **Rule 3** has little or no effect on reducing the average number of downloaded objects per page. Figure 2.3 shows that the resulting performance improvement (by applying the first two rules) is not significant. In the optimistic case, however, performance improvement comes from both bandwidth reduction and reduction in hit rate. The combined effect is much more pronounced. Bandwidth reduction is therefore important only in the sense of reducing client's response time when client bandwidth is the bottleneck, but not in the sense of reducing server overload.

The above feasibility study gives insights into how content should be degraded for maximum performance improvement, as well as estimates how much performance can be improved. It is shown that content adaptation can indeed lead to significant resource savings for a large category of sites.

2.4 Adaptive Content Server Architecture

In this section we discuss an architecture for content adaptation, and instantiations of this architecture that differ in their complexity and capabilities. The architecture will improve server overload

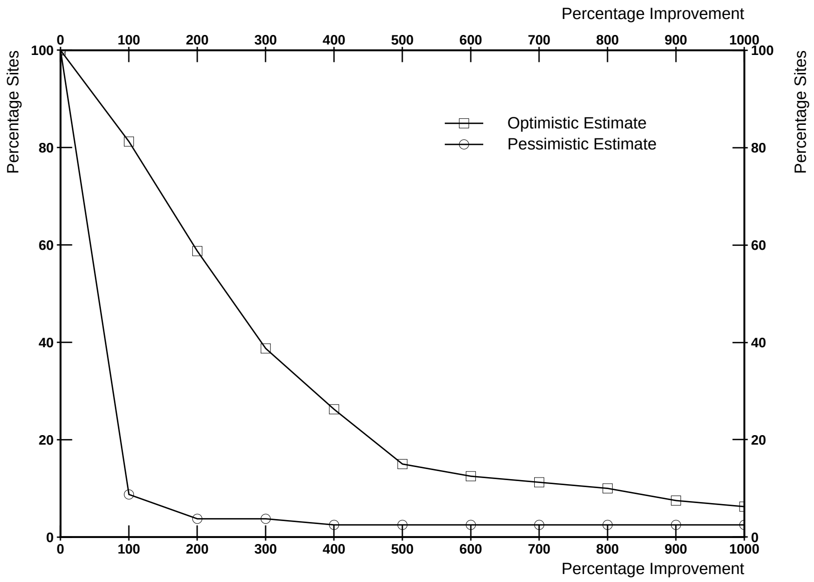


Figure 2.3: Performance Improvement Expected from Mild Degradation

behavior by adapting delivered content to load conditions. In Section 2.5 we generalize this architecture to provide QoS guarantees, performance isolation and service differentiation. Our goal is to achieve these properties transparently to the web server. Existing servers should be able to benefit from our technology with no need for software modification or recompilation. We present two ways such transparency is achieved; the external process approach and the middleware approach. Both assume that content trees of different quality are given *a priori*.

The external process approach places adaptation software in a separate process that runs concurrently with the web server. The process causes content adaptation by switching a link from the server's root service directory to the content tree that matches load conditions. The middleware approach, on the other hand, adapts delivered content by intercepting clients' requests to the web server and changing the path of the requested URL such that it refers to the "right" content tree. Adapting content on a per-request basis results in a much finer-grained control over server utilization and allows developing extensions such as performance isolation, service differentiation, and QoS guarantees. In the following we present the architectural components common to both approaches, then their realization in the context of each. In general, content adaptation software consists of the following components:

- *Load Monitor:* Web server load has to be monitored in order to detect overload conditions.
- *Adaptation Trigger:* The adaptation trigger maps the monitored load value into a decision to

invoke, undo, or change the extent of content degradation as appropriate.

- *Content Adaptor:* Once the trigger fires, indicating a state of server overload or underutilization, action is required to restore server load to desired conditions, if possible.

- *Request Classifier:* In general, the action taken by the content adaptor may depend on the identity of the client or the requested content. For example, a less severe action may be taken with more important clients. A request classification mechanism is therefore needed. The mechanism allows giving preferential treatment to a subset of sites or clients as appropriate to meet QoS guarantees.

We describe different implementations of the above components, their advantages and disadvantages, as well as their impact on achievable server functionality.

2.4.1 The Minimal Adaptive Server

In the simplest case, adaptation software is implemented as a process external to the unmodified web server. The software is allowed to switch, depending on load conditions, between two content trees; one for high quality content and the other for degraded quality content. The architecture is depicted in Figure 2.4. It implements an instance of a load monitor, adaptation trigger and content adaptor as follows.

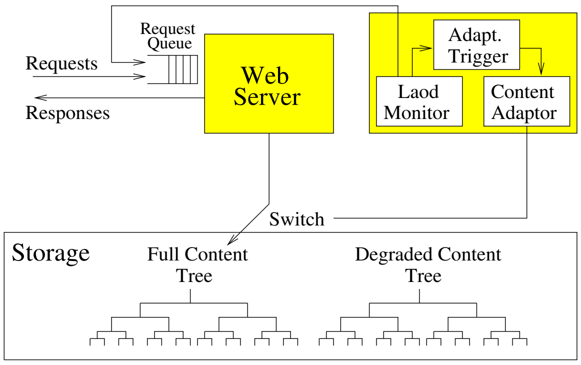


Figure 2.4: The Minimal Adaptive Server

- *Load Monitor:* A simple way of deciding whether or not the server is overloaded is to monitor server's response time. This is accomplished by a local process that periodically sends HTTP requests. Since the server serves requests in FIFO order, response time is proportional to the length of the server's input request queue (e.g., the server's socket listen queue in a UNIX implementation). When the server is underloaded the queue tends to be short (or empty) resulting in small response times. At overload the request queue overflows making the response time grow an order of magnitude. This approximately bimodal behavior of the queue has been verified by our tests as shown in the evaluation section and can serve as a clear overload indicator. The advantage of estimating queue length by monitoring response time (rather than, say, counting queued requests) is that such a mechanism can be implemented outside the server requiring no modification to its code.

- *Adaptation Trigger:* Adaptation is triggered when the monitor senses an increase in server response time beyond a precomputed threshold, $Thresh$. This threshold can be set equal to (or slightly smaller than) the maximum server response time specified in a QoS agreement, if any, thereby causing adaptation when the agreement is about to be violated. In the absence of such a specification, the threshold can be derived from the preconfigured maximum input request queue length, Q, and the average service time, S. For example, if we consider a 90% full queue to be an overload indication, the trigger can be set to $Thresh = 0.9QS$.

- *Content Adaptor:* Once the adaptation trigger is fired, the content adaptor switches transparently from the high quality service tree to the degraded quality service tree in the root service directory. The content adaptor is implemented outside the server. It switches content trees by changing directory links. For example, let the root service directory be "/root", let the high quality tree name be "/full_content", and let the degraded quality tree name be "/degraded_content". The content adaptor creates a link from "/root" to "/root/full_content" to be used during normal operation. At overload, it changes that link to "/root/degraded_content". Thus, when the unmodified server accesses the same file name, such as "/root/my_url.html", it transparently retrieves the file "/root/degraded_content/my_url.html" rather than the file "/root/full_content/my_url.html" when the system is overloaded. The retrieved file size will thus vary depending on load conditions. The approach is easily extendable to a server with more than two content trees.

The advantage of the above-described scheme is that its implementation does not require access to the server source code nor does it require recompiling or relinking the server with new or modified middleware libraries. The solution is completely transparent and fairly portable to different platforms.

Figure 2.5 compares the performance of an adaptive and non-adaptive servers by plotting the connection error probability versus request rate. In this experiment we generated requests for 64K images at an increasing rate. An adapted 8K version of the images was available in the degraded content tree. As shown in the figure, the traditional server suffers an increasing error rate when offered load exceeds capacity at about 160 reqs/s. In contrast, our adaptive server switches to less resource-intensive content thus exhibiting almost no errors up to about 3 times the above rate. In general, the extent of performance improvement will depend on workload and degree of content degradation available.

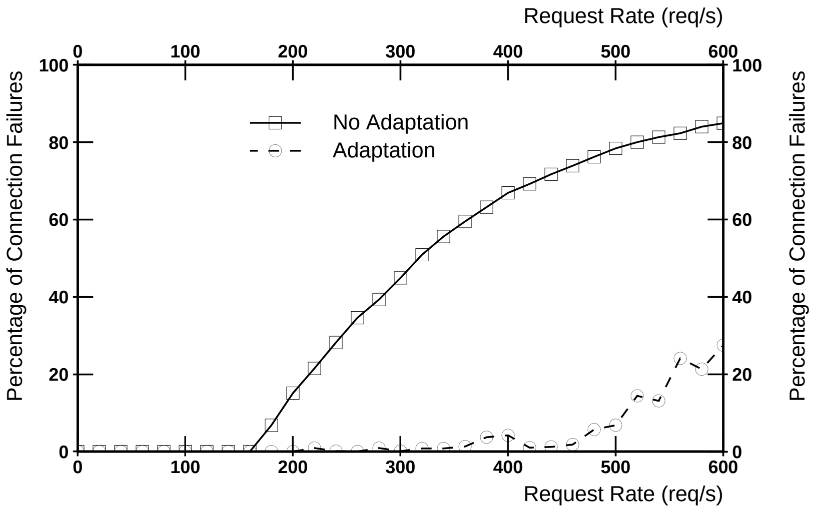


Figure 2.5: Adaptation and Connection Failure Probability

2.4.2 An Enhanced Adaptive Server

While the aforementioned simple adaptation technique increases maximum server throughput and decreases error rate shown in Figure 2.5, it has some limitations. For example, it is not obvious when to switch back from degraded content to high quality content, and the server may become underutilized upon switching to degraded content. Our measurements indicate that server response time is not particularly indicative of the degree of underutilization. The input request queue tends to remain small as long as the server is operating below capacity. It is therefore difficult to tell whether or not reverting to high quality content at a particular time will result in overload. This limitation can be circumvented by measuring the load on the server machine instead of server response time. The measurement should give an idea of how underutilized the server is. When utilization decreases below a configurable value, the server may revert to non-degraded content.

CPU utilization is sometimes regarded as a good load indicator. However, measuring CPU utilization by adaptation software to estimate the load on the server may give incomplete or misleading information. For example, the existence of a low-priority process or thread in the server that implements a busy-wait loop on some event will render CPU utilization identically 100% even when the server is "idle" (i.e., does not have any requests to serve). Furthermore, depending on the ratio of the platform's CPU bandwidth to the communication bandwidth, a server may become overloaded due to communication bandwidth saturation even at low CPU usage. Unfortunately, the bottleneck resource can fluctuate between the CPU and the network depending on the load mix. For example, we observed on our test platform that a large number of requests for small objects tends to saturate the CPU while a smaller number of requests for larger objects tends to saturate the network. Thus, CPU utilization alone is not indicative of how close the system is to its full capacity, since its capacity is bounded by that of the bottleneck resource. A similar concern arises when all server threads or processes are blocked on disk I/O. The resulting CPU idle time cannot be taken advantage of since I/O is the bottleneck.

We developed two different mechanisms for measuring system load that account for the utilization of the bottleneck resource. The methods can be implemented at any software layer where server requests and responses are visible. In particular they can be implemented in a middleware layer below the server, transparently to server software, by embedding them in the *read()* and *write()* socket library calls. These mechanisms are described below.

Server Utilization Measurement

Logically, a server is "fully utilized" when an increase in request rate will result in connection failures. By the same token, a server is not utilized when it has no requests to serve. Between these two extremes a utilization scale can be constructed that inherently accounts for the load on the bottleneck resource. We developed two methods for measuring utilization:

- *The Linear Approximation Method:* We have established that system utilization, U, consumed by processing client requests can be adequately approximated by a linear function of measured request rate R, and delivered bandwidth, BW, such that:

  $U = aR + bBW$

  where constants a and b can be computed by either on-line or off-line profiling as will be described and analytically justified in the evaluation section. The function U is good for estimating offered load as long as it does not exceed server capacity. When capacity is exceeded the measured rate R (and consequently the bandwidth W) saturates and possibly decreases due to overflow of kernel queues and abortion of an increasing number of connections in the kernel. These effects render the above linear approximation invalid under overload, since R, measured above the kernel, no longer reflects all requests. We therefore combine the linear approximation with response time monitoring to determine the load on the server. The combined utilization measurement function is as follows:

- If measured response time is above threshold then let $U = 100\%$
- If measured response time is below threshold then let $U = aR + bBW$.

One advantage of this method is that it is easy to implement in a concurrent system such as a multiprocess web server. Middleware code running in the context of each server process or thread, T_j, records the observed request rate R_j and delivered bandwidth BW_j by that thread and computes its consumed utilization U_j. A separate monitor process or thread then reads and sums up the recorded per-thread values to obtain the aggregate request rate $R = \sum_j R_j$, bandwidth $BW = \sum_j BW_j$, and utilization $U = \sum_j U_j$ of the server. Access synchronization to shared data structures is not required since there is only one writer to any piece of recorded data. The scheme can be implemented transparently to the server. In particular, the request rate, R_j, can be measured transparently by embedding a request counter in the $read()$ socket call used by the web server. The delivered bandwidth, BW_j, can be measured transparently by embedding a response-length accumulator in the $write()$ socket call.

The method allows server capacity planning for QoS guarantees. It provides means for converting a desired request rate and bandwidth into a corresponding resource capacity allocation (i.e., allocated utilization). On the disadvantage side, it requires computing the constants a and b, e.g., via *a priori* profiling. In Section 2.5.4 we describe how this is done.

- *The Gap Estimation Method:* A simpler method for estimating the fraction of time the server spends serving requests is to increment a global counter upon every request arrival and decrement it upon every response departure. As before, the counter can be implemented in middleware. The counter will return to its initial value only when a "gap" is present, i.e., when all current requests have been served, and no additional requests have arrived yet. By summing up the gaps over a period of time, T, a monitor process or thread can compute the total "idle time", G, within that period, where idle time is the time where no requests are pending. The utilization is estimated as $U = (T - G)/T$. The method is illustrated in Figure 2.6, which shows concurrent processing of request bursts separated with idle time. The method does not require *a priori* profiling, and does not require monitoring response time. However, it has the disadvantage of using a global counter that may be updated by multiple writers and thus requires some form of locking or access synchronization that may degrade performance. More importantly, it does not allow mapping rate and bandwidth requirements into system capacity allocation requirements and therefore, in itself, does not provide means for capacity planning for QoS guarantees. Another limitation of this technique is that it computes only aggregate utilization. It does not provide, for example, an accurate way of knowing whether or not an individual hosted site exceeded its capacity allocation, and thus is not a good choice for implementing QoS isolation. Finally, the technique works well only when the bottleneck resource is local to the server. Thus, for example, it works well only when all clients are fast (e.g., on a departmental Ethernet behind a firewall). If a client is the bottleneck, its request may take a very long time to serve for reasons other than server overload. The gap estimation method will erroneously count this service time towards server resource consumption.

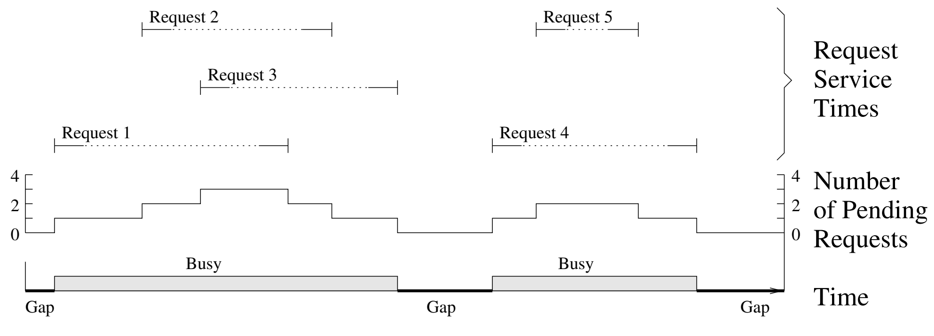


Figure 2.6: Measuring Utilization via Gap Estimation

2.4.3 Server Utilization Control

The adaptation software described so far implements mechanisms for measuring system load that can be used to toggle between two modes of operation: a high quality delivered content mode and a degraded content mode. This bi-modal nature makes it impossible to achieve adequate system utilization when adaptation takes place. When overload is detected all client requests are adapted potentially making the server underloaded. This effect is shown in Figure 2.7 which compares the delivered bandwidth of our server to that of a non-adaptive server for the experiment reported in Figure 2.5. The bandwidth is plotted versus request rate as the servers are driven to overload. Note from Figure 2.5 that the servers saturate at a request rate of about 160 reqs/s. A sharp drop in delivered bandwidth is shown in Figure 2.7 when that particular rate is reached on the adaptive server. This drop occurs because overload is successfully detected at that point, trigerring adaptation. Adaptation takes place by switching to less resource-intensive content leaving server bandwidth underutilized. We would like to avoid overload without underutilizing the server. Our approach to achieve this goal is to degrade only a fraction (rather than the entire population) of clients. The fraction can range anywhere from "no clients degraded" to "all clients degraded". It is determined by a utilization control loop that regulates automatically the number of clients degraded so that the desired system utilization is achieved. In order to remain effective even after all clients have been degraded, the utilization control loop will start rejecting requests, essentially performing an admission control function as well.

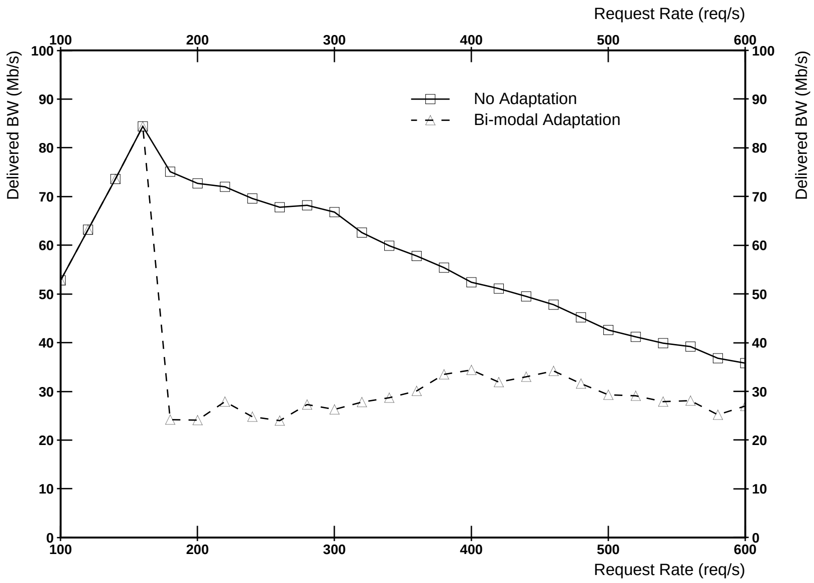


Figure 2.7: Server Underutilization

The utilization control loop described below is one of the more interesting contributions of this thesis. In its simplicity and general applicability it is a very powerful way of controlling resource allocation without kernel-level resource allocation and enforcement mechanisms. It employs admission control and adaptation techniques to regulate resource consumption as desired even in the absence of an accurate application load model and without knowledge of execution times and resource requirements of the service. While the control loop relies only on well-known fundamentals of classical control theory, the use of this theory in the context of web servers is the novelty of this work. The loop is composed of two parts; an adaptation controller that determines the amount of partial degradation required, and an actuator that carries out the degradation. In the application at hand, the actuator implements content adaptation; it degrades delivered content to a specified fraction of clients. We call this actuator a content adaptor. The following subsections describe these two components, starting with the content adaptor.

The Content Adaptor

Let the abstract parameter G be the "control knob" that tunes the extent of partial degradation required from the adaptor. The adaptor accepts input G in the range $[0, M]$, where M is the number of content trees. For a typical adaptive server we expect there to be two content trees, i.e., $M = 2$. For a non-adaptive server, $M = 1$. Trees are numbered $1, ..., M$ from lowest quality tree to highest quality tree. In general, the parameter G is a fractional number composed of an integral part I and a fraction F, such that $G = I + F$. Informally, the two nearest integers to G (namely, I and $I + 1$, where $I < G < I + 1$) determine the two content trees from which requests are served. The fractional part F determines the fraction of requests served from each of the two trees. Requests are served from the two trees with probability $1 - F$ and F, respectively. If G is an integer (i.e., $G = I, F = 0$), all requests are served from tree I. The upper extreme, $G = M$, indicates that all requests are to be served the highest quality content. Decreasing G decreases server load by degrading more clients. $G = 0$, in this numbering scheme, stands for request rejection. The following formal rule is used by the content adaptor to determine which tree to serve an incoming request from:

- If G is an integer (i.e., $G = I,\ F = 0$), the request is served from tree I (or rejected if $I = 0$).
- If G is not an integer, a pseudo-random number r is computed in the range [0, 1] upon receipt of the request. If $r < F$ the request is served from tree $I + 1$. Otherwise it is served from tree I (or rejected if $I = 0$).

As depicted in Figure 2.8, this algorithm provides a continuous degradation spectrum that ranges from serving all requests from the highest quality content tree to rejecting all requests. While

G determines degradation, it does not specify *which* requests should be degraded. The identity of degraded requests depends on the way the number r is computed given the identity of the requesting client. In this section we assume that all requests are equal. Thus, other than the need to fully utilize resources, there is no particular grounds for degrading some but not all of the requests. With that in mind, we present examples of two functions for computing r, and their effects on the client-perceived web server behavior.

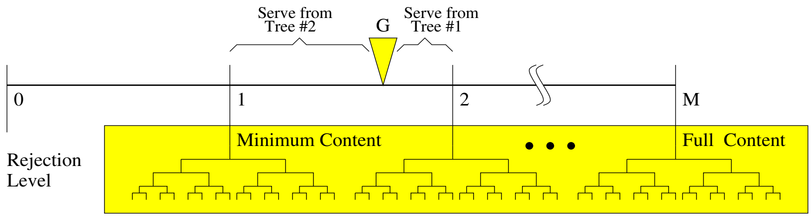


Figure 2.8: The Partial Degradation Range

- *Random (with uniform distribution):* Upon arrival of each request a random number r is chosen with uniform distribution in the range [0, 1]. From the client's perspective this method may result in fluctuations between good and degraded content over the duration of the client's session with the web server, as the quality of content is determined according to the value of r, where r is generated independently for each request in the session using a random function. Such fluctuations may be distracting. It may be more desirable to be consistent in the quality of content presented to a given client.

- *Client-identifier hashing:* A client can be identified using a cookie, or an IP address. A hashing function $h()$ is applied to the client's identifier that transforms it into the number r in the range [0, 1]. This method provides a consistent quality of content for each client accessing the server since each client is always hashed into the same number. Note, however, that the method may permanently discriminate against some of the clients depending on which number their identifier maps to. In particular, clients who map into larger numbers will be degraded first. To prevent such discrimination, the hashing function may be toggled periodically, e.g., between $h()$ and $1 - h()$. The toggling period should be long enough compared to the average client's session length to reduce quality of content fluctuations during a session's lifetime.

The Utilization Controller

The previous subsection describes how partial degradation is achieved once the control parameter G is known. This section presents how the parameter G is determined by the adaptation controller in a self-regulating fashion. We use control theory to stabilize an overloaded server at a desired level of resource utilization. Let U_t be the desired target utilization of the server. Let U be the utilization computed by any of the methods described in this section. Let e be the "utilization error", $e = U_t - U$. Note that positive values of error indicate underutilization while negative values indicate overload. The adaptation controller samples the error e at fixed time intervals and produces an output, G, proportional to the sum of the observed samples since system startup. Let $e[k]$ and $G[k]$ denote the values of error e and controller output G respectively at the kth sampling instant. In the simplest case, at the kth sampling instant the controller performs the following computation:

$G[k] = \sum_{0 \leq i \leq k} Ke[i]$

(or equivalently, $G[k] = G[k-1] + Ke[k]$)

If $(G[k] < 0)$ **then** $G[k] = 0$

If $(G[k] > M)$ **then** $G[k] = M$

where K is a proportionality constant.

Intuitively, if the system is overloaded (i.e., $U > U_t$) the accumulation of negative error values $e = U_t - U$ results in a decrease in G. Since a higher G represents better QoS, a decrease in G results in degrading some clients which decreases server utilization (and vice versa). When the system reaches the target utilization (i.e., $U = U_t$), the error e becomes zero, and this G is fixed.

Classic control literature proposes analytic techniques that compute the value of K in the above equation for best convergence. The apparatus is often referred to as an integral controller. More sophisticated versions of that controller include the Proportional Integral (PI) controller and the Proportional Integral Differential (PID) controller. We use a PI controller in our loop, tuned for quarter amplitude damping, which is a traditional industrial control practice [122]. The controlled process is modeled as a dead time element, of value equal to half the sampling time of the controller. The control loop is depicted in Figure 2.9. For completeness, we describe the mathematics involved in tuning the PI controller.

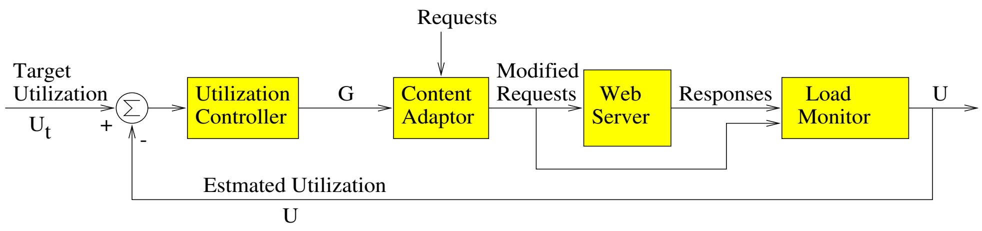


Figure 2.9: The Utilization Control Loop

Tuning the Controller

The utilization control mechanism implemented in our adaptation software uses a PI controller. The standard PI controller is given by the equation:

$$G = K_p(e + K_i \int_0^t e dt) \tag{2.4}$$

where e is the controller's input (the utilization error). Tuning the controller refers to finding the most appropriate values for constants K_i and K_p. Let the controller's transfer function, in the Laplace domain be denoted by $G(s)$, where s is the Laplace transform operator. The Laplace transform of Equation 2.4 is:

$$G(s) = K_p(1 + K_i/s) \tag{2.5}$$

In order to tune the controller, a model of the controlled process must be obtained. In our case, the controlled process is modeled by a dead-time element, and a gain. The dead-time element is one whose output lags behind its input by a certain time T_d. The gain quantifies the amplitude of the change in output resulting from a unit change in input. Process output is the controlled utilization which has a range span of amplitude 1 (since it ranges from 0 to 100%). Process input is the service level determined by the controller, which has a range span of amplitude M. Assuming *linearity*, the process gain is equal to the ratio of the aforementioned range amplitudes, i.e., $1/M$. From a control stability perspective, the control system becomes less stable as the process gain increases. Thus, we design the controller for the worst case (i.e., maximum) process gain. The maximum gain occurs when M is minimum, i.e., for a server containing one tree. The maximum process gain is therefore unity. In addition to the gain, sampling introduces an effective dead-time of half the sampling period, $T_d = T/2$. The Laplace transform of the dead-time element is e^{sT_d}. The transfer function of the process, $P(s)$, is thus given by the Laplace transform:

$$P(s) = 1.e^{sT/2} \tag{2.6}$$

Figure 2.10 depicts the control loop. From this figure, note that the achieved utilization U is given by $U = G(s)P(s)e$, where $e = U_t - U$ is the difference between the target utilization U_t and U. Using simple algebraic manipulation, it can be seen that the transfer function from the desired utilization U_t to the achieved utilization U is given by:

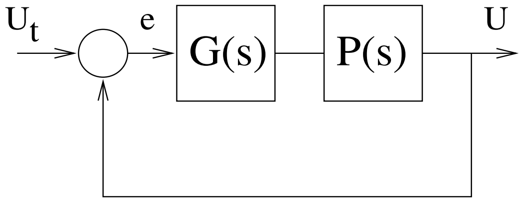


Figure 2.10: The Control Loop

$$U_t/U = P(s)G(s)/(1 + P(s)G(s)) \tag{2.7}$$

Given a change in input U_t, the output U converges to U_t in an oscillatory fashion. The natural frequency of oscillation w of the control loop is obtained by computing the poles of the transfer function in Equation 2.7, with $s = jw$, where j is the imaginary unit vector in the complex number domain. This is done by computing a solution to:

$$1 + P(s)G(s) = 0 \tag{2.8}$$

with $s = jw$. Substituting from Equation 2.5 and Equation 2.6 in Equation 2.8 we get:

$$e^{jwT/2}K_p(1 + K_i/jw) = -1 \tag{2.9}$$

To solve the above complex-number equation, it is transformed into polar coordinates and decomposed into phase and gain components. The phase of the process $e^{jwT/2}$ is $-wT/2$ and the phase of the PI controller $K_p(1 + K_i/jw)$ is $-tan^{-1}(K_i/w)$. It is a common practice in industrial PI controller tuning to set the controller phase to $-\pi/6$ [122]. In order words:

$$tan^{-1}(K_i/w) = \pi/6 \tag{2.10}$$

Thus, the phase component of Equation 2.9 can be written as follows:

$$-wT/2 - \pi/6 = -\pi \quad (2.11)$$

The above expression is an equation in a single unknown, w, which denotes the natural frequency of oscillation for the control loop. The obtained expression for w is:

$$w = 5\pi/3T = 5.24/T\ rad/s \quad (2.12)$$

The parameter K_i of the PI controller can now be computed from Equation 2.10 by substituting in it the computed value of w. Simple algebraic manipulation yields:

$$K_i = 6.38/T \quad (2.13)$$

The second controller parameter, K_p, determines the stability of the control loop. In a stable loop, the controller $G(jw) = K_p(1 + K_i/jw)$ and the process $P(jw) = e^{jwT/2}$ attenuate oscillations causing them to stabilize eventually with output (achieved utilization) equal to input (desired target utilization). For an attenuation gain of γ (in industrial loops $\gamma = 0.5$ is a recommended setting), the following loop gain condition must be satisfied:

$$|e^{jwT/2}|K_p|1 + K_i/jw| = \gamma \quad (2.14)$$

The above condition yields an expression for K_p:

$$K_p = \gamma/(1 + (K_i/w)^2)^{0.5} \quad (2.15)$$

To tune the PI controller, we first used Equation 2.12 to determine the natural frequency of oscillation. Then, constants K_i and K_p were computed from Equation 2.13 and Equation 2.15 respectively with $\gamma = 0.5$.

Testing the Controller

Figure 2.11 depicts the achieved utilization of a server that uses our utilization measurement and control mechanisms. The degree of degradation is managed by the PI controller tuned as described above. In this experiment, the request rate on the server was increased suddenly, at $time = 13$, from zero to a rate that offers a load equivalent to 300% of server capacity. Such a sudden load change is much more difficult to control than small incremental changes, thereby stress-testing the responsiveness of our control loop. The target utilization, U_t, was chosen to be 85%. As shown in

Figure 2.11, the controller was successful in finding the right degree of degradation such that measured server utilization remains successfully around the target for the duration of the experiment. The experiment demonstrates the responsiveness and efficacy of the utilization control loop.

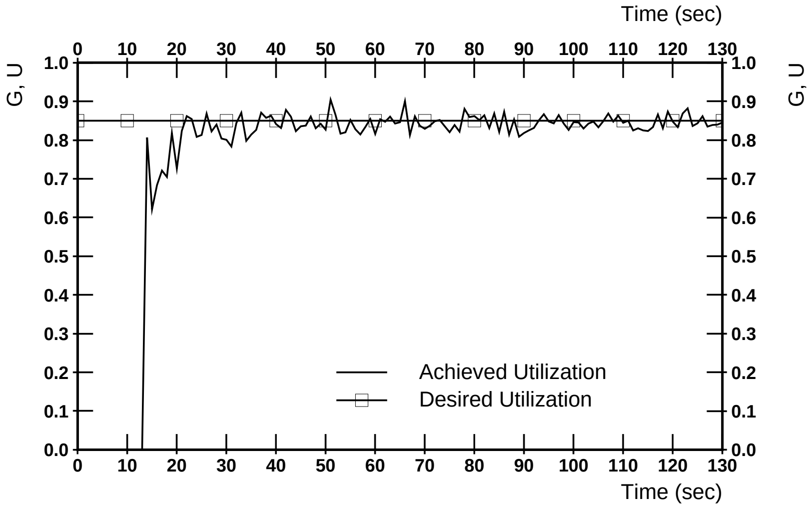


Figure 2.11: Utilization Control Performance

2.5 QoS Extensions for Adaptive Servers

In the preceding sections we described an architecture for content adaptation that avoids overload by degrading content while maintaining a target server utilization. We assumed that all clients are equal and all content was treated the same way. In this section we generalize this architecture to support the following important features:

- *Performance isolation and QoS guarantees*: The web server is extended to export the abstraction of multiple adaptive virtual servers. A different virtual server can be associated, e.g., with each hosted site or user class. The virtual server adapts delivered content such that a configurable maximum request rate and maximum delivered bandwidth are guaranteed independently of the load on other virtual servers thereby achieving performance isolation.

- *Service differentiation*: In addition to achieving performance isolation and QoS guarantees, the web server supports request prioritization. In case of overload, lower priority requests are degraded first. We demonstrate this architecture with two priority classes, although we describe how it can be extended to an arbitrary number of client priorities.

- *Excess-capacity sharing*: When an adaptive virtual server does not consume all its allotted resources, excess capacity is made available to other virtual servers on best effort basis.

2.5.1 Performance Isolation

An adaptive virtual server can be configured for a maximum maintainable request rate R_{max} and a maximum delivered bandwidth BW_{max}. The configuration expresses an agreement whereby the server guarantees the ability to deliver bandwidth BW_{max} as long as the aggregate request rate does not exceed R_{max}. If the request rate condition is violated (i.e., exceeds R_{max}) the bandwidth guarantee is revoked. The server adapts delivered content to deliver the best quality URLs without exceeding its capacity allocation.

The key to achieving performance isolation is capacity planning, load classification, and utilization control as discussed below:

- *Capacity planning:* The maximum maintainable request rate R_{max_i} and the maximum delivered bandwidth BW_{max_i} specification of each virtual server i are converted into a corresponding target capacity allocation, $U_{t_i} = aR_{max_i} + bBW_{max_i}$. The target utilization sum $\sum_i U_{t_i}$ over all virtual servers residing on the same machine should be less than 100% for the guarantees to be realizable. This is checked each time a new virtual server is created. If $\sum_i U_{t_i} > 100\%$ a capacity planning error is returned.

- *Load classification:* A load classifier intercepts input requests and classifies them to identify the virtual server responsible for serving each request. Request classification can be done based on the requested content, addressed site, or client identity depending on system administrator's policy. For example, if each virtual server is associated with a hosted site, requests can be classified based on the site name embedded in the request header. In the current version we implement classification by sender IP, although it is straightforward to add other classification policies.

- *Utilization control:* Once requests are classified, the request rate R_i and delivered bandwidth BW_i for each virtual server i are computed, from which a corresponding utilization value, $U_i = aR_i + bBW_i$, is obtained. The utilization U_i of each virtual server is controlled individually by an instance of the utilization control loop described in Section 2.4.3. The control loop implements the degree of content degradation G_i necessary to keep U_i of the virtual server at its target value, U_{t_i}, thereby achieving the server's individual performance guarantee. The architecture is depicted in Figure 2.12.

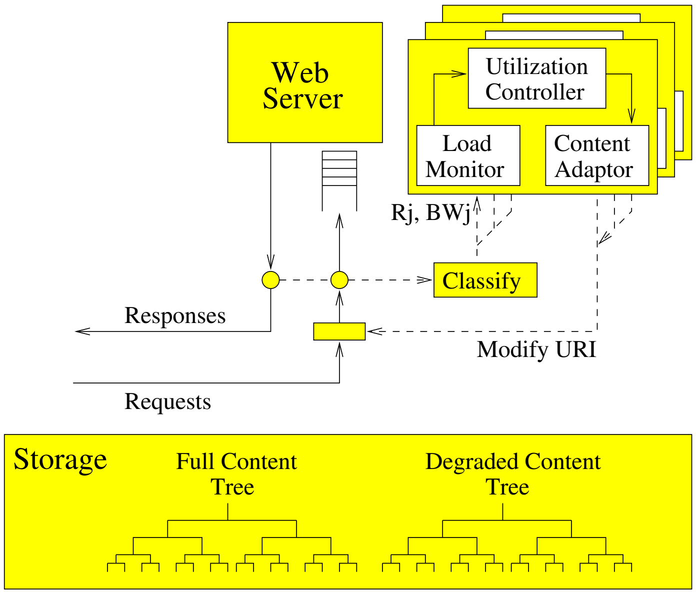


Figure 2.12: Architecture for Performance Isolation

2.5.2 Service Differentiation

In this section we describe how to incorporate service differentiation into our architecture for adaptive content delivery. The goal is to support client prioritization such that lower priority clients are degraded first. Consider a virtual server that supports client prioritization. Let there be m priority classes defined within that server, such that priority 1 is highest, and priority m is lowest. Collectively, clients of the virtual server are allocated a target utilization U_t that may have been derived from a maximum rate and maximum bandwidth specification for that server. This capacity should be made available to clients in priority order. The following extension of utilization control achieves this goal:

- For each priority class j, let the target utilization be $U_{t_j} = U_t - \sum_{i<j} U_i$, where $U_i = aR_i + bBW_i$ is the current measured utilization of the (higher) priority class i.

- Compute the extent of degradation G_j individually for each priority class using the integral controller equation:[5] $G_j = G_j + k(U_{t_j} - U_j)$.

[5]PI and PID control may also be used

- Degrade each class in accordance with the value of G_j as described in Section 2.4.3.

The above scheme allocates the entire virtual server capacity to the highest priority class. The unused capacity of each class is then allocated to lower priority classes. Clients within each priority class are served in accordance with the current capacity, U_{t_j}, allocated for their class. If this capacity is not enough, the clients will be degraded or rejected by the utilization control loop to keep their utilization at the target, U_{t_j}. In practice, client rejection consumes a finite amount of time. Thus, in the presence of low priority traffic, a higher priority class will never achieve its full capacity allocation, U_{t_j}. This can be accounted for in the computation of U_{t_j} as follows:

$U_{t_j} = U_t - \sum_{i<j} U_i - \sum_{l>j} U_{reject_l}$.

where $U_{reject_l} = a_{reject} R_l$ is the overhead of rejecting all current requests of a lower priority class l, the overhead of rejecting a single request being a_{reject}.

2.5.3 Best-Effort Traffic and Excess-Capacity Sharing

An important advantage of grouping several virtual servers on the same machine is the ability to share extra server capacity. Consider two physically separated servers, each of capacity C. If load on one exceeds capacity while the other is underutilized, there is no way to reroute extra traffic to the idle server (unless a gateway is used in front of the server farm to balance load). Idling resources may be wasted on one server while requests are being rejected on another. A single server of capacity $2C$ does not suffer this problem. We therefore extend the preceding mechanisms to allow virtual servers to exceed their contracted target utilization, U_t, as long as there is extra capacity on the machine. Since the virtual server has no contractual obligation to provide the extra capacity in the first place, extra request traffic for any virtual server is uniformly treated on a best-effort basis as non-guaranteed. Non-guaranteed traffic is allowed to occupy the excess capacity on the machine using a mechanism similar to that of service differentiation described in the previous section. Specifically, the degradation level G_n of non-guaranteed traffic is computed from $G_n = G_n + k(100 - U)$, where $U = aR + bBW$ is the current aggregate utilization of the computed from the aggregate request rate and bandwidth. A request for a given virtual server i will be served in accordance with the higher of QoS level G_i for that virtual server or G_n for non-guaranteed traffic. When the individual virtual server is overloaded but the machine as a whole is not, $G_n > G_i$. Consequently, incoming requests are served content with quality determined by G_n which is higher than that warranted by G_i, thus utilizing excess machine capacity. On the other hand, if the machine is overloaded, $G_n < G_i$. Consequently, the quality of content delivered by the virtual server i is determined by G_i. Thus, the individual virtual server is policed not to exceed its capacity allocation. The mechanism allows smooth and intelligent switching between a mode of operation where an individual virtual server i is allowed to exceed its capacity allocation ($G_n > G_i$), and a mode of operation where it is policed to capacity ($G_n < G_i$). This switching is explored in more detail in the evaluation section.

2.5.4 Profiling and QoS Guarantees

Earlier we demonstrated the need to compute parameters a and b that relate server request rate, R, and delivered bandwidth, BW, to consumed capacity, U, where $U = aR + bBW$. A simple way of computing these parameters is to obtain multiple measurements of U and the corresponding R and BW, then apply linear regression techniques to determine a and b that best fit the above equation. Note that R and BW can be measured online by counting requests seen and bytes delivered within a given period. Utilization, U, can be measured in an experimental setting using the gap estimation method described earlier. Given the measured values of R, BW, and U, at successive time intervals, estimation theory provides a way to find a linear fit that minimizes the error. It provides the necessary formulae for computing and updating parameters a and b on-line in view of successive new measurements. An on-line version of the estimator may be used to compute a and b during server operation by sampling periodically the current U, R, and BW then updating parameter estimates.

Alternatively, a and b can be determined by testing the server with a pre-specified workload. Testing can proceed by requesting a URL of a given size at an increasing rate until client connections start timing out indicating server overload. The maximum attained request rate and bandwidth are then recorded. For our purposes, server utilization can be assumed to be 100% at this load condition. The experiment is repeated for different sizes of the requested URL, giving a different rate and bandwidth combination that saturates the server. The resulting set of R, BW, and $U = 100\%$ points is used to construct the line $aR + bBW = 100$ on an R, BW plane. The line intersects the R and BW axes at $100/a$ and $100/b$ respectively, from which a and b are found.

2.6 Implementation

The adaptation software was implemented in C for a UNIX platform. The software was tested on an HP PA-RISK 2.0 workstation running HP-UX 10.20. For the purpose of this experiment an Apache

1.3.0 web server was used. In this section we give more details on software implementation, the testing environment and evaluation of adaptation software.

2.6.1 Web Server Model

In order to handle a large number of clients concurrently, web servers adopt either a multithreaded or a multi-process model. Multithreaded web servers require kernel thread support. Such support is provided in most modern operating systems, e.g., Solaris, Linux, and Windows NT. A separate kernel thread is assigned by the server to each incoming HTTP request. Threads can share common state in global memory. In a multi-process model, common to older UNIX implementations, a separate process is assigned to each incoming request. Since spawning a process is a heavy-weight operation, a pool of processes is usually created at server startup. Created processes listen on a common web server socket, and may communicate via shared memory. A process that accepts a connection handles it until it is closed. Apache 1.3.0, used in our experiments, subscribes to this model.

The adaptation software is designed as a middleware layer between the web server and the underlying operating system. The middleware API may be called directly from the web server if desired, in which case it is not transparent. Alternatively, middleware calls may be made from the socket library used by the server, in which case server code remains unmodified. We begin by describing the API of our adaptation middleware.

2.6.2 Adaptation Software API

Adaptation mechanisms described in this chapter require three entry points. Namely, (i) an initialization point, (ii) a request pre-processing point, and (iii) a request post-processing point. The first point is called once upon server startup. The latter two are called upon receipt of each request and sending of each reply, respectively. The specific calls are described as follows.

- **adaptsoft_init ()**

  This function should be called from the main server process that would later fork off worker processes to handle incoming requests. The call should be made before the aforementioned forking takes place so that child processes inherit the initial data values set by adaptsoft_init(). The function will initialize some global variables (e.g., a shared memory pointer) to be inherited and used by the worker processes. It will also fork off two processes of its own. The *response time monitor* which will monitor server response time by sending periodic HTTP requests, and the *utilization controller* which will implement server utilization control loops. Later in this section we will describe these processes in more detail.

- **adaptsoft_adapt (URI, IP)**

  This function should be called each time an HTTP request is read off from the server's listen queue. It takes as parameters the requested URI and the client's IP address. It invokes a classification mechanism that returns a URI to be served. The returned URI string will be the input URI prepended by the "right" content tree name chosen among those defined in the middleware configuration file. If the request is to be rejected, the returned URI is NULL. Enough extra space must be available in the input string to augment the URI.

- **adaptsoft_log_size (bytecount)**

  This function should be called after each request is served to tell the middleware the size of the served URL (in bytes). The middleware uses this information to perform monitoring and bandwidth control.

