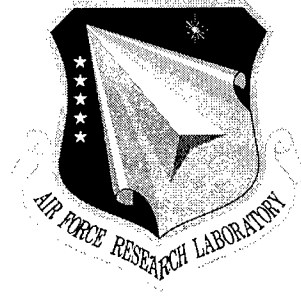


AFRL-IF-RS-TR-2001-193
Final Technical Report
September 2001



END-TO-END RESERVATION SERVICES IN REAL-TIME MACH

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D649/04

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

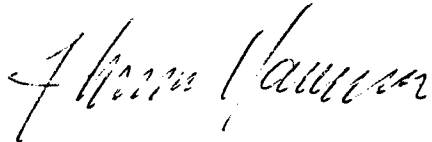
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

20020116 184

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-193 has been reviewed and is approved for publication.



APPROVED: THOMAS F. LAWRENCE
Project Engineer



FOR THE DIRECTOR: WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGA, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

END-TO-END RESERVATION SERVICES IN REAL-TIME MACH

Raj Rajkumar
K. Juvva
A. Molano
C. Lee
J.P. Lehoczky
D.P. Siewiorek
C.W. Mercer
K. Yoshida

Contractor: Carnegie Mellon University
Contract Number: F30602-96-1-0160
Effective Date of Contract: 04 March 1996
Contract Expiration Date: 04 March 1999
Short Title of Work: End-to-End Reservation Services in Real-Time Mach
Period of Work Covered: Mar 96 – Mar 99

Principal Investigator: Raj Rajkumar
Phone: (412) 268-8707
AFRL Project Engineer: Thomas F. Lawrence
Phone: (315) 330-2925

Approved for public release; distribution unlimited.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Thomas F. Lawrence, AFRL/IFGA, 525 Brooks Rd, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Sep 01		3. REPORT TYPE AND DATES COVERED Final Mar 96 - Mar 99
4. TITLE AND SUBTITLE END-TO-END RESERVATION SERVICES IN REAL-TIME MACH			5. FUNDING NUMBERS C - F30602-96-1-0160 PE - 62301E PR - D649 TA - 00 WU - 01	
6. AUTHOR(S) Raj Rajkumar, K. Juvva, A. Molano, S. Oikawa, C. Lee, J.P. Lehoczky, D.P. Siewiorek, C.W. Mercer and K. Yoshida				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie Mellon University Department of Computer Science 5000 Forbes Avenue Pittsburg, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington, VA 22203			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-193	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Thomas F. Lawrence, IFGA, 315-330-2925 DARPA Program Manager: Gary Koob, ITO, 703-696-7463				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The RT-Mach microkernel supports a processor reserve abstraction which permits threads to specify their CPU resource requirements. If admitted by the kernel, it guarantees that the requested CPU demand is available to the requestor. This kernel-supported mechanism is designed to be relatively simple based on the microkernel notion that user-level policies can use this simple mechanism to build more complex and powerful schemes. In this report, the focus is on the needs of such user-level policies in the form of a dynamic Quality of Service (QoS) server. Three goals are sought: 1) explore the necessity, sufficiency, power and flexibility of the kernel-supported reserve mechanism, 2) dynamic management of application quality in real-time and multimedia applications, and 3) investigate our ability to predict and achieve end-to-end application delays in realistic distributed real-time and multimedia applications. A two-pronged approach to accomplish these goals is used. First, the processor reserve abstraction in a user-level dynamic quality of service server is applied. A QoS server can allow applications to dynamically adapt in real-time based on system load, user input or application requirements. Second, the dynamic QoS control capabilities to a distributed multimedia application whose threads have to interact and coordinate with each other within and across processor boundaries are applied. A new notion called continuous thread of control is introduced to assist in bundling processor reserves. The experiments show that we can indeed predict and achieve end-to-end delays in a distributed multimedia application.				
14. SUBJECT TERMS Quality of Service, Multimedia, Real-Time Scheduling, Utility Functions, Operating System Kernel			15. NUMBER OF PAGES 238	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

1.	List of Authors and Papers	ii
2.	Papers:	
	a. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems	1
	b. Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach	16
	c. A Resource Allocation Model for QoS Management	27
	d. Predictable Communication Protocol Processing in Real-Time Mach	37
	e. Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach	47
	f. Operating System Resource Reservation for Real-Time and Multimedia Applications (Thesis)	57
3.	Table of Contents for Thesis	63
4.	List of Figures for Thesis	67
5.	List of Tables for Thesis	69

LIST OF AUTHOR'S AND PAPER'S

- | | | |
|----|--|----|
| 1. | R. Rajkumar, K. Juvva, A. Molano and S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems", To appear in proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998. | 1 |
| 2. | A. Molano, K. Juvva and R. Rajkumar, "Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach", In proceedings of the IEEE Real-Time Systems Symposium, December 1997. | 16 |
| 3. | R. Rajkumar, C. Lee, J. P. Lehoczky and D. P. Siewiorek, "A Resource Allocation Model for QoS Management", Proceedings of the IEEE Real-Time Systems Symposium, December 1997. | 27 |
| 4. | C. Lee, K. Yoshida, C. W. Mercer and R. Rajkumar, "Predictable Communication Protocol Processing in Real-Time Mach", In Proceedings of the Real-Time Technology and Applications Symposium, June 1996. | 37 |
| 5. | C. Lee, R. Rajkumar and C. W. Mercer, "Experiences with Processor Reservation and Dynamic QoS in Real-Time Mach", In Proceedings of Multimedia Japan, March 1996. | 47 |
| 6. | C. W. Mercer, "Operating Systems Resource Reservation in Real-Time and Multimedia", Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, June 1997. | 57 |

Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems

Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shuichi Oikawa

Real-Time and Multimedia Laboratory¹

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

{raj+, kjuvva, amolano, shui}@cs.cmu.edu

Abstract

We consider the problem of OS resource management for real-time and multimedia systems where multiple activities with different timing constraints must be scheduled concurrently. Time on a particular resource is shared among its users and must be globally managed in real-time and multimedia systems. A resource kernel is meant for use in such systems and is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

We identify the specific goals of a resource kernel: applications must be able to explicitly state their timeliness requirements; the kernel must enforce maximum resource usage by applications; the kernel must support high utilization of system resources; and an application must be able to access different system resources simultaneously. Since the same application consumes a different amount of time on different platforms, the resource kernel must allow such resource consumption times to be portable across platforms, and to be automatically calibrated. Our resource management scheme is based on resource reservation [25] and satisfies these goals. The scheme is not only simple but captures a wide range of solutions developed by the real-time systems community over several years.

One potentially serious problem that any resource management scheme must address is that of allowing access to multiple resources simultaneously and in timely fashion, a problem which is known to be NP-complete [5]. We show that this problem of simultaneous access to multiple resources can be practically addressed by resource decoupling and resolving critical resource dependencies immediately. Finally, we demonstrate our resource kernel's functionality and flexibility in the context of multimedia applications which need processor cycles and/or disk bandwidth.

1. Motivation for Resource Kernels

Example real-time systems include aircraft fighters such as F-22 and the Joint Strike fighter [19], beverage bottling plants, autonomous vehicles, live monitoring systems, etc. These systems are typically built using timeline based approaches, production/consumption rates [9] or priority-based schemes, where the resource demands are mapped to specific time slots or priority levels, often in ad hoc fashion. This mapping of resources to currently available scheduling mechanisms introduces many problems. Assumptions go undocumented, and violations go undetected with the end result that the system can become fragile and fail in unexpected ways. We argue for a resource-centric approach where the scheduling policies are completed subsumed by the kernel, and applications need only specify their resource and timing requirements. The kernel will then make internal scheduling decisions such that these requirements are guaranteed to be satisfied.

Various timing constraints also arise in desktop and networked multimedia systems. Multi-party video conferencing, mute but live news windows, recording of live video/audio feeds, playback of local audio/video streams to remote participants etc. can go on concurrently with normal computing activities such as compilation, editing and browsing. A range of implicit timeliness constraints need to be satisfied in this scenario. For example, audio has stringent jitter requirements, and video has high bandwidth requirements [8]. Disk accesses for compilation should take lower precedence over disk accesses for recording a live telecast.

Two points argue in favor of resource-centric kernels we call "resource kernels":

- Firstly, operating system kernels (including microkernels) are intended to manage resources such that application programs can assume in practice that system resources are made available to them as they need them. In real-time systems, system resources such as the disk, the network, communication buffers, the protocol stack and most obviously the processor are shared. If one application is using a large portion of the system resources, then it implies that other applications get a less portion of the system resources and consequently can take longer to

¹This research was supported by the Defense Advanced Research Projects Agency in part under agreement E30602-97-2-0287 and in part under agreement F30602-96-1-0160. Mr. Molano was funded by a research grant from the Community of Madrid and by the National R&D Program of Spain under contracts TIC96-0982 and TIC97-0438.

execute. In other words, their timing behavior is adversely affected. Letting kernels take explicit control over resource usage is therefore a logical thing to do to prevent such unexpected side effects.

- Secondly, our resource model captures the essential requirements of many resource management policies in a simple, efficient yet general form. The implementation of the model can actually be done using any one of many popular resource management schemes (both classical and recent) without exposing the actual underlying resource management scheme chosen. User-level schemes can be used to dynamically downgrade (upgrade) application quality when new (current) resource demands arrive (leave). This feature of the resource model leads to minimal changes from existing infrastructure while retaining flexibility and offering many benefits.

Other alternatives to resource kernels include user-centric and application-centric kernels:

- A user-centric kernel can emphasize multi-user capabilities, and also track and facilitate the needs of specific users. Unix in general and Unix filesystems in particular can be viewed as providing such support. At the same time, Unix attempted to present and manipulate all system entities as files. In resource kernels, we adopt a similar approach and attempt to present all system resources using a uniform model for guaranteed access. Our implementation of the resource kernel is orthogonal to user-centricity, but tighter integration between the two may be possible. Currently, specific user-level requirements must be translated by intermediate layers into resource demands at the resource kernel interface.
- Application-centric kernels are typically custom executives with built-in support for the applications they are intended to serve. As an example, kernels used in telecommunication switches are application-centric and deal explicitly with the high-performance, upgrading, availability, billing and auditing requirements of telecommunication paths. Conceptually, the notions of resource kernels to guarantee timely access to resources can be applied to such kernels as well. For example, consider a postscript printer. It has an executive running a postscript interpreter and control of the physical printing operations. Precisely timed control and concurrency management of downloading new print tasks in such executives can also benefit from the support available in resource kernels.

1.1. Comparison with Related Work

A wealth of resource management schemes and scheduling algorithms exist from which one can draw. Our resource management work builds on and significantly extends established real-time scheduling theory and derived processor reservation work reported in [25]. The work in [25] did not deal with the management of multiple resource types, concurrent accesses to different resources, explicit timeliness control, feedback about resource usage, behavioral control on resource overruns, management of interactions between resource users, and considerations of portability, com-

patibility and automation. In brief, our resource management scheme supports the abstractions behind real-time priority-based scheduling for periodic activities, and service schemes for aperiodics in that framework.