2.6.3 Adaptation Software Implementation

The implementation of our adaptation software is best understood by following the path of a request inside the web server. When a request is first dequeued from the server socket's listen queue by some worker process, W_i, the function *adaptsoft_adapt ()* is called in the context of W_i. This function classifies the request as belonging to virtual server j. In the current implementation, classification is done based on sender's IP address, although it can also be done based on the accessed site name, or client's identity. The function then increments a counter, $r_i[j]$, that accumulates the number of requests for virtual server j seen by worker process W_i. When W_i has finished processing the request, it sends out the response and calls *adaptsoft_log_size()* passing it the number of bytes sent. The function *adaptsoft_log_size()* updates a counter, $b_i[j]$, that accumulates the total bytes sent by process W_i on behalf of virtual server j.

Load Monitoring

Periodically, a call to *adaptsoft_adapt ()* by process W_i also invokes a load analysis function. The function computes on behalf of each virtual server k the request rate $R_i[k] = r_i[k]/t$ that process W_i has seen for virtual server k within the last t time units. It also computes the bandwidth $BW_i[k] =$

$b_i[k]/t$ that process W_i has delivered on behalf of virtual server k within that time interval. Finally, it computes the utilization $U_i[k] = aR_i[k] + bBW_i[k]$ that process W_i consumed on behalf of virtual server k, and stores $U_i[k]$ in shared memory. Upon computing $R_i[k]$, $BW_i[k]$, and $U_i[k]$ for each virtual server k, all counters $r_i[k]$ and $b_i[k]$ are reset to zero in preparation for the next period. The period at which load analysis is carried out need not be fixed. In our implementation, the load analysis function is invoked in each process W_i by the first call to *adaptsoft_adapt ()* that occurs after some minimum interval t_{min} has elapsed since the last invocation of load analysis in W_i. The arrangement implements periodic utilization monitoring.

Utilization Control

The utilization controller is implemented as a separate process forked off by *adaptsoft_init()* during startup. The process executes a loop that wakes up periodically to compute the extent of degradation for each virtual server then sleeps until the next period. Upon waking up, the controller aggregates utilization values $U_i[k]$ to compute the utilization $U_k = \sum_i U_i[k]$ of each virtual server. This utilization is then compared to the desired utilization and the degree of degradation G_k is computed for the virtual server k as described in Section 2.5.

Content Adaptation

The content adaptor is implemented in *adaptsoft_adapt ()*. When *adaptsoft_adapt ()* is invoked upon request arrival, and the request is classified as belonging to some virtual server k, the load adaptor is called to determine which content tree this request is to be served from. The adaptor uses the value of G_k, as described in Section 2.4.3 to determine the content tree. The name of that content tree is then prepended to the URL name passed. If the content tree is determined to be 0, the adaptor, by convention, resets the URL name to NULL, indicating that this request must be rejected.

While in the above discussion we focused on virtual server traffic, one can easily see the scheme works the same way for best-effort traffic as well, the difference being in how G_k is computed by the controller as described in Section 2.5.2. Appendix B shows the actual configuration file used to configure the adaptation software for web server QoS.

2.7 Evaluation

In this section we present a performance evaluation of the developed adaptation software. This software was run in conjunction with an Apache 1.3.0 server on an HP-PA RISK workstation running HP-UX 10.20. To emulate a large number of web clients we used httperf, a testing tool that can generate concurrently a large number of HTTP requests for specified URLs at a specified rate. In order to overload the web server, httperf was run on 4 workstations collectively emulating the community of clients. The workstations were connected to the server via a 100Mb switched Ethernet.

2.7.1 Estimating Service Time

In our first experiment, we profiled the Apache server to determine the time, $T_s(x)$, it takes to serve a URL of size x. Measuring server response time was found not to be indicative of service time $T_s(x)$ because the former includes queuing time, network latency, etc. We therefore measured service time by obtaining the inverse of the maximum throughput. The idea is that if the server can serve no more than n requests per second, then, for all practical purposes, each request takes $1/n$ to serve. The experiment was repeated for different sizes of the requested URL. Table 2.1 shows the maximum throughput and the corresponding service time for each URL size.

URL Size (KB)	Max Request Rate (reqs/s)	T_s ms/req
1	586	1.706
2	578	1.73
4	538	1.858
8	482	2.075
16	383	2.611
32	301	3.322
64	169	5.917
128	85	11.76
256	42	23.81
512	21	47.62

Table 2.1: Service Time vs. Request Size

Table 2.2 compares the measured service times to service times computed using the linear approximation $T_s(x) = A + Bx$ where x is URL size, $A = 1.604$, $B = 0.063$. The constant A can be thought of as the time it takes to serve a zero-size URL. The constant B is the additional service time required per KB of URL size. It can be seen that the quality of this approximation is very good for smaller URL sizes, but deteriorates significantly as URL size increases. The reason is that the service time computed from the linear expression approximates the *end-system's* service time. When the retrieved URLs are small the maximum request rate is determined by the end-system's bandwidth (including both CPU and disk access) making the approximation accurate. As URL size increases, the bottleneck shifts from the end-system to the network. Since the end-system is no longer the bottleneck, the estimated service time falls below the observed service time dominated by that of the bottleneck resource.

URL Size (KB)	Measured T_s	Computes T_s	Error
1	1.706	1.677	-1.7%
2	1.73	1.73	0%
4	1.858	1.856	-0.1%
8	2.075	2.108	1.6%
16	2.611	2.612	0%
32	3.322	3.62	8.9%
64	5.917	5.636	-4.7%
128	11.76	9.668	-17.8%
256	23.81	17.73	-25.5%
512	47.62	33.86	-28.9%

Table 2.2: Approximating Service Time

In order to model service time more accurately we use a composition of two linear approximations, one estimates service time if the end-system is the bottleneck and the other estimates service time if network bandwidth is the bottleneck. While the former is given as before by $T_s(x) = 1.604 + 0.063x$, Table 2.2 suggests that the latter be given by $T_s(x) = 0.093x$, which is equivalent to stating that the network saturates at a transfer rate of approximately $86Mb/s$. We then take the larger of the two service times to account for the bottleneck resource. Thus, the combined expression for T_s is:

$$T_s(x) = \max\{1.604 + 0.063x, 0.093x\}$$

The accuracy of the above approximation is shown in Table 2.3. We can see that the approximation is accurate over most of the range of URL sizes. The larger error at size 32K is due to particulars of the OS implementation. It appears that HP-UX is optimized for long TCP transfers, making CPU service time increase sublinearly with transfer size thus falling below the linear estimate.

URL Size (KB)	Measured T_s	Computed T_s	Error
1	1.706	1.667	-2.3%
2	1.730	1.730	0%
4	1.858	1.856	-0.1%
8	2.075	2.108	1.6%
16	2.611	2.612	0%
32	3.322	3.62	8.9%
64	5.917	5.952	0.6%
128	11.76	11.90	1.2%
256	23.81	23.81	0%
512	47.62	47.62	0%

Table 2.3: Approximating Service Time

The total service time T_N of N requests is $\sum_{1 \leq i \leq N} T_{s_i}(x_i)$, where x_i is the requested URL size in the ith request, and $T_{s_i}(x_i)$ is the service time of that request. Substituting for $T_{s_i}(x_i)$ we get:

$T_N = \max\{1.604N + 0.063 \sum_{1 \leq i \leq N} x_i, 0.093 \sum_{1 \leq i \leq N} x_i\}$

where $\sum_{1 \leq i \leq N} x_i$ is the total bytes requested. Let us denote it by S. Thus, $T_N = \max\{1.604N + 0.063S, 0.093S\}$. If N requests were served by the server within some time interval T, system utilization is $U = T_N/T = \max\{1.604N/T + 0.063S/T, 0.093S/T\}$. Note in this expression that N/T is the observed request rate R, and S/T is the delivered bandwidth BW. Thus:

$$U = \max\{1.604R + 0.063BW, 0.093BW\} \tag{2.16}$$

In practice, requests for URLs above 64KB will constitute only a small fraction of all requests on the server. Thus, it is probably safe to assume that the first term will usually dominate in the above expression. This reduces it to the linear approximation $U = aR + bBW$ we described earlier in this chapter, where $a = 1.604$ and $b = 0.063$. When using the automated profiling feature, constants a and b were found to be 1.4 and 0.05, respectively. We attribute this difference to the way automated profiling computes utilization. During automated profiling, utilization is computed via the gap estimation method. Connection errors begin to occur when the utilization computed by the gap estimation method is only 80% and not 100% as hypothesized in the manual profiling analysis. Constants a and b computed by automated profiling are subsequently lower by 20%. Thus, if automated profiling is used, the server should be configured for maximum capacity of only 80% instead of %100. This does not mean that the server will be less efficient. It merely means that a, b and U values are uniformly scaled by a factor of 80/100.

Note that in Equation (2.16), U is expressed on a scale from 0 to 1, R is expressed in req/ms and BW is expressed in ms/kB. It is more natural to expressed R in $reqs/s$, and BW in Mb/s. After the appropriate conversion of units (in particular note the conversion from kilobytes to Megabits), we get the more natural expression:

$$U = 0.001604\, R(reqs/s) + 0.007875\, BW(Mb/s) \tag{2.17}$$

The a and b parameters are robust to changes in workload (e.g., changes in request rate and requested URL size). However, since they represent, in part, the computational overhead of TCP/IP connections, these parameters might change depending on the average number of retransmissions and the number of segments required to send a given amount of bytes. Thus, for example, the a and b parameters might be smaller for clients accessing the server locally via a high bandwidth LAN and larger for clients accessing the server across a congested or lossy wide area network. In the preceding experiments clients were accessing the server via a LAN. We have not experimented with server access over a wide area network to estimate parameter robustness under these conditions. We expect, however, that a and b will remain stable enough in the face of gradual client population changes for the automated profiling to update them in a timely and accurate manner.

2.7.2 Measuring Response Time

In our experiments, we found that Apache server response time when measured across a fast network (or from a client residing on the same machine with the server) has two important properties. First, it is essentially bi-modal. It remains low until the server becomes overloaded, at which time it increases dramatically. Second, the “knee” in response time seen at overload is roughly equal to the product of service time, T_s, and the maximum length of the listen queue configured for the server. For example, Figure 2.13 plots server response time versus request rate when the listen queue was configured for maximum length of 48, 192, and 768, respectively. In this experiment all requests were for URLs of size 64KB. The sudden increase in server response time when the request rate increases beyond 160 reqs/s makes a clear overload indicator. Figure 2.14 plots response time versus request rate when the URL size is changed. In this experiment the listen queue was configured for a maximum length of 48. The requested URL size was 8KB in one experiment, and 64KB in another. As before a clear rise in response time was observed when server capacity was exceeded. The response time remains small until the threshold of overload and is not particularly indicative of server load until the server is overloaded.

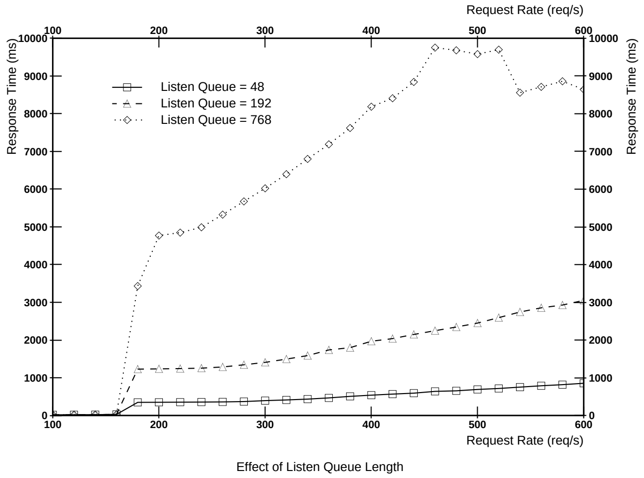


Figure 2.13: Server Response Time for Different Listen Queue Lengths

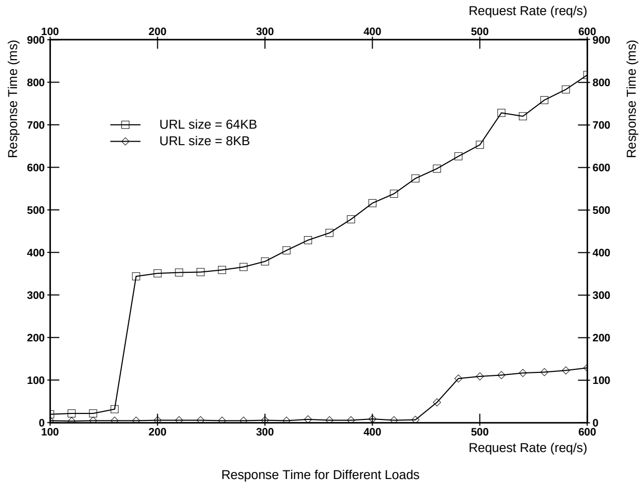


Figure 2.14: Server Response Time for Different URL Sizes

2.7.3 Adaptation at Overload

Content adaptation reduces the load on the server thereby avoiding connection failures. As request rate increases on the server a threshold, $R_{degrade}$, is reached where content has to be degraded in order to prevent overload. As request rate continues to increase beyond $R_{degrade}$, more clients must be degraded until, eventually, a point R_{reject} is reached where no further degradation is possible. If request rate increases beyond R_{reject} some clients must be rejected to prevent indiscriminate connection failures. A server that does not adapt exhibits connection failures starting at rate $R_{degrade}$, while a server that adapts will continue to serve all requests up to the higher rate R_{reject}. In Section 2.4.1 we presented an experiment where the request rate was increased for URLs of size 64KB. An adapted 8K version of the same URL was used for degraded content. In this experiment we found, approximately, that $R_{degrade} = 160$ and $R_{reject} = 460$. The ratio $R_{reject}/R_{degrade}$ is the the maximum sustainable request rate of an adaptive server as compared to the maximum sustainable request rate of a non-adaptive server. The value $R_{reject}/R_{degrade} - 1$ is the net improvement in the maximum sustainable request rate due to adaptation. This improvement depends on the requested URL size. Figure 2.15 plots the net improvement (in percents) versus the average requested URL size, x. The degraded content, in all cases, was 8 times smaller in size than the full-length content, but the required number of server accesses was the same. The percentage improvement in maximum sustainable rate is illustrated both when the accessed URL is a static file of size x, and when it is a CGI script returning a URL of size x. In the latter case a static memory buffer of the specified size was returned by the script with no initialization and no meaningful content. The CGI scripts were written in C. Results for interpreted Perl scripts were slightly lower (not shown in Figure).

It can be seen that the percentage improvement in sustainable rate decreases as the requested URL size decreases. This is because the smaller the requested URL the more dominated is service time by the fixed size-independent processing overhead, rather than the size-dependent data transfer cost. The rate improvement achieved by compressing the URLs is relatively insignificant (less than 100%) for URL sizes below 32K. Thus, content dominated by smaller objects should be degraded by reducing the number of embedded objects per page, rather than reducing the bytes per object. Also note that CGI scripts are not amenable to degradation by reducing the size of generated content. The fixed overhead involved in invoking the script is so great that the additional data-size dependent costs are insignificant unless the returned data volume is substantial. We therefore suggest that

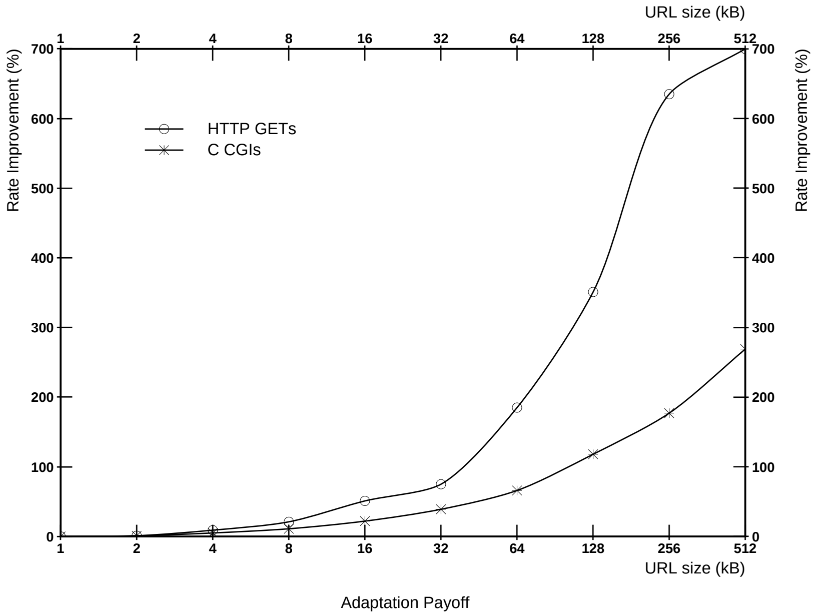


Figure 2.15: Adaptation Payoff: Increase in Maximum Sustainable Rate

dynamic content be degraded by converting it to static whenever possible.

2.7.4 Rejection Overhead

Our adaptive server rejects clients to control utilization when no further degradation is possible. The server can either silently close a client's connection, or return an error message such as "Service not Available". In either case some processing occurs on the end-system before the request is rejected (e.g., protocol processing). To quantify the amount of time spent in processing an eventually rejected request, we instrumented the server to reject all requests by closing the connection as soon as the request is read off the server socket. The request rate on the server was then increased, and the maximum response rate was recorded. The maximum rate was found to be around 900 reqs/s, which is the maximum rate at which rejection can be processed. The time wasted on each rejected request (the inverse of the maximum rejection rate) is thus approximately 1.1 ms/req. This is to be compared with 1.604, the time it takes to serve a zero-size URL (denoted by constant A in Section 2.7.1). The difference is believed to be due to file system access associated with serving the URL. It appears that this difference is not substantial. More than one millisecond of processing time is wasted on each request even if it is rejected. Request classification and rejection should thus be done at the earliest point possible upon request reception in order to conserve end-system's resources. One suitable place for this mechanism is at the bottom of the protocol stack in the operating system's communication subsystem. The difficulty in performing classification at the bottom of the protocol stack lies in the necessity to violate protocol boundaries and peek into headers of higher-level protocols such as HTTP.

It is interesting to compare the aforementioned rejection overhead to the overhead wasted on each failed connection in a server that does not support rejection. Let us denote it by T_f. To compute T_f, consider the top graph in Figure 2.7 which depicts the delivered bandwidth in a regular (non-adaptive) Apache server subjected to an increasing request rate. The maximum delivered bandwidth (of about 84Mb/s) occurs at the overload threshold (at rate 160 reqs/s). Onset of overload indicates that the server is unable to serve successfully more than 160 reqs/s. Substituting in Equation (2.17), the following equation holds:

$$160a + 84b = U_{max} \tag{2.18}$$

where U_{max} is the maximum server utilization at overload. As overload continues to increase, the delivered bandwidth declines to only 36 MB/s at rate of 600 reqs/s. For the worst-case estimate of T_f, assume that the decline in bandwidth is attributed solely to the overhead of handling failed requests. Since the server cannot serve more than 160 requests successfully out of the 600 it receives every second, the number of failed requests is at least 440 reqs/s. The following equation holds:

$$160a + 36b + 440T_f = U_{max} \tag{2.19}$$

By subtracting Equation (2.18) from Equation (2.19), solving for T_f, then substituting for the value of b (as determined in Section 2.7.1, $b = 0.007875s/Mb$), we get $T_f = 0.86ms$ in the worst case. Note that this number is less than the 1.1 ms request rejection overhead.

The implications of the above are interesting. User-level admission control mechanisms trivially require that all requests be seen by the server (so that an admission control decision can be made for each). This implies that each request, whether it ends up being rejected or not, will have to consume platform resources up to the point where it leaves the kernel and is inspected by the server or middleware. As shown above, each request rejected by the server consumes $1.1ms$, on average.

A best-effort server, on the other hand, will serve requests in a FIFO order. As a result, under overload, its socket's listen queue will overflow in the kernel. Many client connections will time out and fail early in the OS before being seen by the server. As shown, a request failed in the kernel consumes only $0.86ms$. As a result, the resources wasted per failed request are less (about 22%

less on our platform). The remaining capacity available to requests that do get through is therefore higher in a best effort server. Thus, while a user-level admission control mechanism will improve the average response time of requests that are not rejected, it will necessarily increase the average rejection rate over the failure rate of a server with no such mechanism. This fact motivates using user-level adaptation instead of rejection as a way to control server overload whenever possible. Content adaptation is especially suited for alleviating light to moderate overload conditions when the server has enough capacity to serve a fraction of, but not all, requests. In cases of severe overload, the server may suffer the receive livelock problem which may preclude serving any requests at all. Methods for resolving the receive livelock problem such as kernel level classification and admission control are beyond the scope of this thesis.

2.7.5 Performance Isolation

We described a performance isolation mechanism that allows creating multiple adaptive virtual servers with individual rate and bandwidth guarantees. The mechanism provides protection among individual virtual servers, as well as protection between the virtual servers and the non-guaranteed best-effort traffic. Figure 2.16 demonstrates these features. In this experiment all requests were for 32KB URLs.[6] A background best-effort load of 300 reqs/s was applied to overload the machine (see Table 2.1 for maximum sustainable rate of 32KB requests). In addition, two adaptive virtual servers, V_1 and V_2, were configured. Server V_1 was configured for a maximum guaranteed bandwidth of 13 Mb/s, and a maximum guaranteed rate of 50 reqs/s. During the experiment, a constant load of 50 reqs/s was applied to that server requiring a bandwidth of 12.8 Mb/s, i.e., just within the allocated server capacity (note that bandwidth in Mb/s is 32KB/req times 8 b/B times 50 reqs/s). Server V_2 was configured for a maximum guaranteed bandwidth of 27 Mb/s, and a maximum guaranteed rate of 100 reqs/s. The load on server V_2 was increased gradually from 0 to 100 reqs/sec, giving rise to a bandwidth requirement of up to 25.6 Mb/s, which is also within server capacity. It is important to note that while each virtual server in isolation was loaded within its individual capacity limit, the aggregate load on the machine (including non-guaranteed traffic) was well above the overload threshold because of best-effort load.. Figure 2.16 depicts the offered load on each virtual server (in terms of bandwidth in Mb/s assuming no content degradation), as well as the actual bandwidth delivered by each server. Both are plotted versus the aggregate request rate. For clarity, the best-effort load is not shown. It can be seen that the actual bandwidth delivered follows closely the

[6] In a real-life situation the workload is likely to be less severe.

offered load on each virtual server. Thus, despite server overload, virtual servers V_1 and V_2 achieve their performance guarantees and suffer no content degradation. Furthermore, variations in load on virtual server V_1 do not affect virtual server V_2. Performance isolation is thus achieved in the sense of maintaining the QoS guarantees independently for each virtual server regardless of other load.

For comparison, we repeated the experiment using a regular Apache server that does not use our adaptation extensions. As before, a best-effort load of 300 reqs/s was applied in addition to a 50 reqs/s load on server V_1 and an increasing 0 to 100 reqs/s load on server V_2. Figure 2.17 depicts the results of this experiment. It can be seen that the delivered bandwidth of both virtual servers falls short of the offered load. The difference reflects the fraction of connections that fail and don't get served due to overload. Note also how the increase in delivered bandwidth of server V_1 results in a decrease in delivered bandwidth of server V_2. No performance isolation is observed. The comparison of Figure 2.16 and Figure 2.17 illustrates the advantage of the developed adaptation software.

2.7.6 Service Differentiation

Adaptation software allows defining multiple priority classes of requests. In this section we experiment with defining two priority classes, namely a basic class B and a premium class P. Requests of class P are treated as higher priority than those of B. In the experiment, we offered a constant load of 100 premium class requests per second. We then gradually increased the rate of basic class requests. Figure 2.18 plots the delivered premium and basic bandwidth versus request rate. It also shows the offered load of both premium and basic clients. Note that when the server becomes overloaded, basic clients are degraded before premium clients thus achieving service differentiation.

2.7.7 Policing vs. Excess-Capacity Sharing

As we argued earlier, an important advantage of colocating several adaptive virtual servers on the same machine is the ability to utilize unused capacity of one virtual server by another that is overloaded. The overloaded server should be allowed to exceed its individual capacity allocation when extra capacity is available, as long as it does not affect other virtual servers. When the machine is overloaded, however, each virtual server should be policed to its individual capacity allocation in order to achieve performance isolation and overload control. These two features are provided by the excess capacity sharing mechanism described in Section 2.5.3. To evaluate the efficacy of this

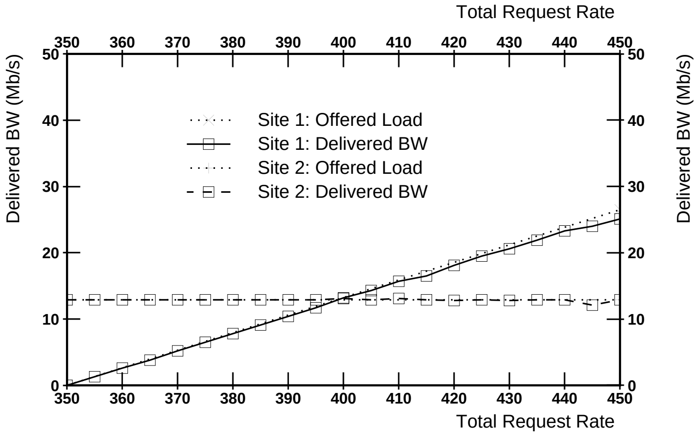


Figure 2.16: Performance Isolation in Adaptive Server

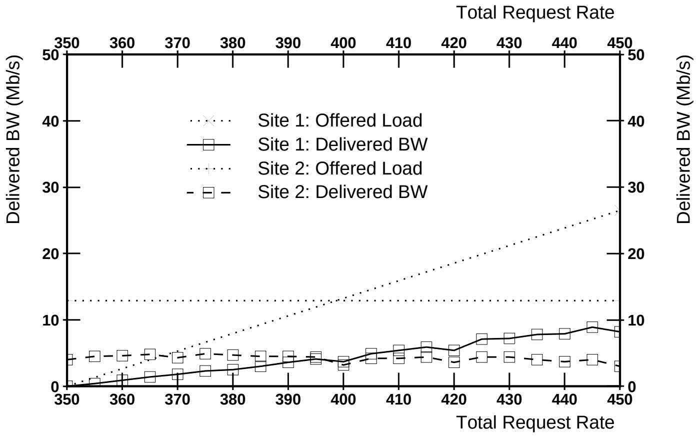


Figure 2.17: Regular Apache Performance

mechanism we conducted two experiments. In the experiments a virtual server V_1 is created whose offered load at run-time exceeds its capacity allocation. Low background load is used in the first experiment. As a result, virtual server V_1 overruns its capacity allocation utilizing the excess capacity on the machine. In the second experiment, high background load is applied. As a result, the virtual server is policed to its individual capacity limit. Moreover, in both experiments a second virtual server, V_2, is also used. Server V_2, which operates within its capacity limit at all times, is shown to deliver its offered load without degradation despite the (controlled) capacity overrun of server V_1, and the background load. Excess-capacity sharing is thus shown not to interfere with performance isolation.

Figure 2.19 depicts the results of the first experiment. It shows the contracted as well as the

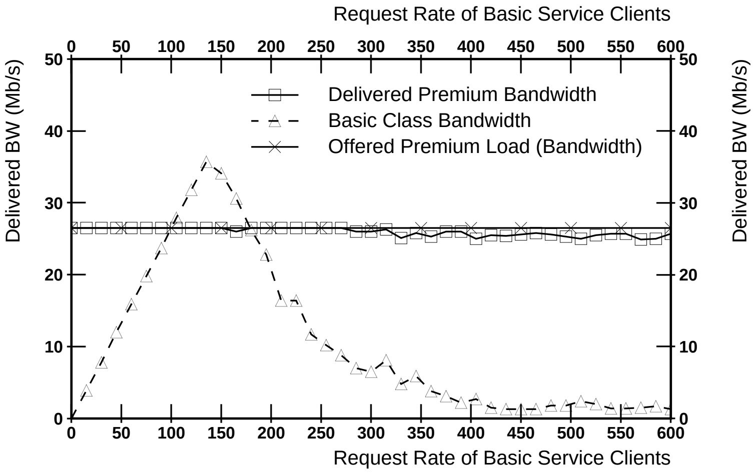


Figure 2.18: Service Differentiation

actual bandwidth of servers V_1 and V_2. Server V_1 is configured for maximum bandwidth of 13Mb/s, and maximum request rate of 100 reqs/s. Server V_2 is configured for maximum bandwidth of 27Mb/s and maximum request rate of 100 reqs/s. At run-time, the request rate of V_2 is held constant at 100, offering a total bandwidth requirement of 25.6Mb/s, i.e., just within its capacity limit. The request rate on server V_1 is increased gradually from 0 to 250 reqs/s. The aggregate rate of both servers is shown on the horizontal axis. It can be seen that server V_2 overruns its capacity allocation delivering a peak of about 35Mb/s at a rate of 140 reqs/s (at which the aggregate rate is 240 reqs/s in Figure 2.19). This is to be compared with its guaranteed maximum bandwidth of 27Mb/s and maximum request rate of 100 reqs/s. Server V_1 remains unaffected, since the excess capacity sharing mechanism ensures performance isolation.

The experiment is repeated with a background load of 100 reqs/s. It can be seen that V_1 is made to deliver exactly its maximum guaranteed bandwidth (27Mb/s) when its rate reaches the maximum guaranteed rate (100reqs/s). This is equivalent to traffic policing, except that in adaptive virtual servers it is achieved via content degradation. The bandwidth consumed by V_1 drops below its guarantees value when the maximum rate guarantee is violated by the community of clients. This is to ensure that the total system capacity utilization of that virtual server remains constant. Similarly, the server is allowed to deliver more than its maximum guaranteed bandwidth when its rate is below the maximum guaranteed rate. This is an optimization that makes use of the capacity allocated to the server to deliver more bandwidth when the request rate hasn't reached its maximum value. Again, server V_2 is not affected due to correct performance isolation.

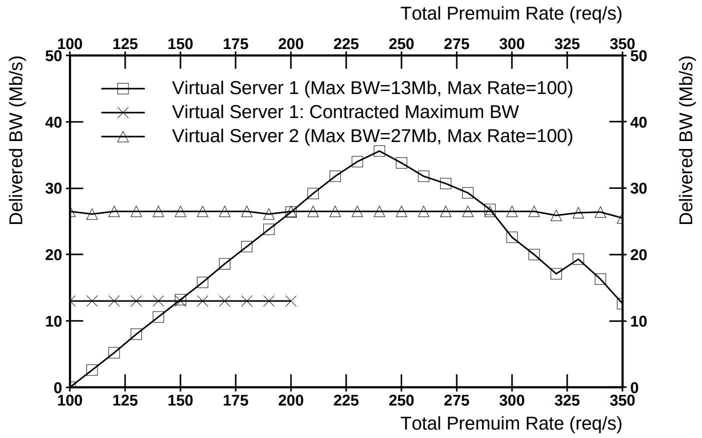


Figure 2.19: Excess Capacity Sharing (Low Background Load)

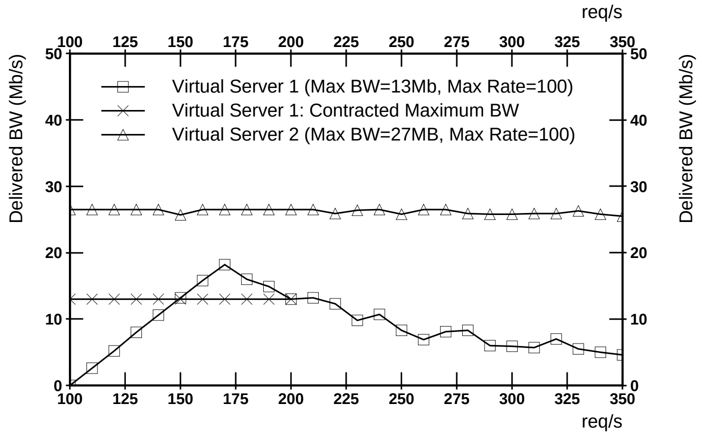


Figure 2.20: Excess Capacity Sharing (High Background Load)

2.8 Conclusions

In this chapter we presented motivational material for building Adaptware. We focused on an important application and illustrated the need for adaptation technology in the context of that application. We built an adaptation software prototype and evaluated its resulting performance. Unlike present day non-adaptive servers, and unlike servers that implement "binary" admission control, we demonstrated an adaptation mechanism that enables a server to cope with overload in a graceful manner. We described the design and implementation of a utilization control loop that adapts service performance in a way that enforces a logical target resource allocation in the presence of variable server load while virtually eliminating connection errors. We proposed several extensions to this mechanism that provide performance isolation, service differentiation, sharing excess capacity, and QoS guarantees. We demonstrated that the architecture can be implemented in a middleware layer transparently to existing server and browser code thereby facilitating deployment.

To generalize adaptation software and separate it from specific application requirements, we extract from this case study certain useful abstractions and mechanisms. The first important abstraction is that of QoS levels. Different content trees represent different levels of quality of service exported to the client. Adaptation software switches QoS levels depending on load conditions.

The second abstraction is that of a QoS contract. For example, authors of a hosted web site may require from their hosting service that it allocates enough resources for their site to meet a hit rate of R for an average URL size of x. The service provider can interpret that specification as a request rate, R, and a delivered bandwidth, $BW = xR$. A utilization budget $U = aR + bBW$ can then be allocated for the hosted site to meet the contracted specification. In general, a QoS contract can have multiple QoS levels represeting a range of acceptable conditions, each is mapped into corresponding resource requirements to be allocated to the contract.

Guaranteeing the contracted performance entails enforcing the particular resource capacity allocation to the contract. For example, each hosted site must be guaranteed its chunk of server resources. In the described architecture, enforcement of capacity allocation is essentially achieved by traffic policing using the utilization control loop. The loop ensures that the contract consumes no more than the capacity allocated to it. The efficacy of this approach has been experimentally verified in Figure 6.8. Alternatively, policing could have been performed on outgoing traffic to regulate server load. This mechanism may be more appropriate for multimedia servers where outgoing flows have considerable duration and bandwidth. The advantage of using policing mechanisms for enforcing resource allocation is that implementing such mechanisms does not require kernel modifications for enforcing capacity allocation. QoS management mechanisms that avoid kernel extensions are preferred because of their relative ease of development and maintainability.

The goal of adaptation is to maximize service utility. In this application, utility was trivially maximized by the utilization control loop by virtue of serving the highest QoS level to the largest possible number of clients without overloading the server. In a more complex application, where some clients are more important than others, more complex utility optimizing resource allocation algorithms are needed to perform logical resource allocation that consider individual client importance and resource requirements. In the remaining chapters we generalize the abstraction of QoS levels and QoS contracts, define a formal notion of utility, develop utility optimizing resource allocation algorithms, and demonstrate enforcement mechanisms of resource allocation that police QoS contracts.

CHAPTER 3

ADAPTWARE AND UTILITY OPTIMIZATION

3.1 Introduction

In this chapter we generalize the architecture developed in Chapter 2 and introduce the basic elements of *Adaptware*; an architecture for adaptive QoS management on server end-systems. QoS-sensitive resource management on server end-systems is motivated by the multitude of emerging Internet applications, such as multimedia streaming and e-commerce, which require predictable performance and contractual performance guarantees. The consequences of failure to satisfy these guarantees range from mild inconvenience (e.g., when potential buyers choose a different shopping site due to failure to download a page) to severe financial loss (e.g., due to failures in on-line trading services). QoS-sensitive applications need a minimum amount of resources in order to operate in a way that is acceptable to the users. With an exponential growth of the Internet user population, popular servers are becoming centralized points of bottleneck. As we demonstrated in the previous chapter, in case of overload, traditional "fair" distribution of resources among server connections/sessions may degrade applications performance equally and indiscriminately below their minimum required level, thus resulting in low "user-perceived utility" while collectively consuming a significant amount of resources. For example, an overloaded web or e-commerce server may be unable to complete most purchase transactions while consuming significant bandwidth on eventually-abandoned connections. Service quality can be improved by allocating bottleneck resources on the server end-system in a QoS-sensitive manner to minimize unnecessary resource consumption, and dynamically adapt the resource allocation to changing load conditions. One natural place for such QoS extensions is the server operating system or middleware.

Adaptware enables business-critical and QoS-sensitive applications to write "QoS contracts" between the server and clients. We show how widely different applications, such as video-on-demand and guaranteed commercial web-hosting, with different QoS semantics and different load characteristics, can make use of our QoS contracts. Given the contracted QoS specification, we describe in this chapter QoS mapping mechanisms and resource allocation policies that optimize service utility under given load and resource constraints. While the policies compute the desired resource allocation, we need mechanisms to ensure that this allocation is enforced. Description of allocation enforcement mechanisms is deferred to Chapter 4.

The success of QoS-management policies and mechanisms in server end-systems depends not only on their efficacy but also on the ease of "retrofitting" them into the existing best-effort service infrastructure. We pay special attention to reducing the cost of retrofitting legacy software with QoS extensions. As mentioned above, unlike several other efforts, we propose extensions that do not require modifying the OS kernel. Implementing QoS extensions outside the kernel has lower development costs, and results in easily maintainable and upgradable code.

New QoS policies and mechanisms usually require new QoS manipulation APIs. While new APIs may benefit software design practices in the future, they raise concerns regarding their utility for today's legacy best-effort code. To address this issue, we demonstrate how the new APIs can be hidden in transparent middleware by exploiting dynamic shared libraries to provide legacy applications with QoS extensions beneath a regular socket API without modifying application code.

The rest of this chapter is organized as follows. Section 3.2 elaborates on the notion of QoS used in this work. It proposes a flexible form of QoS contracts suitable for emerging QoS-sensitive services. Section 3.3 describes our general architecture for embedding QoS provisioning into best-effort server platforms. Section 3.4 addresses the utility-optimizing resource allocation problem, which is NP-complete, and explores simple polynomial-time resource allocation solutions. The chapter concludes with Section 3.5.

3.2 The QoS Contract

In *Adaptware*, QoS requirements must be specified to the server's communication subsystem. This specification is called a *QoS contract*. Most real-time applications have a certain degree of flexibility in terms of resource requirements. For example, video applications can adapt to bandwidth limitations by image compression and frame rate reduction, while web servers can adapt delivered content. To express the flexibility of adaptive applications, a QoS contract C_i contains the following information:

- A desired QoS level, $L_{desired_i}$, the server will attempt to deliver. The semantics of a QoS level will be discussed later in this section.
- The maximum tolerable service degradation: this degradation is specified by a minimum acceptable QoS level $L_{min_i} \leq L_{desired_i}$ from the server under this contract.
- The utility (or charge rate) $R_i[k]$ for each QoS level L_k in the range [L_{min_i}, $L_{desired_i}$].
- The QoS-violation penalty, V_i, for failing to meet the requirements of the minimum level L_{min_i} of an established contract. It is useful to think of contracts as having an extra QoS level called the *rejection level*, with no resource requirements (no service) and a "reward" of $R_i[k] = -V_i$. It quantifies the penalty of disrupting service to the client (e.g., closing the connection in the middle of transmitting a movie).

Normally, the server should avoid running applications below the QoS threshold L_{min_i}, or above the desired level $L_{desired_i}$. The server, however, may choose to terminate a connection at the cost of paying the QoS violation penalty to release resources for a more important client. While the client's perception of service utility might be a continuous function of QoS between L_{min_i} and $L_{desired_i}$, the server may not be able to deliver arbitrary QoS in that range. Instead, the server may export only a finite number of QoS levels, $1, ..., n$, for example, either low-quality audio, standard audio, or high-fidelity audio. Those QoS levels that fall between L_{min_i} and $L_{desired_i}$ are said to be *acceptable* under contract C_i. Ideally, the server must choose, for each contract C_i, a QoS level k_i among those acceptable under the contract, such that the aggregate utility of delivered QoS, $\sum_i R_i[k_i]$, is maximized for the community of clients. If clients pay for their received QoS, this maximization translates into maximization of the server's total revenue.

We have deliberately avoided in the above discussion any mention of semantics of QoS levels. In applications such as video-on-demand where clients request an online movie transmission, QoS levels may represent frame rates and average frame sizes. The contract is "signed" between the server and the requesting client. The server will attempt to deliver picture quality $L_{desired_i}$, but can degrade quality down to L_{min_i} in case of overload.

In other applications contracts may be defined on connection aggregates. For example, consider commercial web hosting. The server may host several independent web sites. A contract C_i is signed with each site owner that guarantees a portion of dedicated server capacity to the site. The guarantee controls aggregate site traffic. Although the QoS levels specified in the contract may in principle have arbitrary semantics, in *Adaptware* we specify a QoS level k of contract C_i with the following two parameters:

- *Aggregate Service Rate, $\mu_i[k]$:* expressed as $M_i[k]$ *units of service* per specified period $P_i[k]$, i.e., $\mu_i[k] = M_i[k]/P_i[k]$. The units of service are arbitrary, but all contracts with a particular server must use the same unit. Examples of service rate are: $M_i[k]$ served URLs per period (e.g., in web servers), $M_i[k]$ served packets per period (e.g., in guaranteed communication), or $M_i[k]$ served frames per period (e.g., in audio/video servers). *Adaptware* localizes the interpretation of service rate to a single plug-in.

- *Aggregate Data Bandwidth, $W_i[k]$:* specifies the aggregate bandwidth in bytes per second to be allocated for the contract. Aggregate bandwidth is orthogonal to service rate, because the unit of service (such as a request, frame or packet) does not necessarily have a fixed number of bytes.

Aggregate service rate and data bandwidth are useful QoS parameters because communication resource consumption, as shown in Chapter 2, can be generally approximated by two components: (i) a fixed average per-unit-of-service consumption (such as per-packet protocol-processing cost), and (ii) a data-size-dependent consumption (such as data copying and transmission cost). This approximation becomes increasingly valid with increased levels of traffic aggregation, as the aggregate approaches the average behavior. Thus, aggregate service rate and data bandwidth are a very useful and scalable way of describing resource consumption in large servers. The accuracy of this approximation has already been established for web servers [4, 5]. We do not deal with jitter and end-to-end response-time constraints, since their satisfaction depends largely on network support that cannot be guaranteed by the end-system alone. Furthermore, today's Internet application technology adapts successfully to delay variations with adequate client-side buffering, making the perceived QoS more a function of connection rate and bandwidth.

3.3 Architecture

This section discusses our architecture and deployment model of QoS extensions. We focus on enabling legacy application software to reap QoS benefits without code modification or recompilation.

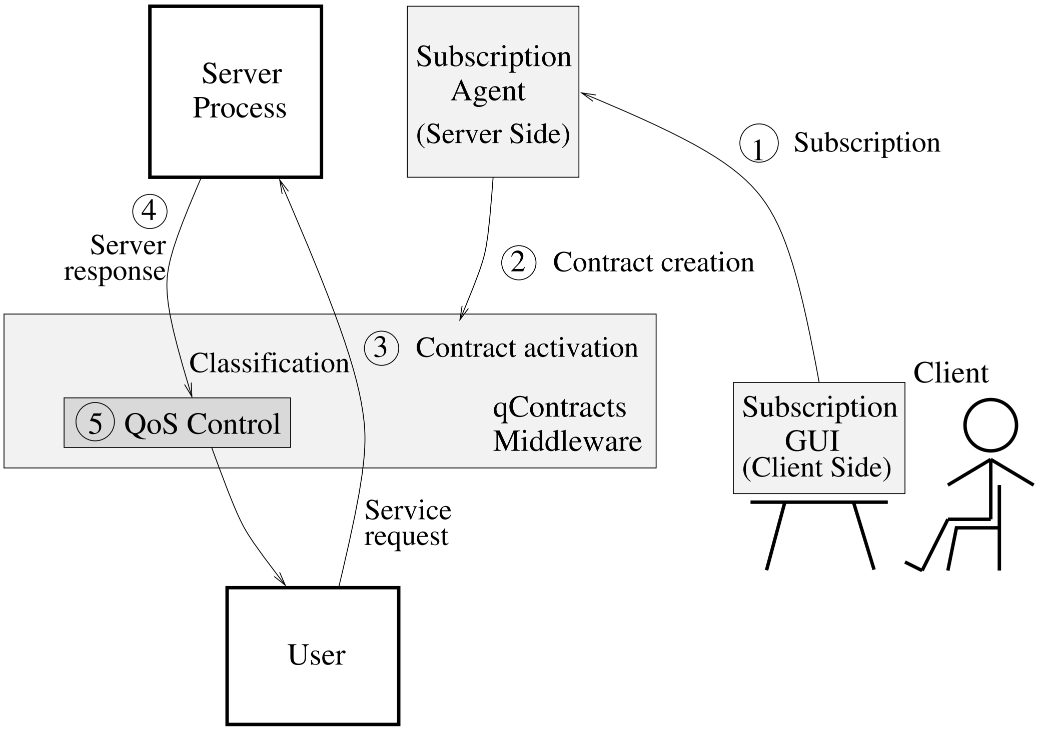


Figure 3.1: Architecture for QoS

Figure 3.1 gives a high-level architectural view of performance-assured services, showing important components and their interactions. The shaded regions are the software components we add to the existing infrastructure to provide QoS-contract guarantees. Unshaded regions represent best-effort legacy code. In our model, clients desiring QoS provisioning will subscribe (via some convenient API such as a web browser) to receive "premium" service. Subscriptions are processed via a subscription agent, which is a process or CGI script separate from the server process, invoked on the server machine. The agent creates QoS contracts with the machine's communication subsystem on behalf of the subscribed clients by calling the QoS-sensitive API extensions exported by the communication subsystem. These clients may be identified to the operating system, for example, by their IP address. The OS then enforces the contract.

3.3.1 Contract Establishment

Contract establishment refers to contract creation, activation, and admission control. We make a logical distinction between contract creation and contract activation. Video-on-demand clients, for example, may want to subscribe to the service (as they would to a premium cable channel), in order to receive contracted QoS whenever they connect to the service. In such a case, the contract is created once, and stored by the subscription agent with the option to be activated internally by the communication subsystem whenever a socket is created with this particular client. Upon contract activation resources should be allocated to serve the contract. Contract activation therefore triggers a utility-optimizing admission-control algorithm within the communication subsystem that determines the amount of resources to be allocated in order to sustain an acceptable QoS level. The contract is admitted if this resource capacity is available. If the contract is not admitted, contract activation fails and the client is served on a best-effort basis (presumably at no charge). A client may request that contract creation and activation be atomic, in which case a created contract is activated immediately. This model is more suitable for services such as web hosting, where the web site owner has the contract created and activated with the server for the lifetime of the hosted web site. We permit contract parameters to be modified after contract activation. This is equivalent to an atomic termination of an old contract and activation of a new one with the modified parameters.

3.3.2 Contract Enforcement

Admitted contracts are enforced by *QoS control* which essentially enforces the appropriate resource allocation. Since we are interested in portable QoS control mechanisms, resource allocation is achieved without OS kernel enforcement mechanisms such as capacity reserves [88]. Thus, resource allocation is *logical*. Our approach is to rely on policing mechanisms to ensure that actual resource consumption coincides with the logical resource allocation. To describe how this is done, it is useful to classify contracts depending on the flows they control. In services with long-lived flows and per-flow contracts (e.g., video transmission), load is controlled most efficiently via flow-control mechanisms that police the outbound server connections to limit resource consumption, so it does not exceed its allocation. In services with short-lived flows and contracts defined on flow aggregates (e.g., web hosting), load is controlled more efficiently by policing the inbound request rate. The scheme uses monitoring and feedback mechanisms to admit only as many requests as necessary to utilize but not exceed the allocated contract resources. Our architecture contains both outgoing flow-control and incoming request rate policing mechanisms. Request rate policing is *not* to be confused with admission of new contracts into the system. While the latter mechanism ensures that the machine is not overloaded as a whole, the former ensures that each individual contract does not use more resources than its allocated share.

The operating system classifies and prioritizes connections transparently to the server. OS calls such as $accept()$, performed on the server's well-known port, return connections to the server in priority order. Rejected connections are closed by the OS, causing no further resource consumption. Non-contracted clients receive a lower priority, and thus are served only when extra resources are available to serve them.[1] Outgoing server connections, such as video flows, are classified by contract within the OS. The OS serves them in a QoS-sensitive fashion, implementing flow shaping and policing mechanisms as appropriate for each QoS contract. Server threads overdrawing their allotted resources are blocked on communication calls such as $write()$, allowing other threads to run. Server code can remain best-effort in nature.

3.3.3 QoS Extensions

In summary, *Adaptware*'s architectural extensions to present service software fall in the following three categories:

- *Contract Establishment and Utility Optimization:* A contract C_i is created and activated for a new client, admission control is invoked, an optimal QoS level is chosen, and resources are logically allocated for C_i.
- *Classification:* Client requests and server responses must be classified in order to "charge" request execution to the corresponding contract's resource budget. While we introduce default classification mechanisms (e.g., classification by IP), we allow applications to perform other types of classification on their own as will be described later.
- *Enforcement and QoS Control:* After classification, request processing must be charged to the corresponding contract's resources such that a contract's resource consumption does not exceed its resource allocation. We will compare an OS implementation of QoS enforcement with a middleware implementation, and discuss the advantages and limitations of each.

In the remainder of this chapter we focus on the utility optimization problem. Namely, in the presence of multiple clients with different importance and different customized contracts how to allocate resources to them such that service utility is maximized. Enforcement is discussed in Chapter 4.

3.4 Utility Optimization

Let each created contract C_i have multiple acceptable QoS levels, such that the resource requirements of QoS level k are given by $U_i[k]$ (which may be a vector), and the utility of delivering this

[1] Hopefully, this policy will encourage subscription.

QoS level is $R_i[k]$ (which may depend on client importance). It is desired to find the resource allocation that achieves the maximum aggregate utility $\sum_i R_i[k]$ given a finite amount of available resource capacity. Since, in our model, QoS-sensitive applications have minimum resource requirements to yield non-zero utility to the end-user, the optimization algorithm must limit the number of clients served concurrently. Otherwise, the per-client share of resources will eventually drop below the required minimum, as load increases on the server, thus reducing service utility to zero.

The nature of the utility-maximization policy depends in large part on user tolerance to QoS violation. Users' tolerance to violation of contract C_i is captured by the contract's QoS-violation penalty V_i. As one may expect, if the QoS-violation penalty is high (e.g., the end-user may be very unhappy if a video-on-demand transmission was terminated abruptly by the server), reservation-based mechanisms achieve higher total utility since the resources allocated to a QoS contract are never taken away. Later in this section we compare a simple FCFS reservation policy to the optimal policy and show near-optimal performance. We also show that if the QoS-violation penalty is negligible, simple priority scheduling will asymptotically approach the optimal utility-maximizing solution as the number of clients increases. We describe how resource requirements of contracts are estimated from QoS level specification (Section 3.4.1), present an optimal resource-allocation policy (Section 3.4.2), compare it with FCFS reservation and prioritization (Section 3.4.3), and derive near-optimal load-sensitive FCFS resource allocation policies.

3.4.1 QoS Mapping

The resource requirements imposed by each QoS contract must be known before utility optimization can be made. In the previous chapter we reported, in the context of web servers, that (i) the time for the end-system to process a unit of service (e.g., packet, frame, or URL) is accurately approximated by $a + bx$ (where a and b are constants, and x is the size of data served), and that (ii) the corresponding consumed network bandwidth is approximated by cx. Parameters a, b, and c are determined by linear regression, and need to be re-evaluated only when the platform is upgraded. Aggregating the capacity consumed by processing a sequence of service units during some observation period, the bottleneck resource utilization required to meet the QoS level, k, of contract C_i, is given by:

$$U_i[k] = max(aM_i[k]/P_i[k] + bW_i[k],\ cW_i[k]) \tag{3.1}$$

where $M_i[k]/P_i[k]$ and $W_i[k]$ are the service rate and bandwidth parameters of the QoS level.

The first term in this expression represents CPU consumption which increases with both the request rate and data bandwidth served. The second represents communication link consumption. The bottleneck resource is the one whose consumption is higher. Thus, from Equation (3.1), if $aM_i[k]/P_i[k] + bW_i[k] > cW_i[k]$ the bottleneck is the CPU. Otherwise it is the link. Using simple algebraic manipulation, the bottleneck resource is identified as follows:

$$\begin{array}{rlll} if & W_i[k]/\mu_i[k] > a/(c-b) & bottleneck: link \\ if & W_i[k]/\mu_i[k] \leq a/(c-b) & bottleneck: CPU \\ where & \mu_i[k] = M_i[k]/P_i[k] & \end{array} \tag{3.2}$$

Different contracts may in theory overload different resources. In practice, however, we optimize for the case where the bottleneck resource does not depend largely on the individual contract. This is justified because the average size, $W_i[k]/\mu_i[k]$, of a unit of service does not vary considerably from one contract to another. For example, the average frame size does not depend considerably on a particular movie, and the average URL size does not depend considerably on the particular web site (although there are always exceptions). Therefore, the values of $W_i[k]/\mu_i[k]$ for all contracts tend to be clustered, and likely to be on the same side of the constant $a/(c-b)$. Substituting this observation in Inequalities. (3.2), we conclude that clients generally overload the same bottleneck resource. This important property justifies using a single utilization value and one-dimensional resource optimization for a multiple-resource end-system.[2] We therefore define the abstract end-system resource utilization as a scalar value such that 0% utilization indicates no load, and 100% utilization indicates saturation of the server's bottleneck resource. Optimal QoS levels within the admitted contracts must be selected to maximize the aggregate utility while keeping end-system utilization below 100%. This is described in the following section.

3.4.2 Optimal Resource Allocation

Suppose there are n QoS contracts handled by the server. Let contract C_i specify m_i acceptable QoS levels with utility $R_i[1], \ldots, R_i[m_i]$, respectively, and let the QoS-violation penalty be V_i. Each QoS level k is now given by the utilization requirement $U_i[k]$ computed as described in Section 3.4.1. We introduce an additional (artificial) QoS level for each contract, called the *rejection level* (at which

[2]This property does not hold for applications that can trade one resource for another, e.g., adaptive on-line video compression.

the client receives no service). This level has no resource requirement (i.e., $U_i[k] = 0$), and incurs a negative utility equal to its QoS-violation penalty (i.e., $R_i[k] = -V_i$) for an established contract, and 0 for a contract being considered for admission.

To reduce this NP-complete problem to a polynomial-time problem, we consider a subclass where $U_i[k]$ can only take discrete values which are multiples of some arbitrary small constant δ. This problem is solvable in polynomial time with dynamic programming. We construct a grid of subproblems, $S(i, U)$, where $S(i, U)$ is the subproblem of optimally assigning QoS levels to contracts $C_1, \ldots, C_i$ without exceeding a utilization U. For notational simplicity, $S(i, U)$ also denotes the resulting aggregate service utility. Given n contracts, we need to solve the problem $S(n, 100)$. Using dynamic programming, we construct the following recursive relation:

$$S(i, U) = \max_{1 \leq k \leq m_i} \{R_i[k] + S(i-1, U - U_i[k]) | U_i[k] \leq U\}. \tag{3.3}$$

For the special case of $i = 1$,

$$S(1, U) = \max_{1 \leq k \leq m_1} \{R_1[k] | U_1[k] \leq U\}. \tag{3.4}$$

Since the utilization is discretized, there is only a finite number, $K = 100/\delta$, of possible utilization values in the range $[0, 100]$. Thus, there are a total of nK subproblems $S(i, U)$ to be solved. The complete algorithm is given below.

Algorithm A

for $i = 1$ **to** n
    for $U = 0$ **to** 100 `in steps of` δ
        `compute` $S(i, U)$ `from Equation 3.3`
return $S(n, 100)$

The returned solution assigns a QoS level for each contract. In the worst case, the solution returned by the above algorithm will specify the rejection level for all contracts. Note from **Algorithm A** that when contract C_n is being admitted (activated), the sets of problems $S(1, *)$, ..., $S(n-1, *)$ will have already been computed at the admission time of contract C_{n-1}. Thus, at the admission time of C_n, we need to compute only the incremental set of problems $S(n, *)$, which is an $O(l_{av})$ computation, where l_{av} is the average number of acceptable QoS levels per contract. Furthermore, of all problems in set $S(n,*)$, only $S(n,100)$ is in the "critical path" of admitting the new contract. The other problems in $S(n,*)$ can be computed later in the background. The incremental utility-optimizing contract admission-control algorithm is thus as follows:

admission_control($contract\ C_n$**)**

1 `compute` $S(n,100)$ `from Equation 3.3`

2 **if** $S(n,100) < S(n-1,100)$ **reject** C_n

3 **else accept** C_n

4 `compute remaining problems in set` $S(n,*)$

We shall use **Algorithm A** and its incremental form as a basis for comparison with simpler QoS-maximizing heuristics to assess the quality of the heuristic solutions. This comparison gives insight on the best OS mechanisms to use for QoS optimization for a particular application.

3.4.3 Suboptimal Policies

We now evaluate the efficacy of FCFS and priority-based policies in finding near-optimal resource allocation solutions. We begin by proving, analytically, the near-optimality of FCFS under non-restrictive assumptions. Assume that the server exports n QoS levels, $L_1, ..., L_n$. Assume that the resource requirements of each QoS level are fixed and determined only by the level itself (e.g., the resources needed for movie transmission depend only on image quality and not the identity of the recipient). Let the QoS level with the highest absolute reward be denoted L_{hi}, and the QoS level with the highest reward per unit of consumed utilization be denoted by L_{lo}. Let the consumed utilization $U_i[k]$ be denoted by h and l for the two QoS levels respectively. Assume that contracts assign a random uniformly-distributed utility $R_i[k]$ to each QoS level, such that the utility of level L_{lo} ranges between Min_{lo} and Max_{lo}, and the utility of level L_{hi} ranges between Min_{hi} and Max_{hi}.

The optimal policy will always keep the clients with the largest ratio of utility to resource consumption. Thus, the optimal policy will achieve, at best, a utility of $max(Max_{hi}/h,\ Max_{lo}/l)$ per unit of resources. The FCFS policy will keep the clients that arrive before the processing capacity saturates. Since utility is uniformly-distributed, FCFS will achieve the expected utility of $(Max_{hi} + Min_{hi})/2h$ per resource unit if it assigns QoS level L_{hi}, and $(Max_{lo} + Min_{lo})/2l$ per resource unit if it assigns QoS level L_{lo}. In general, by assigning L_{hi} to new clients under low load and assigning L_{lo} under high load, FCFS allocation policy can achieve an average utility of $max((Max_{hi}+Min_{hi})/2h,\ (Max_{lo}+Min_{lo})/2l)$ per unit of resource consumption. For the sake of finding a lower bound on achieved utility, the above expression is minimized by setting Min_{hi} and Min_{lo} to zero. In this case the FCFS achieves half of the optimal utility, which constitutes the lower bound. FCFS is thus proven to be a near-optimal policy.

The difference between the optimal policy and FCFS decreases when the QoS-violation penalty is taken into account. QoS violation penalty is never incurred by FCFS since it never reallocates resources assigned to already admitted clients. The optimal policy can increase utility by taking resources away from initially-accepted clients and allocating them to more important ones at the cost of paying the QoS violation penalty. Naturally, the larger the penalty the less beneficial such resource reassignment may be, and the closer the optimal policy becomes to FCFS.

Figure 3.2 depicts simulation results that compare FCFS allocation and fair allocation to an optimal QoS maximizing resource allocation policy. By fair, we mean the prevailing policy in contemporary servers, where each client gets an equal share of resources on the average. All contracts were assumed to have two QoS levels; L_{hi}, which requires 2$\%$ utilization per client, and L_{lo} which requires 1$\%$. Rewards are uniformly distributed in their respective ranges. The figure plots *average normalized utility* defined as the aggregate utility achieved for the community of clients by the given resource allocation policy normalized by that of the optimal policy and averaged over 100 experiments. The average normalized utility is plotted versus server load, expressed in the number of accessing clients. Note that the maximum number of clients supportable at QoS level L_{hi} is 50, and the maximum number supportable at L_{lo} is 100. The server is underutilized when clients < 50, and overloaded when clients > 100. The FCFS policy assigns the highest QoS level, L_{hi}, at low load. At high load it assigns the QoS level with the highest reward per unit of resource consumption. Several curves are shown for the FCFS policy which differ in the QoS violation penalty V of the application, expressed as percentage of maximum reward ($\max R_i[k]$).

The figure shows that FCFS is trivially optimal (by selecting QoS level L_{hi}) when the server is underutilized. As load increases the performance of FCFS drops since assigning L_{hi} may become wasteful of resources. Eventually, as load increases, our FCFS policy switches to assigning L_{lo} to incoming clients thus approaching the optimal policy again. When it becomes impossible to serve all clients, the optimal policy, unlike FCFS, can increase utility further by replacing current less important clients by arriving more important ones assuming the QoS violation penalty is small.

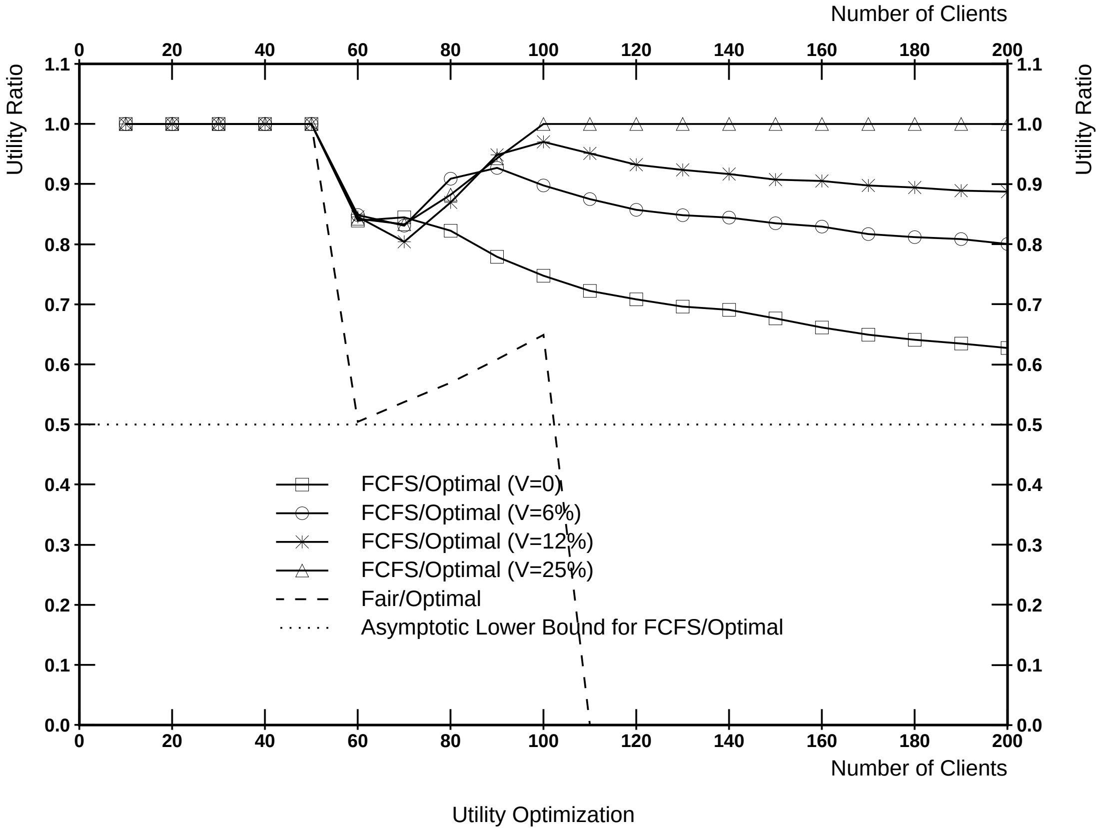


Figure 3.2: Near Optimal Resource Allocation

As more clients access the server the probability of such replacement increases, thus increasing the optimal aggregate utility over that achievable by FCFS. This explains the slight decline in the relative performance of FCFS as the load increases beyond 100 clients in Figure 3.2. It also explains why FCFS is closer to the optimal when the QoS violation penalty is higher. For critical applications (such as e-commerce) where the QoS violation penalty is very high, FCFS becomes optimal for a large range of load conditions.

The figure also shows that fair resource distribution quickly approaches zero utility in a staircase fashion as the machine gets overloaded, thus motivating QoS-sensitive resource allocation. A drop in utility is seen with fair distribution when per-client resource allocation decreases below the minimum requirements of a particular QoS level.

A priority based policy, on the other hand, may serve clients in the order of reward per unit of resource consumption. This is equivalent to the knapsack problem. The utility achieved by placing the most valuable items (clients) in the knapsack first approaches asymptotically the optimal solution as the resource consumption of each item decreases approaching a continuous problem formulation. If all items are the same "size" (same utilization requirements), then placing the the most valuable items first is an optimal utility-maximizing solution. This reasoning assumes no QoS-violation penalty. When the penalty is taken into consideration priority-based resource allocation will achieve less utility since, by neglecting the QoS-violation penalty, the server is likely to pay it more often than it would under an optimal policy. This results in decreasing the utility achieved by prioritization compared to the optimal policy. Since we are interested in applications with high QoS violation penalty we do not consider prioritization any further.

3.5 Conclusions

We proposed a notion of QoS contracts that captures the flexibility in applications' QoS constraints. A QoS contract specifies acceptable QoS levels along with their utility, and a QoS-violation penalty. We generalized the adaptation techniques described in Chapter 2 with a particular focus on solving the general utility optimization problem in applications with elastic QoS requirements specified by QoS contracts. We presented an optimal algorithm for maximizing aggregate user-perceived service utility on server end-systems, and compared it with both reservation- and prioritization-based solutions. We demonstrated that a simple FCFS policy with load-sensitive QoS-level assignment is near-optimal. We presented methods to estimate a contract's resource requirements. We discussed how API extensions for QoS specification and management can be hidden from legacy applications by making the new API calls transparently to the server. In summary, we argued for deploying QoS management on server end-systems, and demonstrated the benefits of such deployment in terms of higher aggregate service utility, as well as practicality of utility optimization.

CHAPTER 4

END-SYSTEM ARCHITECTURE FOR ADAPTIVE QOS

4.1 Introduction

In the preceding chapter we described and evaluated policies for logical resource allocation. To complete the description of *Adaptware* we need mechanisms to ensure that this allocation is enforced. Contracts should be able to access their resource budgets and should be able to consume only those resources logically allocated to them. Such allocation enforcement mechanisms are described below.

We implement two non-intrusive architectures for QoS-contract enforcement that reside in user space. User-space implementations have a significantly lower development cost. For microkernel-based operating systems we investigate a solution for QoS enforcement implemented in a communication subsystem server which is the main focus of this chapter. The communication server runs on the Open Group microkernel Mk7.2, and exports a socket-like API with limited QoS extensions. For monolithic operating systems we describe extensions implemented in middleware. They handle communication resources on behalf of the OS communication subsystem so as to meet the requirements of QoS contracts. The advantage of a middleware implementation is greater portability, which is an important goal of *Adaptware*. The two approaches differ primarily in the way per-contract resource reservation is enforced. In the OS implementation, we control user-level communication resource scheduling. In the middleware implementation, on the other hand, our software has little control on the way communication processing is scheduled. Instead, coarse-grained policing mechanisms are applied to provide a given amount of resources for each contract. We begin by describing the OS implementation.

4.2 Design Overview

The communication subsystem is structured as a multi-threaded communication server with QoS extensions for expressing and enforcing QoS contracts and their resource allocation. In order to achieve per-contract QoS we use a thread-per-contract model. A dedicated thread, called a *contract-handler thread*, is assigned within the communication server to each QoS contract to serve its communication flows. These dedicated contract-handler threads are scheduled according to the contracted QoS parameters. For connections with no QoS contracts, a default handler thread is used. This handler is created at server boot time and assigned a lower priority than that of other handlers. A user-level scheduler, implemented in the communication server, is responsible for sequencing the contract-handler threads, thus decoupling their QoS-specific scheduling policy from the generic fixed-priority scheduling support in the kernel. If the network link is the bottleneck, a complementary mechanism is needed to prioritize traffic transmission on the link. For this purpose, outgoing packets are queued in a common heap at the bottom of the protocol stack implemented within the server. The heap is sorted by handler priority. Packets are dequeued from this heap in priority order when the network device is not busy.

The communication subsystem exports an extended socket API to the application. Applications perform IPCs to invoke the communication primitives supported by the server. The "send" IPC call wakes up an API thread in the communication server that queues up the outgoing message in an input message queue for transmission by the corresponding contract handler. There is one input message queue per QoS contract, including one for the default handler. When a new contract is created, an input message queue is created for it, as well as a dedicated contract-handler thread. The contract handler is signaled when its input message queue becomes non-empty. This results in periodic execution of the handler until the queue is drained. By default all outgoing traffic is handled by the default handler until an explicit $socketBindContract$ call is made. The call binds a socket to a specified (non-default) QoS contract. From then on, all outgoing communication via this socket will be deposited into that contract's message queue and processed by its corresponding contract-handler thread. Essentially, this technique classifies outgoing traffic in accordance with the socket on which it is sent.

A different set of contract handler threads process incoming traffic. If the client has QoS extensions it can tag its request by the identity of a QoS contract. Otherwise, the request is handled by the default handler (for incoming traffic) and is typically classified later by the server which does an $accept()$ on the client and binds the resulting socket to the proper contract. Figure 4.1 presents the communication subsystem architecture illustrating QoS mapping, admission control, QoS-level selection, scheduling, contract-handler threads, traffic classification, message queues, and the outgoing packet heap.

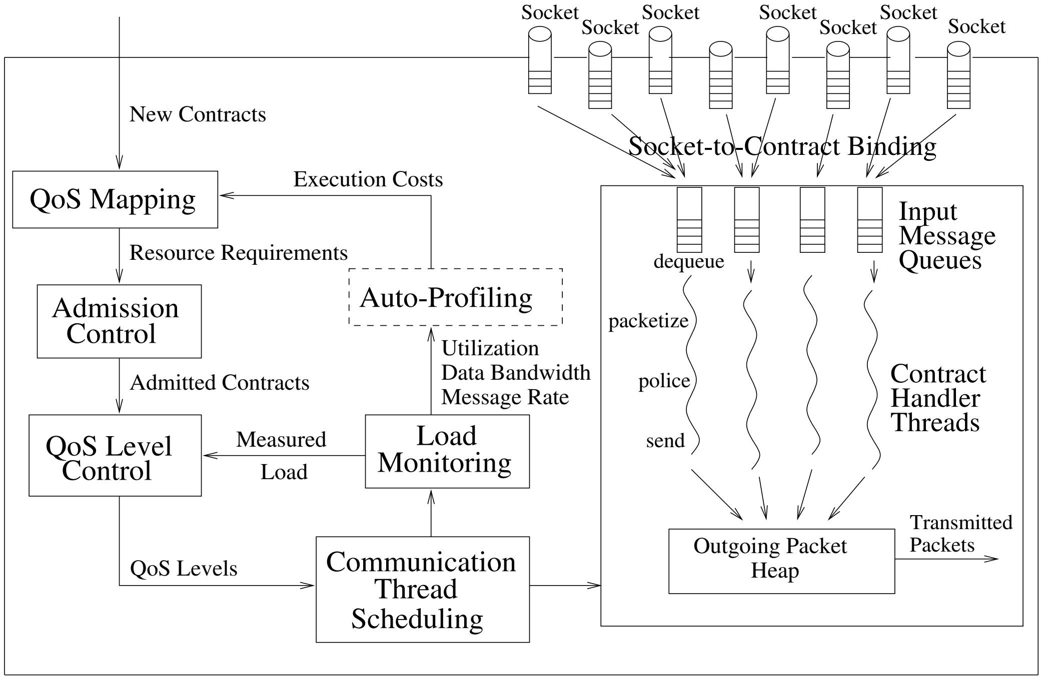


Figure 4.1: Communication subsystem architecture

QoS mapping, admission control, and QoS level selection have already been discussed in the previous chapter. We incorporate these algorithms into the communication subsystem so they are invoked upon establishment of each new contract to assign a QoS level to it and compute the corresponding logical resource requirements. In this chapter, we focus primarily on enforcing QoS contracts and ensuring that each contract gets its logically allocated resources. This is done by means of QoS-sensitive communication thread scheduling, as well as load monitoring and automated profiling to adapt to changes in dynamic load conditions and platform speed. Conceptually, in order to enforce logical resource allocation, the communication subsystem must ensure that the contract-handler thread for contract C_i gets the optimal processing utilization $U_i[k_i^*]$, as computed by the QoS-optimizing algorithms described in Section 3.4.2 and Section 3.4.3. To meet this requirement, the contract-handler thread for C_i is invoked by the user-level scheduler periodically once every $P_i[k_i^*]$ seconds. The scheduler allocates to the thread a budget $E_i[k_i^*] = U_i[k_i^*]P_i[k_i^*]$ during each period. The thread gets blocked when its budget expires. The budget is replenished at the start of the next period at which time the contract handler may be resumed. The arrangement polices outgoing server traffic so it does not exceed its resource allocation. Hence our end-host communication subsystem meets the following goals:

- Provides per-flow or per-service-class guarantees on the end-host. Once a QoS contract is established with a client, it is enforced.

- Maximizes the aggregate reward (for delivered QoS) across all clients. In applications such as video-on-demand servers where clients pay for their received QoS, this maximization translates into maximization of the total server's revenue.

- Adapts responsively to transient load disturbances and resource shortage. This is in sharp contrast to indiscriminate degradation that applies to premium-service clients as well as basic-service clients alike.

We describe in Section 4.3 our implementation platform. In Section 4.4 we refine the definition of QoS-levels to provide both latency and bandwidth guarantees on the end-system, and review the notion of QoS contracts used in our communication subsystem. Section 4.5 describes $q$threads, a user-level scheduling package to enforce contractual obligations. Section 4.6 describes *CLIPS*, a Communication Library for Implementing Performance-assured Services and discusses several implementation issues related to the platform at hand. Evaluation and testing results are presented in Section 4.7. Section 4.8 discusses extensions of the framework to aggregated flow contracts and presents a middleware realization called qContracts. The chapter concludes with Section 4.9.

4.3 The Implementation Platform

We implemented the proposed communication architecture on Pentium-based PCs, running the MK 7.2 microkernel developed by the Open Group (OG).[1] MK 7.2 is a derivative of CMU's Mach. The communication subsystem architecture was implemented using the facilities of OG's CORDS environment which are based on x-kernel support originally developed at the University of Arizona. The advantage of using x-kernel in our platform lies in its uniform protocol interface that allows us to compose arbitrary protocol stacks in a flexible manner. An x-kernel protocol stack can be configured from protocol objects by specifying a protocol graph of the communication subsystem. The specification is then compiled into the desired protocol stack. This flexibility enables

[1] OG is formerly known as the Open Software Foundation.

the separation between the protocol stack and the resource-management mechanism. CORDS adds to the x-kernel framework the notion of per-connection passive resource reservation. It exports the abstraction of *paths* which can be associated with particular contracts. The framework allows charging the consumption of resources such as memory bandwidth and message buffers at different layers of the protocol stack to the same path, which facilitates the implementation of per-contract resource management. It also implements per-path input thread pools to shepherd incoming traffic. The framework allows realizing differential services by assigning different priorities to threads of different paths. We changed CORDS to use threads whose priority is automatically set by our communication subsystem scheduler in accordance with the current QoS-level selection.

While a server-based implementation is natural for a microkernel operating system, it may perform poorly compared to user-level protocol libraries due to excessive data copying and context switching [85, 134]. Implementing the service as a protocol library, however, distributes the functions of admission control and run-time resource management among several address spaces. Since applications may each compete for communication resources, controlling system-wide resources is more effectively done when these functions are localized in a single domain. The CORDS framework provided on MK 7.2 gives an easy way of migrating CORDS server code into the kernel by setting appropriate kernel compile-time flags. Thus, the communication server can be developed more effectively in user space, then migrated into the kernel for efficiency if desired. An additional advantage of using the CORDS framework is its availability on widely-used platforms such as Windows NT. Figure 4.2 shows the end-host architecture, especially the communication server implementing the communication protocol stack. It also shows the interface between the application(s) and the server in our implementation environment.

To enforce QoS guarantees, we implement a user-level thread package whose scheduler transparently executes the near-optimal QoS level selection algorithms described in Sections 3.4.2, and assigns thread scheduling priorities appropriately to satisfy the selected QoS levels. This scheduling package, called *q*threads, is novel in that its threads are explicitly aware of their own QoS levels. For the particular case of a communication subsystem server, a generic contract handler was implemented to perform protocol processing, packetization/depacketization, data buffering, policing, and client aggregation.

We now review the notion of QoS guarantees, generalizing it to specify both rate and latency requirements, then describe and evaluate the implementation of *q*threads, and a communication library, called *CLIPS*, that implements our QoS-sensitive communication-specific functions. To-

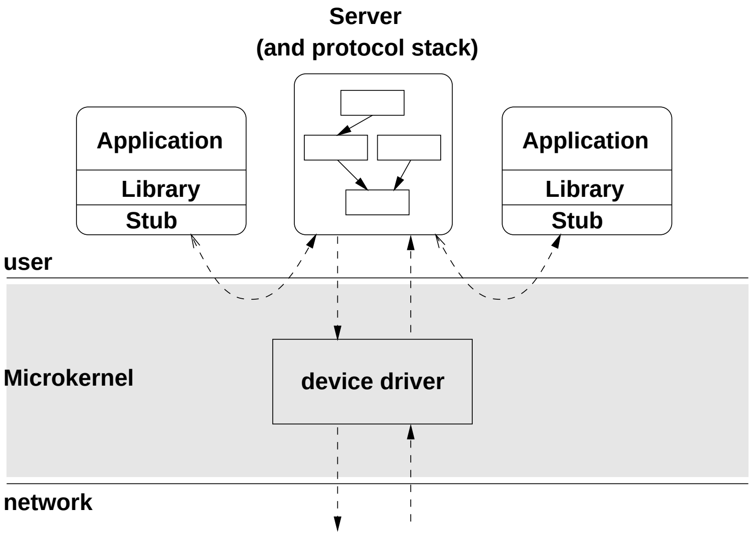


Figure 4.2: Communication architecture

gether, these components off-load the burden of QoS enforcement from designers of real-time services and allow QoS adaptation to platform capacity and load.

4.4 Real-time Communication Guarantees

The generalized semantics of a QoS level are described in Section 4.4.1. For completeness, Section 4.4.2 reviews the notion of QoS-contracts based on expressing multiple acceptable QoS levels. The issue of quantifying the relative desirability (reward) of these QoS levels by the client is elaborated on in Section 4.4.3.

4.4.1 QoS Levels

A QoS level, L_i, specifies a desired bandwidth and latency guarantee for a socket or a set of sockets. In the simplest case, it is expressed by a number M_i of packets, a period P_i, a bandwidth W_i, and a buffer size parameter B_i. The bandwidth guarantee states that the socket will be allocated enough resources so that at least W_i bytes can be transmitted per second assuming that the packet rate is no larger than M_i packets every P_i units of time. The buffer parameter specifies the maximum number of packets to be buffered in the communication subsystem due to the sender's burstiness. Without loss of generality, we can assume that B_i is specified in multiples of M_i. The maximum number $N_{max}(t)$ of packets that can be generated within an interval of length t is $(B_i + \lfloor t/P_i \rfloor)M_i$. Since $\lfloor t/P_i \rfloor M_i$ of these packets will have been transmitted by time t, the expression for $N_{max}(t)$ implies that at most B_iM_i can be awaiting transmission at the sender at any given time. A packet generated at time t at the source is said to be *conformant* if the total number of packets generated by time t is less than $N_{max}(t)$. Packets generated in excess of the above are said to be *non-conformant*. Latency guarantees are given to conformant packets only. Since M_i conformant packets are guaranteed to be transmitted every P_i units of time and since the maximum number of conformant packets awaiting transmission is B_iM_i, a conformant packet will be transmitted within B_iP_i time units after its generation at the source. Thus, if a conformant packet is generated at time t, the latency guarantee states that it will be transmitted by the deadline $D_i = t + B_iP_i$. Non-conformant packets may be dropped or delayed until they become conformant in which case they will be transmitted within B_iP_i time units after their conformance time.

A QoS level for a receiving connection specifies a budget that holds enough computing resources to receive a flow of bandwidth W_i assuming that no more than M_i packets are received every P_i time units. Packets in excess of M_i received within a period P_i are non-conformant. The buffer size parameter B_i specifies the maximum size of the delivery buffer in multiples of M_i. This parameter is useful when the receiver task is not synchronized with the communication service, in which case the service must buffer incoming packets until the application executes a receive. A packet is said to have been *delivered* when it has been processed by the protocol stack at the receiver and deposited in a delivery buffer. The communication service guarantees a conformant data packet to be deposited into the delivery buffer no later than P_i time units after it has been received. The application must read (i.e., drain) the delivery buffer within B_iP_i time units of the packet's reception or else the buffer may overflow.

A single socket may have two contracts associated with it; one for outgoing traffic and one for incoming traffic. The two contracts may have different QoS levels. Note that the QoS level as defined above is an end-host level abstraction, and not an end-to-end connection-level abstraction. It can be used with connectionless protocols such as UDP. In particular, it does not imply that network bandwidth is actually reserved for a connection to satisfy the bandwidth guarantee. The network might not support per-flow bandwidth reservation. We decouple the concerns of end-host resource reservation from those of network resource reservation. The latter concern should be addressed at the network communication protocol layer and in routers, while the former is a concern of end-host resource management covered by our architecture.

4.4.2 The QoS Contract

The QoS contract Q_i specifies alternative *QoS levels*, their corresponding *rewards*, and a *QoS-violation penalty* as defined in Section 3.2. Contracts can be defined separately for outgoing and incoming traffic sharing the same socket. Thus, contracts can be either of "sender" or "receiver" type. Each QoS level is expressed in terms of its M, P, W, and B traffic parameters described in Section 4.4.1. All rewards must have the same units across clients. For example, in one implementation, rewards are specified in a range from 0 to 100%, which is then scaled within the communication server by the importance level of the client. In another implementation, clients are given "credit lines" in accordance with their importance, then allowed to specify reward values that are debited to their capital. The QoS-violation penalty is the cost incurred if none of the requested alternative QoS levels can be provided by the communication subsystem. Technically, we should separate between the QoS-violation penalty and the QoS-*rejection* penalty. The former is incurred if the communication subsystem violates an existing contract by degrading a client below the minimum contracted QoS-level. The latter is incurred if a new request is rejected prior to contract establishment. Depending on the application these penalties can be set by the client or the service provider. For some applications it may be argued that there should be no rejection penalty when no contract has been established yet. Figure 4.3 shows an example connection-establishment request.

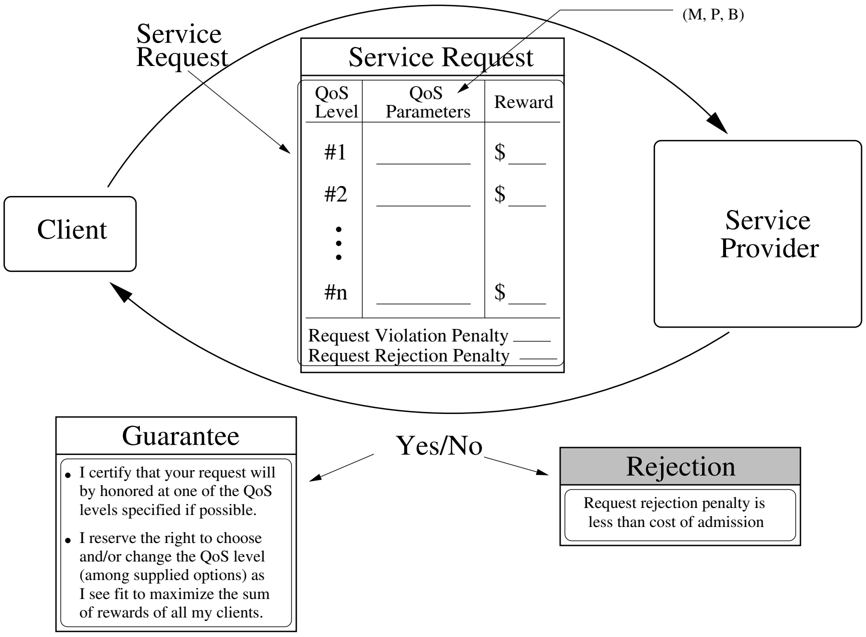


Figure 4.3: Connection establishment request

Rewards and violation penalty are not a redundant expression of the same concept. They quantify different aspects of client preferences. For example, a multimedia user may attach a very high violation penalty to an important lecture while assigning low rewards to the resolution of video transmission during the lecture if he/she is more interested in the audio contents. The same user may attach a relatively low violation penalty on viewing Jurassic Park while assigning sufficiently high rewards to the resolution of the movie's transmission. This tells the system that in case of overload the resolution of the lecture should be degraded first, before that of Jurassic park, but if resources are no longer sufficient to maintain both connections, Jurassic Park should be the one to be terminated. Note, also, that specifying only one acceptable QoS level with a default (e.g., infinite) violation penalty reduces this interface to a traditional on-line admission control API. An infinite violation penalty would mean that QoS-violation results in system failure, as is the case in hard real-time systems. If such is the case, the underlying system should also support resource reservation, and the worst case resource requirements of arriving requests should be known at arrival time. Since our interest is primarily in applications which have flexible QoS requirements we do not make the above assumptions.

Should the QoS contract be "signed" by the server, the client is guaranteed to receive the service at *one* of its requested QoS levels, or be "paid" the specified amount of QoS-violation penalty. The selection of a particular QoS level is delegated to the server (in this case, the communication subsystem). Furthermore, the communication subsystem may change the delivered QoS to another level in the client's QoS contract at its discretion, when appropriate. This gives the subsystem enough leeway for QoS adaptation and the optimization of aggregate reward.

4.4.3 Reward Estimation

The problem of specifying rewards for different QoS levels (as well as the QoS levels themselves) warrants additional discussion. In real-time applications interacting with a human user (rather than a physical environment) QoS levels and rewards are essentially subjective. For example, the parameters of QoS levels may be set by application designers to correspond to "poor", "good" and "excellent" performance, respectively. In a previous chapter we illustrated a web application where different QoS levels correspond to different content trees. The reward information may either be prespecified by the service provider or set by the end users at the time of contract establishment using an intuitive GUI (such as a slide rule which quantifies user satisfaction with the predefined QoS levels). In certain applications, an "expert" user interface may allow users to customize their applications by manipulating the preset definition of “poor”, “good” and “excellent” QoS. This is somewhat similar, for example, to adjusting the brightness and contrast of a regular TV screen, or adjusting the multiple bands of a stereo equalizer for a certain playback quality. In a system deployed for commercial purposes, QoS levels and rewards would typically be selected by the service provider. For example, an e-commerce server can use reward information based on a combination of customized user preferences and user membership fees.

QoS-level and reward specification for applications that deal with an external physical environment, as opposed to a human user, should be done based an objective performance measure. For example, in [3] we illustrated a case where QoS levels and rewards were specified by an AI agent representing an application domain expert for an automated flight application to minimize mission failure probability. In general, the specification of QoS levels and rewards is an application-specific policy that should be discussed in the context of the particular application. Our main goal in *Adaptware* is to provide general mechanisms for such specification that are rich enough to express elastic service requirements, whenever such information is available.

4.5 *q*threads User-level Scheduling

We enforce the contractual guarantees mentioned in Section 4.4 using QoS-sensitive scheduling of contract handler threads within the communication server. For this purpose we built a generic scheduling package, called *q*threads, targeted for future performance-assured services. It exports QoS contracts as a first class abstraction. QoS contracts can be created, deleted, or modified. Threads can be assigned to contracts, in which case they are called contract handler threads. The execution period and budget of threads assigned to a particular contract are selected by the user-level scheduler in accordance with the current QoS level at which the contract is executed. In order to improve the portability of the *q*threads scheduler, we divide it architecturally into two layers. The bottom layer abstracts the machine and operating system into a real-time virtual environment that exports quasi-periodic threads described below. The top layer is machine- and OS-independent. It implements QoS contracts on top of the API exported by the quasi-periodic thread layer. In the following subsections we describe the abstractions and architectural components of the user-level scheduler.