Some of our goals (such as resource centricity and portability) are similar to those of Microsoft Research's Rialto kernel among others. The reservation model also has its counterparts in network reservation protocols as used in ATM and RSVP. However, the operating system problem seems more complex in one sense since inherently different resource types must be dealt with, while networks essentially deal with one type (namely network packets). In another sense, network reservations must be homogeneous, scalable and efficient, making its realization harder in a different sense.

Despite its origins in real-time scheduling theory, we expect our resource management model to be compatible with resource management schemes with their origins in networks such as proportional fair-sharing schemes such as PGPS, WF²Q, virtual clocks and lottery scheduling. The notion of fairness has for long been deemed to be anti-thetic to real-time systems and the management of timeliness [38]. Weighted allocation schemes such as proportional fair-sharing, however, can still be applied to the real-time model. This can be done by dynamically recomputing the weights so as *not* to be proportional or fair, but instead to obtain a fixed share of a resource when new requests arrive or current requests complete. Our scheme employs a different period for each real-time activity, and guarantees a "share" of that period to the activity. As a result, the dynamic recomputing of weights in a proportional fair-share scheme can be viewed as a special case of our model as having a single (small) fixed period for all resource allocations. The primary difference that we see is that our work advances system capabilities to include non-traditional resources such as disk bandwidth that can be used in conjunction with processor scheduling.

Finally, Blazewicz et al. [5] have shown that the problem of scheduling activities which need multiple resources simultaneously and have timeliness constraints is NP-complete. In our work, we therefore strive for practical and acceptable alternatives which can guarantee access to different resource types.

1.2. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we present the goals to be satisfied in designing a resource kernel, and based on well-established principles of real-time resource management, defines a resource reservation model and its parameters. In Section 3, we describe the implementation of our resource reservation model in the context of processor scheduling, and evaluate it. In Section 4, we detail the implementation of the resource reservation model in the context of disk scheduling, and evaluate those schemes. In Section 5, we address other issues that arise in the context of using the resource kernel in practice including calibrating an application's resource demands automatically and in portable ways. Finally, in Section 6, we con-

clude with some remarks outlining our research contributions and future work.

2. Designing a Resource Kernel

The challenges for a resource kernel are many. We characterize these challenges below as a set of goals that resource-centric kernels should aim to satisfy.

2.1. Design Goals of a Resource Kernel

G1. Timeliness of resource usage. An application using the resource kernel must be able to request specific resource demands from the kernel. If granted, the requested amount of resources must be guaranteed to be available in timely fashion to the application. An application with existing resource grants must also be able to dynamically upgrade or downgrade its resource usage (for adaptation and graceful degradation purposes). This implies that the kernel must support an admission control policy for resource demands. Conventional real-time operating systems do not provide any such admission control mechanisms, even though an equivalent feature (without enforcement capabilities) could be built at user level.

G2. Efficient resource utilization. The resource kernel must utilize system resources efficiently. For example, a trivial and unacceptable way to satisfy G1 would be to deny all requests for guaranteed resource access. In other words, if sufficient system resources are available, the kernel must allocate those resources to a requesting application. This goal implies that the admission control policy used by the resource kernel have provably good properties. Such proof may be analytical or empirical but our version of the resource kernel provides analytically proven properties. It must be noted that this goal is subordinated to G1, in that guaranteed resource access is the primary goal, and efforts to improve efficiency and throughput cannot happen at the expense of the guarantees.² Traditional real-time operating systems leaves the matter completely open to the developers, each of whom must use their own schemes to obtain better utilization for their applications.

G3. Enforcement and protection. The resource kernel must enforce the usage of resources such that abuse of resources (intended or not) by one application does *not* hurt the guaranteed usage of resources granted by the kernel to other applications. Traditional real-time operating systems such as those compliant with the POSIX Real-Time Extensions [30] do *not* satisfy this goal.

G4. Access to multiple resource types. The resource kernel must provide access to multiple resource types such as processor cycles, disk bandwidth, network bandwidth, communication buffers and virtual memory. The communication protocol stack on the system may potentially be viewed as a resource type as well, but in most systems, they use the processor and hence can be managed by appropriate scheduling and allocation of processor cycles. For example, see [23]. Traditional real-time operating systems provide mechanisms that can *only* be used to guarantee access to processor cycles.

G5. Portability and Automation. The absolute resource demands needed for a given amount of work can unfortunately vary from platform to platform due to differences in machine speed. For example, a signal processing algorithm can take 10ms on a 200MHz Pentium but take 20ms on a 100MHz Pentium. Ideally, applications must have the ability to specify their resource demands in a portable way such that the same resource specification can be used on different platforms. In addition, there must exist means for the resource demands of an application to be automatically calibrated.

G6. Upward compatibility with fielded operating systems. A large host of commercial and proprietary real-time operating systems and real-time systems exist. Many of these systems employ a fixed priority scheduling policy [12] to support provide real-time behavior, and the rate-monotonic [18] or deadline-monotonic [17] priority assignment algorithm is frequently used to assign fixed priorities to tasks. Basic priority inheritance [33] is used on synchronization primitives such as mutexes and semaphores to avoid the unbounded priority inversion problem when tasks share logical resources. For example, Solaris [11], OS/2, Windows, Windows NT, AIX, HP/UX all support the fixed priority scheduling policy. The Java virtual machine specification also does. Priority inheritance on semaphores is supported in all these OSs (except Windows NT). POSIX real-time extensions, Unix-derived real-time operating systems such as QNX and LynxOS, and other proprietary real-time operating systems such as pSOS, VxWorks, VRTX, OS/9000, and iRMX support priority inheritance and fixed priority scheduling. To be accepted, the resource kernel must be upward compatible with these schemes. The priority inheritance scheme is also used or being considered for use in multimedia-oriented systems [28, 40].