4.5.1 The Quasi-Periodic Thread Abstraction

Informally, a quasi-periodic thread is a thread that executes periodically for a finite duration, starting at an arbitrary point in time. More formally, a quasi-perdiodic thread has a dynamic budget $E(t)$, and a dynamic execution period $P(t)$, where t denotes time, such that:

- $E(0) > 0$.
- The *invocation time* of a quasi-periodic thread is the time, I, at which it is called using $start\_quasiperiodic()$.
- The *finish time* of a quasi-periodic thread is the time F at which it calls $stop\_quasiperiodic()$.
- The *logical arrival times* of a quasi-periodic thread are all instances
  $a_k = T + k \sum_{0 \le i \le k-1} P(a_i) < F$,
  where $k = 0, 1, 2, ...$,
  and T is:
  - the thread's invocation time I, if the budget was non-zero at time I, or
  - the budget replenishment time, if the budget was zero at time I.

  The budget is replenished at $a_k + P(a_k)$.
- The quasi-periodic thread can do at most $E(a_k)$ units of work in any interval $[a_k, a_k + P(a_k)]$, where k is an integer.

Quasi-periodic threads are the contract handler threads in our architecture. Turning periodic execution on and off with $start\_quasiperiodic$ and $stop\_quasiperiodic$ is useful, for example, to avoid repeatedly re-invoking a service thread by the scheduler when there are no requests to serve. In fact, in our implementation, $start\_quasiperiodic$ is invoked by arrival of service requests, while $stop\_quasiperiodic$ is called after the served request queue has been drained. Figure 4.4 depicts an instance of executing a quasi-periodic thread. The period and budget of quasi-periodic threads may in general change with time due to adaptation. We describe how quasi-periodic threads are implemented on operating systems with POSIX-compliant kernel threads and fixed priority scheduling.

4.5.2 Implementing Quasi-Periodic Threads

For each quasi-periodic thread, Q_i, the *q*threads scheduler maintains a thread data structure that associates Q_i with an underlying POSIX-compliant kernel thread T_i (the kernel need not support

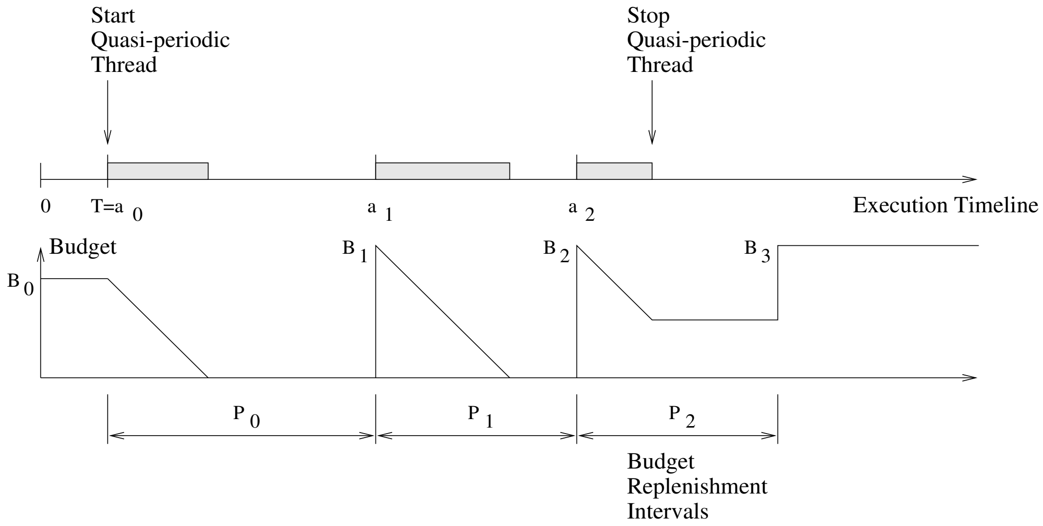


Figure 4.4: Quasi-periodic Execution

periodic thread scheduling), an underlying kernel semaphore S_i, a period P_i, a periodic budget E_i, and an eligibility flag F_i. The budget is replenished every period P_i by a periodic timer event. The eligibility flag determines whether periodic execution of the thread is enabled or disabled. It can be set or reset by the external *q*threads API calls $start\_quasiperiodic$ and $stop\_quasiperiodic$ respectively. Initially the *q*threads scheduler starts out with an empty set of quasi-periodic threads. A kernel thread T_i can register itself at will with the *q*threads scheduler, in which case a quasi-periodic thread data structure Q_i is created for it, including the creation of semaphore S_i. By default, the eligibility flag of the newly created quasi-periodic thread is turned off. Upon registration with the *q*threads scheduler, the thread T_i is blocked on the kernel semaphore S_i. Once the eligibility flag is set, the semaphore S_i will be signaled periodically with period P_i. The thread will be allowed to execute within each period only until its budget, E_i, expires. When the eligibility flag is reset, the thread will no longer be signaled.

An event requesting service from the thread will use *q*threads API to start the specified quasi-periodic thread, which turns on the eligibility flag. If the thread has non-zero budget, it will be put in the ready queue of the *q*threads scheduler immediately. Otherwise, it will be put in the ready queue when the budget is replenished.

The *q*threads scheduler maintains a list of ready threads. At any given time, only one of these threads can be scheduled for execution. This is ensured by keeping all registered threads blocked on their respective semaphores, except one thread whose semaphore is signaled. An arbitrary scheduling policy can be implemented by controlling the sequence in which semaphores are signaled. Context switches occur when threads voluntarily yield the CPU. This implies that threads must periodically execute a yield call. If the calling thread's budget has expired, or if a higher priority thread is waiting, the yield results in a context switch performed by signaling the next thread's semaphore then blocking the yielding thread on its own semaphore. This cooperative preemption model controls the instances at which threads can be preempted, thereby substantially reducing the complexity and execution cost of concurrent applications, (e.g., by reducing the the locking and unlocking of shared data structures). The model implies that we trust the thread programmer. This implication is justified since *q*threads is designed for building servers. We assume that server code can be trusted by its OS.

During intervals when no threads are running, a high priority timer invokes the *q*threads scheduler periodically to check for arrival times of quasi-periodic threads. Eligible quasi-periodic thread identifiers reside in a heap sorted by their future arrival times. The timer tick dequeues all entries whose arrival time has come and enqueues corresponding entries for the respective subsequent invocations. Dequeued entries are then put in the *q*threads scheduler ready queue which is a heap sorted by thread priority. Arbitrary priorities can be assigned to quasi-periodic threads. Two policies are of particular interest. These are, EDF priorities and rate-monotonic priorities. In EDF scheduling, the priority of the thread invocation is set equal to the end of its invocation period, i.e., its deadline. Threads will earlier deadlines are invoked first. In rate-monotonic scheduling, the priority is equal to thread period, such that smaller periods have higher (i.e., numerically smaller) priorities.

4.5.3 Semaphore Operations

Since the *q*threads scheduler does not have its own threads, but rather "borrows threads from the kernel", a kernel thread registered with *q*threads may upset *q*threads scheduling if it blocks on kernel semaphores. This is because unless the blocking thread notifies the user-level scheduler that it is about to block, the scheduler does not know when it blocks and will not schedule another thread for execution. Any semaphore operations of quasi-periodic threads, therefore, have to use *q*threads semaphores. We implemented our own $semCreate$, $semWait$ and $semSignal$ operations. Let j be some semaphore created by *q*threads' $semCreate$. When a quasi-periodic thread Q_i calls $semWait$ on j, if the semaphore is already locked, the thread is blocked on its own kernel semaphore S_i, and entered into js queue maintained by the *q*threads scheduler. If the highest priority thread blocked on j is T_k, a subsequent $semSignal$ on j signals the kernel semaphore S_k. Thus, blocked threads are awakened in priority order. The priority queue of the semaphore allows semaphore operations to obey whatever prioritization policy is implemented in the *q*threads scheduler. Priority inheritance and priority ceiling protocols can easily be implemented if desired.

4.5.4 Thread Budgets

Thread budget manipulation in *q*threads is one of the more unique aspects of the *q*threads scheduler. The execution budget of quasi-periodic threads is defined in abstract units of work with bandwidth semantics measured in machine-independent application-level units such as frames per second, URL hit rate, or byte throughput. Using machine-independent execution budgets for threads is in sharp contrast to operating system abstractions such as capacity reserves where budgets are defined in machine specific units such as milliseconds of CPU time. The separation between budget units and machine speed in our architecture allows *Adaptware* to adjust itself easily to changes in the underlying platform capacity without changing server code. Such adjustment is done by periodically computing, within the *q*threads scheduler, the mapping function from units of work to machine utilization. We require that a quasi-periodic thread, Q_i, decrements its budget, E_i, using a $decrementBudget(\Delta)$ call upon performing each unit of work. The call will block when the budget E_i expires. The decrement, Δ, is in general a vector of values that describe the unit of work performed. These values are converted by the operating system to a scalar decrement in the execution budget E_i assigned to Q_is contract, C_i, by QoS level control. Thus, $\delta E_i = \theta.\Delta$, where θ is a vector of linear coefficients determined by a self-profiling module. The length of vectors Δ and θ is a configurable parameter instantiated at system initialization time. In view of the QoS contract definition presented in Section 4.4.2, we set the vector Δ to be (δ_R, δ_{BW}), where δ_R is the number of packets sent in the unit of work (e.g., 1 if the unit of work is a packet), and δ_{BW} is the number of bytes sent (which will vary depending on the size of the sent packets). From the QoS mapping results described in Section 3.4.1, we know that the decrement in the execution budget is given by $\delta E_i = a\delta_R + b\delta_{BW}$, where a and b are the profiling parameters. Thus, $\theta = (a, b)$. Alternatively, we can restrict θ to be of length 1, in which case $\delta E_i = a^{`}\delta_R$. The latter case is more suited for multimedia servers where all packets in the sent stream have the same packet size. In this case, $\delta_{BW}$ is proportional to $\delta_R$. Therefore, $\delta E_i = a\delta_R + b\delta_{BW} = (a + bs_{MTU})\delta_R = a^{`}\delta_R$, where s_{MTU} is the uniform packet size.

Contract handler threads are trusted with decrementing the budget appropriately every unit of work. Since the *q*threads package is developed for server designers we assume that the designer can be trusted. From the designer's perspective, a unit of work may denote a served packet, a served

URL, a served video frame, a served audio frame, or other similar service rate measures depending on the application. In addition to manipulating the budget, $decrementBudget$ internally calls $yield$ which will result in a context switch if a higher priority thread is ready. Thus, cooperative preemption of quasi-periodic threads discussed in Section 4.5.2 is implemented transparently on unit of work boundaries.

4.5.5 Self-profiling

An important goal of *Adaptware* is automatic adaptation to platform capacity. Platform capacity, therefore, needs to be determined dynamically without manual instrumentation and measurement every time resources are upgraded, code is changed, or the application is re-executed on different OS or hardware. For this purpose we employ a self-profiling subsystem. A profiling module is executed as a high priority thread invoked periodically by the scheduler at a preconfigurable period τ. The job of the profiling module is to update the measurement vector θ, which is a global scheduler data structure. Outside the profiling module, the scheduler makes no assumptions regarding the semantics of the measurement vector, θ, beyond the stipulation that the logical budget consumed by a quasi-periodic thread in processing an abstract unit of work is given by the dot product of the two vectors, θ and Δ, where Δ is the vector of values passed by quasi-periodic application threads in $decrementBudget$ calls. This separation of concerns enhances the maintainability and upgradeability of *Adaptware*. For example, in order to change the semantics of QoS contracts (i.e., the semantics of Δ), changes to *Adaptware* code may be localized to the profiling module that computes θ) with no alterations required to the rest of the scheduler architecture and enforcement mechanisms. This is in contrast to other operating system implementations where the nature of enforced guarantees is hardwired into kernel code. In our particular implementation of the profiling module it is assumed that, $\Delta = (\delta_R, \delta_{BW})$, where δ_R and δ_{BW} denote the packets and bytes sent respectively. The profiling subsystem looks up, every period τ:

- the number of packets, $N = \sum x_R$, transmitted within τ,
- the number of bytes, $W = \sum x_{BW}$, transmitted within τ, and
- the percentage of time μ during which at least one quasi-periodic thread was ready or running. It represents system utilization by the scheduled communication threads (taking into account the tasks that preempt their execution).

Let the vector X be defined as (N, W). Linear regression is used to compute vector θ, such that $\tau\mu = X\theta$. The known least square estimator yields $\theta = (X^T X)^{-1} X^T \tau\mu$. For better computational efficiency, we use a recursive form of the least squares estimator. The recursive least squares estimator computes the parameter estimates that minimizes the standard deviation of measurement error, assuming white measurement noise. Let the profiling vector at the kth invocation of the profiling module be denoted by X_k. Let the measured utilization be μ_k. Let the current estimate of profiling parameters be given by the vector θ_k. These estimates can be initialized to zero at the start of estimation. Let P_k be a square matrix whose initial value P_0 at the start of estimation is set to a unit matrix. The estimator's equations at sampling instant k are:

$$\gamma_k = (X_k^T P_{k-1} X_k + 1)^{-1} \tag{4.1}$$

$$\theta_k = \theta_{k-1} + P_{k-1} m_k \gamma_k (\tau\mu_k - X_k^T \theta_{k-1}) \tag{4.2}$$

$$P_k = P_{k-1} - P_{k-1} X_k \gamma_k X_k^T P_{k-1} \tag{4.3}$$

To improve convergence and increase statistical confidence in the estimates, we increase the observation period τ to yield consistent results, thus filtering out high frequency variations. The automated computation of θ decouples thread budget definitions from platform capacity considerations, essentially computing the mapping between machine-independent budgets and platform speed adaptively in a transparent manner within the operating system. We believe that the ability to perform such mapping is an important quality of future operating systems that support real-time applications on arbitrary platforms.

4.5.6 Programming with QoS Contracts

On top of the quasi-periodic thread abstraction described in preceding sections, we implemented QoS contracts, the second main abstraction of *q*threads. A contract is created by a $createContract$ call. Quasi-periodic threads can the be assigned to the contract via a $contrcatAllocateThread$ call. The values of QoS parameters are declared via the $contractRegisterLevel$ call which defines QoS levels. The call specifies the byte and/or packet rate parameters. It invokes QoS mapping, described in Section 3.4.1, to compute the corresponding utilization budget. The contract cannot be activated until it passes the admission control test. A contract is submitted to admission control via the $contractSubmit$ primitive, which executes the QoS optimization algorithm of Section 3.4.2 that determines whether the contract should be admitted and assigns an optimal QoS level k^* for each admitted contract. Upon admitting a new contract, the period and budget parameters of all quasi-periodic threads assigned to the contract are set to those of the chosen QoS level. In the implementation, the budget of a quasi-periodic thread is represented by a pointer, which allows sharing the same budget among multiple threads. Thus, all the threads assigned to the same contract share the same budget. The following is an application code sample that creates a QoS contract:

```
1   contractId = contractCreate ();
2   contrcatAllocateThread (contractId, threadId);
3   contractRegisterLevel (contractId, QoS_Level_1, $0.02);
4   contractRegisterLevel (contractId, QoS_Level_2, $0.01);
5   contractRegisterLevel (contractId, QoS_Level_3, $0.006);
6   contractRegisterLevel (contractId, No_Service, $-V);
7   contractSubmit (contractId);
```

In this example, the first two lines initialize contarct data structures for the newly-created contract and assign a contract handler thread to it. Lines 3-5 define the QoS levels acceptable in the QoS contract (passing a data structure that describes appropriate QoS parameters), and their rewards. Line 6 specifies the QoS violation penalty. The negative sign in $-V$ indicates that the penalty is debited rather than credited to the server. The last line submits the contract for the admission-control algorithm.

The QoS level selected for each contract is kept in the contract's state. The present QoS level can be polled by the contract handler thread using the primitive $contractGetQoSLevel$ which returns the QoS level. The returned value can be used to determine which code version needs to be executed by the thread during the current iteration of its service loop. Different QoS levels may give rise to different versions of code. For example, data compression may be performed in one level but not another.

4.5.7 Load Monitoring and QoS Level Adjustment

While contract admission control sets the QoS levels of quasi-periodic threads based on load estimates obtained by self profiling we need to adjust these QoS levels in view of actual measured load when the estimates are inaccurate. Thus, we check for overload or under-utilization conditions periodically, and make a small incremental adjustment to the QoS levels of some threads around the operating point computed by the admission control module. The purpose of QoS-level adjustment is to keep the system from getting under-utilized or overloaded due to transient load or inaccurate profiling estimates θ. Adjusted QoS levels are always selected among those specified in the QoS contract for the threads in question. In our implementation, overload conditions are detected by checking whether or not thread deadlines are violated. This is accomplished by adding a missed-deadline counter to the *q*threads scheduler. When d deadlines are violated, "overload" is flagged. In the current implementation, deadline violations are never "forgotten" until overload is flagged (at which point the counter is reset). In cases where the application can tolerate a certain rate of deadline misses, it may make more sense to "forget" about deadline violations that occurred more than a specified time interval V ago. Overload should be flagged if d deadlines have been violated in the last V time units. The parameter d (and V) can be configured according to application's tolerance to deadline misses. Under-utilization conditions are detected by inspecting the effective utilization μ. When it drops below a pre-configured threshold *and* some of the threads are scheduled below the maximum QoS level $L_{desired}$ specified in their contract, "under-utilization" is flagged. Once overload or under-utilization is flagged, a QoS-adjustment heuristic is executed to adapt QoS levels accordingly. It executes a fast greedy algorithm attempting to maximize the aggregate achieved reward as shown below:

QoS Level Adjustment Heuristic

if $overload$ **then**
    Choose the *q*thread, Q_i, whose degradation by one QoS level results
    in the $minimum$ decrease in reward
        (i.e., $R_i[current] - R_i[current-1]$ is minimum)
    Degrade Q_i to next QoS level.
if $underutilization$ **then**
    Choose the *q*thread, Q_i, whose promotion by one QoS level re sults
    in the $maximum$ increase in reward
        (i.e., $R_i[current+1] - R_i[current]$ is maximum)
    Promote Q_i to next QoS level.

Since the heuristic offers a non-optimal solution, we may need to recompute QoS levels using an optimal algorithm periodically, with a larger period. Together, the optimal algorithm and the heuristic adjustment keep service utility maximized for the community of clients.

4.6 *CLIPS* Communication Resource Management

We describe our implementation of the multithreaded communication server using $q$threads support. The architecture is called *CLIPS* (Communication Library for Implementing Performance-assured Services). It offers an adaptive communication resource-management mechanism on end-hosts. Figure 4.5 depicts the main components of the communication server. It illustrates both the send and receive data paths. As seen from Figure 4.5, three different types of threads are involved in handling data on each path, namely, API threads, contract handler threads and kernel interface threads. These threads are discussed in more detail in the following subsections.

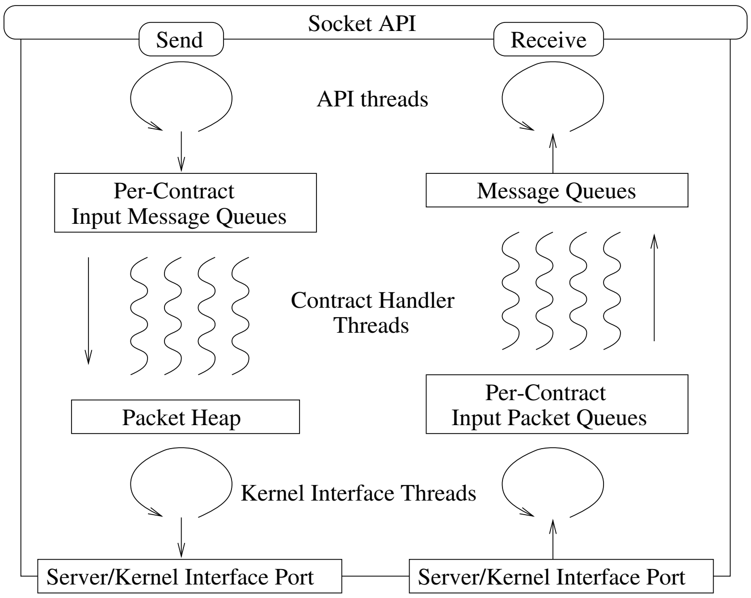


Figure 4.5: The Communication Server

4.6.1 Application-Server IPC and API Threads

Application messages accumulate in the kernel queues for delivery to the server. In a QoS-sensitive system the length of such a queue should be derived from the application traffic specification, for example message rate, size, and burst. If the queue is too small, application messages may be dropped or result in frequent overflow and context switches thereby affecting performance. If the queue is overly long, application messages may reside in it longer than desired and result in violations of contracted latency guarantees. Unless we modify kernel code, we cannot control the allocation of kernel-level IPC queues (Mach port queues in our implementation). Therefore, our strategy is to drain them as fast as possible, transferring messages to contract-specific queues in the communication server. These queues are sized in accordance with the connection's traffic specification. We create API threads within the server whose function is to consume messages from the corresponding Mach port queue. Once an application sends a message to the server, an API thread reads it from the Mach port and queues it for the corresponding contract handler. The thread, whose execution time is charged to the handler's budget, runs at handler priority, and is allowed to continue running at background priority when the handler's budget expires.

One API thread is created per socket in the communication server. The same thread handles both sent and received data. Normally, the API threads are blocked waiting for I/O. When an application invokes an API call on a particular socket, the corresponding API thread is unblocked. The thread identifies the requested operation and invokes the corresponding routine in the server. If the operation is a send, the thread queues up the data into the input message queue of the contract handler associated with the given socket. If the queue fills up, the thread is blocked on the message enqueue operation until queue space becomes available essentially blocking the application until the enqueue operation can be completed. If the operation is a receive, the thread reads the received message queue. Note that in this scheme a blocked send operation will prevent a receive from taking place on the same socket since there is only one API thread per socket. We allow this situation to happen because in most servers sockets to particular clients are used unidirectionally to send outbound traffic. Clients access the server via a different socket (e.g., the server's well known port) to send incoming requests.

4.6.2 Contract Handler Threads

Contract handler threads are the main worker threads in our communication subsystem. There are two contract handler threads per contract; one for sending data and one for receiving. Collectively, the tasks implemented by the contract handler threads in serving a contract are as follows:

- *QoS-level-sensitive message dequeuing:* The handler of contract C_i dequeues messages from its input message queues at a fixed rate, asynchronously with the enqueue operation, every period $P_i[k]$ as specified by its QoS level, k. If several sources deposit messages into the same queue (as might be the case with client aggregation), WFQ can be implemented to achieve fair bandwidth sharing, and guarantee each client a minimum share of the contract's bandwidth. The message queue is sized to $B_i[k]M_i[k]$ in accordance with the QoS contract.

- *Protocol processing:* Communication protocol processing, packetization and depacketization is performed by the contract handler. The handler can be configured to execute an arbitrary protocol stack. We gain such flexibility by using x-kernel for protocol development. x-kernel exports a uniform protocol interface and allows configuring arbitrary protocol stacks.

- *Per-contract policing:* In order to prevent clients from violating their traffic specification, client traffic is policed on a per-handler basis to ensure that each connection is conformant to the rate of service dictated by its current QoS level. Traffic is policed by limiting each contract handler to send (receive) no more than $M_i[k]$ packets during each period $P_i[k]$. This is achieved by calling $contractDecrementBudget$ after each packet transmission as described in Section 4.5.4. The call blocks the calling thread when the budget expires. Note that such voluntary blocking (yielding) indirectly enforces a CPU run-time limit on each invocation of the handler.

- *Outgoing packet queuing:* The contract handler, on the sender side, deposits processed packets in an outgoing queue. When the communication link becomes available to the sending host, link bandwidth must be allocated to outgoing packets in proper priority order to provide the QoS levels selected by *q*threads. Outgoing packets are therefore queued in a priority heap sorted by thread deadline, as assigned to the thread by the *q*thread scheduler.

- *Receive side demultiplexing:* Incoming packets are demultiplexed by the receiver which then invokes the contract handler responsible for serving the particular connection. The communication subsystem scheduler then schedules each contract handler to run in accordance with the deadline assigned to it from its QoS level.

4.6.3 Outgoing Network Device Handler Thread

The bottom layer of the protocol stack, in our server architecture, interfaces with the kernel device driver via the kernel's IPC mechanism. Device output is initiated by *CLIPS* as close as possible to the device driver without being in the kernel. Outgoing packets are dequeued from the priority heap in which they are deposited by contract handlers, and sent to the network device in the kernel. Being in user space, the dequeue operation cannot be invoked directly by the kernel device driver in response to transmission completion interrupts unless the underlying OS supports user-level upcalls. In the absence of such support user-level code cannot be executed in interrupt context. Instead, we utilize a high priority user-level thread to perform packet dequeuing and device transfers. The thread is signalled when the packet heap becomes not empty. It loops dequeuing successive packets and sending them synchronously to the kernel until the heap is drained. Packets are sent to the kernel synchronously in order to avoid packet accumulation in the FIFO queue of the kernel's device driver. Ideally, we would have implemented the priority heap in the device driver, eliminating the need for the packet heap and the dequeuing kernel interface thread in user space. This solution requires modifying the kernel unless the operating system allows inserting customized device drivers without changing kernel code.

4.6.4 Incoming Packet Classification Thread

Resource reservation must be coordinated in a QoS-sensitive fashion. CORDS provides two abstractions, *paths* and *allocators*, for reservation and allocation of system resources. Resources associated with paths include dynamically allocated memory, input packet buffers, and input threads that shepherd messages up the protocol stack [49]. Paths, coupled with allocators, provide a capability for reserving and allocating resources at any protocol stack layer on behalf of a particular connection, or contract. With packet demultiplexing at the lowest level at the receiver (i.e., performed in the device driver), it is possible to isolate packets of different contracts from each other early in the protocol stack. Incoming packets are stored in buffers explicitly tied to the appropriate contract and serviced by contract handler threads previously allocated to that contract.

Proper handling of received data requires that a packet's contract be identified as early as possible in the protocol stack, and that packets be queued and served accordingly. One solution for this problem would be to prepend a contract identifier to sent packets to allow early contract-based demultiplexing at the receiver. While this technique is natural for networks supporting a notion of virtual circuit identifiers (VCI) such at ATM, it is not so for traditional data link technologies such as Ethernet. In the case of Ethernet, the CORDS driver has the capability to add a new *path identifier* to the data link header that serves as a contract identifier. This, however, would create a non-standard Ethernet header that would not be understood by hosts not running the CORDS framework.

In our implementation, the unmodified in-kernel network device driver simply relays received packets to the communication server in FIFO order via the available IPC mechanism, in our case Mach ports. This FIFO ordering has two main disadvantages. First, it does not respect connection QoS requirements, since urgent packets can suffer unbounded priority inversion when preceded by an arbitrary number of less urgent packets in the queue. Second, since the same queue is used for guaranteed and non-guaranteed traffic, depending on packet arrival-time patterns, guaranteed data maybe dropped when the queue is filled by non-guaranteed packets. These two problems cannot be solved without modifying the kernel device driver. To ameliorate this unpredictability, a high priority thread waits on the communication server's input port and dequeues incoming packets as soon as they arrive, depositing them in their appropriate contract-specific queues. This prevents FIFO packet accumulation in the kernel and allows the server to service packets in priority order according to their contract.

4.7 Evaluation

We conducted several experiments to test the performance of the implemented communication subsystem server and verify its ability to meet the contracted QoS requirements. To emulate natural operating conditions, the server machine was placed on one segment of a shared departmental $10Mb$ Ethernet serving a couple of hundred machines in the department (204 to be exact). The Ethernet is connected to a campus-wide network via an FDDI ring that is in turn connected to an Internet backbone. First, we identified the cost of QoS adaptation in the proposed architecture. Our profiling results indicate that the periodic load monitoring overhead is about $7\mu s$ per iteration, the QoS-optimization overhead is approximately $32\mu s$ per QoS contract, and the heuristic QoS-adjustment overhead is $8\mu s$. Currently, monitoring and heuristic QoS adjustment are performed once every $100ms$, and QoS-optimization is invoked every 5 seconds. For 10 contracts created in the system (due to client aggregation the number of clients can be much more), the above figures indicate that the aggregate overhead consumed by QoS-adaptation mechanisms is less than 0.1 %. In the following we describe two sets of experiments. The first was conducted under conditions where CPU power was the bottleneck resource. The second was conducted when the network was the bottleneck. As we shall see, different elements of the architecture were sensitized in the two sets of experiments, giving rise to different performance characteristics and problems.

4.7.1 Experiments with a CPU bottleneck

We conducted a set of experiments with QoS contracts in an environment where the end-system was the bottleneck resource. Since in our testbed the server was fast enough to saturate the link, we turned on the (fairly heavy) debugging mode to slow down the server enough to become the bottleneck.

Contract Policing

In response to contract establishment requests, we created a set of contract handler threads, each of which processes application messages that are generated persistently by "dummy" applications and sent using our communication subsystem API. Each application would simply sit in a tight loop generating fake data. The contract handler thread for each connection enforces its QoS contract and polices the application in accordance with its selected QoS level. Connection throughput was measured at the receivers. Figure 4.6 shows the measured throughput versus time for each of three UDP flows. In this experiment the contacted rates for the UDP flows were $1.5Mb/s$, $1Mb/s$ and $0.5Mb/s$, respectively.[2] The figure shows that although the applications dump messages persistently to the communication subsystem without regard to a maximum rate, their measured throughput does not violate the contracted QoS due to appropriate policing. Similar results were seen when we added non-guaranteed traffic in the background. Since non-guaranteed traffic is served by a lower priority (default) handler, it does not get through unless no guaranteed contract handlers are running.

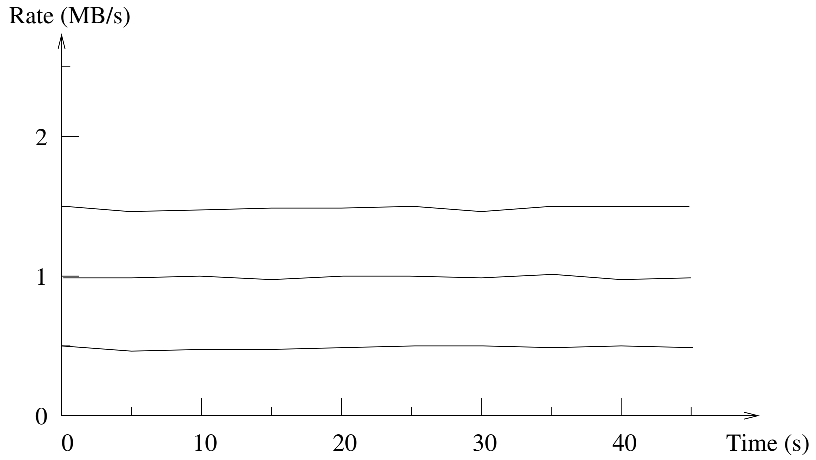


Figure 4.6: Policing effect

[2] Where b, in this section, refers to *bits*.

Admission Control

The rate-policing and traffic prioritization mechanisms alone, however, are not sufficient for enforcing the QoS contracts. In Figure 4.7 we show what happens when we increase the number of guaranteed connections above the schedulable limit. In this experiment, we incrementally add new connections, creating a $1Mb$ contract for each. The QoS contract for each connection has $M_i = 100kb$, and $P_i = 100ms$. As we can see, the system becomes overloaded and the average connection throughput decreases below its contracted value as the aggregate bandwidth consumed by the system saturates. In our experiments the maximum aggregate consumed bandwidth was found to be approximately $3.7Mb$ (due to the heavy debugging mode). The inability of individual connections to receive their contracted rate calls for an admission-control mechanism to ensure that the set of all guaranteed connections is schedulable.

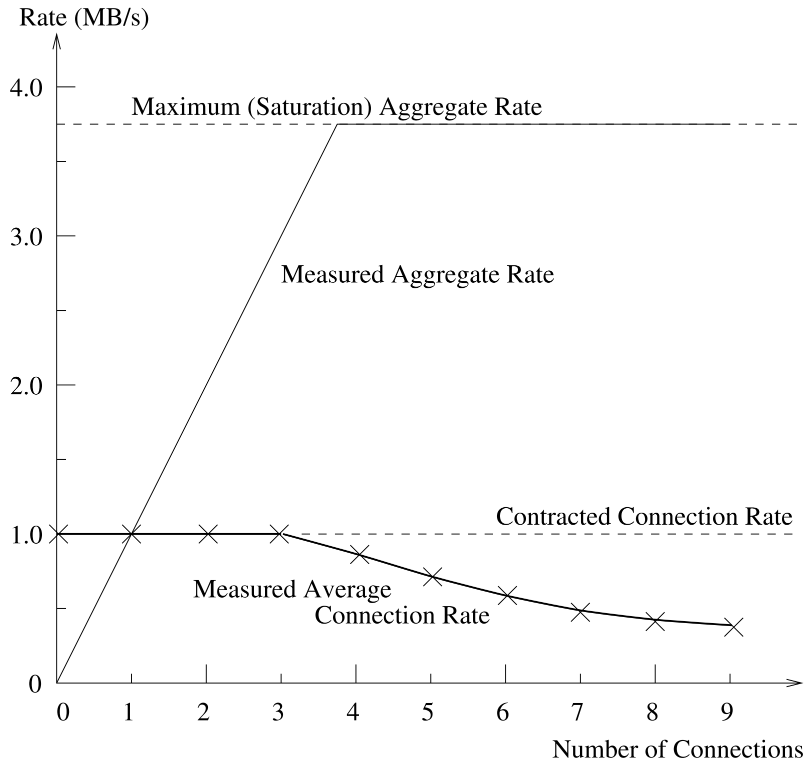


Figure 4.7: Overload and violation of contracted QoS

We incorporated the admission-control algorithm presented in Section 3.4.2. We initialized the length of the profiling vector θ to unity, essentially reducing it to a scalar, a. We used Ethernet packets as the unit of work in our architecture, decrementing the budget of contract handler threads upon each packet transmission. The on-line estimated value of a was about $3.25ms/pkt$, which yields a maximum throughput of about 307 (Ethernet) packets per second. This throughput permits admitting only three $1Mb$ connections. From Figure 4.7 we can see that running more than 3 connections results in QoS contract violation, indeed. Thus, our combination of automated profiling and admission control does succeed in adjusting to platform capacity such that overload is prevented and QoS guarantees are maintained.

QoS Adaptation

We analyzed system response to transient load disturbances. In this experiment, we fixed the number of connections, defined multiple QoS levels for each, then varied the load on the host letting the optimization algorithm implemented by the QoS level control module run periodically (every 5 seconds) to recompute the QoS levels based on the most recent estimate of a. Two long compilation tasks were started concurrently with packet transmission to overload the CPU. Figure 4.8 shows the results of a representative run. The top part of the figure shows 5 contracted connections, labeled $C_1, ..., C_5$, where C_1 is the least important connection, and C_5 is the most important. The QoS contract for each connection had 3 QoS levels of bandwidth $1Mb$ ($M_i = 100kb, P_i = 100ms$), $0.33Mb$ ($M_i = 67kb$, $P_i = 200ms$) and $0.11Mb$ ($M_i = 33kb$, $P_i = 300ms$) respectively. Rewards were assigned proportionally to the bandwidth and weighted by connection importance. Thus, the reward for QoS level k of connection C_i was $R_i[k] = iM_i[k]$. The figure depicts the QoS level selected for each connection at every invocation of the QoS selection/optimization algorithm. The bottom part of the figure shows the change in the measured cost-per-packet, a, as well as the number of missed deadlines between successive invocations of QoS optimization (A deadline miss means that M_i bits weren't transmitted within P_i time units.) As can be seen from the figure, less important connections were degraded during overload intervals to keep aggregate system throughput below saturation. Since the QoS level optimization algorithm runs at a slow period (5 seconds), we observed a relatively large number of deadline violations. This is because QoS level selection was not responsive to short term load fluctuations which caused some deadlines to be missed. When the experiment was repeated with the fast QoS level adjustment heuristic enabled to change the QoS levels in response to transient load disturbances, most of the deadline misses were eliminated, as shown in Figure 4.9.

The above-presented experiments illustrate the function and effectiveness of the main architectural elements of our design. Namely, our architecture is shown to (i) provide per-contract guarantees on the end-host, (ii) maximize the aggregate reward of the end-host's communication service

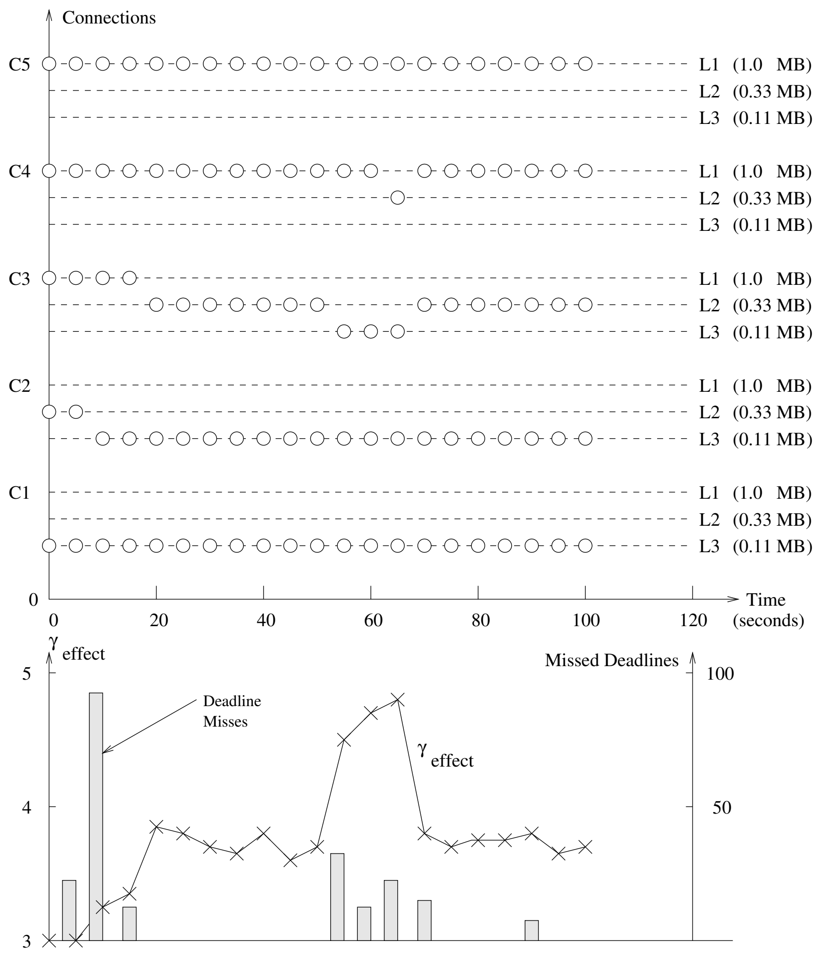


Figure 4.8: QoS level adaptation

across all clients, and (iii) adapt responsively to transient overloads and resource shortage.

4.7.2 Experiments with a Link Bottleneck

In order to test the performance of our communication subsystem when the communication link is the bottleneck, we recompiled the server with debugging code disabled, causing it to become significantly faster. The new executable was fast enough to saturate the $10Mb$ Ethernet. The experiments reported in this section were conducted on a private Ethernet to prevent saturating the department's network with large UDP flows. To verify that the contracted QoS is still enforced we created 3 contracted UDP flows of bandwidth $1Mb$, $2Mb$ and $3Mb$, respectively. A non-guaranteed UDP flow was sent concurrently with gradually increasing bandwidth. We observed that once the aggregate outgoing flow saturates the Ethernet link, the server is unable to maintain bandwidth guarantees for contracted flows. This effect is demonstrated in Figure 4.10 by the decline in guaranteed flow

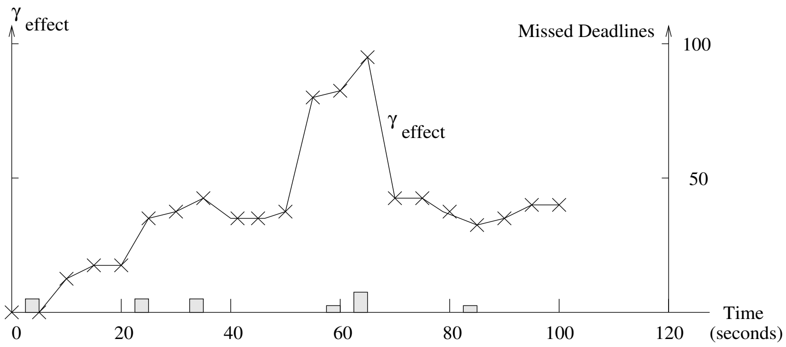


Figure 4.9: Improved responsiveness due to QoS level adjustment heuristic

bandwidth after the communication link gets saturated.

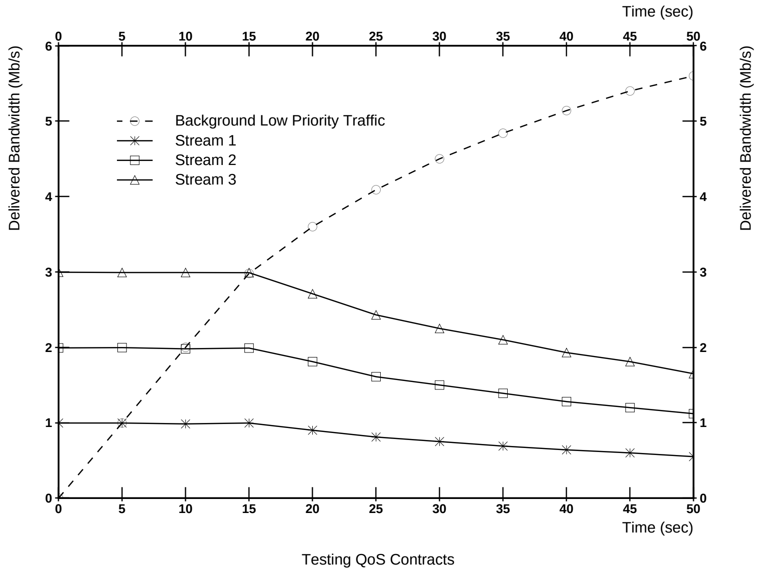


Figure 4.10: A Glitch in Enforcing QoS Guarantees

The Buffer Size Dependency

Ideally, guaranteed bandwidth should have remained constant, since non-guaranteed flows are served at a lower priority and should not have interfered with guaranteed flows. The explanation of this interference lies in communication buffer overflow on the end-system. We used a version of code in this experiment where an internal packet buffer is associated with each outgoing connection. When the network is slow, the server causes these buffers to saturate with the server's outgoing data awaiting transmission, which blocks the sending contract handler threads. As a result, the size of the outgoing data buffers plays a critical role in determining the outgoing connection throughput.

A low-priority UDP connection with a large buffer may send more data during periods of network overload than a high-priority connection with a small buffer which makes priority scheduling ineffective.

To demonstrate the effect of buffer size on UDP flows, a QoS contract was established with a guaranteed client A, specifying the packet rate of $R_i = M_i/P_i = 600$ pkts/s (1500 bytes/pkt). A non-contracted client B was created. The aggregate data generated by both clients saturates the server's 10 Mb/s Ethernet link causing an overload. The ratio of the outgoing data buffer sizes of the two connections was changed gradually from 0.2 to 4, and the ratio of the connection throughputs was recorded for each ratio of buffer size. Figure 4.11 plots the observed ratio of bandwidth delivered to the two clients versus the buffer size ratio of their respective connections. The figure shows both (i) the case where the server was instrumented to let the default handler run at the same priority as client A's handler, and (ii) the case where A's handler has the higher priority. It can be seen that regardless of contract handler priority, the delivered bandwidth ratio followed closely the connection buffer-size ratio showing a strong correlation between outgoing buffer size and connection bandwidth. We fixed the aforementioned problem by aggregating all outgoing packets in a single buffer implemented as a priority heap and dequeuing packets in priority order.