Goals G1-G4 are integral to resource kernels, while goals G5 and G6 are for practicality and convenience. Goals G1, G2, G5 and G6 can be satisfied by appropriate extensions to traditional real-time operating systems which support fixed priority CPU scheduling. For example, a user-level server can perform admission control using a resource specification model similar to ours, and assign fixed priorities based on the resource parameters used by our model. However, in order to satisfy goals G3 and G4, the internals of these operating systems need to be modified in ways similar to our resource kernel design and implementation.

²This emphasis on guarantees and timeliness may understandably seem to bias the resource kernel away from multimedia systems. (In real-time systems, missed deadlines may potentially lead to system failure, and possible loss of life and/or property). However, we believe that as multimedia applications on desktops and internet appliances mature, users will come to expect smooth video frame changes, jitterless audio, and synchronized audio and video. It is to be noted that VCR/TV/satellite technologies have been delivering such guaranteed timing behavior for years. It seems rather illogical to expect anything less from computers at least when a user is willing to pay for it, particularly if virtual reality environments must seem real, or for applications such as tele-medicine and tele-surgery.

2.2. An Historical Perspective of our Real-Time Resource Management Model

Many deployed real-time systems have been built and analyzed using the fixed priority periodic task model first proposed by Liu and Layland [18]. This model employs two parameters, a maximum computation time C needed every periodic interval T for each activity that needs to be guaranteed. The rate-monotonic scheduling algorithm [18] assigns fixed priorities³ based only on T and is an optimal fixed priority scheduling algorithm. Instead of using priorities, if the $\{C, T\}$ model is directly used in a real-time system, the assumptions underlying the Liu and Layland model can be monitored and enforced at run-time. Following this strategy, the "aperiodic server" model [13, 37] uses artificially introduced C and T values for new "server tasks" which can then service aperiodic tasks within a periodic setting. This bounded periodic usage was adopted by the processor reservation work carried out in [25].

We build on this proven trend by identifying, designing, implementing and evaluating significant kernel extensions to the Liu and Layland work along multiple dimensions:

- using arbitrary deadlines [16, 17] to obtain fine-grained control timeliness of concurrent activities,
- applying the priority inheritance solutions explicitly to the unbounded priority inversion problem when activities share resources [2, 31, 34],
- dealing with new resource types such as disk scheduling, a problem which has not been studied in depth in the Liu and Layland model, and
- combining the scheduling of multiple resources into a single common framework observing that the problem of scheduling multiple resources with deadlines is known to be an NP-complete problem [5].

2.3. The Resource Reservation Model

The resource kernel gets its name from its resource-centricity and its ability of the kernel to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage,

The resource kernel attains these capabilities by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritizes them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

2.4. Explicit Resource Parameters

Our resource reservation model employs the following parameters: computation time C every T time-units for managing the net utilization of a resource, a deadline D for meeting timeliness requirements, a starting time S of the resource allocation, and L , the life-time of the resource allocation. We refer to these parameters, $\{C, T, D, S$ and $L\}$ as explicit parameters of our reservation model. The semantics are simple and are as follows. Each reservation will be allocated C units of usage time every T units of absolute time. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval. The guarantees start at time S and terminate at time $S + L$.

2.5. Implicit Resource Parameter

If various reservations were strictly independent and have no interactions, then the explicit resource parameters would suffice. However, shared resources like buffers, critical sections, windowing systems, filesystems, protocol stacks, etc. are unavoidable in practical systems. When reservations interact, the possibility of "priority inversion" arises. A complete family of priority inheritance protocols [31] is known to address this problem. All these protocols share a common parameter B referred to as the blocking factor. It represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its B is unbounded, a reservation cannot meet its deadline. The resource kernel, therefore, implicitly derives, tracks and enforces the implicit B parameter for each reservation in the system. Priority (or reservation) inheritance is applied when a reservation blocks, waiting for a lower priority reservation to release (say) a lock. As we shall see in Section 4.5, this implicit parameter B can also be used to deliberately introduce priority inversion in a controlled fashion to achieve other optimizations.

2.6. Reservation Type

When a reservation uses up its allocation of C within an interval of T , it is said to be *depleted*. A reservation which is not depleted is said to be an *undepleted* reservation. At the end of the current interval T , the reservation will obtain a new quota of C and is said to be *replenished*. In our reservation model, the behavior of a reservation between depletion and replenishment can take one of three forms:

- *Hard reservations*: a hard reservation, on depletion, cannot be scheduled until it is replenished. While appearing constrained and very wasteful, we believe that this type of reservation can act as a powerful building block model for implementing "virtual" resources, automated calibration, etc.
- *Firm reservations*: a firm reservation, on depletion, can be scheduled for execution only if no other undepleted reservation or unreserved threads are ready to run.
- *Soft reservations*: a soft reservation, on depletion, can be scheduled for execution along with other unreserved threads and depleted reservations.

³A lower T yields a higher priority.

2.7. System Call Interface to Reservations

System Call	Description
Create	Create a reservation port
Request	Request resource on reservation port
Modify	Modify current reservation parameters
Notify	Set up notification ports for resource expiry messages.
Set Attribute	Set attributes of reservation (hard, firm or soft reservation)
Bind	Bind a thread to a reservation.
GetUsage	Get the usage on a reservation (accumulated or current).

Table 2-1: A subset of the reservation system call interface for each resource type.

Our resource reservations are *first-class* entities in the resource kernel. Hence, operations on the reservations must be invoked using system calls. A select subset of the system call interface for the resource reservation model is given in Table 2-1. A reservation modification call allows an existing reservation to be upgraded or downgraded. If the modification fails, the previous reservation parameters will be restored. In other words, if an application cannot obtain higher resources because of system load, it will at least retain its previous allocation. A notification registration interface allows the application to register a port to which a message will be sent by the resource kernel each time the reservation is depleted. A binding interface allows a thread to be bound to a reservation. More than one thread can be bound to a reservation. Query interfaces allow an application to obtain the current list of reservations in the system, the recent usage history of those reservations (updated at their respective T boundaries), and the usage of a reservation so far in its current T interval.

3. Processor Resource Management

In this section, we shall discuss and evaluate our implementation of the resource reservation model for the processor resource.

3.1. Admission Control

Description	Overhead (μ s)
Processor admission control with 1 reservation	25
Processor admission control with 10 reservations	120
Processor admission control with 20 reservations	195
Processor reservation creation (excluding admission control)	150

Figure 3-1: Processor Admission Control Policy Overhead w/ Exact Schedulability Conditions

Our processor reservation scheme employs a fixed priority scheme due to its widespread popularity (as mentioned in the description of goal G6 in Section 2.1). In other words, each reservation is assigned a fixed priority, which is equal to its period T or deadline D , depending upon whether a

rate-monotonic or a deadline-monotonic scheme is used respectively. Our admission control policy does *not* employ (oft-used) static utilization bounds (e.g as given in [18]) since they can be very pessimistic in nature [14]. Instead, we use an exact schedulability condition which provides the best admission control for a given set of real-time threads. This algorithm is described in detail in the Appendix (Section 2). The algorithm, being a complex non-linear function of the thread periods, their relationships and their computation times, does not have a standard degree of complexity. However, it is easily coded and can be computed efficiently. The computational cost of the scheme for a wide range of thread counts is shown in Figure 3-1. As can be seen, the overhead is acceptable. It is also incurred only when a thread requests a new reservation (or upgrades an existing reservation).

3.2. Tracking Implicit Parameter B

When a lower priority reservation blocks a higher priority reservation⁴, the former inherits the reservation (and therefore its priority). When the higher priority reservation finally unblocks, the inheritance is revoked. However, the duration for which the inheritance was in place is priority inversion. The resource kernel tracks and accumulates the duration of priority inversion during a reservation's T . If it exceeds the maximum B that can be tolerated by that reservation, a message is sent to the reservation's notification port.

3.3. Performance Evaluation

Reservation Type	Initial Reservation				Upgraded to				Downgraded to			
	C_i (ms)	T_i (ms)	D_i (ms)	C_i/T_i	C_i (ms)	T_i (ms)	D_i (ms)	C_i/T_i	C_i (ms)	T_i (ms)	D_i (ms)	C_i/T_i
Hard	8	80	60	0.1	12	80	60	0.15	4	80	60	0.05
Firm	15	80	70	0.19	19	80	70	0.24	11	80	70	0.14
Soft	20	80	80	0.25	24	80	80	0.3	20	80	80	0.25

Table 3-1: The processor reservation parameters used for Figures 3-2, 3-3 and 3-4

We now evaluate the processor reservation scheme by running different workloads with and without the use of reservations. All our experiments use a PC using a 120MHz Pentium processor with a 256KB cache and 16MB of RAM. We illustrate two basic points in these experiments:

1. the nature of the three types of reservations, and
2. the flexibility to upgrade and downgrade different reservations. dynamically.

In the experiments of Figures 3-2 and 3-3, three threads running *simultaneously* in infinite loops are bound to the three reservations listed in Table 3-1. In the experiment of Figure 3-2, only these three threads are running. In contrast, in the experiment of Figure 3-3, many other unreserved threads in infinite loops are also running in the background and competing for the processor. The behavior

⁴This terminology means that a thread bound to a lower priority reservation is blocking a thread bound to a higher priority reservation.