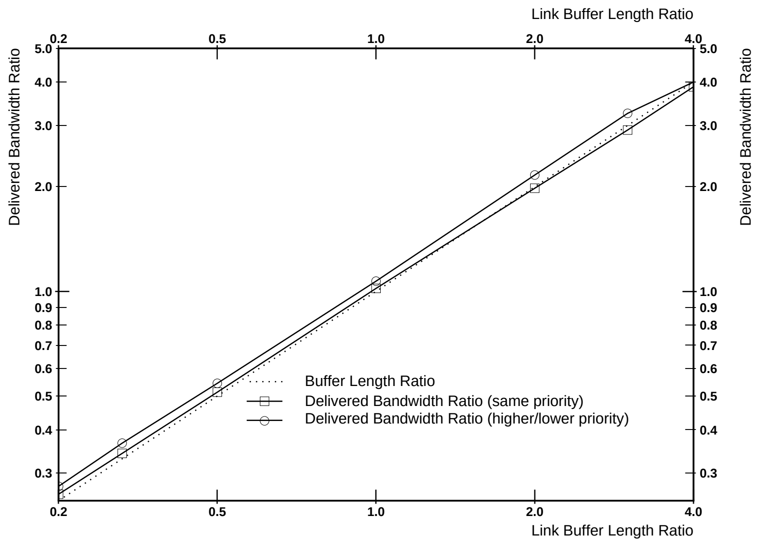


Figure 4.11: Bandwidth dependency on buffer size

The Semaphore Dependency

In order to test the new communication subsystem prototype, two contract handler threads were created in the server to process two data streams: one to a guaranteed client A, and the other to a non-guaranteed client. The handlers process outgoing data and deposit it in the common packet heap, blocking on a common semaphore when the heap is full. Handler priorities were such that the thread serving the guaranteed client was higher in priority than that serving the other client. We policed both clients to a packet rate of R_i that was gradually increased, as shown by the dotted line in Figure 4.12, to cause overload. The solid lines in Figure 4.12 show the corresponding actual packet rate received by each client on their respective destination machines. It can be seen that each client receives the generated rate R_i up until the aggregate packet generation rate at the server increases beyond $700pkt/s$ (i.e., until $R_i = 350pkt/s$ for each client). This aggregate rate saturates the 10Mb/s Ethernet. When R_i is increased beyond 350, the server fails to deliver packets at rate R_i even to the higher priority client, although it delivers an aggregate rate much higher than R_i. We conclude that clients are not served in priority order.

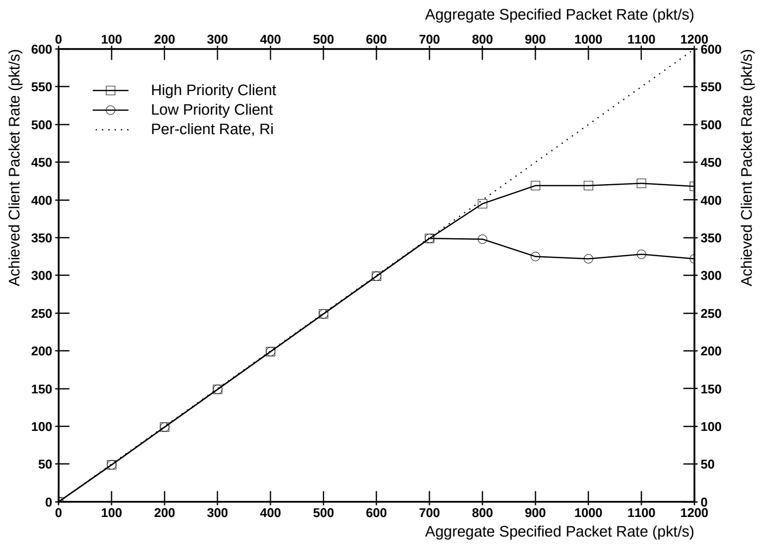


Figure 4.12: Effect of priority-insensitive semaphores

The reason this time had to do with semaphore implementation. The Mach kernel does not ensure that when a semaphore is signaled, the highest-priority thread waiting on the semaphore gets the CPU. Instead, a waiting thread is awakened in FIFO order. The use of a single semaphore for the two communication subsystem threads randomizes thread execution in a way that does not allow the guaranteed (high-priority) client to properly utilize the available bandwidth. In order to verify that the semaphore implementation is the cause of the observed problem, we implemented *q*thread semaphores (on top of Mach semaphores) with a priority queue of their own. The *q*thread semaphores ensure that the highest-priority waiting thread is resumed when a semaphore is signaled. The above experiment was repeated. Figure 4.13 shows that the higher-priority client now achieves its ideal service rate, R_i, much more closely at the expense of the lower-priority client.

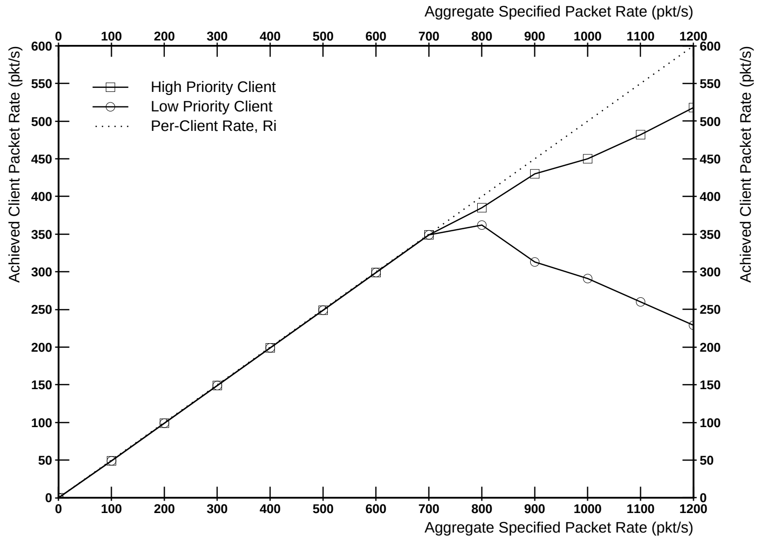


Figure 4.13: Prioritization with improved semaphore implementation

Residual Effects

We conjecture that the residual inaccuracy in achieving the ideal service rate by the guaranteed client is due to the interaction between the policing mechanism and priority scheduling. When the budget of a contract handler expires the handler blocks, allowing the default lower priority handler to run. Since the CPU is not the bottleneck the low priority handler can run until it saturates the outgoing packet heap. When the high priority contract handler is eventually resumed it may not be able to send its data in time. These problems arise only under overload. In the absence of overload, the system performs to specification, e.g., as seen in Figure 4.12, for aggregate packet rates below 700 pkts/s. We, therefore, utilized a policing mechanism to "emulate" priorities. A utilization budget is created for the default low-priority handler. Its value is set to the remaining system capacity left unused by guaranteed clients, i.e., to $100\% - \sum_i U_i$. The low-priority handler is policed not to over-draw its allocated resources by implementing it as a quasi-periodic thread of an appropriate period and budget.

Figure 4.14 illustrates the goodness of this approach at mimicking the thread prioritization effect when the network is the bottleneck. The previous experiment was repeated with one guaranteed and one non-guaranteed client, except that this time the non-guaranteed connection was policed to prevent overload. When the network is saturated, the data rate of the non-guaranteed client is forced to decline, via policing, with the increase in the contracted rate of the guaranteed client to keep their sum constant at the maximum network bandwidth. It can be seen from Figure 4.14 that the guaranteed client is now able to receive its contracted service rate.

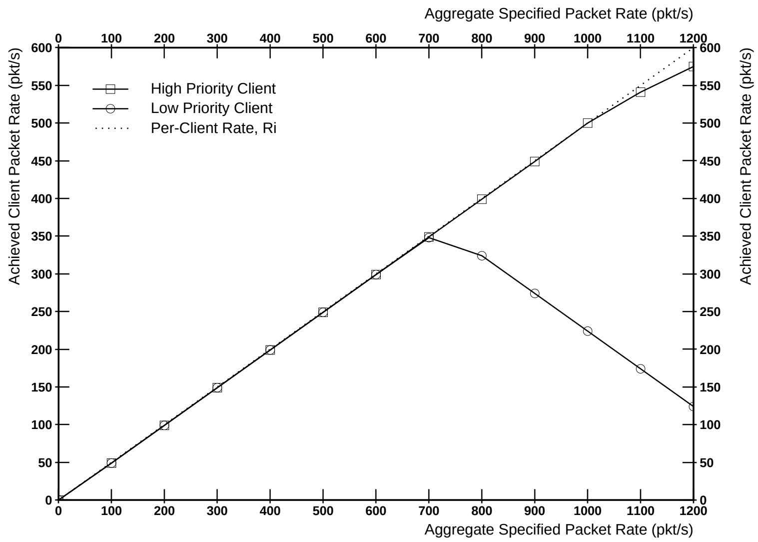


Figure 4.14: Achieving QoS-sensitive behavior

Figure 4.15 shows the consumption of network bandwidth, which is the bottleneck resource. Note how the network saturates at an aggregate delivered bandwidth of 10 Mb/s (measured at the receivers), which is its maximum capacity, when the aggregate service rate increases beyond 700 requests per second. This verifies that the network is indeed the bottleneck resource in the presented experiments.

We now repeat the experiment shown in Figure 4.10 using the communication server enhanced with aforementioned modifications. The same UDP flows were created of contracts for $1Mb$, $2Mb$ and $3Mb$, respectively. A non-guaranteed UDP flow was sent concurrently with gradually increasing bandwidth. This time we observed that each guaranteed flow received its contracted bandwidth. We also observed that the non-guaranteed flow was policed to the remaining bottleneck resource capacity. These results are demonstrated in Figure 4.16. We conclude that our adaptive communication subsystem is able to ensure QoS guarantees regardless of the bottleneck resource. This feature separates our design from other architectures which often focus either on link bandwidth management (typically in the networking community), or end-system resource scheduling (typically in the operating system community). The obtained guarantees are adapted to platform resources and load.

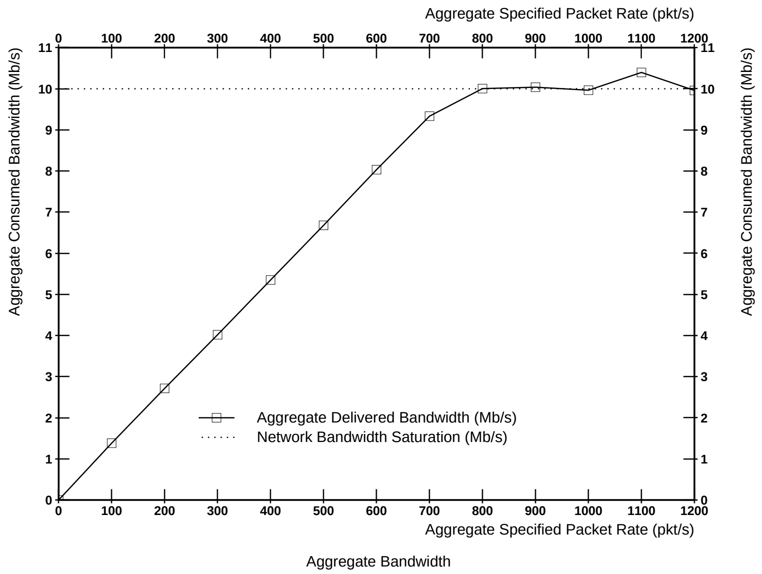


Figure 4.15: Network bandwidth saturation

4.8 Extensions

The mechanisms described in the previous sections can be implemented in middleware on top of a regular operating system such as UNIX. QoS extensions we introduce fall roughly into (i) profiling extensions executed periodically and independently of other scheduled activities, (ii) admission control extensions invoked at contract creation and activation time, and (ii) enforcement extensions embodied into the *q*thread quasi-periodic thread policing mechanism used to police contract handler threads. Of these extensions, (i) and (ii) can be executed outside the operating system in a middleware layer as described in Chapter 2. Policing needs to be redesigned, since the middleware does not have access to thread scheduling in the communication subsystem. The middleware associates resources with contracts (instead of threads). It makes no assumptions on thread-to-client mapping in the server process and communication subsystem, which separates this middleware implementation of QoS-provisioning from other mechanisms for performance isolation and QoS guarantees, such as [3, 55, 63, 75, 86, 88, 95, 141]. These solutions usually addressed QoS provisioning as a per-thread or per-process concept.

4.8.1 Contract Policing in Middleware

To police QoS contracts, our middleware, called *q*Contracts, keeps a periodic timer. At the beginning of every period P_i, an execution budget of P_iU_i is allocated for each admitted contract

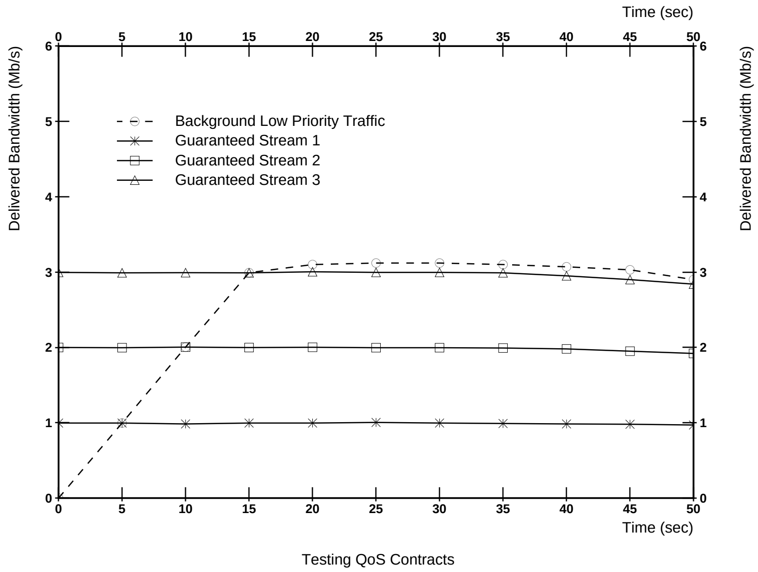


Figure 4.16: Enforcing QoS Guarantees

Q_i, where U_i is the logical utilization allocated to the contract. Policing must ensure that no contract Q_i exceeds its allocated execution budget. The middleware executes policing code in the context of the server's worker threads when they perform socket calls. qContracts exports the $contractChargeBudget(client\_id, x)$ primitive for the purpose of policing. The primitive is called upon execution of each service unit, where x is the byte size of the served unit. The call reduces the budget U_iP_i by $max(a + bx,\ cx)$, the processing time attributed to the served unit. If the budget expires the call either blocks or, in its non-blocking version, returns an error. To describe how to this policing mechanism achieves its purpose, we make a distinction between two general types of servers:

- *Servers with long-lived flows:* A server is said to have long-lived flows if the single connection's flow lasts much longer than P_i. In that sense, servers with persistent HTTP 1.1 connections, for example, do not necessarily belong to the long-lived flow category since their connections may be idle most of the time with only sporadic short bursts of flow. Video servers, on the other hand, are a good example of this type. Serving a single request will take the entire duration of a movie (i.e., in the order of hours). Service rate, $R_i = M_i/P_i$, in this case is typically the rate at which the single request is served, e.g., the frame rate at which the movie is transmitted. In servers with long-lived flows, either the server's worker thread or the $write()$ library call that sends the frames out to the client is instrumented to call the blocking version of $contractChargeBudget(client\_id, frame\_size)$ upon each frame transmission.

The call will block in qContracts when the execution budget expires, and will unblock when it is replenished. Since the budget is replenished to $U_i P_i$ every period P_i, the average utilization due to request execution on behalf of the i-th client cannot exceed U_i.

- *Servers with short-lived flows:* a server is said to have short-lived flows if a large number of server responses can be sent to completion within P_i. This, for example, is true of web servers which typically have the capacity to process hundreds of requests per second. In servers with short-lived flows, contracts are typically defined on flow aggregates. Service rate, $R_i = M_i/P_i$, in this case, defines the aggregate request rate (e.g., the hit rate on a particular web site). Bandwidth W_i defines the aggregate delivered byte rate. In this case, the use of the blocking version of $contractChargeBudget(client\_id, x)$ is inappropriate for policing because, upon budget expiration, the call blocks only the calling thread. It leaves it possible for a different thread to pick another request whose processing is charged to the same contract. In the worst case, all server threads may encounter, during some period P_i, requests charged to a contract whose budget has expired. All these threads will thus block on $contractChargeBudget(client\_id, x)$, bringing the entire server to a halt until that budget is replenished. The situation cannot arise in servers with long-lived flows because the service time of a request is much larger than P_i, making it impossible for more than one request to be charged to the same contract during any single budget replenishment period. To avoid this problem, in servers with short-lived flows, the $read()$ library call is instrumented to call $contractCheckBudget(client\_id)$ as each request is read in. The latter call returns an error if the budget has expired, in which case the instrumented $read()$ will discard this request (for violation of the contracted rate) and read the next. The $write()$ library call that sends the response to the client is instrumented to call a nonblocking $contractChargeBudget(client\_id, response\_size)$ upon response transmission to update the budget accordingly. (Note that if server source is available, it may be easier to call $contractCheckBudget(client\_id)$ and $contractChargeBudget(client\_id, response\_size)$ directly from the server's worker thread.) Since no requests are served after the $U_i P_i$ budget expires, and since the budget is replenished every period P_i, the average utilization by the i-th client cannot exceed U_i.

4.8.2 Discussion

Not surprisingly, there is a significant difference in the means of enforcing QoS contracts between long-lived flows (e.g., multimedia) and short-lived flows (e.g., HTTP). Multimedia applications, for example, are dominated by long-lasting connections of considerable bandwidth delivered to the client. Essentially, QoS enforcement is achieved by policing individual outbound server flows as described above. In web servers, on the other hand, contracts are defined for aggregate traffic, e.g., the aggregate of all the flows from a particular hosted site. In the worst case, all server threads may encounter, during some execution interval, requests charged to a contract whose budget has expired. All these threads will thus block on I/O bringing the entire server to a halt until that budget is replenished. To prevent such scenario, assigning different (non-overlapping) server thread pools to different traffic categories becomes essential for proper QoS management of aggregate flows whenever the underlying OS mechanisms or network traffic shaping mechanisms impose upper bounds on aggregated flow bandwidth (as in proportional share OS resource allocation, and weighted fair queuing).

To reuse today's best-effort server code, where a single pool of threads or processes serves all requests, our policing solution essentially relies on request admission control to make the outgoing aggregate flow bandwidth conform to specifications. Since the resource requirements imposed by a single service request are often unknown, *measurement-based* admission control is used to regulate the fraction of admitted requests based on bandwidth measurements. In web servers, the length of requested URLs, for example, can differ by a couple of orders of magnitude from one request to another. Measurement-based admission control will be effective as long as the resource requirements of the largest request remain a small fraction of the total server capacity, making it possible to apply the laws of large numbers to aggregated flows and use a fluid flow model. In Chapter 2 we detailed a a measurement-based admission control technique that relies on a control loop which adjusts the fraction of admitted requests and the degree of degradation depending on load conditions.

An important question with admission control is which requests to reject. One might argue that indiscriminate rejection is acceptable within the same traffic class. This, however, is not always a good policy. Consider a web server hosting two sites, A and B, and assume that the request rate on site A has increased beyond its contracted limit. Assume all clients of site A are equal. If rejection is indiscriminate, the rejected requests may originate from any client. If clients' sessions are comprised of multiple requests, all clients are likely to see rejections during the session, and none is able to complete their session satisfactorily. From the perspective of maximizing the aggregate utility of the service, it may be more desirable to adjust admission control such that a subset of clients are rejected consistently under overload, while others see consistently good service (i.e., actually complete their sessions). In [5] such an admission control mechanism is described, where the pool of consistently admitted clients is sized dynamically based on measured server load.

Both policing and measurement-based admission control can be used to implement service differentiation. A low-priority aggregate flow is forced (by either method) to fit within the bandwidth left unused by higher-priority flows. This mechanism successfully “fakes” priorities in middleware, even when identical same-priority threads handle different traffic classes in the server. It requires an estimate of aggregate server capacity as well as a measurement of the bandwidth used by each priority class. While the approach works well for a limited number of classes, it is unclear if can scale up to a large number of priority levels because admission control in each level depends on measurements of all higher priority levels, and measurement accuracy decreases with the granularity of a traffic class.

Another important problem that arises with flow prioritization is that of consistent flow-priority management across multiple resources. Higher-priority flows should receive precedence over lower-priority flows in accessing not only the CPU resources, but also memory buffers, disks, and communication bandwidth. Unless the CPU is the bottleneck resource, threads will block waiting on the real bottleneck which must then be allocated in the same priority-sensitive fashion.

Classification of requests into one of several flows poses a particularly important concern. Such classification is often dependent on application-specific data (e.g., the site named in the HTTP header). Implementing request classification in the socket library transparently to the server may result in a large run-time overhead since it will require parsing all requests and interpreting their content (such as the HTTP header fields) in the context of $read()$ calls. Classification, however, is already performed in the context of usual server execution, so it is advantageous to have access to server code. Unfortunately, performing classification in user space (i.e., in the server or middleware) is not efficient for another reason. Since admission control is performed *after* classification (to discriminate among different classes), rejected requests will have consumed a substantial amount of resources in the kernel by the time they are rejected by the server. A substantial amount of execution resources can thus be wasted on eventually rejected connections. This concern calls for OS support for early classification in the kernel.

Finally, if *Adaptware* is called from within a socket library, care must be taken to call it only on socket operations related to the server. Thus, only the server should be linked with the new library. Furthermore, only the sockets representing connections with clients should activate *Adaptware* API. The library must keep track of these sockets and distinguish between them and regular file descriptors (e.g., log files) or other communication (e.g., with CGI scripts). A possible solution is to indicate the server's well-known port in the middleware's configuration file. Only operations on sockets whose descriptors are returned by an $accept()$ call to the well-known port should invoke *Adaptware* API.

4.9 Conclusions

We presented a new abstraction and structuring methodology for communication subsystems on end-hosts based on a more flexible notion of a QoS contract. We elaborated on the QoS specification API which allows the application developer to define flexible communication requirements by specifying a set of acceptable QoS levels, rewards, and QoS violation penalty. We presented a thread-per-contract architecture that enforces the QoS contract. We described an implementation of contract handlers as *q*threads scheduled by the communication subsystem to maximize aggregate reward. Scheduling contract handlers involves appropriate selection of a QoS level for each handler in accordance with current resource load and capacity. We described a mechanism for QoS-level selection based on a combination of periodic optimization and fast heuristic adjustment. A reward optimization algorithm uses a dynamically computed estimate of per-packet handling cost to determine the QoS-levels. A simple heuristic was described that dynamically adjusts the selected QoS levels in response to transient load disturbances. We discussed mechanisms for overload and underutilization detection, as well as policies for adjusting QoS levels in response to such conditions. A communication server architecture, called *CLIPS*, was described and contrasted against a middleware implementation, called *qContracts*. The impact of such mechanisms and policies has been briefly illustrated using experimental results from an actual implementation of the communication subsystem. The analysis shows that the design is capable of meeting its stated goals regardless of the bottleneck resource, platform speed, and load mix.

CHAPTER 5

ADAPTATION IN DISTRIBUTED SYSTEMS

5.1 Introduction

This chapter extends the QoS adaptation framework to achieve predictability and graceful degradation in long-lived *distributed* real-time services. By "long-lived" we mean that a service request, if granted, will hold its resources for a relatively long period of time. Predictability in real-time applications has traditionally been achieved by reserving resources and employing admission control under some *a priori* assumed load and failure conditions. Graceful QoS degradation, on the other hand, requires resources to be reallocated dynamically in order to cope with changing load and failure conditions while maximizing system utility. Both predictability and graceful QoS degradation are necessary for real-time applications, but pose conflicting requirements. The proposed framework attempts to reconcile these requirements in distributed applications.

We incorporate the notion of QoS contracts proposed earlier into a processing capacity management middleware service called *RTPOOL*. The service is designed and implemented to support timeliness guarantees for distributed real-time applications. We review the proposed QoS-adaptation model in the context of *RTPOOL* (Section 5.2), then describe *RTPOOL* architecture highlighting the synergy between components of the service and the QoS-negotiation support (Section 5.3). We present relevant details of *RTPOOL* implementation and the negotiation API (Section 5.4), and describe the use of *RTPOOL* in the context of an automated flight control application (Section 5.5). Flight performance is evaluated (Section 5.6), illustrating the efficacy of QoS adaptation support, followed by a brief summary of contributions (Section 5.7).

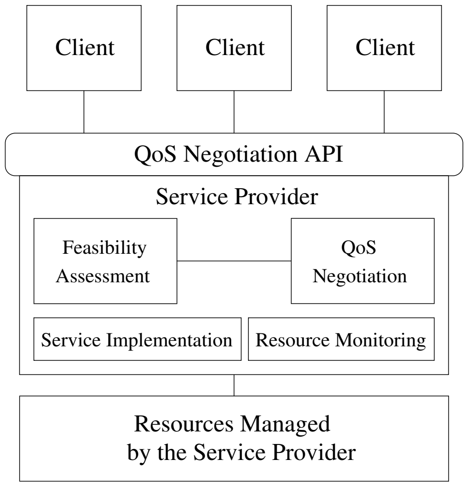


Figure 5.1: Service provider architecture.

5.2 Model

We consider a class of distributed real-time systems in which various software components perform tasks to accomplish a single overall "mission." We will henceforth call this mission an *application*. Flight control, shipboard computing, automated manufacturing, and process control applications generally fall under this category. The application is composed of a set of tasks, each of which requires a set of resources/services. Following the main QoS contract model of *Adaptware*, a client requesting service specifies in its request a set of QoS levels and their rewards to the service provider, as well as a rejection and violation penalty. To control system load in a way that ensures predictable service, the service provider must subject the client's request to on-line admission control which determines whether to guarantee or reject the request. The architecture of the service provider is given in Figure 5.1. The provider runs on top of a pool of resources whose size may vary dynamically, and serves a dynamic set of real-time clients. The underlying resources available to the provider are monitored by the resource monitoring module. The provider exports a QoS-negotiation API to its clients based on QoS levels, rewards and penalties. The QoS-negotiation module is responsible for selecting the appropriate QoS level for each client so that overall utility is maximized. The feasibility assessment module is responsible for checking whether or not the selected QoS levels of the respective clients can be sustained using currently-available resources. Assisted by the feasibility assessment module, the QoS-negotiation module performs admission control on incoming service requests.

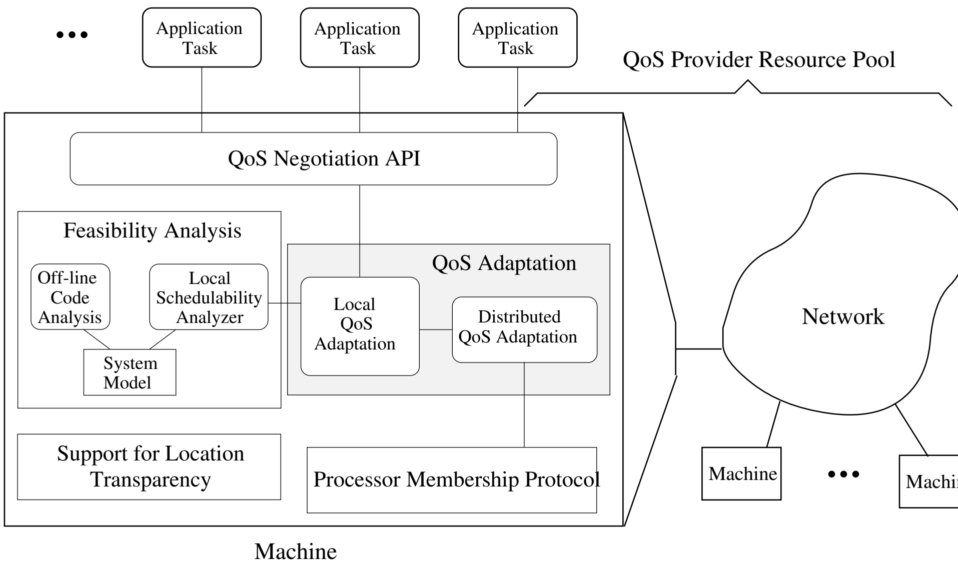


Figure 5.2: General overview of *RTPOOL*.

5.3 RTPOOL

We designed an instance of the service provider we call *RTPOOL*, shown in Figure 5.2, to export the abstraction of a single distributed computing resource. This service is responsible for load sharing on a pool of processors to guarantee timeliness. It employs a *processor membership protocol* that keeps track of processor pool membership and reports processor failures. In order to provide timeliness guarantees, schedulability analysis is used.

Clients of *RTPOOL* are application tasks. *RTPOOL* service requests are used to guarantee the timeliness of new incoming tasks. *RTPOOL* assumes periodic tasks, while handling aperiodic tasks with periodic servers. A task is composed of a set of modules, and has a deadline by which all of its modules must be completed. The modules may have arbitrary precedence constraints among themselves, thus specifying their execution sequence. We assume that task arrivals (guarantee requests) are independent, so we do not support precedence constraints among *different* tasks.

Each task's request includes its rejection penalty, and parameters of its QoS contract that specify different QoS levels and their respective rewards. A client task's QoS level is specified by the parameters of its execution model. For an independent periodic task, the parameters consist of task period, deadline, and execution time. We model period and deadline as negotiable parameters. This represents a significant departure from most scheduling literature, although the authors of [115] articulate on the alterability of task periods in real-time control systems using system stability and performance index. Task execution time, on the other hand, depends on the underlying machine speed and thus should not be hardcoded into the client's request. Instead, each QoS level in the negotiation options specifies which modules of the client task are to be executed at that level. This allows the programmer to define different versions of the task to be executed at different QoS levels, or to compose tasks with mandatory and optional modules. The reward associated with each QoS level tells *RTPOOL* the utility of executing the specified modules of the task with the given period and deadline.

Requests for guaranteeing tasks may arrive dynamically at any machine in the pool. Tasks normally receive higher QoS than their minimum QoS level. It is therefore highly probable for the new arrival to be guaranteed on the local machine. To guarantee a request at the local machine, *RTPOOL* executes a *local QoS-optimization heuristic*. The heuristic (re)computes the set of QoS levels for all local clients (including the new one just arrived) which maximizes the sum of their rewards. Recomputing the QoS levels may involve degrading some tasks to accommodate the new one. The task is rejected if *both* (i) the new sum of rewards (including that of the newly-arrived task) is less than the existing sum prior to its arrival, *and* (ii) the difference between the current and previous sums is larger than the new task's rejection penalty. Otherwise, the requested task is guaranteed. As a result, task execution requests will be guaranteed unless the penalty from resulting QoS degradation of other local clients is larger than that from rejecting the request. When a task execution request is rejected by the local machine, one may attempt to transfer and guarantee it on a different machine using a load-sharing algorithm.

Note that conventional admission control schemes would always incur the request rejection penalty whenever an arrived task makes the set of current tasks not schedulable. By offering QoS degradation as an alternative to rejection we can show that the reward sum (or perceived utility) achieved using our scheme is *lower bounded* by that achieved using conventional admission control schemes given the same schedulability analysis and load sharing algorithms. Thus, in general, our proposed scheme achieves higher perceived utility.

Figure 5.3 gives an example of the local QoS-optimization heuristic. The heuristic implements a gradient descent algorithm, terminating when it finds a set of QoS levels that keeps all tasks schedulable, if any. Note that unless all tasks are executed at their highest QoS level, the machine suffers from *unfulfilled potential reward*. The unfulfilled potential reward, UPR_j, on machine N_j, is the difference between the total reward achieved by the current QoS levels selected on the machine and the maximum possible reward that would be achieved if all local tasks were executed at their

```
Let each client task T_i have QoS levels M_i[0],...,M_i[best_i] with
rewards R_i[0],...,R_i[best_i], respectively.

  1. Start by selecting the best QoS level, M_i[best_i], for each
     client T_i.

  2. While the set of selected QoS levels is not schedulable, do
     Steps 3 and 4.

  3. For each client T_i receiving service at level M_i[j] > M_i[0],
     determine the decrease of local reward, R_i[j] - R_i[j - 1], re-
     sulting from degrading this client to the next lower level.

  4. Find client T_k whose R_k[j] - R_k[j - 1] is minimum and degrade it
     to the next lower level.

  5. Go to Step 2.
```

Figure 5.3: Local QoS optimization heuristic

```
  1. On source machine, N_i, find client T_k whose removal will
     result in maximum increase, W, in total reward.

  2. N_i request reassigning T_k, with reward W.

  3. Each machine N_j, where UPR_i - UPR_j > V, receives the re-
     quest and recomputes QoS levels for its local clients plus
     T_k.  If its total reward is higher with T_k, N_j bids for T_k
     with the reward increment W_j resulting from accepting it.

  4. N_i transfers T_k to highest bidder.
```

Figure 5.4: Distributed QoS optimization protocol

highest QoS level. This difference can be thought of as a fractional loss to the mission. Often this loss is unavoidable because of resource limitations. However, such loss may also be caused by poor load distribution, in which case it can be improved by proper load sharing.

RTPOOL employs a load-sharing algorithm that implements a *distributed QoS-optimization protocol*. The protocol uses a hill climbing approach to maximize the global sum of rewards across all clients in the distributed pool. It is activated between two machines N_i and N_j when the difference $UPR_i - UPR_j$ exceeds a certain threshold V. The protocol is given in Figure 5.4.

Close examination of the local QoS optimization heuristic and the distributed QoS optimization protocol reveals that neither makes assumptions about the nature of the client and the semantics of its QoS levels.[1] For *RTPOOL* this means complete independence between the task model used by the feasibility assessment module, and the QoS-negotiation mechanism. As a result, it is easier to enhance *RTPOOL* to handle more elaborate task models, constraints, and QoS-level parameters/semantics without affecting its QoS-negotiation mechanism. The disadvantage of this separation of concerns compromises optimality somewhat, as illustrated by example in Section 5.6.

5.4 Implementation and API

In this section we highlight implementation details of the *RTPOOL* service, particularly those related to its QoS-specification API. *RTPOOL* is currently running on top of TOG Mach RT, MK7.2, on a PC platform, and is implemented as a user-level library, which exports the abstraction of tasks, threads, QoS levels, and rewards. Highlighted below are the components of the implemented prototype.

5.4.1 Scheduling and QoS Negotiation

We use *q*threads, described in Chapter 4, for thread scheduling and QoS adaptation support. The package implements a user-level local scheduler on each machine. The local scheduler is the lowest layer of *RTPOOL*. It supports quasi-periodic threads, whose period can be changed at run-time in response to changes in the QoS level.

On top of quasi-periodic threads, we export the abstraction of tasks, QoS levels, and rewards. Its API permits the user to create tasks, create threads within each task, define QoS levels for the task, and specify rewards. It also permits the user to specify for a given thread the QoS levels in which the thread is eligible to execute. When new load (i.e., task or a set of tasks) arrives it is submitted to the system via a corresponding primitive that invokes admission control. As a result, QoS levels are re-calculated, and a new value for unfulfilled potential reward is computed.

Dynamic arrival of a task at the local machine occurs when (i) another task requests it to be executed there, or (ii) the load-sharing heuristic migrates it from another machine to the one under consideration. QoS levels are also recomputed on a machine when its local load decreases. This occurs when (i) a periodic thread is terminated (i.e., will no longer be invoked periodically), or (ii) a task is migrated away from the local machine. Once a set of new active QoS levels has been

[1] The distributed QoS-negotiation protocol, however, assumes service to a given client can be migrated to another node.

established, the *q*threads package adjusts the parameters of the periodic threads known to the local scheduler. This includes modifying their deadlines and periods when applicable. In the current implementation, all created tasks execute in the same address space. The application is compiled into a single executable image at system start time.

5.4.2 Invocation Migration

On top of *q*threads we provide an invocation migration mechanism to implement the distributed QoS optimization protocol described in Section 5.3. The mechanism is completely transparent to the application. We call it *invocation* migration, because the transfer occurs between two successive invocations of a periodic task (i.e., when one invocation has terminated and the next hasn't started yet).

When the distributed QoS optimization heuristic determines that a task is to be migrated, the *state variables* of each thread in the transferred task are sent to the new machine, the threads belonging to the task are destroyed at the source and recreated with the transferred state at the target. A state variable is one whose value needs to be preserved across successive invocations. In the current implementation, state variables of a thread have to be indicated to *RTPOOL* using a corresponding library call at thread initialization time. The QoS levels are recomputed on the source and target after the transfer to update the QoS levels accordingly. If a task must always execute on a certain machine (e.g., because of some I/O devices it needs on that machine), the task can be *wired* to that machine by calling a *wire_task()* primitive.

5.4.3 Pool Membership API

A membership algorithm is used to maintain a consistent view of the current membership of the shared resource pool. Our group membership algorithm is a derivative of [2]. The only interface the user sees to that algorithm is the *subscribe_to_pool()* call which causes the machine on which the call was executed to join the named pool. Typically, when the application starts up on a given machine, it executes the join primitive which broadcasts a join request to the given pool. If the request is not answered (i.e.,the pool doesn't yet exist), the machine creates a singleton pool of itself and attempts to run the application on its own. The machine designates itself as the *group leader* for the pool (initially, consisting of itself only). When a group leader receives a join request from a different machine, it broadcasts a membership message [2] to all group members including the new machine.

When a new machine subscribes to (joins) the pool each machine in the pool adds the new member to the group. Since the new machine does not run any application task, its unfulfilled potential reward is zero. Due to our load-sharing heuristic, machines whose unfulfilled potential reward is above a given threshold will attempt to offload tasks to the new member. The new machine accepts tasks one at a time, recomputing QoS levels after each arrival, and recomputing its unfulfilled potential reward. Task transfer will continue until the unfulfilled potential reward is balanced among all machines within a certain threshold, which stops the distributed QoS optimization.

When a machine fails, its failure is detected via the membership protocol and broadcast to all group members. Task assignment information is replicated on each machine in the pool. The group leader (the machine with the highest number in the pool) re-creates the destroyed tasks. The load-sharing heuristic redistributes the load, if necessary. When the group leader fails, its successor (the machine with the next highest number in the pool) becomes the leader. Task state is lost in crash. Loss of state can be avoided by task replication. Wile the current implementation does not support it, it is easy to extend this implementation to replicate tasks such that the state of failed tasks may be retrieved from their replicas.

5.4.4 Communication API

An application need not be aware of where each of its tasks is executing. The decision of where to run each task is left up to the load-sharing heuristic. This requires location-independent communication primitives. Tasks communicate in our architecture via location-independent *send()* and *receive()* primitives which determine the destination from the target task identity then use the communication subsystem architecture described in Chapter 4 to send messages to the recipient. Our communication protocol stack is implemented using x-kernel 3.2 [60], and is layered on top of a UDP/IP stack. Based on *CLIPS*, the communication subsystem architecture on each host supports prioritized, bounded-time message delivery which enables *RTPOOL* to determine the effect of communication delays on the schedulability of periodic tasks as will be described in the next chapter which details schedulability analysis.

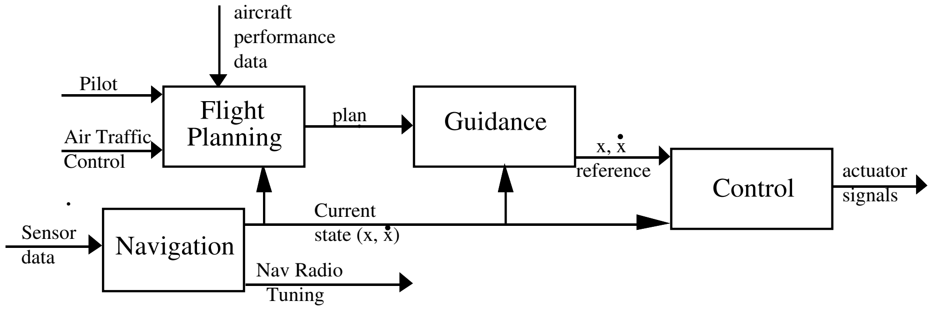


Figure 5.5: Flight management system functions.

5.5 Application

We have used *RTPOOL* to provide negotiable timeliness guarantees for several real-time tasks required in a fully-automated flight control system.[2] The system was used to fly a simulated model of an F-16 fighter aircraft. Details of the automated aircraft flight problem are provided in Section 5.5.1, followed by a description of a method to determine the involved task QoS levels and rewards from application domain knowledge (Section 5.5.2). Section 5.5.3 summarizes the set of tasks, QoS levels, and rewards that describe the application.

5.5.1 Automated Flight Control

Current Flight Management Systems (FMS) perform several flight control functions, including flight planning, navigation, guidance, and control [79]. Figure 5.5 illustrates these FMS tasks and their interconnections; details of each module are provided in [79] and [110]. In current FMS, real-time execution guarantees exist for the navigation, guidance, and control modules, allowing critical function deadlines to be met. Schedulability guarantees for these systems are typically computed off-line. Our QoS-negotiation scheme will allow the system to gracefully degrade performance when enough resources are lost to violate the off-line guarantees. We consider the case where all tasks have a known bounded execution time. Issues in dealing with potentially unbounded on-line computations, such as run-time intelligent mission planning, are discussed in [14].

Aircraft guidance commands are typically issued in terms of aircraft altitude (z), and compass heading (h). To control a simulated F-16 aircraft we employ a control loop to compute the continuous-valued commands for a set of *primary* actuators, including the elevator, ailerons, rudder, and throttle. The elevator, ailerons, and throttle generate aerodynamic forces which directly

[2]The application described in this section is given by Ella Atkins who collaborated with the author on testing *RTPOOL*.

affect aircraft roll and pitch attitude, and, via dynamic coupling, indirectly affect aircraft heading and airspeed. Our controller is also capable of commanding a *secondary* set of actuators whose effect is to achieve better flight performance, but that are not as critical for flight safety. Secondary actuators include the F-16's afterburner which generates extra engine thrust, as well as the flaps and speed brake, used to provide extra control for airspeed and altitude.

In a parallel research effort [13] a set of linear controllers have been implemented to calculate the *primary* actuator commands to achieve the desired reference altitude (z_{ref}) and heading (h_{ref}) for the aircraft. The inputs to the controllers include sensor measurements of current aircraft altitude z, heading h, pitch angle p and roll angle r. Equation (5.5.1) shows the control laws used during our experiments, adopted from those used in [14].

$$\begin{pmatrix} elevator \\ ailerons \\ rudder \end{pmatrix} = \begin{pmatrix} K_1 & 0 & -K_{p_1} & -K_{d_1} & 0 & 0 \\ 0 & K_2 & 0 & 0 & -K_{p_2} & -K_{d_2} \\ 0 & K_3 & 0 & 0 & -K_{p_3} & -K_{d_3} \end{pmatrix} \begin{pmatrix} z_{ref} \\ h_{ref} \\ p \\ \dot{p} \\ r \\ \dot{r} \end{pmatrix}$$

In the higher-performance controller modes (see Section 6.3), the system also exerts control over the set of *secondary* actuators. We control each with a discrete-valued command so that each assists the continuous-valued *primary* actuators appropriately. For details on the discrete control law refer to [13].

5.5.2 Computing QoS Levels and Rewards

Our QoS-negotiation scheme enables the application domain expert to express application-level semantics to *RTPOOL* using QoS levels, rewards and rejection penalty. In this section we briefly highlight how this support can be taken advantage of using the analytic techniques developed in the context of CIRCA (the Cooperative Intelligent Real-time Control Architecture) [14]. Based on a user-specified domain knowledge base, CIRCA's main goal is to build a set of control plans to keep the system "safe" (i.e., avoid catastrophic failures such as an aircraft crash) while working to achieve its performance goals (e.g., arrive at its destination on time). In order to deal successfully with an inherently non-deterministic, perhaps poorly modeled, environment of a complex real-time system CIRCA employs *probabilistic* planning which models the system by a set of states and transition probabilities. System failure is modeled by temporal transitions to failure states (TTFs). CIRCA's mission planner uses its domain knowledge base to compute the actions to be taken (set of tasks) and their timing constraints (QoS levels), so that the probability of TTFs is reduced below a certain threshold. The reward decrease corresponding to degrading a task from one QoS level to another, or rejecting a task altogether, is computed from the corresponding increase in failure probability.