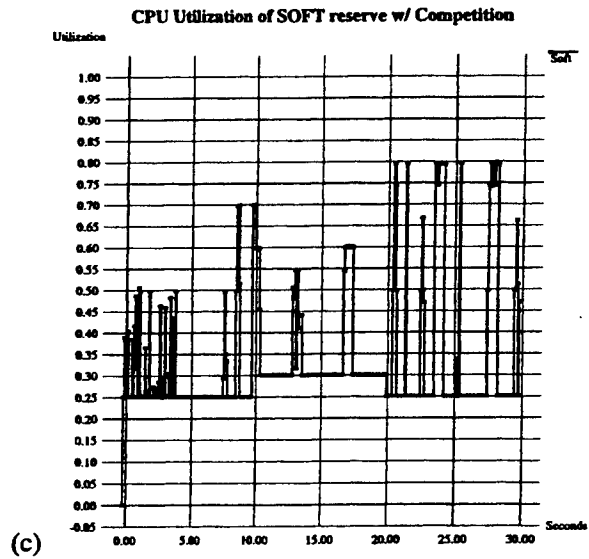
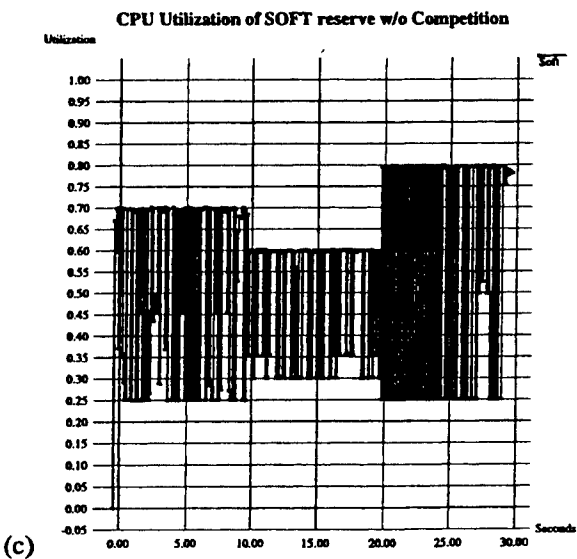
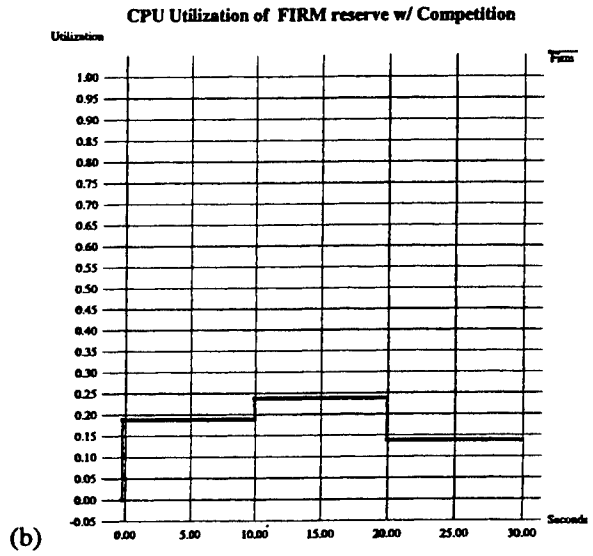
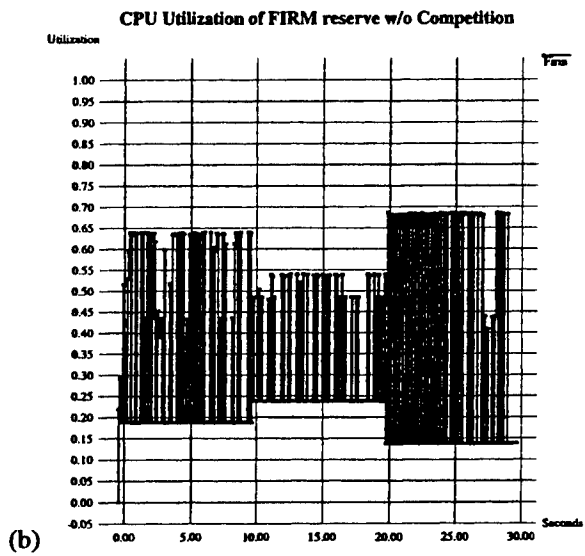
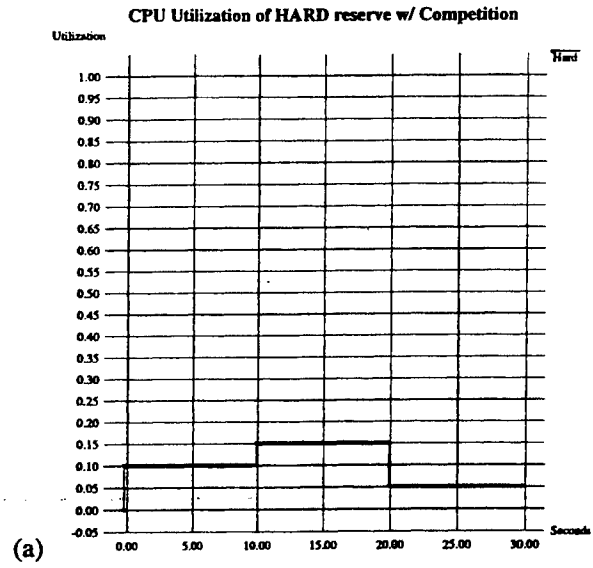
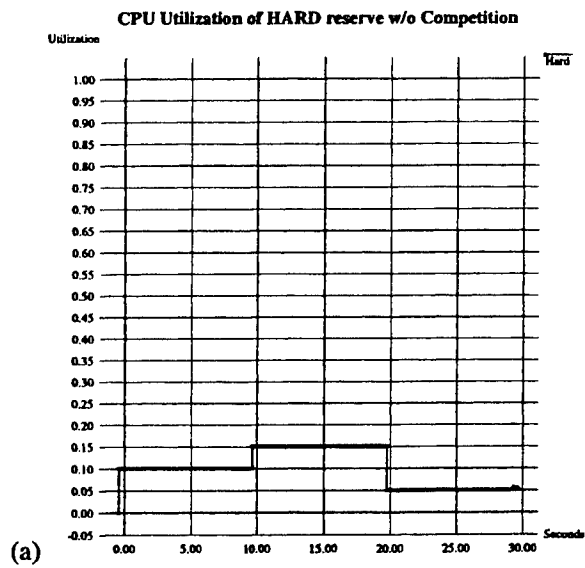


Figure 3-2: Behavior of *reserved* infinite loop threads *without* unreserved competition

Figure 3-3: Behavior of *reserved* infinite loop threads *with* unreserved competition

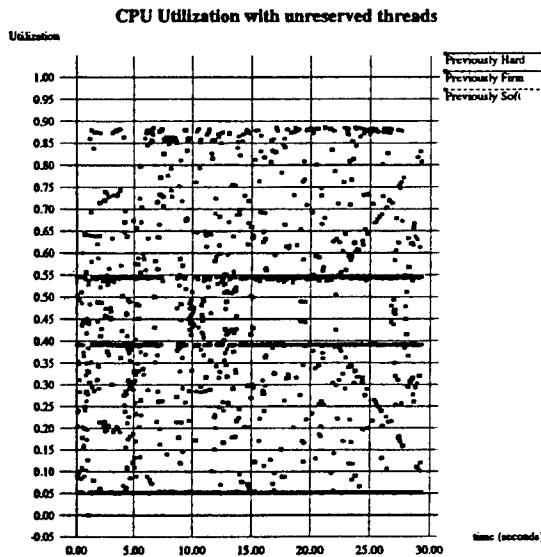


Figure 3-4: Behavior of *unreserved* infinite loop threads with unreserved competition

of the three types of reservations is illustrated between these two figures.

- The first reserved thread is bound to a hard reservation and should not consume more than its granted utilization which is initially 10%, explicitly raised to 15% at time 10, and then explicitly dropped to 5% at time 20. As can be seen in (a), this thread, despite running in an infinite loop and the presence of many competing threads, obtains exactly this specified usage in *both* Figures 3-2 and 3-3.
- The second reserved thread is bound to a firm reservation, and is allocated 19% of the CPU initially, upgraded to 24% at time 10, and then downgraded to 14% at time 20. In Figure 3-2-(b), when there is no unreserved competition, this thread obtains a minimum of its granted utilization. In addition, it obtains "spare" idle cycles from the processor since there are no unreserved competing threads. However, in Figure 3-3-(b), when there is *always* unreserved competition, this thread obtains only its granted utilization. Thus, as intended, a firm reservation behaves like a hard reservation when the processor is *not* idle, and like a soft reservation when idle processor cycles are indeed available.
- The third reserved thread is bound to a soft reservation, which is allocated 25% initially, upgraded to 30% at time 10, and then downgraded to 25% at time 20. A soft reservation can compete for cycles left behind by any threads with currently undepleted reserves. As a result, this thread obtains more cycles than its granted utilization in both Figures 3-2-(c) and 3-3-(c). It must be noted that the thread obtains a minimum of its granted utilization during all its instances. It can also be seen that this thread obtains more processor cycles in Figure 3-2 since it competes only with one thread bound to a firm reservation.

It must be recalled that the three threads of Figures 3-2 and

3-3 are running simultaneously. The completion times of this same set of threads (with the background competition of Figure 3-3) when run without using any reservations are plotted in Figure 3-4. The same threads which behave extremely predictably in Figure 3-3 now exhibit an enormous amount of seemingly random and practically unacceptable unpredictability. This demonstrates that our resource management scheme works as expected; without the scheme, resource usage is not predictable.

C_i (ms)	T_i (ms)	D_i (ms)	$U_i=(C_i/T_i)$
5	20	10	25%
10	40	30	25%
10	60	45	16.66%

Table 3-2: The processor reservation parameters used for the experiment of Figures 3-5 and 3-6

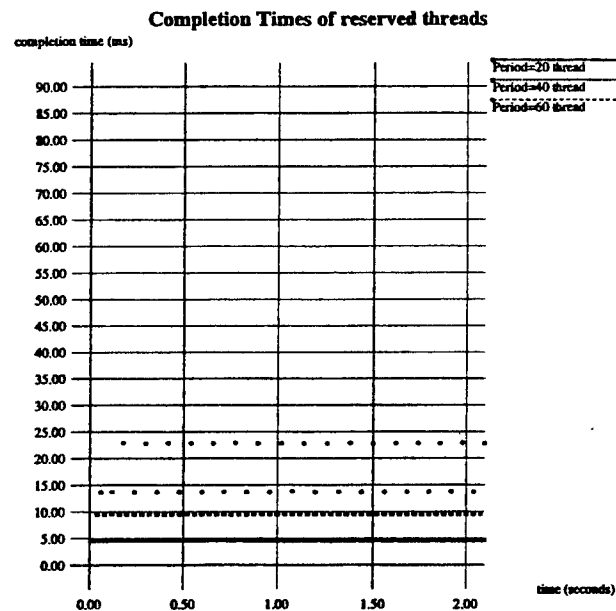


Figure 3-5: Completion times of reserved threads in the presence of competing threads

We now run another experiment where each thread imposes only finite demands, but the completion times of these demands can be predictable only with explicit resource management. The reservation parameters used for this experiment are listed in Figure 3-2. Note that the deadlines are substantially smaller than the reservation periods giving finer grained control over timeliness. One thread for each of the three reservations is created with the same period and (slightly less) computation time as its corresponding reservation. When using reservations in our resource kernel, the completion times for each of these threads as they execute with their different periods was time-stamped. The corresponding results are plotted in Figure 3-5. As can be seen, all the three threads complete their executions well ahead of their deadlines. Thread 2 also has a constant completion time despite its lower priority because of its