For example, the planner computes a maximum period for each task based on the notion of preempting TTFs [13]. For any state, an outgoing TTF is considered to be preempted if its probability (computed by CIRCA using transition temporal properties and any applicable task's maximum period) is below the specified probability threshold value. To define alternative QoS levels, CIRCA's planner may compute task alternative periods based on a set of alternative TTF probability thresholds. For example, say a TTF has a cumulative probability distribution that reaches the threshold value when the preemptive task's maximum period is set to 0.2 seconds. But, suppose we may need to relax the task's period requirement under overload. The period of the degraded QoS level is computed from the next higher probability threshold level, and this task is assigned a lower reward that corresponds to the reduction in certainty that the TTF will be preempted. A complete set of QoS levels may be developed by considering each pre-specified probability threshold for each TTF, with one task QoS level associated with each TTF probability threshold value.

5.5.3 Description of Flight Tasks

We have used the Aerial Combat (ACM) F-16 flight simulator [100] for all flight tests, running it on a Sun workstation connected via sockets to the real-time execution platform. We have tested the QoS-negotiation capabilities by flying the simulated aircraft around the pattern illustrated in Figure 5.6. In this pattern, the aircraft executes a takeoff and climb, then holds a constant altitude as it continues around its rectangular course. When turning to final approach, the aircraft descends, using radio navigation sensors (i.e., the ILS) to guide it accurately to the runway. By varying periods of the controllers, course navigation system (i.e., the high-level task that commands the aircraft altitude and heading), and the sensors, we were able to observe the degradation in flight quality (i.e., stability) as a function of each task's selected QoS level.

In this section, we describe the tasks and associated rewards used during our tests of the QoS negotiation algorithms. The goals of our example mission were to complete the flight around a rectangular pattern (illustrated in Figure 5.6), and to destroy observed enemy targets, if any, using the simulated F-16's onboard radar and missiles. Four separate tasks were required to control the aircraft during flight: "Guidance", "Control", "Slow Navigation", and "Fast Navigation." These

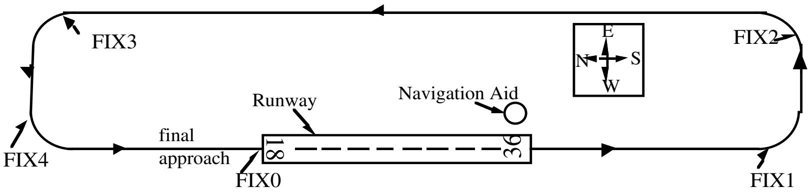


Figure 5.6: Aircraft flight pattern flown during testing.

tasks function much like their similarly-named counterparts in Figure 5.5. The "Guidance" task is responsible for setting the high-level reference trajectory of the aircraft. For our tests, this task specified quantities such as heading and altitude to guide the aircraft from takeoff through landing. The "Control" task is responsible for executing the low-level control loops that compute actuator commands from the commanded high-level trajectory. We have two "Navigation" tasks that read sensor values, distinguished by the update frequency required, and command radio frequency changes as required. The navigation sensor values are used by the "Guidance" task to determine when and how to alter the commanded trajectory, and are used as standard state feedback by the "Controller" task.

Table 5.1 shows the set of QoS levels present for all tasks, including the associated reward, execution time, and period. In our simple set of tests, we set each task deadline equal to its period, although there are no such requirements in our QoS negotiation protocol. Also, because each of these tasks is considered critical to execute (at least at a degraded QoS level), we set all task rejection penalties sufficiently high such that all tasks are always accepted by the QoS negotiator.

In addition to the basic flight control tasks discussed above, we simulate a function necessary during military operation: "Missile Control." The "Missile Control" task is composed of two precedence-constrained threads: "Read Radar" and "Fire Missile". The "Read Radar" thread monitors aircraft radar to detect approaching enemy targets, then, if a target has been detected, the "Fire Missile" thread is used to launch a missile at any enemy targets that appear on radar. As shown in Table 5.1, the simulated "Missile Control" task is computationally expensive and has two QoS levels. If Level 1 is possible, radar will be scanned with sufficient frequency to allow most any enemy target to be detected and destroyed. Otherwise (level 0), fast-moving targets may not be destroyed. During experiments (in Section 7.3), we varied the reward for "Missile Control" QoS Level 1 depending on the "subjective" relative importance of taking down enemy targets vs. flight control performance.

As described above, the "Controller" task is responsible for executing the control loop. At each invocation, the controller uses the currently stored sensor values to compute the appropriate

Task	L	R	ET(ms)	P(sec)	Ver
G	0	10	100	10	def
	1	15	100	5	def
	2	20	100	1	def
C	0	1	80	5	sec
	1	100	60	1	prim
	2	104	80	1	sec
	3	120	60	0.2	prim
	4	124	80	0.2	sec
SN	0	10	100	10	def
	1	20	100	5	def
	2	25	100	1	def
FN	0	1	60	5	def
	1	100	60	1	def
	2	120	60	0.2	def
MC	0	1	500	10	def
	1	30/200	500	1	def

Table 5.1: Flight plan with different QoS levels.

actuator values to control both aircraft attitude and position/velocity. Two versions of this function were tested, one that used the *secondary* actuators (QoS levels 2 and 4) and one that did not (QoS levels 0, 1, and 3). Use of these actuators allows the aircraft to perform better in terms of takeoff distance and climb rate as shown in Section 5.6 below, but these actuators require a larger task execution time and are not critical for maintaining safety. The importance of the control loop period is illustrated by the relatively high reward given to the low-period QoS levels for the "Controller" task, and the small reward changes between the use of the different versions (e.g., level 3 vs. level 4) reflects the fact that the difference in choice of version is not critical for safety.[3]

Next, the "Slow Navigation" task is responsible for reading those sensors that do not require a high sampling rate because they either are not used for the control loop or else they do not change

[3]We defined a QoS "level 0" for the "Controller" and "Slow Navigation" tasks that, as will be shown in Section 5.6, were so slow that the aircraft becomes unstable during turning maneuvers. These levels are included among their task's QoS negotiation options for illustrative purposes only, and would be there otherwise.

rapidly. All navigation sensors are grouped into this task because they are used by the "Guidance" task to determine the high-level altitude and heading commands, but not by the more safety-critical control loop task. The Table 5.1 reward/period values for "Slow Navigation" reflect the non-critical nature of this task. Finally, the "Fast Navigation" task is responsible for maintaining all sensor data used by the "Controller" task. Since the system must read this data to maintain sufficient state variable accuracy, the periods and rewards are similar in structure to that used by the "Controller" task (although we had only one version of this task, since a bulk sensor data read was the only computational task required).

5.6 Evaluation

We have concentrated our test efforts on demonstrating the use of the QoS negotiation architecture for the one particular example describe above: aircraft flight control. In this section, we show test results which illustrate specifically how QoS negotiation can help flight control degrade gracefully. In Section 5.6.1, we assess the accuracy of the QoS negotiation heuristic for our set of flight tasks by observing QoS degradation of the same task set with lower machine speeds. In Section 5.6.2 we study aircraft performance during flight as a function of the "Controller" task's QoS level, illustrating one example in which the QoS negotiator allows graceful performance degradation. In Sections 5.6.1 and 5.6.2, we focus on tests which use a single machine, and consider only the flight guidance, navigation, and control tasks. We conclude our experiments (Section 5.6.3) with tests which employ the full set of flight and missile control tasks from Table 5.1 and observe the effects of load sharing between two machines. As shown in Section 5.6.3, our system exhibits graceful performance degradation upon machine failure.

5.6.1 QoS Negotiation Heuristic Testing

In Section 5.3, we described a simple local QoS optimization heuristic to help a service provider select a high-reward set of QoS levels for its clients. Using the QoS levels and rewards listed in Table 5.1, we illustrate the behavior of the presented heuristic. In this experiment we kept the task set fixed, and decreased the underlying CPU speed (increasing task execution times), then observed the corresponding decrease in task QoS levels. Figure 5.7 plots the observed QoS levels versus CPU speed, normalized by the minimum CPU speed for which the task set is schedulable.

Since, the heuristic uses only reward information to guide its search for a feasible set of QoS

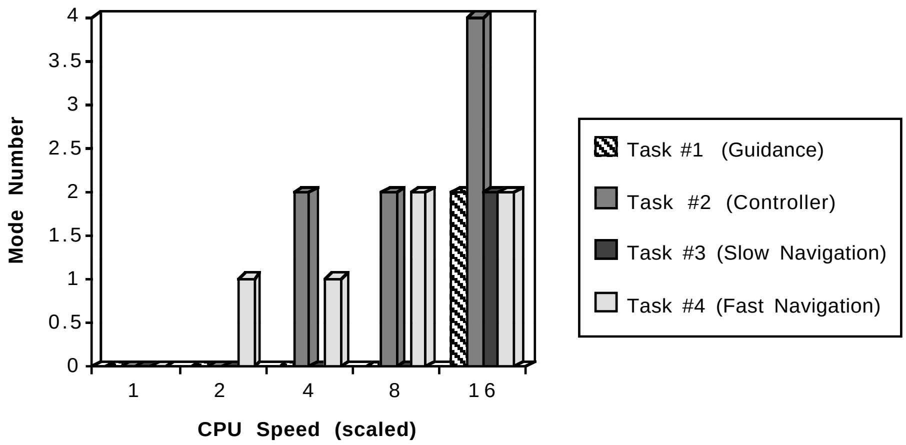


Figure 5.7: QoS levels selected vs. CPU speed for flight tasks.

levels (thus being applicable as is in any service that uses our QoS negotiation scheme) optimality is compromized yet "graceful degradation" of QoS modes is still illustrated.

5.6.2 Aircraft Performance

We evaluated the performance of our system by studying its ability to control the aircraft simulator during flight. In this section, we consider only the flight control tasks as they execute on one machine, saving discussion of the load sharing protocol and missile control task for the next section. As shown in Figure 5.7, since the "Controller" and "Fast Navigation" tasks required the smallest execution period, these tasks are the bottlenecks for execution, so changes in aircraft performance are most easily observed by looking at changes in QoS levels for these tasks. Since these tasks are tightly coupled (i.e., the "Controller" task uses results from "Fast Navigation"), we considered a matrix of test cases in which we varied the "Controller" QoS level from its highest value (4) to the lowest specified level (0), and simply ensured that the "Fast Navigation" level acted with at least as low a period as was present in the "Controller" level.

As shown in Table 5.1, the "Controller" task QoS levels are a function of two variables: task period and version. We look at tests illustrating the major differences in performance for each of these variables in this section. Figure 5.8 shows aircraft altitude during the takeoff and climb phase of flight. This figure illustrates the main performance difference between the two versions of the task in the "best performance" case of each (i.e., period = 200 msec). In level 4, the afterburner and flaps are operational, which requires a larger "Controller" task execution time, while in level 3, the afterburner and flaps remain off. As shown in the Figure, the afterburner and flaps allow the

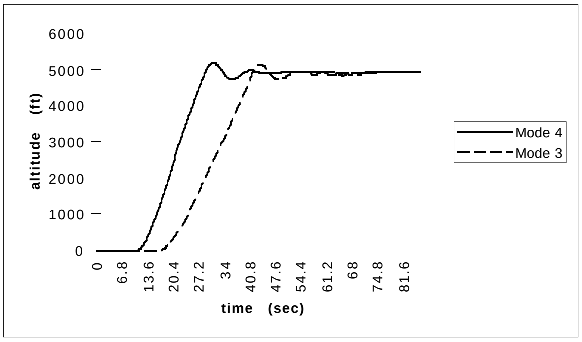


Figure 5.8: Aircraft altitude performance with and without "extra" control actuation.

aircraft to take off and climb more quickly, but both achieve the stable altitude of 5000 ft. eventually. This example illustrates why the rewards for these different versions are not very different, since performance for level 4 is only slightly better than that in level 3. This example illustrates how QoS negotiation can achieve graceful degradation. Overall processor utilization is decreased by reducing the "Controller" task to level 3, but safety (i.e., controller stability) is not compromised.

In the next series of tests, we studied aircraft performance as "Controller" QoS level changes affect the task period. For these tests, we isolated version from period modification effects with a tests series in which the afterburner and flaps were consistently actuated (levels 0, 2, and 4). Similar trends result using the other task version (levels 1 and 3). To illustrate the changes, we consider three different QoS levels: level 4 with a period of 0.2 seconds (200 msec), level 2 with a period of 1 second, and level 0 with a period of 5 seconds. We include level 0 among the Controller's negotiation options for illustration purposes only, as a comparative example showing controller instability. Of course, no unstable QoS levels should be defined among a client's negotiation options, since the client should not "ask" for instability as a service option.

Figures 5.9 through 5.12 show state variables as a function of time from takeoff and climb through the turn to a heading of East after reaching FIX 1 (as shown in Figure 5.6). Figure 5.9 shows the aircraft altitude for the different controller periods. The plot for Period=0.2 (QoS level 4) is the "best" case; the aircraft climbs quickly to 5000 ft and stabilizes there. When the controller period is increased to one second, the aircraft is still stable in altitude, but it is much more sluggish, climbing to altitude more slowly and taking longer to stabilize. The range of periods between 0.2 and 1.0 seconds illustrates graceful degradation, since all will result in a stable controller, just not as capable of responding quickly. Finally, to illustrate the necessity of real-time response, we tested the same controller at a period of 5.0 seconds (QoS level 0). In this case, the controller becomes unstable and the aircraft eventually crashes.

Figure 5.10 shows aircraft heading as a function of time for the three different “Run Controller” periods during takeoff, climb, and turn from a heading of South to a heading of East. The change in Period from 0.2 sec to 1.0 sec simply slows system response, but a controller period of 5.0 seconds drives the system unstable.

Figures 5.11 and 5.12 show plots of pitch angle and roll angle, respectively, for the two “stable” controller periods. We do not include period = 5.0 seconds here because the instability obscures the other plots. Since pitch angle and altitude are coupled, the pitch angle has largest magnitude whenever the altitude is climbing (or descending), and as illustrated in this plot, the increase in period to 1.0 seconds causes a large pitch angle to be required for a longer time, a stable but undesirable performance trait for aircraft flight. The roll angle plot (Figure 5.12) also shows a delay and longer roll angle deviation from 0.0 for the slower-period control cycle. Additionally, Figure 5.12 illustrates another possible problem with the slow period: overshoot. As shown in the Figure, with a period of 0.2 seconds, the roll (or bank) angle quickly reaches the desired value for a constant-rate turn from South to East. However, bank angle for the slower controller far overshoots its desired value before the controller notices and reacts. In both cases, the controllers are stable, but the pilot and any passengers would certainly not be very happy about the rather severe bank angle temporarily achieved (and if the period were much slower at all, the bank overshoot would be sufficiently large to drive the aircraft out of the our controller’s stable region).

5.6.3 Load Sharing

Load sharing capabilities have been implemented in RTPOOL, and we have performed a final set of tests which included both the flight control tasks (with performance characteristics shown above) and a missile control task as described in Section 5.5.3. In these tests, we start the system with two machines available for task execution. Because, as defined in Table 5.1, the missile control task was computationally expensive, the load sharing protocol places all flight control tasks on one machine and the missile control task (both the “read radar” and ”fire missile” threads) on the other machine.

When the two machines function normally, both the flight and missile control tasks ran in their maximum performance levels. In this case, enemy targets are quickly detected and fired upon, while flight control is identical to the best performance profiles in the Section 5.6.2 plots. For the next test set, we began operation with two functioning machines, then shut one down (simulating machine

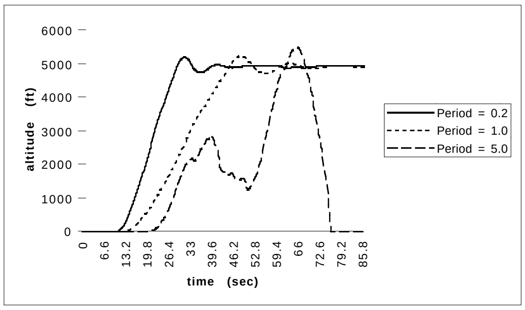


Figure 5.9: Aircraft altitude performance for different controller task levels.

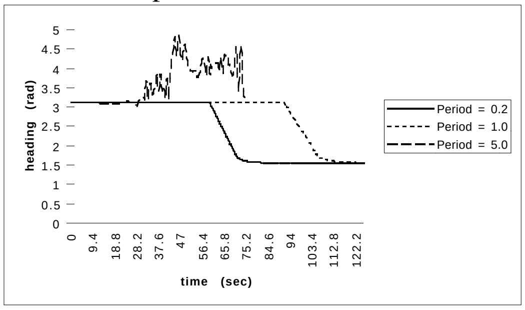


Figure 5.10: Aircraft heading performance for different controller task levels.

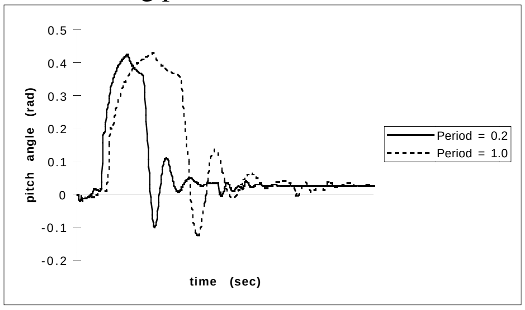


Figure 5.11: Aircraft pitch performance for different controller task levels.

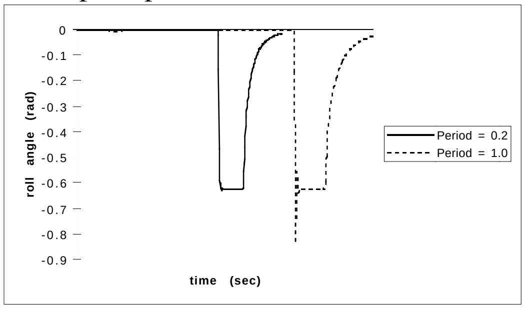


Figure 5.12: Aircraft roll performance for different controller task levels.

failure) just after takeoff. This requires the load sharing and QoS negotiation algorithms to function dynamically, such that the one functional machine now has to execute both the flight and missile control tasks. If “Missile Control” reward is relatively low (we set it to 30), the system chooses to degrade the “Missile Control”, “Guidance”, and “Slow Navigation” functions (to level 0), but manages to keep the “Controller” and “Fast Navigation” tasks at safe levels. In this manner, the flight control is a bit sluggish but stable However, the aircraft is unable to launch missiles at most targets.

Alternatively, this system may be aboard an expendable drone whose most important function is to destroy a target or attack enemy aircraft. In this case, the reward set may be structured such that the missile control task takes precedence over accurately maintaining flight control.[4] To illustrate such changes in the task reward set, we altered the reward for QoS level 1 of the “Missile Control” task to 200 (as shown in Table 5.1). Now, when the second machine shuts down, the QoS negotiator reduces all flight control levels to 0, since the missile controller is perceived as the most important task. After one machine fails, the aircraft eventually becomes unstable (as illustrated in Section 5.6.2), but it is still able to quickly detect and respond to enemy targets that appear on radar.

It is important to note that, if we had used traditional algorithms for schedulability analysis which do not allow for negotiated QoS degradation, the system would have failed to guarantee/accept the entire task set on the same processor which may lead to a complete failure.

5.7 Conclusions

We extended *Adaptware* to provide a novel scheme for QoS negotiation in distributed real-time applications. This scheme may be applicable for the design of server farms, extending the interface of such services in that (i) it adopts a modified notion of QoS contracts that allows for defining QoS compromises and supports graceful QoS degradation, and (ii) it provides a generic means to maximize service utility for the community of clients in a distributed server. Our QoS negotiation method improves the guarantee ratio over traditional admission control algorithms and increases the application-level perceived utility of the system.

The proposed QoS-negotiation architecture has been incorporated into *RTPOOL*, an example middleware service which implements a computing resource manager for a pool of processors. *RT-*

[4] In our tests, when the missile control takes precedence over flight control during single machine operation, the aircraft becomes unstable. This is more extreme than one might want for an actual system, since one can’t launch missiles if the aircraft has crashed.

POOL is used as a computing server for a flight control application to demonstrate the efficacy of QoS negotiation. We demonstrated that the application does have negotiable parameters/constraints and can thus benefit from the added flexibility of our QoS contract model. We also showed that application QoS levels and their respective rewards can be analytically derived from system failure probability. QoS-negotiation support, while guaranteeing maximum QoS levels during normal operation, is shown to provide graceful QoS degradation in case of resource loss.

CHAPTER 6

SCHEDULABILITY ANALYSIS

6.1 Introduction

In the previous section we avoided discussing schedulability analysis techniques that may be employed by *RTPOOL* to accommodate real-time applications. Real-time application tasks have individual deadlines to be met by *RTPOOL*. In general, these tasks may need to communicate with one another, which imposes additional constraints on their allocation in order for their deadlines to be met. In hard real-time systems, failure to meet the deadline of a task may result in catastrophic consequences. Each task must therefore be guaranteed *a priori* to meet its timing constraints. Efficient techniques for schedulability analysis are needed. These techniques should be used at admission time of new applications, as well as by *RTPOOL*'s distributed QoS optimization heuristic when tasks are offloaded from one machine to another such that each task is guaranteed to meet its timing constraints on its prospective destination before it is admitted or reassigned. Since periodic tasks are the base load of most real-time systems, and since *q*threads' main abstractions rely on periodic tasks, we shall focus in this chapter on a periodic task model and describe how to schedule it. Sporadic tasks may be considered periodic by using, for example, a sporadic server.

Task precedence constraints and resource requirements need to be accounted for by schedulability analysis. The algorithm should find a feasible schedule (i.e., one that meets all deadlines), whenever such a schedule exists. We cast the schedulability problem into that of minimizing maximum task lateness, where lateness is defined as the difference between task completion time and deadline. A feasible schedule would then correspond to a solution where maximum task lateness is non-positive. If the optimal schedule has positive lateness, then no feasible schedule exists. This chapter addresses the problem of finding the optimal[1] schedule for hard real-time tasks given a particular task-to-processor assignment, known periodic task arrival times, precedence constraints and resource requirements. It constitutes a pre-run-time analysis stage that compliments *RTPOOL*'s run-time scheduling mechanisms (such as EDF scheduling) that enforce the desired schedule. As we describe later in the chapter, the analysis stage computes task deadlines, a part of QoS level specification, that make the resulting EDF schedule optimal in the sense of globally minimizing the maximum task lateness of the distributed application. The algorithms described are needed only for hard real-time applications where the deadline of each individual task invocation must be met. For soft real-time tasks, such as request handling in distributed web server farms, this analysis is not required.

The problem of minimizing maximum task lateness in hard real-time systems was optimally solved by Xu and Parnas [146] for uniprocessors. An attempt to extend their approach to several processors was made by Shepard and Gagne [119], but their algorithm occasionally fails to find existing feasible schedules as we pointed out in [3]. Xu remedied this shortcoming by proposing optimal multiprocessor scheduling with precedence and exclusion constraints [147]. However, his model is not suitable for distributed systems, since it assumes that tasks can be resumed on any processor at no additional cost, neglects the cost of intertask communication, and does not address the problem of scheduling inter-machine messages.

In distributed hard real-time systems, inter-machine message communication affects task schedulability, and thus, has to be accounted for. One way to solve the combined task and message scheduling problem is to separate message communication from task scheduling. The communication architecture described in Chapter 4 can guarantee bounded latency on message transmission and reception on the host. Communication protocols such as RSVP or Real-time Channels can guarantee bounded-time message transport in the network. Assuming the existence of such support in the network (typically in a closed embedded system), the task scheduling problem can be solved given the fixed and known message delay bounds. An optimal algorithm for solving such task scheduling problem is presented in [98]. A disadvantage of separating message scheduling from task scheduling is that the bounded message delays guaranteed by the communication subsystem depend on message priority (or class) which in turn depends on the sending/receiving task schedule from which the urgency of the message is determined (i.e., the slack time available for its transport such that the receiver does not miss its deadline). The schedule of communicating tasks,

[1] in the sense of minimizing maximum task lateness

however, cannot be computed without knowing the communication delay bounds between them in the first place. Because of this tight coupling between message delay bounds and task schedules, the problems of task scheduling and message scheduling should be solved *together*. We develop a *combined* approach to schedulability analysis that takes into consideration both tasks and intertask messages.

Several heuristics have been proposed to solve the combined problem, e.g., [8] and [61]. A flexible scheme which combines off-line analysis with on-line guarantees is suggested in [34] for uniprocessors. In [94], a rather similar scheme is described for distributed systems. It uses off-line analysis to convert task precedence and communication constraints into pseudo deadlines of tasks and messages, then employs an on-line guarantee routine to find a runtime task and message schedule that minimizes the number of tasks missing deadlines. An algorithm combining this problem with that of task allocation was presented in [104].

In contrast, we propose an *optimal* algorithm for scheduling tasks in a distributed real-time system which interacts with the problem of message scheduling, thereby improving the quality of solution. Assuming that the real-time channel paradigm [65] is used for message communication, the problem of message scheduling reduces to that of an appropriate choice of message deadlines. We define a *message-priority space* in which each point corresponds to a different message-priority assignment. Our algorithm may be viewed as searching the message-priority space *and* the space of all task schedules for a point where the task scheduling problem has an optimal solution in the sense of globally minimizing maximum task lateness. Conceptually, the search proceeds in two orthogonal dimensions. The first searches the message priority space. At a given point in the message-priority space, the second searches the space of all possible task schedules for a schedule that minimizes maximum task lateness. Due to the complexity of the combined problem, optimality is guaranteed in the second dimension, while a near optimal solution is sought in the first dimension.

Since it often suffices in hard real-time systems to find a *feasible* schedule which satisfies all deadlines as opposed to an optimal one, the search algorithm can be terminated after finding a first feasible schedule. This does not eliminate the need for casting the search as a lateness minimization problem. By finding the minimum-lateness schedule infeasible, the minimization approach allows the algorithm to terminate when the task set is unschedulable without explicitly expanding the entire solution space. Thus, the search algorithm remains efficient whether the answer to the schedulability problem is positive or negative. This is in contrast to other algorithms [104] which generally find a feasible schedule quickly if one exists, but cannot positively determine that a task set is unschedulable until they expand the entire search space.

The rest of this chapter is organized as follows. Section 6.2 presents the basic algorithm. Section 6.3 evaluates its performance. A fast heuristic derived from it is presented in Section 6.4. It uses a greedy (instead of optimal) technique to search for feasible schedules. The basic algorithm is generalized to a more practical model for resource requirements in Section 6.5. The chapter concludes with Section 6.6.

6.2 The Basic Algorithm

This section presents the basic algorithm proposed for combined task and message scheduling. Section 6.2.1 describes the system model and notation, while Section 6.2.2 presents a general solution approach. Simulation results are provided in Section 6.3.

6.2.1 System Model

The distributed system is composed of a set, P, of *p processing nodes* (PNs), $PN_1, \cdots, PN_p$, connected by an arbitrary network N. PNs run a set T of *n* hard real-time tasks, $T_1, \cdots, T_n$. Each task is assumed to reside permanently on one processor. Tasks may be assigned to processors using one of several heuristics, e.g., as described in *RTPOOL*. Each task T_k in the distributed system has a *known* arrival time a_k, total execution time c_k, and deadline d_k. The known arrival time is usually computed from task periodicity. Periodic tasks are invoked once every period P_k. The arrival time of periodic task T_k is associated with its individual invocations, such that the arrival time $a_k[j]$ of its jth invocation, $T_k[j]$, is the beginning of its period $a_k = (j-1)P_k$. The deadline of a periodic task invocation is set relative to its arrival time. For periodic tasks it suffices to analyze the system within an interval of time equal to the least common multiple (LCM) of all task periods. We call such interval the *planning cycle*. Table 6.1 gives an example set of periodic tasks assigned to two PNs. The duration of the planning cycle is $LCM(3, 6, 12) = 12$. This set will be used throughout the rest of the chapter to illustrate the solution approach.

A task may be composed of one or more *modules*. Each module M_i of a task invocation $T_k[j]$ has a worst-case execution time C_i which bounds its actual execution time, an arrival time A_i which denotes the earliest time the module can be invoked, and a deadline D_i which is the latest time it can finish execution. Initially, $A_i = a_k[j]$, and $D_i = d_k[j]$ of the corresponding task invocation. In a particular schedule γ, the time $S_i(\gamma)$ when module M_i is first given the CPU is called the *module*

$Task_k$	PN	c_k	$d_k - a_k$	P_k
T_1	1	1	3	3
T_2	1	2	5.5	6
T_3	1	3	11	12
T_4	2	3	4	6
T_5	2	1	9	12
T_6	2	0.5	3.5	6

Table 6.1: An example task set.

start time and the time $E_i(\gamma)$ when the module finishes execution is called the *module completion time*. The completion time of a task invocation $T_k[j]$ is the completion time E_{last} of its last module M_{last}. The *lateness* of task invocation $T_k[j]$ in schedule γ is defined as the lateness of the task's last module, $E_{last}(\gamma) - D_{last}$. A positive task lateness indicates that the task missed its deadline. Schedule lateness, $lateness(\gamma)$ is the maximum lateness over all task invocations in the schedule.

Modules may have *synchronization constraints* which are either precedence constraints (e.g., when one module waits for the results of another) or *mutual exclusion constraints* (between pairs of modules accessing the same serial resource). A precedence constraint $M_i\ precedes\ M_j$ means that M_i and M_j must be scheduled such that $E_i(\gamma) \leq S_j(\gamma)$. A mutual exclusion constraint $M_i\ excludes\ M_j$ means that neither of the two modules can have an execution interval between the other's start time and completion time. In other words, the modules must be scheduled such that either $M_i\ precedes\ M_j$ or $M_j\ precedes\ M_i$. Note that $excludes$ is commutative. The set $Sync_{init}$ is the set of all synchronization constraints defined initially for the system. In this section we cast resource constraints as mutual exclusion constraints between (entire) modules which implies that modules lock/unlock *all* their required resources *together*, and hold them throughout the entire interval of their execution. Thus, there is no possibility of deadlock. This restriction will be relaxed in Section 6.5. Modules residing on different processors communicate via message passing. A message from module M_i to module M_j in a planning cycle is denoted by $m_{i,j}$. Communication among modules residing on the same node is assumed to incur a fixed overhead, which is included in the execution time of the sending module.

For our running example, Table 6.2 depicts all task invocations in the task set shown in Table 6.1 within the planning cycle. For simplicity of illustration, all tasks except one (T_3) are chosen to consist of only one module. To illustrate synchronization constraints, we let task T_5 (module M_{11}) use some results computed in module M_7 of task T_3 that are sent to T_5 via a message $m_{7,11}$. We also let each odd-numbered invocation of task T_4 (i.e., module M_9) communicate a message to the fourth invocation of task T_1 (module M_4) in the same planning cycle. Furthermore, we let tasks T_4 and T_5 use the same serial resource thus creating a mutual exclusion constraint between each invocation of T_4 (modules M_9 and M_{10}) and task T_5 (module M_{11}). The resulting set of synchronization constraints $Sync_{init}$ is shown in Table 6.2. This example task set will be used throughout the chapter to illustrate the algorithm.

Invocation	Module	a_k	c_k	d_k
$T_1[1]$	M_1	0	1	3
$T_1[2]$	M_2	3	1	6
$T_1[3]$	M_3	6	1	9
$T_1[4]$	M_4	9	1	12
$T_2[1]$	M_5	0	2	5.5
$T_2[2]$	M_6	6	2	11.5
$T_3[1]$	M_7	0	1	11
	M_8	0	2	11
$T_4[1]$	M_9	0	3	4
$T_4[2]$	M_{10}	6	3	10
$T_5[1]$	M_{11}	0	1	9
$T_6[1]$	M_{12}	0	0.5	3.5
$T_6[2]$	$M_[13]$	6	0.5	9.5

Messages: $m_{7,11}, m_{9,4}$

$Sync_{init} =$

$\{M_7\, precedes\, M_{11}, M_7\, precedes\, M_8, M_9\, precedes\, M_4, M_9\, excludes\, M_{11}, M_{10}\, excludes\, M_{11}\}$

Table 6.2: The module set.

6.2.2 The Solution Approach

Our objective is to find an optimal task schedule and a near-optimal message priority assignment in the sense of minimizing schedule lateness (defined in Section 6.2.1) across all PNs. In doing so we consider (C1) a message communication paradigm, (C2) an optimal task scheduling algorithm, and (C3) a message priority assignment heuristic.

The message communication paradigm, C1, defines the mechanism used for message transport on the target system. The paradigm itself is not a contribution of our algorithm but rather a parameter of the underlying system. It must guarantee bounded communication delay for messages. We compute (i) a message priority order using some heuristic C3 that attempts to reduce task lateness, (ii) message delay bounds using paradigm C1, and (iii) an optimal schedule using the optimal task scheduling algorithm C2 that minimizes task lateness for given message delays. The real-time channels presented in [65] guarantee bounded transport delays in the network, which *CLIPS* ensures bounded end-system processing time of communicated messages. They will be used as an example for the paradigm C1. The general idea of real-time channels is to reserve resources in the network (e.g., on network routers) to guarantee bounded-time processing of a stream of messages specified by a given period, maximum message size, and worst-case jitter.

A B&B technique is used to minimize the lateness of the distributed communicating tasks. It can be viewed as a search, by implicit enumeration, through the entire *valid solution* space. A valid solution, γ, is a schedule with the following properties.

- Every module M_i in γ starts no earlier than its arrival time, i.e., $S_i(\gamma) \geq A_i$, and is given the CPU for a total of C_i time units.

- All task synchronization (i.e., precedence and exclusion) constraints in the set $Sync_{init}$ are satisfied.[2]

- Messages delays are computed using paradigm C1 and are accounted for in the schedule. That is, if module M_i sends a message $m_{i,j}$ to module M_j, and the delay bound of $m_{i,j}$ computed by paradigm C1 is $d_{i,j}$, then $S_j(\gamma) \geq d_{i,j} + E_i(\gamma)$.

A valid solution γ is *feasible* if it satisfies the additional constraint that every module M_i in γ finishes before its deadline (i.e., $E_i(\gamma) \leq D_i$). The B&B search can be viewed as traversing a search tree. The root vertex, V_{root}, of the search tree represents the space of all possible valid solutions.

[2]We have defined in Section 6.2.1 what it means to satisfy the precedence and exclusion constraints.

Branching from vertex V is a subdivision of the solution space of the parent among a set of child vertices. We denote by $Space(V)$ the set of all valid solutions represented by vertex V. Thus, for a parent vertex V, branching subdivides the solution space $Space(V)$ among a set of child vertices. In other words, the union of $\{Space(C) : C \text{ is a child of } V\}$ over all children of V amounts to $Space(V)$. Bounding a vertex V is the estimation of a value, $bound(V)$, that lower-bounds schedule lateness of all valid solutions in $Space(V)$. That is, $\forall \gamma \in Space(V) : lateness(\gamma) \geq bound(V)$. Bounding allows us to prune vertices whose bounds are higher (i.e., worse) than the lateness of the best solution found so far, say $BestLateness$, since such vertices cannot lead to an optimal solution. To enable pruning, a tentative schedule, $solution(V)$, is computed at each expanded vertex V out of the set $Space(V)$. Vertices whose bound is greater than the lateness of $solution(V)$ are then pruned. The algorithm continues until an optimal solution is found, i.e., all vertices have been pruned except one, and no further branching is possible. The complete algorithm is thus listed below.

1. Set up V_{root}. Let $ActiveVertexSet = \{V_{root}\}$.
   Let $BestLateness = lateness(solution(V_{root}))$

2. Let V_{expand} be the vertex with minimum $bound(V)$ among all vertices, $V \in ActiveVertexSet$. Pop V_{expand} out of $ActiveVertexSet$.

3. Find $solution(V_{expand})$. If $lateness(solution(V_{expand})) < BestLateness$ let $BestLateness = lateness(solution(V_{expand}))$. Find the children of vertex V_{expand} applying the branching function, $branch(V)$ to V_{expand}.

4. Prune the vertices that do not improve on the best solution found so far, i.e., vertices V for which $bound(V) \geq BestLateness$.

5. Add the remaining vertices to the set $ActiveVertexSet$.

6. If $ActiveVertexSet$ is non-empty, go to step 2. Otherwise, return the solution with the current $BestLateness$.

In the following subsections, we (i) show how the root vertex is set up, (ii) describe the function $solution(V)$ that computes a schedule at vertex V out of the space of valid schedules, $Space(V)$, represented by the vertex, (iii) give details of the branching function, $branch(V)$ that returns a set of child vertices given a parent vertex V, and (iv) derive the bounding function that determines, given some vertex V, a lower bound on schedule lateness for all schedules in $Space(V)$. These functions in conjunction with the pseudo-code given above completely specify our B&B algorithm. As with any B&B algorithm, its optimality is guaranteed as long as (i) branching does not leave any part of the solution space unreachable, and (ii) bounding computes a true lower bound of the performance measure for each vertex [69]. These properties are proved when discussing branching and bounding respectively.

Setting up the Root Vertex

The root vertex represents the entire space of all valid solutions. $Space(V_{root})$ is implicitly specified by (i) module arrival times and computation times, and (ii) a set of synchronization constraints $Sync(V_{root}) = Sync_{init}$, as given in the problem input (e.g., see Table 6.2). Any valid solution to the scheduling problem must satisfy (i) and (ii), (as well as account for message delay bounds computed by paradigm C1). A valid *feasible* solution would also satisfy all deadlines.

For the purpose of computing an initial schedule $solution(V_{root})$, we also compute an initial message priority order, $Ord(V_{root})$ using heuristic C3. The "urgency" of each message is first estimated as follows. The precedence constraints associated with messages among modules running on different PNs are neglected, and modules on each PN are scheduled using EDF subject to the remaining precedence and exclusion constraints. Let the resulting schedule be $\gamma_{no\ constraints}$. For each message $m_{i,j}$ from module M_i to module M_j we note the completion time $E_i(\gamma_{no\ constraints})$ of the sending module M_i in schedule $\gamma_{no\ constraints}$ which is the time message $m_{i,j}$ is sent. The message must make it to the receiver M_j in time for it to execute by its deadline, D_j. Thus, the relative deadline for message $m_{i,j}$ is set to $D_j - C_j - E_i(\gamma_{no\ constraints})$.

Messages are ordered by their relative deadlines, such that messages with tighter relative deadlines have higher priorities. Ties are broken arbitrarily. This results in the initial message priority order $Ord(V_{root})$. Figure 6.1 demonstrates the schedule $\gamma_{no\ constraints}$ for one planning cycle of the task set in Table 6.2. It shows the intervals from message transmission time to receiver deadline for the two messages in the planning cycle, $m_{7,11}$ and $m_{9,4}$. From the figure it is seen that $m_{7,11}$ must have higher priority than $m_{9,4}$. Thus, $Ord(V_{root}) = m_{7,11}, m_{9,4}$ (in decreasing priority order).

Computing $solution(V)$

In order to prune vertices in the search space we compute a valid solution at each visited vertex V. The solution is drawn from the solution subspace $Space(V)$ represented by the vertex. The lateness

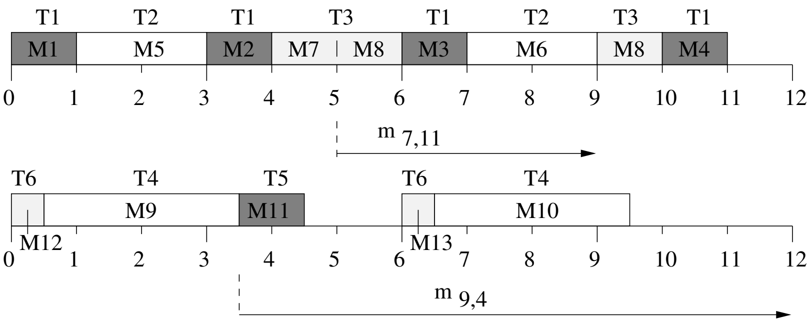


Figure 6.1: Computing message priority order $Ord(V_{root})$.

of the computed solution is used to prune vertices whose $bound()$ is higher. Thus, the goodness of the function $solution(V)$ in picking a low-lateness schedule out of $Space(V)$ determines how efficient the pruning process is. In this subsection we describe the function $solution(V)$.

Given a complete message priority order $Ord(V)$ at vertex V, and a set $Sync(V)$ of synchronization constraints, we need to (i) compute message delays, (ii) compute a valid task schedule, and (iii) find schedule lateness. Of these, computing message delays is not performed by our algorithm. Instead, it is performed by the underlying communication paradigm, C1 (e.g., real-time channels). In our running example, we would thus use C1 to establish a channel for the higher priority message $m_{7,11}$ first, then establish a channel for message $m_{9,4}$. For the purpose of illustration, assume that the delay bounds computed by C1 for messages $m_{7,11}$ and $m_{9,4}$ are 1.75 and 3, respectively.

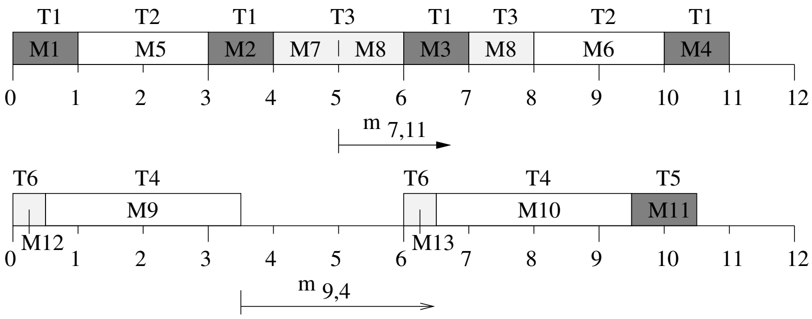
Once the message delay has been established for each message, the function $solution(V)$ computes a new arrival time A_i, and deadline D_i for each module M_i to account for message communication delays and precedence constraints. The following recursive equations are used.

$$A_i = \max(A_i, \ \{A_j + C_j + m_{ji} \mid M_j \ precedes \ M_i\}) \tag{6.1}$$

$$D_i = \min(D_i, \ \{D_j - C_j - m_{ij} \mid M_i \ precedes \ M_j\}). \tag{6.2}$$

where C_j is the worst-case computation time of M_j, and m_{ji} (m_{ij}) is the computed communication delay between M_j and M_i (M_i and M_j), if any. This is a standard technique for deadline and release time modification in order to account for precedence constraints. Variations of it have been used in several publications, e.g., [94, 98, 118, 147]. For example, in the task set of Table 6.2, $A_{11} = \max(0, A_7 + C_7 + m_{7,11}) = 2.75$, and $D_7 = \min(12, D_{11} - C_{11} - m_{7,11}, D_8 - C_8) = \min(12, 9 - 1 - 1.75, 12 - 2) = 6.25$

Next, $solution(V)$ computes an EDF schedule subject to the above arrival times and deadlines, as well as precedence and exclusion constraints in set $Sync(V)$. We choose EDF because it is a locally optimal preemptive scheduling policy. To prevent unbounded (dynamic) priority inversion, a module which blocks others with earlier deadlines inherits the earliest of these deadlines. We call this policy *EDF with Deadline Inheritance* (EDF-DI). Note that we do not use dynamic priority ceilings [32], because deadlocks cannot occur in our simplified model. Figure 6.2 shows the schedule computed by $solution(V)$ at the root vertex for the task set in Table 6.2. In this example, schedule lateness is, $lateness(solution(V_{root})) = 1.5$, which is the lateness of T_5 (module M_{11}).



$Ord(V) = m_{7,11}, m_{9,4}$

$Sync(V) = \{M_7\ precedes\ M_{11}, M_7\ precedes\ M_8,$

$M_9\ precedes\ M_4, M_9\ excludes\ M_{11}, M_{10}\ excludes\ M_{11}\}$

$lateness(solution(V)) = 1.5$

Figure 6.2: Root schedule.

Branching $branch(V)$

The branching function, $branch(V)$, subdivides the valid solution space $Space(V)$ represented by a parent vertex V into a set of subspaces, each represented by a child vertex. To make sure that no parts of the solution space are "lost", the union of the subspaces $Space(C)$ over all children C of vertex V must amount to $Space(V)$. The set of valid solutions $Space(V)$ at an arbitrary vertex V is implicitly represented by (i) module arrival times and computation times at vertex V, and (ii) the set of synchronization constraints $Sync(V)$. The solution space is subdivided by the branching function in order to help pruning subsets of that space. We prune vertices whose lower bound is worse (i.e., higher) than the best lateness of a schedule found by $solution(V)$. Thus, we need $solution(V)$ to return progressively better schedules as we get deeper in the search tree to enable pruning more vertices. In what follows, we describe how branching is done, and present the methodology used to attempt to improve the lateness of $solution(V)$ at each descendant.

Consider some vertex V in the search tree generated by our B&B algorithm. Let T_m be the task with the maximum task lateness in the schedule computed by $solution(V)$. If there is a tie, let T_m be the task with the maximum lateness that finishes first. This tie-breaking rule is important to guarantee "progress" (i.e., prevent an infinite loop in the algorithm) as will be described later in this section. Let M_{last_m} be the last module of task T_m. For example, in the root schedule shown in Figure 6.2, M_{last_m} is M_{11}. Unless the schedule at V happens to be optimal, there exists a way to reduce schedule lateness. In other words, it is possible to let module M_{last_m} finish earlier. In the following we consider all possible ways of letting M_{last_m} finish earlier. In order to categorize and describe these ways, we use the concept of a *busy period* [146]. Informally, the *busy period*, B_i, of module M_i is the interval $[G_i, E_i]$, where E_i is the completion time of M_i, and G_i is the start of the period of *continuous* processor utilization that includes M_i. For example, in the root schedule shown in Figure 6.2 the busy period of the latest module M_{11} is $B_{11} = [6, 10.5]$. The busy period B_i of module M_i is defined recursively as follows:

1. $M_i \in B_i$,
2. While $\exists$ M_k whose completion time satisfies $t < E_k < E_i$ (where t = min $\{ A_j \mid M_j \in B_i)$ $\})$ let $M_k \in B_i$.

In order to reduce schedule lateness we consider the following three cases, exactly one of which will be satisfied in any schedule (since their ORing amounts to unity).

Case 1: No module M_i in B_{last_m} has predecessors on other processors (according to set $Sync(V)$), and no module M_i in B_{last_m} has a deadline $D_i > D_{last_m}$.

Case 2: $\exists$ some module M_i in B_{last_m} whose deadline $D_i > D_{last_m}$.

Case 3: $\exists$ some module M_i in B_{last_m} who has a predecessor M_j on a different processor (according to set $Sync(V)$), and no module in B_{last_m} has a deadline $D_i > D_{last_m}$.

Case 1: No module M_i in B_{last_m} has predecessors on other processors (according to set $Sync(V)$), and no module M_i in B_{last_m} has a deadline $D_i > D_{last_m}$. In any schedule with a lower lateness than $solution(V)$ the module M_{last_m} (which has the maximum lateness in $solution(V)$) must complete earlier. However, since all other modules in B_{last_m} have tighter deadlines, the schedule where M_{last_m} executes last in B_{last_m} (i.e., $solution(V)$) is optimal. That is to say, it is the optimal solution within the subset $Space(V)$ of all possible schedules represented by vertex V. No further branching from that vertex is possible. The branching function returns a null set of children.

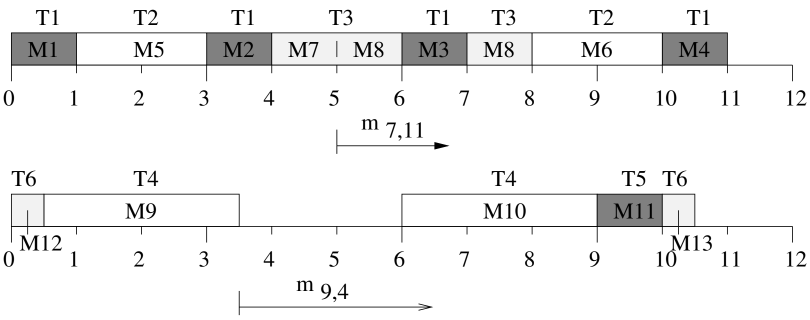
Case 2: There exists some module M_i in B_{last_m} whose deadline $D_i > D_{last_m}$. Since EDF scheduling is used to obtain $solution(V)$, modules scheduled before the latest module M_{last_m} in its busy period must necessarily have *earlier* deadlines. The only way some module M_i in B_{last_m} can have $D_i > D_{last_m}$ yet be scheduled before M_{last_m} is because of priority inversion due to a mutual exclusion constraint which prevents that module from being preempted. In other words, Case 2 implies that $\exists M_j \in B_{last_m}$ such that $M_i\ excludes\ M_j \in Sync(V)$. For example, in Figure 6.2 we can see a situation where the latest module M_{11} which becomes ready upon delivery of message $m_{7,11}$ at time $t = 6.75$ cannot preempt a less urgent module M_{10} due to a mutual exclusion constraint even though $D_{10} > D_{11}$. We also see a case where M_{13} of intermediate priority delays less urgent M_{10} before M_{11} becomes ready, thus indirectly delaying the higher priority module M_{11} because of the mutual exclusion constraints.[3] "Eliminating" the exclusion constraint would resolve the problem of priority inversion, potentially resulting in a better schedule. To eliminate an exclusion constraint, $M_i\ excludes\ M_j$, the branching function $branch(V)$ generates two children C_1 and C_2 such that $Space(C_1)$ is the subset of all schedules in $Space(V)$ where $M_i\ precedes\ M_j$, and $Space(C_2)$ is the subset of all schedules in $Space(V)$ where $M_j\ precedes\ M_i$. In other words, we set the children's synchronization constraints such that $Sync(C_1) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_i\ precedes\ M_j\}$ and $Sync(C_2) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_j\ precedes\ M_i\}$. Since an exclusion constraint $M_i\ excludes\ M_j \in Sync(V)$ means that in any valid solution either $M_i\ precedes\ M_j$ or $M_j\ precedes\ M_i$, it can be seen that $Space(C_1) \cup Space(C_2) = Space(V)$. However, in each child independently, the exclusion constraint has been replaced with a precedence constraint.

For the sake of illustration, Figure 6.3 gives the $Sync()$ sets of the two children of the root vertex whose schedule, shown in Figure 6.2, satisfies Case 2. The figure shows that applying the $solution()$ function to each child gives a better schedule (in terms of maximum task lateness) than that of the parent. (Compare the maximum lateness of the children in Figure 6.3 to that of root schedule in Figure 6.2.) $Child_1$ results in decreasing the deadline of module M_{10} (see Equation 6.2) thus preventing M_{13} from preempting M_{10}. As a result, both M_{10} and M_{11} finish earlier, thus reducing schedule lateness. $Child_2$ results in causing M_{10} to wait until M_{11} is finished instead of

[3] EDF-DI does not prevent such priority inversion because it cannot keep M_{10} from being preempted *before* it inherits the deadline of M_{11}

blocking it due to mutual exclusion, also reducing schedule lateness. This lateness improvement leads to more efficient pruning. Intuitively, the improvement is attributed to the local optimality of EDF scheduling when no mutual exclusion constraints are present.

Child 1:

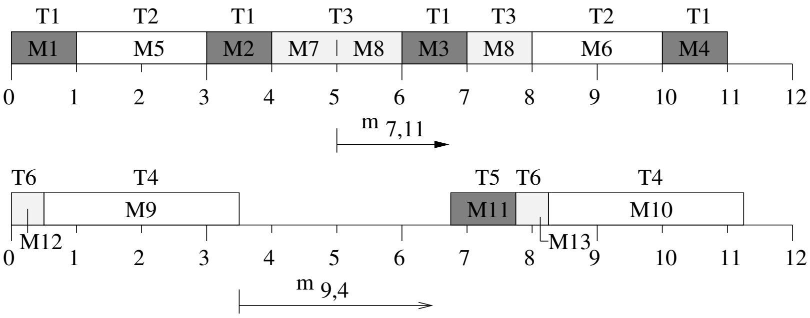


$Ord(V) = m_{7,11}, m_{9,4}$

$Sync(V) = \{M_7\ precedes\ M_{11}, M_7\ precedes\ M_8, M_9\ precedes\ M_4, M_9\ excludes\ M_{11},$
$M_{10}\ precedes\ M_{11}\}$

$lateness(solution(V)) = 1$

Child 2:



$Ord(V) = m_{7,11}, m_{9,4}$

$Sync(V) = \{M_7\ precedes\ M_{11}, M_7\ precedes\ M_8, M_9\ precedes\ M_4, M_9\ excludes\ M_{11},$
$M_{11}\ precedes\ M_{10}\}$

$lateness(solution(V)) = 1.25$

Figure 6.3: Branching in Case 2

Case 3: There exists some module M_i in B_{last_m} who has a predecessor M_j on a different processor (according to set $Sync(V)$), and no module in B_{last_m} has a deadline $D_i > D_{last_m}$. Following the same argument as in Case 1, the schedule generated in Case 3 for the busy period $B_{last_m} = [G_{last_m}, E_{last_m}]$ is locally optimal. The latest module has the largest deadline and is therefore scheduled last in the busy period. However, since in Case 3, $\exists M_i \in B_{last_m}$ which has a remote predecessor M_j, schedule lateness may be improved by either increasing the priority of message $m_{j,i}$, if any, to let it arrive earlier at the destination, or else by rescheduling M_j earlier on its processor. In either case, the busy period B_{last_m} as a whole may start earlier thus potentially reducing schedule lateness. For example, consider the schedule of $Child_2$ shown in Figure 6.3. The latest module M_{last_m} is M_{10} whose lateness is 1.25. The busy period B_{last_m} is $B_{10} = [6.75, 11.25]$. There exists a module $M_i = M_{11}$ in B_{last_m} with a remote predecessor $M_j = M_7$, thus Case 3 is satisfied. Message $m_{7,11}$ already has top priority. So we consider scheduling M_7 earlier to decrease the lateness of the latest task. In general, a remote predecessor M_j is forced to start earlier by decreasing its deadline such that it inherits the lateness of M_{last_m}. Thus, the branching function in Case 3 returns a set of children as follows.

- If $\exists$ a message $m_{i,j}$ from a remote predecessor M_j immediately preceding some $M_i \in B_{last_m}$ (i.e., $M_j\ precedes\ M_i \in Sync(V)$) increase the priority of $m_{i,j}$ (if possible). The algorithm shown in Figure 6.4 specifies how message priority is increased. Essentially, the first promoted message gets the highest priority. Each time another message is promoted, its priority is set one level below the priority of the previously promoted message. The variable $priority\_limit(V)$ tells which level message priority should be increased to at search vertex V. The variable is set to the highest priority, 1, at the root, and is incremented each time a message has been promoted. We do not claim optimality with respect to setting message priorities, although we expect that increasing the priority of a message on the critical path is likely to improve schedule lateness.

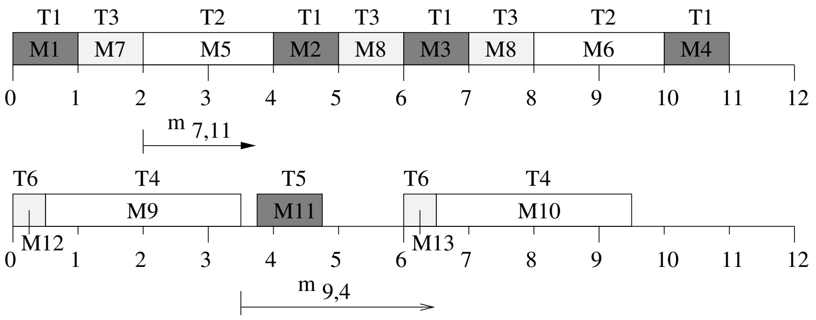
- If message priority cannot be increased (or there are no messages) then for all remote predecessors M_j immediately preceding some $M_i \in B_{last_m}$ (i.e., $M_j\ precedes\ M_i \in Sync(V)$) create a child vertex with same module arrival times, computation times, and synchronization constraints, and let M_j inherit the lateness of the latest task, i.e., reduce the deadline of M_j in the child to $\min(D_j, E_j - lateness(M_{last_m}))$, where $lateness(M_{last_m}) = E_{last_m} - D_{last_m}$. Note that changing a module deadline makes the heuristic function, $solution()$, return a different (potentially better) EDF schedule for child C than it does for its parent V, leading to pruning more vertices. However, $Space(C) = Space(V)$ because the valid solution space is defined independently of deadline values.

If the deadline of a predecessor M_j of the latest module is advanced, the predecessor will either

```
message_priority_increase(V)
 if V = V_root
   priority_limit (V) = 1
   let message_priority = priority_limit (V)
 else
   priority_limit (V) = priority_limit (parent (V)) + 1
   if message_priority is lower than priority_limit (V)
     let message_priority = priority_limit (V)
```

Figure 6.4: The branching function.

(i) finish earlier in the new schedule (i.e., a different schedule, γ_C, results at the child), or (ii) will become the latest task itself. In either case "progress" is guaranteed in the sense that the child differs from the parent in either the schedule returned by $Solution(V)$ or the latest task. Note that case-(ii) above is because the predecessor's deadline has been advanced to $D_j = E_j(\gamma_V) - lateness(M_{last_m})$, where γ_V is the schedule at the parent vertex V. Thus, if the predecessor does not finish earlier in the new schedule (i.e., if $E_j(\gamma_C) = E_j(\gamma_V)$), its lateness will be $E_j(\gamma_C) - D_j = E_j(\gamma_V) - D_j = lateness(M_{last_m})$. Since our tie breaking rule chooses the latest task to be the one with the minimum completion time among those with the maximum lateness, it will choose M_j rather than M_{last_m} as the latest task in the new schedule, γ_C. This guarantees progress. Note that branching in Case 3 is similar to substituting a precedence constraint between the latest task and one of its remote predecessors with an artificial deadline on the predecessor. Removal of a precedence constraint between tasks residing on different processors brings the EDF schedule at the child vertex closer to the optimal. EDF *is* optimal when all such precedence constraints (and all exclusion constraints) are removed. Figure 6.5 illustrates branching in Case 3. It shows the schedule obtained by branching from vertex $Child_2$ shown in Figure 6.3 by letting the predecessor M_7 inherit the lateness of the latest module M_{10}. The new deadline of M_7 becomes $D_7 = 5 - 1.25 = 3.75$. Intuitively, M_7 must complete by that deadline in order for M_{10} to complete in time. The resulting new EDF schedule shifts M_7 earlier (in accordance with its new deadline) which happens to result in a globally optimal schedule. Figure 6.6 summarizes the branching function.



$Ord(V) = m_{7,11}, m_{9,4}$

$Sync(V) = \{M_7\ precedes\ M_{11}, M_7\ precedes\ M_8, M_9\ precedes\ M_4, M_9\ excludes\ M_{11},$
$M_{11}\ precedes\ M_{10}\}$

Figure 6.5: Branching in Case 3.

Bounding $bound(V)$

Our bounding function determines a lower bound on lateness at a vertex V by removing from set $Sync(V)$ (i) all the mutual exclusion constraints, and (ii) all precedence constraints among modules on different processors, then computing vertex cost subject to the remaining set of local precedence constraints, say $Sync_{rem}(V) \subset Sync(V)$, and module arrival times and deadlines as described earlier. (Note that message priorities are irrelevant here, because precedence constraints and delays associated with interprocessor messages have been ignored in $Sync_{rem}(V)$.)

The lateness of the computed EDF schedule is globally optimal among all schedules that satisfy $Sync_{rem}(V)$. This is because (i) since no exclusion constraints are present in $Sync_{rem}(V)$ EDF is locally optimal, and (ii) since all constraints in $Sync_{rem}(V)$ are between modules on the same processor, modules on each processor are "independent" of modules on every other processor. Therefore, the set of locally optimal uniprocessor schedules is a globally optimal schedule. Let the aforementioned optimal lateness be called L_{min}.

Finally, since any solution $S \in Space(V)$ to the original task scheduling problem satisfies the constraint set $Sync(V)$, it satisfies, by implication, the constraint subset $Sync_{rem}(V)$. Thus, the lateness of S cannot be less than the global optimum L_{min}. Thus, L_{min} is a true lower-bound on lateness for any valid solution in $Space(V)$. The optimality of the B&B algorithm follows from the correctness of the bounding function, and the ability of the branching function to encompass the entire solution space. Intuitively, the branching function transforms the set of initial precedence and exclusion constraints into an equivalent set of constraints with no mutual exclusion and no

branch()

if $Case\ 1$ **return** $solution(V)$ optimal

if $Case\ 2$

Find $M_i, M_j \in B_{last_m}$ where $D_i > D_{last_m}$ and $\exists\{M_i\ excludes\ M_j\} \in Sync(V)$

Generate Child C_1 where $Sync(C_1) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_i\ precedes\ M_j\}$

Generate Child C_2 where $Sync(C_2) = Sync(V) - \{M_i\ excludes\ M_j\} + \{M_j\ precedes\ M_i\}$

if $Case\ 3$

if $\exists m_{j,i}$ where $M_i \in B_{last_m}$, and M_j, M_i run on different processors

Increase the priority of $m_{j,i}$ as shown in Figure 6.4

Let $Sync(C) = Sync(V)$

else $\forall\ M_j$ where $M_j\ precedes\ M_i \in Sync(V)$, $M_i \in B_{last_m}$ and M_j, M_i run on different processors

Generate Child C with $Sync(C) = Sync(V)$ and $D_j = \min(D_j, E_j + E_{last_m} - D_{last_m})$

Figure 6.6: The branching function.

precedence constraints across different processors; a case in which EDF is globally optimal.

6.3 Evaluation

To demonstrate the utility of the algorithm, a simulator was constructed, generating arbitrary task graphs on which the algorithm can be applied. On each run, the algorithm was given a task graph, an optimal solution was found, and the number of generated vertices was recorded. The numbers of modules, processors, precedence and exclusion constraints, were varied to determine the trends in algorithm performance. We restrict the following discussion to the most tangible results we found, namely, the effects of application concurrency, CPU utilization, and module interaction on the performance of the algorithm. We considered systems of 300 modules running on 4 processors. The effect of application concurrency is analyzed by varying the number of concurrent application threads per processor. CPU utilization is varied by changing the average module execution time. Finally, module interaction is controlled by varying the number of communicated messages between different modules. In the figures given below, each point is obtained by averaging 25 readings.

Figure 6.7 shows the effect of concurrency. The number of concurrent application threads per processor is varied from 1 to 9. Note that the total number of application threads may be much more than the above. By concurrent threads we mean only those that become ready to execute during the same time intervals. CPU utilization is fixed at 90%, and the number of messages in the system is fixed at 150 (half the number of modules). It can be seen that an optimal schedule is found near the root in most cases when the degree of concurrency is low. This is explained by the fact that EDF scheduling (performed at the root) is locally optimal. Exploiting this characteristic may lead to an optimal solution when application concurrency is low. For concurrency of 6 threads per processor or less, the average number of generated vertices is less than 5. As the concurrency increases, the local optimality of EDF scheduling becomes less and less sufficient. Thus, our algorithm expands progressively more vertices to find a globally optimal solution.

Figure 6.8 demonstrates the effect of CPU utilization. CPU utilization is the total computational workload per processor divided by schedule length. The number of messages in the system was fixed at 150, and the average degree of concurrency was 8. The algorithm performs very well for utilization up to 80%. The average number of generated vertices over that range is less than 10. As utilization increases, the algorithm runtime increases abruptly, due mainly to the accompanying increase in the length of the busy period, and therefore the increase in branching factor.

Finally, Figure 6.9 illustrates the effect of module interaction measured in number of communicated messages within the system. CPU utilization was fixed at 90%, and the degree of concurrency was fixed at 8. Unlike the other curves, the algorithm runtime increases almost *linearly* with the number of messages. As the number of messages increases, so does the branching factor. However, since each message introduces a precedence constraint, increasing the number of messages tends to constrain the task graph and decrease scheduling options at any given time, thus reducing the depth of the search tree.

In general, for a wide range of workloads the algorithm generates an optimal solution at or near the root of the search tree. A similar observation was reported in [119]. This is due to the nature of the performance measure being optimized. Schedule lateness refers to the lateness of only *one* module. If we happen to be unable to reduce the lateness of the latest module, the algorithm terminates even though there may be ways to decrease the lateness of other modules. The generation of vertices is inexpensive. For the task sets considered in this section, the worst-case computation time of a run was in the order of a few seconds on a Sun Ultra workstation.

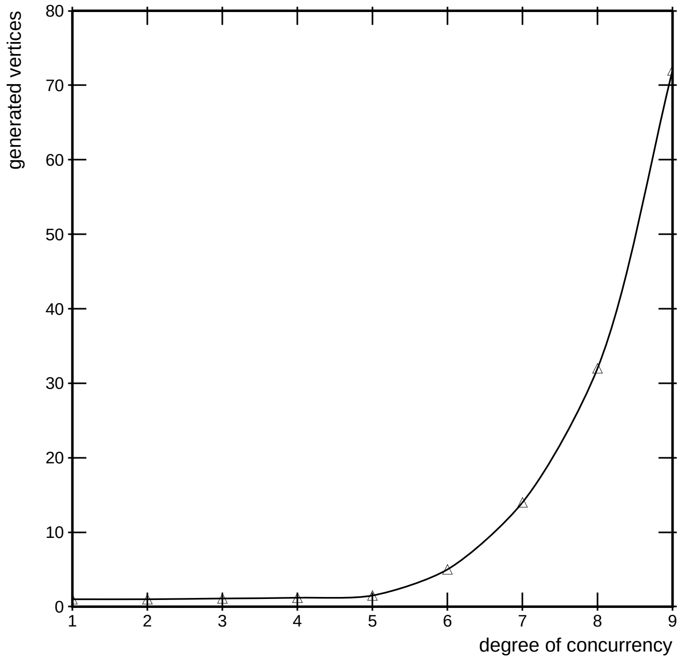


Figure 6.7: Effect of application concurrency.

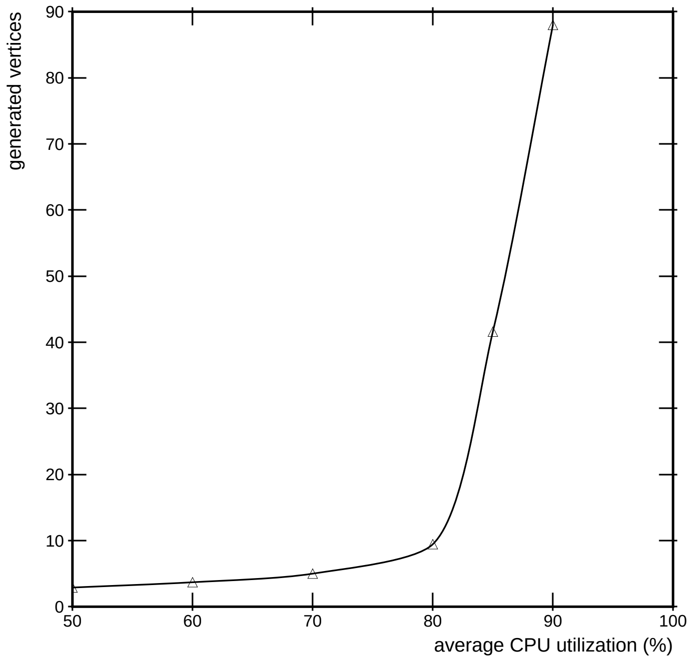


Figure 6.8: Effect of processor utilization.

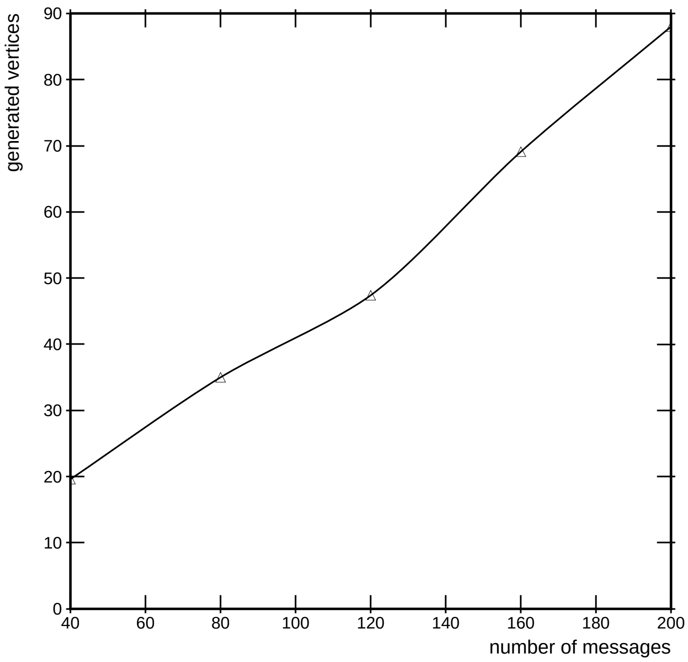


Figure 6.9: Effect of module interaction.

6.4 A Simple Heuristic

The complexity of the algorithm presented in Section 6.2 prevents it from being deployed on-line. However, this complexity can be reduced by employing a greedy heuristic which performs depth-first search with no backtracking. The proposed heuristic expands each vertex V by generating all its children, then branches to the minimum-cost child, ignoring all others. Thus, at each vertex, after generating its children, a complete schedule is computed for each child (as opposed to a lower bound). Children whose schedule lateness is more than that of the parent are pruned. Among the surviving ones, the child with the minimum schedule lateness is selected for expansion next. The algorithm continues until a vertex is reached that has no children, (or until the first feasible schedule is found, if so desired). Although pathological cases may be constructed where the heuristic fails to find an existing feasible schedule, it was able to arrive at the optimal schedule in 29 out 30 randomly-generated cases of periodic task sets with 90% CPU utilization. This number, however, depends much on the nature of the task set. In the case where tasks arrive at random times with random deadlines the heuristic performs worse than in the case where tasks arrive at regular intervals and have the same deadline at each invocation. To compare the costs of running the two algorithms, 30 randomly-generated periodic task sets (of 400 tasks each) were constructed and each algorithm was run on each set. The number of generated vertices was *not* used as a measure for algorithm comparison since the amount of computation per vertex is higher for the heuristic algorithm. This is because it computes a *complete schedule* for each child vertex, while the optimal one computes only a lower bound. Since the computation of a schedule at a vertex was found to be the most costly element of both B&B algorithms, the *number of complete schedules computed* until a solution is found was taken as the measure for their comparison. The heuristic was found to generate 74% less schedules than the optimal algorithm for the task set size considered before the best schedule is found. It is projected that the savings are greater for larger task sets.

6.5 A More General Resource Constraint Model

In the previous sections we presented an algorithm for combined task and message scheduling in distributed real-time systems. The algorithm uses a simple model for resource requirements. The model has two limitations:

- Resources are assumed to be locked/unlocked *together*;
- All resources are locked at module start and unlocked upon module termination.

Figure 6.10 illustrates consequences of these limitations. For example, we cannot model the fact that resource R2, in Figure 6.10, must be locked only for a fraction of a module's execution time. Instead, in the model, it has to be locked throughout the entire interval of the module's execution. A more severe consequence is that all resources have to be unlocked at module termination, whereas communication can only occur at module boundaries. Thus, in Figure 6.10, we must create a module boundary at the point when a message is sent. This will erroneously imply that all held locks are released at that point. As a result, there is nothing to prevent the scheduler from inserting a module which uses R1 or R2 between M_1 and M_2 violating the actual exclusion constraints. In what follows, we present a more realistic version of resource constraints, and describe how the algorithm in Section 6.2 can be modified to accommodate it.

6.5.1 A General Exclusion Model

We extend the notion of an exclusion constraint to include exclusion between *strings of modules* where a string of k modules $M_1\ M_2\ \cdots\ M_k$ is a sequence $M_1\ precedes\ M_2\ \cdots\ precedes\ M_k$.

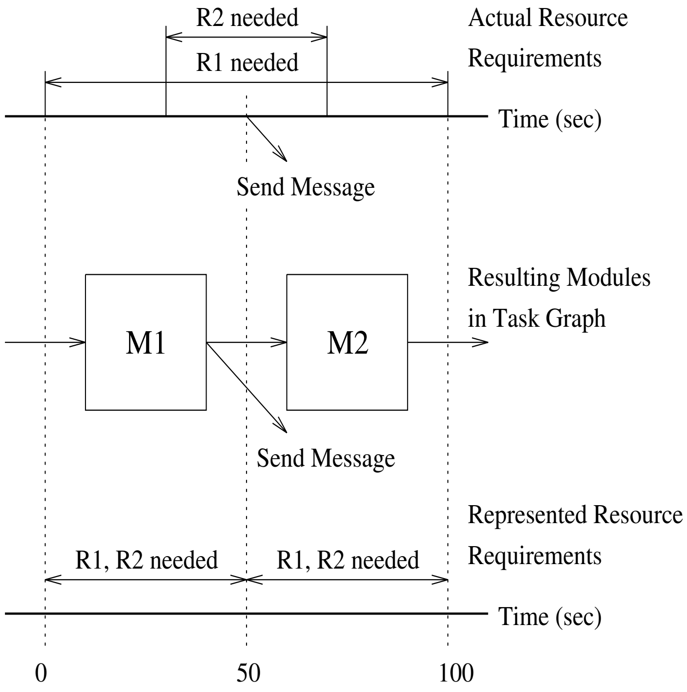


Figure 6.10: Accounting for contention.

Thus, the constraints have the general form $ModuleString_1$ $excludes$ $ModuleString_2$, meaning that all modules in one of the strings have to terminate before any module in the other can start. For example, Figure 6.11 depicts the resource requirements of two tasks. The following exclusion constraints can be derived:

- C1: M_2 M_3 M_4 M_5 $excludes$ M_9 M_{10} (because of R1)
- C2: M_3 M_4 $excludes$ M_8 M_9 M_{10} M_{11} (because of R2)
- C3: M_5 $excludes$ M_8 M_9 M_{10} M_{11} (because of R3)

Note that under this model, a deadlock may occur. For example, in Figure 6.11 a potential deadlock arises because $task_1$ and $task_2$ lock resources R1 and R2 in different orders.

6.5.2 Algorithm Modifications

A close look to the algorithms described in previous sections may reveal that no modifications are necessary to accommodate the new type of exclusion constraints. This is because an exclusion constraint such as M_2 M_3 M_4 M_5 $excludes$ M_9 M_{10} can simply be viewed as an equivalent exclusion constraint M_a $excludes$ M_b, where $M_a = M_2M_3M_4M_5$, and $M_b = M_9M_{10}$. Our only concern is

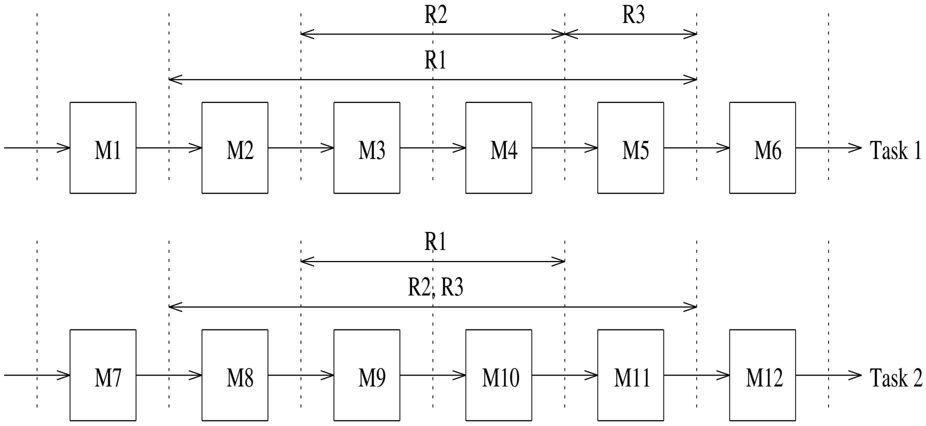


Figure 6.11: An example of mutual exclusion.

to avoid deadlocks when computing a valid solution, $Solution(V)$. Thus, to accommodate the exclusion model presented above, we slightly modified the definition of a valid schedule. In particular, in addition to the former requirements, a valid schedule must also be *deadlock-free*. As a result, the function $Solution(V)$ must have a way to detect deadlocks in the computed solution. This function is performed off-line, and can utilize any of the known methods for deadlock detection. (Deadlock detection is not investigated in this work.) A trivial way of detecting deadlock is to schedule all tasks until no further tasks can be scheduled. If some tasks remain unscheduled then a deadlock is present. The problem is how to modify the schedule once a deadlock is detected.

To answer the above question, note that if there were no exclusion constraints, deadlocks would not develop. We had already demonstrated one way of removing exclusion constraints by replacing them with precedence constraints. When a deadlock is detected by $Solution(V)$, it is circumvented using a similar technique. Consider a deadlock that occurs during the computation of a solution schedule $Solution(V)$ at some search vertex V. A typical deadlock detection algorithm can then identify a cycle of modules in which each module cannot run because it is waiting for another module in the cycle to proceed. Such waiting may be due to either precedence or exclusion constraints. Since there are no cycles in the precedence constraint graph in the problem input, at least one of the edges in the deadlock cycle must be due to an exclusion constraint. This exclusion constraint can be replaced by one of two possible precedence constraints.[4] Consequently vertex V branches into two children. If the replacement results in a cyclic precedence constraint graph in any of the two generated constraint sets, the corresponding child vertex is destroyed. Bounding is then performed on the

[4]The replacement of an exclusion constraint by complementary precedence constraints has been described in Section 6.2.2.

surviving vertices. As in the original search algorithm, $Solution(V)$ is re-applied to the vertex with the lowest bound. It will no longer run into the same deadlock since the cycle causing the deadlock has been broken. In essence, to avoid the deadlock we have "eliminated" an exclusion constraint the same way we described earlier in the context of branching. Eventually enough of the exclusion constraints will be eliminated to produce a deadlock free schedule. The aforementioned technique does not compromise the optimality of the solution since it does not compromise the ability of the branching function to encompass the entire solution space.

6.6 Conclusion

We presented a new B&B algorithm for off-line *combined* task and message scheduling in distributed real-time systems. The algorithm computes task deadlines and message priorities such that the maximum task lateness is minimized. It accounts for precedence and exclusion constraints between task modules. Furthermore, it exploits the coupling between task completion times and message delays by recomputing message priorities and deadlines during the search to reduce the maximum task lateness.

A simulation study has shown that the algorithm scales well with respect to system size and the number of communicated messages among tasks. A heuristic version of the algorithm is presented, where a greedy technique is used to trade optimality for speed. We also suggest a way to generalize the algorithm for a more practical resource model. The algorithm can be used in *RTPOOL* for admission control of incoming hard real-time tasks. It would compute a set of deadlines that constitute part of their QoS level specification. The hard real-time task set is guaranteed to be schedulable subject to the computed deadlines. Soft real-time tasks can be accounted for in this analysis by a single independent periodic task (budget) of the aggregate utilization of the soft real-time task set.

CHAPTER 7

RELATED WORK

Adaptware reflects a recent change in research direction towards adaptive real-time computing. Traditionally, real-time computing has been concerned mainly with predictability, not adaptation. Predictable performance has usually been achieved using task assignment and scheduling, resource reservation, and admission control. We briefly describe the evolution of these mechanisms from means to provide predictable real-time performance to components of adaptation architectures for QoS-sensitive applications.

7.1 Real-Time Computing: A Historical Perspective

Real-time computing has emerged as a distinct computing discipline in the late 60s and early 70s with the publication of key results on predictable scheduling under time constraints, such as [80]. Since then, task assignment and scheduling algorithms that guarantee predictable temporal behavior of the computing system have been studied extensively, both in operations research and real-time systems [11, 16, 22, 25, 35, 36, 48, 113, 143]. Since most real-time applications have been (and still are) periodic in nature, the periodic task model has received special attention. For example, Dhall and Liu [43] and their colleagues developed various assignment algorithms based on the rate monotonic scheduling algorithm [80], or intelligent fixed priority algorithm [114]. Classical methods for task assignment in distributed systems have been developed by minimizing the sum of task processing costs on all assigned processors and interprocessor communications (IPC) costs, including graph-theoretic solutions [132, 133] and integer programming solutions [84] among others [82, 84, 117, 124]. Since the problem of assigning tasks subject to precedence constraints is generally NP-hard [40, 50, 73, 77], some form of enumerative optimization or approximation using heuristics needed to be developed for this problem [36, 37, 66, 117]. Ma *et al.* [84], and Sinclair [124] derived optimal task assignments to minimize the sum of task execution and communication costs with the branch-and-bound (B&B) [68] method. The computational complexity of this method was evaluated using simulation in [84, 124].

With the proliferation of real-time applications in various application domains, predictable resource scheduling algorithms have been reported for process control [10, 12], turbo engine control [72], autonomous robotic systems [83], and avionics [56]. AI-based approaches that utilize application domain knowledge were described in [10, 56, 83]. Solutions to the resource allocation problem have also been presented for specific hardware topologies such as hypercubes [139], hexagonal architectures [120] and mesh-connected systems [150]. Simulated annealing [67] has been proposed as an optimization heuristic. Different flavors of using simulated annealing in the context of real-time task assignment and scheduling can be found in [21, 33, 137, 142].

Abstract execution models of increasing complexity have been developed to capture the resource requirements and computing characteristics of real-time applications. For example, [123] considers an abstract problem where a given task graph is invoked periodically under an end-to-end deadline. A task allocation and message schedule are computed such that the end-to-end deadline is satisfied for each invocation. In [105, 135, 145] efficient methods are considered for allocating periodic tasks where different tasks may have different deadlines. Graph-based heuristics, which attempt to minimize interprocessor communication, are used for task assignment in [135, 145]. Analytic models for load sharing have been developed in [121]. Off-line schedulability analysis [6, 119, 146, 147] was used to verify that the reserved resources are sufficient to meet all timing constraints.

Optimal solutions for task allocation problems in hard real-time systems have been reported. For example, [147] describes an optimal branch-and-bound (B&B) algorithm for task assignment and scheduling on multiprocessors subject to precedence and exclusion constraints. In [109] a task assignment and scheduling algorithm was presented to minimize the *total execution time* (TET) of an arbitrary task graph in a distributed real-time system. An optimal task scheduling algorithm is presented in [98] for communicating periodic tasks and used in [99] to help optimally allocate communicating periodic tasks in heterogeneous distributed real-time systems.

Fault-tolerant real-time systems literature also touched on issues in real-time resource allocation and management. For example, a k-Timely-Fault-Tolerant problem is solved in [97] where an assignment and schedule are found for replicated tasks such that all deadlines are met in the presence of up to k processor failures. In [57, 105] replicated tasks with precedence constraints are considered.

For periodic task sets, the generalized rate monotonic scheduling theory [116] and holistic schedulability analysis [136] have proven to be particularly useful for pre-run-time analyses. These algorithms were coupled with concurrency control methods such as priority ceiling [52], and dynamic priority ceiling protocols [32]. The pre-run-time resource allocation and schedulability analysis methods discussed so far share in common the fact that they are static in nature. They require an exact *a priori* characterization of worst-case offered load and processing capacity. Such characterization, however, is difficult to obtain for practical systems.

In the 80s, with the increasing complexity of real-time applications, such as the space shuttle and Mars pathfinder, the real-time system community introduced the concept of "next generation" *dynamic* real-time systems [127]. It was postulated that in future real-time applications, the computing system may operate in unpredictable environments where load patterns are not known in advance. This was a significant departure from previous literature which typically assumed full knowledge of the task set and execution requirements. The dynamic real-time system model was pioneered by the Spring kernel project [129, 130] which introduced planning-based scheduling and online guarantees for randomly arriving tasks [70, 94, 106, 126, 128, 153, 154].

7.2 Related Work on Operating Systems

More recently, resource reservation has been applied for temporal isolation of real-time applications [63, 75, 88] in an attempt to safely colocate real-time and best-effort processing via appropriate operating system support. While earlier real-time operating systems [39, 138] provided a priority-based interface, new kernel extensions have been proposed to provide real-time guarantees for QoS-sensitive applications. For example, capacity reserves [88] have been used in Mach to allocate processing capacity for multimedia applications [75], and flexible CPU reservation was used in Rialto for efficient scheduling of time-constrained independent activities [63]. Resource shares have been suggested as another mechanism for performance isolation. Examples include hierarchical CPU scheduling [55], proportional-share lottery and stride scheduling [141], and proportional-share allocation for time-shared systems [131]. Proportional-share scheduling has also been used to schedule system services [62]. Other notable real-time kernel extensions include the SMART scheduler in Solaris [95], a real-time scheduler for Linux [125], and the concept of resource-centric kernels [101]. Recently, resource containers have been proposed as an operating system extension motivated by server-side processing needs [19]. Resource containers allow fine-grained control of resource management in monolithic kernels. Different thread activities, as well as kernel processing on their behalf, can be charged to different resource containters as appropriate for the application. In essence, resource containers are similar to capacity reserves with the distinction that the latter was developed for microkernels. As with reserves, containers are specified in platform specific terms (such as CPU cycles) unlike QoS contracts, and require kernel modification.

To complement operating system research on QoS-sensitive scheduling, several operating system extensions have been developed for QoS-sensitive communication. QoS-sensitive operating system communication subsystems have been investigated in [76, 86, 148]. QoS-guaranteed protocol stack implementation in the user space has been proposed in [54, 78]. Real-time upcalls (RTUs) [53] were proposed as a mechanism to schedule protocol processing for networked multimedia applications via event-based upcalls [38]. Rate-based flow control of multimedia streams via kernel-based communication threads is proposed in [149]. Explicit operating system support for communication has been a focus of the Scout operating system, which uses the notion of paths as a fundamental operating system structuring technique [91]. The CORDS path abstraction [49], which is similar to Scout paths, provides a rich framework for development of real-time communication services.

Recent efforts have also addressed an important problem associated with data reception, namely, receive livelock [103]. Receive livelock has been addressed at length in [90] via a combination of techniques (such as limiting interrupt arrival rates, fair polling, processing packets to completion, and regulating CPU usage for protocol processing) to avoid receive livelock and maintain system throughput near the maximum system input capacity under high load. Another approach is to schedule applications in a proportional share manner and use the cumulative rate to limit packet processing to solve the receive livelock problem [62]. Lazy receiver processing (LRP) [44], while not completely eliminating it, significantly reduces the likelihood of receive livelock even under high input load.

7.3 Related Work on Networking

While the real-time and operating system research focused on the end-system, end-to-end QoS has been investigated in the networking community. An extensive survey of such QoS architectures is provided in [29], which provides a comprehensive view of the state of the art in the provisioning of end-to-end QoS. Reservation-based protocols were suggested to provide QoS guarantees for communication services [18, 65]. Support for QoS or preferential service in the network has been examined integrated and differentiated services on the Internet [24, 26, 39, 144]. Several classes of service were considered, including guaranteed service which provides guaranteed delay bounds, and controlled load service which has more relaxed QoS requirements. Issues involved in sharing link bandwidth across multiple classes of traffic are explored in [46]. The signaling required to set up reservations for application flows can be provided by RSVP [151], which initiates reservation setup at the receiver, or ST-II [42], which initiates reservation setup at the sender. Real-time communication services [87] were developed in the context of the Armada project [1] that focused on middleware for real-time communication and fault-tolerance.

7.4 QoS Adaptation and Multimedia Applications

In mid 90s, with the advent of less critical classes of real-time applications, such as multimedia, and with the relative maturity of operating system and networking research on QoS guarantees and service differentiation, the real-time systems community introduced the concept of QoS adaptation into its real-time scheduling, resource management, and admission control literature. Recent research efforts considered general adaptive resource management frameworks for real-time applications with *elastic* QoS constraints. QoS-adaptive service models were presented in [30, 58, 59]. In [3] we described a QoS negotiation framework that attempts to maximize system utility. We proposed a flexible QoS-specification interface for applications with elastic QoS requirements and demonstrated its applicability in the context of a flight control system. This work was extended for communication-oriented applications in [7] which advocated a new architecture for OS communication subsystems. The Q-RAM architecture [102] introduced QoS-sensitive near-optimal resource allocation algorithms for applications with multiple resource requirements and multiple QoS dimensions. FARA [108] presents a hierarchical adaptation model for complex real-time systems. An end-to-end QoS model similar to ours is presented in [59] in the context of a middleware approach to QoS management that requires application cooperation. The approach is extended in [27] to account for practical limitations such as inaccuracies in estimating application resource requirements. In [47] a dynamic distillation method was proposed to adapt to network and client variability via on-line compression techniques. The technique, however, is inapplicable for dealing with server overload.

Novel real-time operating systems and communication architectures were developed to embody QoS adaptation support. The Rialto operating system [64], targeted at multimedia applications, took the approach of dynamically maximizing aggregate system "value" using a resource planner. The Nemesis operating system designed in the context of the Pegasus project [78] investigated support for adaptive multimedia applications. In the multimedia community, various communication architectures have been proposed to support adaptive QoS guarantees. Examples include the QoS-A framework [28], the Heidelberg QoS model [140], V-net [45], NetWorld [31], the QoS-adaptation model of [7], COMETS' Extended Integrated Reference Model (XRM) [74], the OMEGA end-point architecture [93], and the QoS Broker [92]. Odyssey [96], presents a framework for experimenting with application-aware adaptation on mobile computing platforms. A novel RSVP-based QoS architecture supporting integrated services in TCP/IP protocol stacks, running on legacy (e.g., Token Ring and Ethernet) and high-speed ATM LAN networks is described in [20]. The AQUA system [71] has developed QoS negotiation and adaptation support for allocation of CPU and network resources. A native-mode ATM transport layer has been designed and implemented in [9]. It provides support for traffic policing and shaping, but no support is provided for scheduling protocol processing and accounting for implementation overheads and constraints. A good survey of such communication architectures can be found in [15]. Predictable graceful degradation has also been investigated in the context of fault-tolerant real-time computing. Examples are the overload management [107], imprecise computation technique [81], adaptable redundancy [23] and mandatory/optional task scheduling [41].

7.5 Contrast with Adaptware

While earlier architectures for QoS adaptation concerned themselves primarily with introducing new APIs, kernel or network support for QoS, *Adaptware* is concerned with developing a transparent layer between the kernel and the application, that performs QoS management. Thus, we take a non-intrusive approach to QoS adaptation. Our work differs from operating system approaches in that it doesn't require the underlying OS kernel to be redesigned. Instead, we consider the design of QoS-adaptive middleware services on top of commonly-available kernel support. Our approach, therefore, makes our implementation more portable and adaptable. Similarly, plethora of legacy application code can benefit from our approach, since it does not require redesigning the application to make use of the new mechanisms. Concentrating on middleware, our work is complementary to the research on real-time communication protocols [17, 65, 111, 152], network support for QoS [51, 112], and real-time kernel support [63, 76, 89, 101]. Unlike previous middleware approaches [27, 59] which introduced QoS extensions for applications with per-flow QoS constraints, *Adaptware* addresses the issue of transparency of QoS extensions to applications where constraints are imposed on flow aggregates. It implements proper per-traffic-class QoS management even when used by legacy multithreaded (or multiprocess) best-effort servers in which a single pool of identical same-priority threads serves all traffic classes in FCFS order. Thus, while *Adaptware* introduces new programming abstractions that encourage a QoS-sensitive application design methodology, it does not preclude re-use of existing mainstream server code in new QoS-sensitive contexts.

Unlike QoS adaptation methods arising in the multimedia community [28, 31, 93, 140], we consider a QoS-negotiation model suitable not only for multimedia applications but also for web services and embedded systems. While multimedia applications are dominated by the high volume of long-lived communicated data whose source and destination are typically fixed, in web servers flows are short-lived and server load is dominated by incoming request processing, while in embedded systems (e.g., process control) computation is more dominant than communication and dynamic task allocation for better load sharing is an important concern.

Ideally, QoS adaptation should be coordinated at both ends and all internal/intermediate points of the connection. Real-time communication protocols such as RTP [111], can be used with our scheme to help clients adapt application-level performance to network delays. RSVP [152], or real-time channels [17, 65, 86] can be used to reserve host and network bandwidths, when applicable. Further development of such techniques for flexible end-to-end guarantees is beyond the scope of this thesis. Unlike "hard real-time" communication architectures [86], we make no assumptions about the existence of resource-reservation support both on the OS kernel and in the network, although we can make good use of such support if available.

We do not require *a priori* system-load characterization and profiling information for proper maintenance of QoS contracts. Instead, as a part of our adaptation scheme, a self-tuning mechanism is incorporated to adjust itself to the measured load and resource conditions. We develop novel mechanisms for controlling resource allocation that rely on classical control theory. These mechanisms are targeted for server platforms. Since such platforms typically run a single application (the server), we do not deal with issues of trust among independent applications with conflicting requirements for which kernel-level enforcement solutions are more suitable. Overall, the thesis is a major step towards applying the wealth of real-time and classical control concepts in more mainstream domains and emerging applications of Internet computing in order to provide predictable and adaptive performance guarantees.

CHAPTER 8

CONCLUSIONS

In this thesis we investigated building adaptation middleware and operating system extensions for next generation adaptive real-time applications. We first presented motivational material for building *Adaptware*. We focused on web servers as an important application and illustrated the need for adaptation technology in the context of that application. Unlike present state-of-the-art, we demonstrated an adaptation mechanism that enables a server to cope with overload in a graceful manner. We demonstrated how the adaptation mechanism can be used to provide performance isolation, service differentiation, sharing excess capacity, and QoS guarantees.

The goal of adaptation is to maximize service utility. We generalized the adaptation software and separated it from specific application requirements, by extracting from this web case study certain useful abstractions and mechanisms. We further suggested flexible QoS contracts as a main system resource management entity. Guaranteeing the contracted performance entails enforcing the particular resource capacity allocation to the contract. The abstraction must therefore be supported by the operating system or middleware. We developed generalized adaptation techniques with a particular focus on solving the utility optimization problem in applications with elastic QoS requirements specified by QoS contracts. We presented an optimal algorithm for maximizing aggregate user-perceived service utility on server end-systems, and compared it with both reservation- and prioritization-based solutions.

We presented two different architectures for enforcing QoS contracts. The first is implemented within an operating system's communication subsystem. We presented a thread-per-contract end-host communication subsystem architecture and described an implementation of contract handlers as threads scheduled to maximize aggregate service utility. This architecture, called *CLIPS*, was contrasted against a middleware implementation, called *qContracts* that enforces QoS contracts by means of measurement-based adaptation control and load policing. Aspects of classical control theory were re-introduced in the context of a computing environment and used both to analyze the stability of the measurement-based adaptation control loop and to tune the adaptation controller. Experimental results showed that the approach is capable of meeting its stated goals in enforcing QoS contracts regardless of the bottleneck resource, platform speed, and load mix. An important feature of our architecture is the capability for automated self-profiling that estimates platform speed and characterizes the application such that QoS contract specifications can be mapped into resource requirements without *a priori* offline pre-computation.

We extended *Adaptware* to provide a novel scheme for QoS negotiation in distributed real-time applications. This scheme may be applicable for the design of server farms, extending the interface of such services in that (i) it adopts a notion of QoS contracts that allows for defining QoS compromises and supports graceful QoS degradation, and (ii) it provides a generic means to maximize service utility for the community of clients in a distributed server. Our distributed QoS negotiation method improved the guarantee ratio over traditional admission control algorithms and increases the application-level perceived utility of the system. The proposed QoS-negotiation architecture was incorporated into *RTPOOL*, an example middleware service which implements a computing resource manager for a pool of processors.

Finally, we presented techniques for schedulability analysis when *Adaptware* is applied to hard real-time applications. A new B&B algorithm we presented for combined task and message schedulability analysis in a distributed real-time system to verify temporal correctness before the system is deployed. A simulation study has shown that the algorithm scales well with respect to system size and the number of communicated messages among tasks. A heuristic version of the algorithm was presented, where a greedy technique is used to trade optimality for speed. We also suggested a way to generalize the algorithm for a more practical resource model.

The presented work on *Adaptware* represents a seed for many interesting possible ramifications and future research directions. Of particular interest is the use of the well developed control theory in novel computing contexts such as web servers. Several extensions are possible in that regard. For example, Robust Control Theory provides theoretical foundations for choosing control parameters and algorithms such that the resulting system performance is insensitive to the inaccuracies and unpredictability in the controlled system model. Applied to web servers, such theory can produce robust load control algorithms that achieve the desired application QoS even when server load profiles, execution costs, and available resources are inadequately modeled, unpredictable, or partially unknown. A complimentary direction to this research is to produce more accurate mechanisms for online characterization of service model parameters to be used, e.g., for proper QoS mapping. For example, while we succeeded in modeling web service execution times accurately for HTTP 1.0 requests, it may be interesting to investigate the accuracy of this method with HTTP 1.1 persistent connections.

On a different dimension, the work presented in this thesis focused mostly on resource management on a single machine, more research is needed on distributed resource management, e.g., within server farms. For example, it is interesting to investigate load sharing algorithms that globally optimize service utility in the farm. Integration of these algorithms with core networking support (such as diff-serv architecture), or network appliances such as proxy caches is another avenue for future extensions.

From a performance evaluation standpoint, while the thesis demonstrates a proof of concept on a non-standard operating system, it is interesting to implement the presented mechanisms in Linux or a similar widely-used OS and investigate the resulting actual performance in terms of predictability and adaptability, as well as the actual overheads associated with QoS-sensitive resource management.

On a more abstract scale, it is interesting to view *Adaptware* as a core for multi-dimensional QoS optimization. Adaptation mechanisms presented in this thesis consider only a single dimension where all QoS levels can be linearly sorted by their importance. In more general systems, QoS may have several dimensions such as timeliness, fault-tolerance, security, etc. A theoretical framework is needed for multi-dimensional QoS optimization in such applications.

APPENDICES

APPENDIX A

Surveyed Shopping Sites

The table below lists the surveyed sites, the size of each site in Kbytes, the Average size (in KB) of HTML files, GIFs and JPGs, and the percentage of total bytes attributed to images. All of the surveyed sites belong the shopping category in the sense that they advertize and allow purchasing items or services on-line. These include clothing items, greeting cards, household articles, furniture, arts, flowers and adult services.

Site Name	Total Size	HTML	GIF	JPG	Images
http://artessentials.com/products.htm	332K	13.5K	4.7K	3.7K	26%
http://www.rlcgroup.com/oracle/	211K	1.9K	10.3K	24.8K	98%
http://www.caroline-b.com/	761K	7.2K	3.7K	9.1K	65%
http://www.vidmail.com/	88K	3.7K	4.7K	-	16%
http://www.crossdress.net/	1896K	5.5K	23.1K	34.7K	92%
http://www.videocatalog.com/	20K	2.2K	-	11.4K	54%
http://www.lacis.com/	2167K	8.1K	6.8K	7.9K	88%
http://artessentials.com/products.htm	332K	13.5K	4.8K	3.7K	26%
http://www.catscan.com/nancy/index.htm	299K	3.1K	5.3K	13.7K	82%
http://www.shearcomfort.com/	144K	11.2K	23.7K	7.7K	91%
http://www.enterpriseart.com/	784K	5.8K	10.3K	4.8K	75%
http://home.erols.com/swimwear/	897K	1.1K	1.6K	28.2K	97%
http://www.iinet.net.au/ elirab/	1206K	5.0K	4.2K	15K	74%

continued

http://www.crafts2urdoor.com/	223K	5.8K	12.8K	-	63%
http://www.mysticvaly.com/	966K	7.5K	14.3K	17.9K	81%
http://www.reunionblues.com/	518K	7.1K	11.3K	-	83%
http://www.coorsandco.com/CA_index.html	818K	5.5K	9.1K	-	83%
http://www.rollerwarehouse.com/	177K	0.8K	4.2K	7.3K	57%
http://www.indiansilk.com/	749K	1.7K	33.2K	-	97%
http://www.indiaworld.co.in/home/narita/index.html	627K	4.5K	16.1K	24.2K	83%
http://vermontcountrystore.com/vcs/vcs.htm	539K	2.9K	13.1K	35.1K	93%
http://www.cuddledown.com/	1249K	8.0K	13.5K	17.2K	81%
http://www.tweeds.com/weltotweed.html	8416K	4.8K	7.7K	26.3K	89%
http://www.damartusa.com/	3395K	4.1K	16.9K	-	85%
http://www.sabaki.com/Products/English/index.html	1054K	4.2K	30.7K	-	96%
http://www.ishops.com/hc/	176K	6.7K	2.9K	60.3K	53%
http://heels.sexyshoe.com/	2682K	9.6K	20.6K	21.1K	74%
http://www.silhouettes.com/	1144K	5.2K	2.1K	20.2K	47%
http://www.bcsupernet.com/users/sativa/	99K	4.7K	9.7K	4.5K	74%
http://www.smith-hawken.com/	426K	7.7K	2.7K	6.2K	44%
http://www.mind.net/darnell/	3348K	3.8K	28.4K	-	99%
http://www.bugleboy.com/virtualstore/	1215K	3.6K	1.8K	10.7K	75%
http://www.ellemag.com/	8657K	6.1K	4.1K	7.9K	4%
http://www.llbean.com/	53K	1.4K	0.8K	16.5K	74%
http://www.jantzenswim.com/	219K	1.3K	7.1K	8.4K	83%

continued

http://www.viamall.com/fredholly/	21446K	4.5K	13.8K	83.8K	94%
http://www.baroness.com/features/	1124K	2.4K	11.2K	20.2K	90%
http://www.r2intertec.com/basicapparel/	366K	8.4K	15.9K	17.4K	78%
http://www.shoesforcrews.com/catalog/	1514K	9.0K	9.3K	27.8K	86%
http://www.furs.com/FUR/	139K	5.5K	0.67K	-	6%
http://www.bullock-jones.com/	200K	0.9K	33K	151K	91%
http://www.aldenshoes.com/	69K	1.9K	0.96K	3.6K	20%
http://www1.viaweb.com/austads/	605K	5.9K	2.5K	-	35%
http://www.koalaswim.com/main.htm	106K	4.1K	2.7K	17.7K	8%
http://venusmodelsearch.com/	241K	5.6K	-	3.7K	28%
http://www.onehanesplace.com/	49K	2.9K	6.1K	5.4K	34%
;http://www.ullapopken.com/	5137K	4.1K	3.5K	13.0K	7%
http://www.tallclassics.com/	170K	3.6K	5.1K	8.0K	84%
http://www.hottouch.com/	102K	4.0K	37.1K	5.5K	79%
http://www.wetride.com/	128K	5.5K	7.2K	8.0K	81%
http://www.erotica.byus.com/	426K	2.7K	7.5K	12.6K	95&
http://www.freelove.com/	292K	1.3K	6.8K	12.6K	42%
http://www.swedenteens.com/	422K	4.0K	6.5K	10.8K	82%
http://www.pornpeepshows.com/	134K	3.7K	5.3K	6.8K	82%
http://3sex.com/guests	418K	1.7K	7.1K	23.4K	76%
http://porn.raunchysexx.com/	161K	5.1K	13.1K	15.2K	83%
http://www.xxxsexphotos.com/	259K	9.3K	5.9K	9.5K	55%
http://www.smutland.com/	446K	3.6K	3.9K	25.4K	90%

continued

http://www.bettersex.com/	265K	3.4K	4.6K	-	59%
http://www.artcat.com/products.htm	4235K	4.6K	10.7K	53.5K	97%
http://www.bi-furniture.com/html/products.htm	475K	4.2K	7.8K	7.9K	75%
http://www.ballard-designs.com/	342K	1.3K	2.5K	15.1K	89%
http://www.ellenburgs.com/	147K	4.5K	6.8K	39.3K	80%
http://www.cf-direct.com/catalog.htm	1822K	8.1K	8.8K	9.9K	78%
http://www.primenet.com/ tashjian/	42K	3.8K	-	-	0%
http://www.ethanallen.com/home_garden/coverpage/	1509K	3.4K	15.0K	6.5K	87%
http://www.majesticproducts.com/	3829K	5.5K	37.6K	17.2K	91%
http://www.countrybed.com/	519K	3.1K	13.5K	26.2K	91%
http://www.safariland.com/	365K	7.9K	5.7K	7.3K	55%
http://www.starkbros.com/CAT_TOP.HTM	383K	9.5K	2.7K	6.7K	29
http://www.Rocknrescue.com/catalog.htm	3229K	9.5K	3.3K	9.4K	78%
http://www.marmot.com/htmpages/homepg.htm	262K	4.8K	8.8K	-	50%
http://www.cookiebouquets.com/	58K	7.3K	2.8K	22.6K	86%
http://www.harborsweets.com/	155K	5.2K	3.5K	-	38%
http://www.sportys-catalogs.com/	166K	5.5K	1.8K	9.2K	38%
http://www.folder-factory.com/	100K	1.8K	49.8K	4.2K	82%
http://artonline.com.br/art/indice_e.htm	194K	2.4K	4.0K	3.1K	41%
http://www.franceantiq.fr/sna/berko/ber_uk.htm	3365K	11.3K	0.6K	29.7K	58%
http://www.franceantiq.fr/cne/manic/manic-uk.htm	17081K	30K	2.5K	13.3K	21%
http://www.neaguild.com/macrodel/	872K	2.4K	19K	-	91%

APPENDIX B

Configuration File

```
# Caution:  Make sure that CONFIG_DIR in http_adaptive.h points to the
# directory where this file is located.

# The Adaptive Web Server has two main modes of operation; a LEARNING mode
# and a standard mode.  These modes can be toggled between by setting/resetting
# the LEARNING flag.  The purpose of the LEARNING mode is to let the server
# perform self-profiling upon installation on a new platform.  The server
# will still deliver content while LEARNING, although no QoS guarantees are
# offered in this mode.  Ideally, the server should be heavily loaded while
# learning.  If the server receives only occasional requests, learning is not
# recommended.  If the load is large enough, the server will learn within 10
# minutes or so.  It should then be configured for desired QoS guaranteed
# (see below) and restarted with LEARNING disabled.

# Execution Flags in LEARNING mode:
# ---------------------------------

# LEARNING

# The above flag enables the LEARNING mode.  Comment it out to turn learning
# off.

# SELF_MODULATION

# The above flag should be turned on in LEARNING mode.  It allows the server
# to modulate internal load at will (e.g., by degradation or rejection) even
# at constant offered external load.  This toggling enhances LEARNING since
```

```
# it effectively changes the load on the server allowing the server to
# "learn" the effects of load change.  If this flag is commented out, you
# must ensure that the load applied to the server while LEARNING has enough
# inherent variability on its own, or else learning will produce erroneous
# results

ESTIMATOR_PERIOD 5

# The above flag defines the period (in seconds) over which load statistics
# are averaged during LEARNING. It is recommended to keep this period
# sufficiently large to reduce the effect of measurement noise.  The flag has
# no effect if learning is disabled.

# General execution flags:
# -------------------------

SERVER_IP 15.4.91.45
SERVER_PORT 8011

# The above flags define server IP and port number.

# VERBOSE
# Comment the above flag out if you don't want excessive printouts to the
# screen.

TARGET_UTILIZATION 80

# The above flag defines the desired target utilization for the server.
# Utilizations above target will be treated as overload and dealt with
# accordingly.  Target utilization is specified as a percentage (0 to 100).
# A target utilization close to 100 will result in bad performance due to
# jitter and frequent listen queue overflow.  It is best to keep target
# utilization at or below 80
HTDOCS_TREE /degraded
HTDOCS_TREE /good

# The above flag defines the subdirectory names containing the different
# content trees.  The definitions must be organized in increasing order of
# quality of content.  The corresponding subdirectories should be located
```

```
# in the main service directory of the server.  If one of the content trees
# is served from the main directory itself it is denoted by "/".  There can
# be only one content tree if desired.  In this case, degradation will be
# accomplished by request rejection.

ENABLE_ADAPTOR

# The above flag enables the content adaptor.  The content adaptor is the
# part of the adaptive server architecture that prepends the "right" content
# tree name to the requested URL in accordance with load conditions.  If the
# adaptor is disabled (i.e., the above flag is commented out) the middleware
# will still execute all its functions as usual, except that the results will
# be masked from the server.  The URL in each request will remian unchanged.
# Disabling the toggle is useful, e.g., to compute the pure overhead of our
# middleware compared to a non-adaptive server

# Performance Isolation and Tiered Services Flags:
# ------------------------------------------------
# The Adaptive Server allows defining virtual private servers of guaranteed
# maximum rate and bandwidth.  The middleware will allocate enough capacity
# for each virtual private server to meet its bandwidth requirement as
# long as the request rate does not exceed the maximum specified rate.
# The format for specifying the maximum rate and badnwidth is:

# PREMIUM_CLASS class_id rate bandwidth

# Currently:  (class_id) is the name of the text file that contains a
# listing of all clients belonging to this class.  Clients are identified by
# their IP address.  The file should contain one IP address per line, and
# should reside in the configuration directory.  (rate) must be specified in
# req/s.  (bandwidth) must be specified in Mb/s (i.e., Megabits/s)

# PREMIUM_CLASS class1 100 50

# PREMIUM_CLASS class2 100 26

# Uncomment the flag below to let each virtual server be strictly confined
# to its allocated resource capacity.  When the flag is commented out virtual
# private servers are allowed to exceed their capacity allocation if excess
```

```
# capacity exists on the server.

# STRICT_POLICING
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. Abdelzaher, M. Bjorklund, S. Dawson, W. Feng, F. Jahanian, S. Johnson, Marron, A. Mehra, T. Mitton, A. Shaikh, K. Shin, J. Wang, and H. Zou, "ARMADA middleware and communication services," *Journal of Real-Time Systems*, vol. 16, no. 1, , January 1999.

[2] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin, "RTCAST: Lightweight multicast for real-time process groups," in *IEEE Real-Time Technology and Applications Symposium*, Boston, Massachusetts, June 1996.

[3] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin, "QoS negotiation in real-time systems and its application to automated flight control," in *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[4] T. F. Abdelzaher and N. Bhatti, "Web content adaptation to improve server overload behavior," in *International World Wide Web Conference*, Toronto, Canada, May 1999.

[5] T. F. Abdelzaher and N. Bhatti, "Web server QoS management by adaptive content delivery," in *International Workshop on Quality of Service*, London, UK, June 1999.

[6] T. F. Abdelzaher and K. G. Shin, "Optimal combined task and message scheduling in distributed real-time systems," in *IEEE Real-Time Systems Symposium*, Italy, Pisa, December 1995.

[7] T. F. Abdelzaher and K. G. Shin, "End-host architecture for qos-adaptive communication," in *IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[8] R. Agne, "A distributed offline scheduler for distributed hard real-time systems," in *Distributed Computer Control Systems. Proceedings of the 10th IFAC Workshop*, pp. 35–40, Summering, Austria, September 1991.

[9] R. Ahuja, S. Keshav, and H. Saran, "Design, implementation, and performance of a native mode ATM transport layer," in *INFOCOM*, pp. 206–214, March 1996.

[10] M. Alfano, A. Di-Stefano, L. Lo-Bello, O. Mirabella, and J. H. Stewman, "An expert system for planning real-time distributed task allocation," in *Proceedings of the Florida AI Research Symposium*, Key West, FL, USA, May 1996.

[11] H. H. Ali and H. El-Rewini, "Task allocation in distributed systems: a split graph model," *J. Combin. Math. Combin. Comput.*, vol. 14, no. 1, pp. 15–32, January 1993.

[12] P. Altenbernd, C. Ditze, P. Laplante, and W. Halang, "Allocation of periodic real-time tasks," in *20th IFAC/IFIP Workshop*, Fort Lauderdale, FL, USA, November 1995.

[13] E. M. Atkins. *Reasoning About and In Time when Building Plans for Safe, Fully-Automated Aircraft Flight*. Ph.D. Thesis Proposal, December 1996.

[14] E. M. Atkins, E. Durfee, and K. G. Shin, "Plan development in circa using local probabilistic models," in *Uncertainty in Artificial Intelligence: Proceedings of the Twelfth Conference*, pp. 49–56, August 1996.

[15] C. Aurrecoechea, A. Cambell, and L. Hauw, "A survey of QoS architectures," in *4th IFIP International Conference on Quality of Service*, Paris, France, March 1996.

[16] K. R. Baker, *Introduction to Sequencing and Scheduling*, Wiley & Sons, 1974.

[17] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang, "The tenet real-time protocol suite: Design, implementation, and experiences," *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pp. 1–10, February 1996.

[18] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang, "The tenet real-time protocol suite : design, implementation, and experiences," *IEEE/ACM Transactions on Networking*, vol. 4, no. 1, pp. 1–10, February 1996.

[19] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Third USENIX Symposium on Operating Systems Design and Implementation*, pp. 45–58, New Orleans, Louisiana, February 1999.

[20] T. Barzilai, D. Kandlur, A. Mehra, D. Saha, and S. Wise, "Design and implementation of an RSVP-based quality of service architecture for integrated services Internet," in *DCS*, May 1997.

[21] J. E. Beck and D. P. Siewiorek, "Simulated annealing applied to multicomputer task allocation and processor specification," in *Proceedings of 8th IEEE Symposium on Parallel and Distributed Processing*, pp. 232–239, October 1996.

[22] A. Billionnet, M.-C. Costa, and A. Sutter, "An efficient algorithm for a task allocation problem," *J. Assoc. Comput. Mach.*, vol. 39, no. 3, pp. 502–518, March 1992.

[23] M. Bizzarri, P. Bizzarri, A. Bondavalli, F. Di-Giandomenico, F. Tarini, P. Laplante, and W. Halang, "Design of flexible and dependable real-time applications," in *20th IFAC/IFIP Workshop*, Lauderdale, FL, USA, November 1995.

[24] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *An Architecture for Differentiated Services*. Internet Draft (draft-ietf-diffserv-arch-01.txt), August 1998.

[25] S. H. Bokhari, "A network flow model for load balancing in circuit-switched multicomputers," *IEEE Trans. Parallel and Distrib. Sys.*, vol. 4, no. 6, pp. 649–657, June 1993.

[26] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: An overview," *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[27] S. Brandt and G. Nutt, "A dynamic quality of service middleware agent for mediating application resource usage," in *Real-Time Systems Symposium*, pp. 307–317, Madrid, Spain, December 1998.

[28] A. Cambell, G. Coulson, and D. Hutchison, "A quality of service architecture," *ACM Computer Communications Review*, April 1994.

[29] A. T. Campbell, C. Aurrecoechea, and L. Hauw, "A review of QoS architectures," *MSJ*, 1996.

[30] S. Chatterjee, J. Sydir, B. Sabata, and T. Lawrence, "Modeling applications for adaptive qos-based resource management," in *Proceedings of the 2nd IEEE High-Assurance System Engineering Workshop*, Bethesda, Maryland, August 1997.

[31] D. Chen, R. Colwell, H. Gelman, P. K. Chrysanthis, and D. Mosse, "A framework for experimenting with QoS for multimedia services," in *International Conference on Multimedia Computing and Networking*, 1996.

[32] M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *Journal of Real Time Systems*, vol. 2, no. 4, pp. 325–346, 1990.

[33] S. T. Cheng, S. I. Hwang, and A. K. Agrawala, "Schedulability oriented replication of periodic tasks in distributed real-time systems," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, Canada, 1995.

[34] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Journal of Real-Time Systems*, vol. 2, no. 3, pp. 181–194, September 1990.

[35] W. W. Chu, "Task allocation in distributed data processing," *IEEE Computer*, vol. 13, pp. 57–69, Nov 1980.

[36] W. W. Chu and L. M. Lan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. on Computers*, vol. C-36, no. 6, pp. 667–679, June 1987.

[37] W. W. Chu and K. Leung, "Module replication and assignment for real-time distributed processing systems," *Proc of IEEE*, vol. 75, no. 5, pp. 547–562, May 1987.

[38] D. D. Clark, "The structuring of systems using upcalls," in *Symposium on Operating Systems Principles*, pp. 171–180, 1985.

[39] R. Clark, E. Jensen, and F. Reynolds, "An architectural overview of the Alpha real-time distributed kernel," in *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, 1992.

[40] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley and Sons, New York, 1976.

[41] R. Davis, S. Punnekkat, N. Audsley, and A. Burns, "Flexible scheduling for adaptable real-time systems," in *Proceedings Real-Time Technology and Applications Symposium*, pp. 230–239, Chicago, IL, USA, May 1995.

[42] L. Delgrossi and L. Berger, "Internet stream protocol version 2 (ST-2) protocol specification - version ST2+," *Request for Comments RFC 1819*, August 1995. ST2 Working Group.

[43] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[44] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd OSDI Symposium*, pp. 261–275, October 1996.

[45] B. Field, T. Znati, and D. Mosse, “V-net: A framework for a versatile network architecture to support real-time communication performance guarantees,” in *InfoComm*, 1995.

[46] S. Floyd and V. Jacobson, “Link-sharing and resource management models for packet networks,” *IEEE Transactions on Networks*, vol. 3, no. 4, pp. 365–386, August 1995.

[47] A. Fox, S. Gribble, E. A. Brewer, and E. Amir, “Adapting to network and client variability via on-demand dynamic distillation,” in *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 160–170, Cambridge, Massachusetts, October 1996.

[48] S. French, *Sequencing and Scheduling*, Halsted Press, 1982.

[49] F.Travostino, E.Menze, and F.Reynolds, “Paths: Programming with system resources in support of real-ti me distributed applications,” in *Proc. IEEE Workshop on Object-Oriented Real-Time Dependabl e Systems*, February 1996.

[50] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[51] L. Georgiadis, R. Guerin, V. Peris, and R. Rajan, “Efficient support of delay and rate guarantees in an Internet,” in *ACM SIGCOMM*, Stanford, California, August 1996.

[52] J. B. Goodenough and L. Sha, “The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks,” *Ada Letters*, vol. 7, no. 8, pp. 20–31, August 1998.

[53] R. Gopalakrishnan and G. M. Parulkar, “A real-time upcall facility for protocol processing with QoS guarantees,” in *Symposium on Operating Systems Principles*, p. 231, December 1995.

[54] R. Gopalakrishnan and G. Parulkar, “Efficient user space protocol implementations with qos guarantees using real-time upcalls,” *IEEE/ACM Transactions on Networking*, 1998.

[55] P. Goyal, X. Guo, and H. Vin, “A hierarchical cpu scheduler for multimedia operating systems,” in *Proceedings of Second Usenix Symposium on Operating System Design and Implementation*, Seattle, Washington, October 1996.

[56] C. M. Hopper and Y. Pan, “Task allocation in distributed computer systems through an ai planner solver,” in *Proceedings of IEEE 1995 National Aerospace and Electronics Conference*, volume 2, pp. 610–616, Dayton, OH, USA, May 1995.

[57] C.-J. Hou and K. G. Shin, “Replication and allocation of task modules in distributed real-time systems,” in *24th IEEE Symposuim on Fault-Tolerant Computing Systems*, pp. 26–35, June 1994.

[58] D. Hull, A. Shankar, K. Nahrstedt, and J. W. S. Liu, “An end-to-end qos model and management architecture,” in *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pp. 82–89, San Francisco, California, December 1997.

[59] M. Humphrey, S. Brandt, G. Nutt, and T. Berk, “The DQM architecure: middleware for application-centered qos resource management,” in *Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, San Francisco, California, December 1997.

[60] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, January 1991.

[61] K. Jeffay, "On latency management in time-shared operating systems," in *Real-Time Operating Systems and Software*, pp. 86–90, May 1994.

[62] K. Jeffay, D. Smith, A. Moorthy, and J. Anderson, "Proportional share scheduling of operating system services for real-time applications," in *Real-time Systems Symposium*, pp. 480–491, Madrid, Spain, December 1998.

[63] M. Jones, D. Rosu, and M.-C. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," in *16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[64] M. B. Jones and P. J. Leach, "Modular real-time resource management in the rialto operating system," Technical Report MSR-TR-95-16, Microsoft Research, Advanced Technology Division, May 1995.

[65] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[66] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. on Computers*, vol. C-33, no. 11, pp. 1023–1029, November 1984.

[67] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[68] W. H. Kohler and K. Steiglitz, *Computer and Job-Shop Scheduling Theory*, chapter Enumerative and Iterative Computational Approach, pp. 229–287, Wiley and Sons, 1976.

[69] W. H. Kohler and K. Steiglitz, "Enumerative and iterative computational approach," *Computer and Job-Shop Scheduling Theory*, pp. 229–287, 1976.

[70] G. Koren and D. Shasha, "D-over: An optimal on-line scheduling algorithm for overloaded real-time systems," in *IEEE Real-Time Systems Symposium*, pp. 290–299, Phoenix, Arizona, December 1992.

[71] L. Krishnamurthy, *AQUA: An Adaptive Quality of Service Architecture for Distributed Multimedia Applications*, PhD thesis, University of Kentucky, 1997.

[72] J. L. Lanet, "Task allocation in a hard real-time distributed system," in *Proceedings of the 2nd Conference on Real-Time Systems*, pp. 244–252, Szlarska Poreba, Poland, September 1995.

[73] E. L. Lawler, *Deterministic and Stochastic Scheduling*, chapter Recent Developments in Deterministic Sequencing and Scheduling: A Survey, pp. 35–74, Reidel, Dordrecht, The Netherlands, 1982.

[74] A. Lazar, S. Bhonsle, and K. Lim, "A binding architecture for multimedia networks," *Journal of Parallel and Distributed Computing*, vol. 30, pp. 204–216, Nevember 1995.

[75] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with processor reservation and dynamic QoS in real-time mach," in *Proceedings of Multimedia*, Japan, March 1996.

[76] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in real-time Mach"," in *Proceedings of the Real-time Technology and Applications Symposium*, June 1996.

[77] J. K. Lenstra and A. H. G. R. Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, vol. 26, no. 1, pp. 23–35, Jan 1978.

[78] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *JSAC*, June 1997.

[79] S. Liden, "The evolution of flight management systems," in *of the 1994 IEEE/AIAA Thirteenth Digital Avionics Systems Conference*, pp. 157–169. IEEE, 1995.

[80] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[81] J. W.-S. Liu, W. Shih, K.-J. Lin, R. Bettati, and J. Chung, "Imprecise computations," *IEEE Proceedings*, January 1994.

[82] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. Comput.*, vol. 37, no. 11, pp. 1384–1397, November 1988.

[83] T. C. Lueth and T. Laengle, "Task description, decomposition and allocation in a distributed autonomous multi-agent robot system," in *Proceedings of International Conference on Intelligent Robots and Systems*, pp. 1516–1523, Munich, Germany, September 1994.

[84] P. Y. R. Ma and et. al., "A task allocation model for distributed computing systems," *IEEE Trans. on Computers*, vol. C-31, no. 1, pp. 41–47, January 1982.

[85] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," in *Symposuim on Operating System Principles*, pp. 244–255, December 1993.

[86] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring communication for quality of service guarantees," in *IEEE Real-Time Systems Symposium*, pp. 144–154, Washington, DC, December 1996.

[87] A. Mehra, A. Shaikh, T. Abdelzaher, Z. Wang, and K. Shin, "Realizing services for guaranteed qos communication on a microkernel operating system," in *IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[88] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[89] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pp. 90–99, May 1994.

[90] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Winter USENIX Conference*, January 1996.

[91] D. Mosberger and L. L. Peterson, “Making paths explicit in the Scout operating system,” in *OSDI*, pp. 153–168, October 1996.

[92] K. Nahrstedt and J. Smith, “The QoS broker,” *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, 1995.

[93] K. Nahrstedt and J. Smith, “Design, imlementation, and experiences with the OMEGA end-point architecture,” *IEEE JSAC*, September 1996.

[94] M. D. Natale and J. A. Stankovic, “Dynamic end-to-end guarantees in distributed real-time systems,” in *Proc. Real-Time Systems Symposium*, pp. 216–227, December 1994.

[95] J. Nieh and M. S. Lam, “The design, implementation, and evaluation of SMART: A scheduler for multimedia applications,” in *16th ACM Symposium on Operating System Principles*, pp. 184–197, Saint-Malo, France, October 1997.

[96] B. D. Noble and M. Satyanrayanan. *Experience with Adaptive Mobile Applications in Odyssey*. to appear in Mobile Networking and Applications.

[97] Y. Oh and S. H. Son, “Scheduling hard real-time tasks with tolerance to multiple processor failures,” *Multiprocessing and Multiprogramming*, vol. 40, pp. 193–206, 1994.

[98] D.-T. Peng and K. G. Shin, “Optimal scheduling of cooperative tasks in a distributed system using an enumerative method,” *IEEE Trans. Software Engineering*, vol. 19, no. 3, pp. 253–267, Mar 1993.

[99] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, “Assignment and scheduling of communicating periodic tasks in distributed real-time systems,” *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 745–758, December 1997.

[100] R. Rainey. *ACM: The Aerial Combat Simulation for X11*, February 1994.

[101] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource kernels: A resource-centric approach to real-time systems,” in *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[102] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, “Practical solutions for qos-based resource allocation problems,” in *Real-time Systems Symposium*, pp. 296–306, Madrid, Spain, December 1998.

[103] K. K. Ramakrishnan, “Performance considerations in designing network interfaces,” *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, February 1993.

[104] K. Ramamritham, “Allocation and scheduling of complex periodic tasks,” in *Proc. Int’l Conf. on Distributed Computing Systems*, pp. 108–115, 1990.

[105] K. Ramamritham, “Allocation and scheduling of precedence-related periodic tasks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, April 1995.

[106] K. Ramamritham, J. A. Stankovic, and W. Zhao, “Distributed scheduling of computing tasks with deadlines and resource requirements,” *IEEE Transactions on Computers*, vol. 38, no. 8, pp. 1110–1123, August 1989.

[107] P. Ramanathan, "Graceful degradation in real-time control applications using (m, k)-firm guarantee," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, Seattle, WA, USA, June 1997.

[108] D. Rosu, K. Schwan, and S. Yalamanchili, "FARA - a framework for adaptive resource allocation in complex real-time systems," in *Real-time Technology and Applications Symposium*, pp. 79–84, Denver, Colorado, June 1998.

[109] P. Scholz and E. Harbeck, "Task assignment for distributed computing," in *Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing*, pp. 270–277, Shanghai, China, March 1997.

[110] J. Schreur, "B737 flight management computer flight plan trajectory computation and analysis," in *Proceedings of the American Control Conference*, pp. 3419–3429, June 1995.

[111] H. Schulzrinne, "RTP: The real-time transport protocol," in *MCNC 2nd Packet Video Workshop*, volume 2, Research Triangle Park, North Carolina, December 1992.

[112] H. Schulzrinne, "A comprehensive multimedia control architecture for the Internet," in *NOSSDAV*, St. Louis, Missouri, May 1997.

[113] S. Selvakumar and C. S. R. Murthy, "Static task allocation of concurrent programs for distributed computing systems with processor and resource heterogeneity," *Parallel Computing*, vol. 20, no. 6, pp. 835–851, 1994.

[114] O. Serlin, "Scheduling of time critical processes," in *Proc. of AFIPS 1972 Spring Joint Computer Conf.*, pp. 925–932, Montvale, N. J., 1972, AFIPS Press.

[115] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On task schedulability in real-time control systems," in *IEEE Real-Time Systems Symposium*, pp. 13–21, Washington, DC, December 1996.

[116] L. Sha, R. Rajkumar, and S. S. Sathaye, "Generalized rate monotonic scheduling theory: A framework for developing real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 68–82, January 1994.

[117] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. on Computers*, vol. C-34, no. 3, pp. 197–203, March 1985.

[118] T. Shepard and M. Gagne, "A model of the F18 mission computer software for pre-run-time scheduling," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 62–69, 1990.

[119] T. Shepard and M. Gagne, "A pre-run-time scheduling algorithm for hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 669–677, Jul 1991.

[120] K. G. Shin and C. J. Hou, "Evaluation of load sharing in harts with consideration of its communication activities," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 7, pp. 724–739, July 1996.

[121] K. G. Shin and C. J. Hou, "Analytic models of adaptive load sharing schemes in distributed real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 740–761, July 1993.

[122] F. G. Shinskey, *Process control systems: application, design, and tuning*, McGraw-Hil, New York, 4th edition edition, 1996.

[123] S. B. Shukla and D. P. Agrawal, "A framework for mapping periodic real-time applications on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 7, pp. 778–784, July 1994.

[124] J. B. Sinclair, "Efficient computation of optimal assignments for distributed tasks," *J. of Parallel and Distributed Computing*, vol. 4, pp. 342–362, 1987.

[125] B. Srinivasan, S. Pather, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *Real-time Technology and Applications Symposium*, pp. 112–120, Denver, Colorado, June 1998.

[126] J. Stankovic, "Decentralized decision making for task reallocation in a hard real-time system," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 341–355, March 1989.

[127] J. Stankovic and K. Ramamritham, *Hard Real-time Systems*, IEEE Press, 1988.

[128] J. A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Transactions on Computers*, vol. 34, no. 12, pp. 1130–1143, December 1985.

[129] J. A. Stankovic and K. Ramamritham, "The design of the Spring kernel," in *Proc. Real-Time Systems Symposium*, pp. 146–157, December 1987.

[130] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A new paradigm for real-time systems," *IEEE Software*, pp. 62–72, May 1991.

[131] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton, "A proportional-share resource allocation algorithm for real-time time-shared systems," in *Real-Time Systems Symposium*, pp. 288–299, Washington, DC, December 1996.

[132] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithm," *IEEE Trans. on Software Engineering*, vol. SE-3, no. 1, pp. 85–93, January 1977.

[133] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer*, vol. 11, pp. 97–106, July 1978.

[134] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," *IEEE Transactions on Networking*, vol. 1, no. 5, pp. 554–565, October 1993.

[135] T.-S. Tia and J. W.-S. Liu, "Assigning real-time tasks and resources to distributed systems," *Internnational Journal of Minim and Microcomputers*, vol. 17, no. 1, pp. 18–25, 1995.

[136] K. Tindell and J. Clark, "Holistic schedulability analysis for hard real-time systems," *Microprocessing and Microprogramming*, vol. 40, pp. 117–134, 1994.

[137] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: An np-hard problem made easy," *J. of Real-Time Systems*, vol. 4, no. 2, pp. 145–166, May 1992.

[138] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proceedings of the USENIX Mach Workshop*, pp. 73–82, October 1990.

[139] B. R. Tsai and K. G. Shin, "Assignment of task modules in hypercube multicomputers with component failures for communication efficiency," *IEEE Transactions on Computers*, vol. C-43, no. 5, pp. 613–618, May 1994.

[140] C. Volg, L. Wolf, R. Herrwich, and H. Wittig, "HeiRAT – quality of service management for distibuted multimedia systems," *Multimedia Systems Journal*, 1996.

[141] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, PhD thesis, Massachusetts Institute of Technology, September 1995.

[142] E. Wells and C. C. Caroll, "An augmented approach to task allocation: Combining simulated annealing with list-based heuristics," in *Proc. Euromicro Workshop*, pp. 508–515, 1993.

[143] R. E. D. Woolsey and H. S. Swanson, *Operations Research for Immediate Applications: A Quick and Dirty Manual*, Harper and Row, 1974.

[144] J. Wroclawski, "Specification of the controlled-load network element service," *Request for Comments (RFC 2211)*, September 1997.

[145] S. S. Wu and D. Sweeping, "Heuristic algorithms for task assignment and scheduling in a processor network," *Parallel Computing*, vol. 20, pp. 1–14, 1994.

[146] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations," *IEEE Trans. Software Engineering*, vol. SE-16, no. 3, pp. 360–369, March 1990.

[147] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 139–154, February 1993.

[148] D. K. Y. Yau and S. Lam, "Migrating sockets for networking with quality of service guarantees," in *International Conference on Network Protocols*, Atlanta, Georgia, October 1997.

[149] D. K. Y. Yau and S. S. Lam, "An architecture towards efficient OS support for distributed multimedia," in *Proc. Multimedia Computing and Networking (MMCN '96)*, January 1996.

[150] S. M. Yoo and H. Y. Youn, "An efficient task allocation scheme for two dimensional mesh-connected systems," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, pp. 501–508, Vancouver, Canada, 1995.

[151] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, pp. 8–18, September 1993.

[152] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Network*, September 1993.

[153] W. Zhao, K. Ramamritham, and J. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Transactions on Computers*, vol. 36, no. 8, pp. 949–960, August 1987.

[154] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 5, pp. 564–577, May 1987.