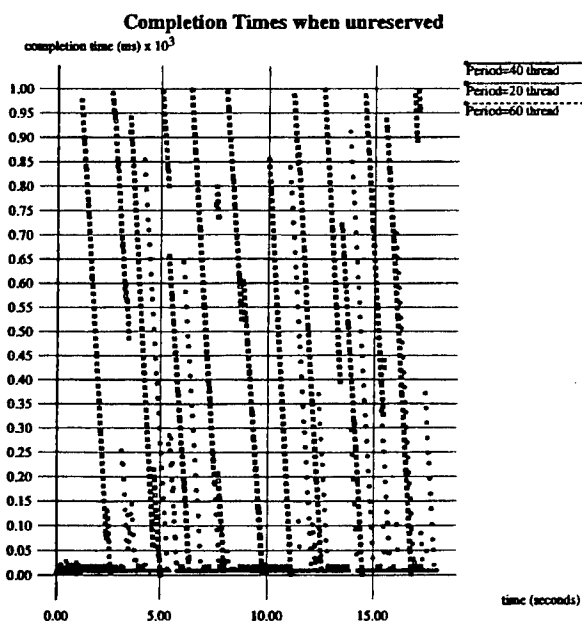


Figure 3-6: Completion times of unreserved threads with competing threads

harmonicity with thread 1. The behavior of the completion times when no reservations are used is depicted in Figure 3-6. As can be seen, the same threads have widely varying completion times and also miss their deadlines rather frequently.

To summarize, the resource kernel provides a guaranteed slice of processor resources to applications independent of the behavior of other applications (including execution in infinite loops). Processor cycles that are idle can also be *selectively* allocated to running tasks.

4. Disk Bandwidth Resource Management

Traditional real-time systems have largely avoided the use of disks. This is in part because they may be relatively slow for some real-time applications. However, many real-time applications can benefit from the use of disks to store and access real-time data (such as real-time database applications). Unfortunately, the use of a disk can (a) introduce unpredictable latencies, and even worse (b) the disk access requests must now be managed in conjunction with the processor scheduling. On the processor side, fixed priority algorithms allow a mix of tasks with different periodicity, and hence the disk subsystem must do too. This problem has not been studied extensively partly because the multiple resource problem with deadlines is known to be NP-complete [5]. Some exceptions can be found in [1, 20, 21] but their resource specification models and metrics are very different from the ones we study. The closest scheduling model to ours is found in [6] but its approach is one of using fixed priority scheduling, minimizing blocking through the use of "chunking" and using a static task set. Also, only simulation studies were carried out. In contrast, we use dynamic priority scheduling, exploit blocking instead of minimizing it and evaluate an implementation within our resource framework. In addition, we deal with processor needs that must be dealt with concurrently.

Desktop multimedia systems also need to read from (or write to) disk storage relatively large volumes of video and audio data. In addition, these streams represent continuous media streams, and must therefore be processed by the disk subsystem in real-time. In other words, it would be practically very useful in practice if disk bandwidth can also be guaranteed in addition to managing processor cycles.

In this section, we present a simplistic disk scheduling algorithm based on earliest deadline scheduling. We then improve the algorithm by exploiting "slack" in the reservations to obtain a hybrid of earliest deadline scheduling and a traditional scan algorithm. Our evaluations of these schemes that guaranteed disk bandwidth reservation can be obtained at only a small loss of system throughput.

4.1. Filesystem Bandwidth Specification

Our resource specification model for disk bandwidth is identical to that of processor cycles. In other words, a disk bandwidth reservation must specify a start time S , a processing time C to be obtained in every interval T before a deadline of D . The processing time C can be specified as # of disk blocks (as a portable specification) or in absolute disk bandwidth time in native-platform specification.

4.2. Admission Control

Our simplest disk head scheduling scheme employs the earliest deadline scheduling algorithm [18]. The earliest deadline scheduling algorithm is an optimal scheduling algorithm for our scheduling model and can guarantee 100% resource utilization under ideal conditions. In other words, a higher priority reservation must be able to preempt a lower priority reservation preemptively, and $D_i = T_i$. However, instantaneous preemptions are not possible in disk scheduling. An ongoing disk block access must complete before the next highest priority disk block access request can be issued. This introduces a blocking (priority inversion) factor of a single filesystem block access (as per [35]). Also, when $D_i < T_i$, the required earlier completion time (of $T_i - D_i$, must be added to the blocking factor. A detailed discussion of this admission control policy is beyond the scope of this paper and can be found in [27].

4.3. Scheduling Policy

Instances of a disk bandwidth reservation become eligible to execute every T_i units (at times $S_i, S_i + T_i, S_i + 2T_i, S_i + 2T_i, \dots$). Consider an instance which arrives at time $S_i + nT_i$. This instance has a deadline of $S_i + nT_i + D_i$. Similarly, all instances of all outstanding disk bandwidth reservations have corresponding deadlines. After each disk block access is completed, the disk scheduler makes another scheduling decision. It picks the next ready reservation instance with the earliest deadline and issues a disk access command corresponding to that instance's next disk access request. If there are no pending requests, the disk remains idle.

4.4. The Architecture of the Reserved Filesystem

The architecture of the reserved filesystem follows a traditional scheme. A *Real-Time File Server* running on top of our resource kernel (based on the RT-Mach microkernel) manages the reserved real-time filesystem. RT-FS has multiple worker threads which receive and process filesystem access requests from real-time clients. Each worker thread stores the incoming request it is processing into a common io-request queue. The worker thread responsible for issuing the current disk block access waits for its completion. It then awakens, and determines the next request based on the scheduling policy above. If the next request corresponds to another worker thread, that thread is signaled. Else, the worker thread continues to service its remaining disk access requests, if any.

4.5. Exploiting 'B': Just-In-Time Disk Scheduling

The earliest deadline disk scheduling described above blindly picks the next block with the earliest deadline irrespective of the current position of the disk head. Since the physical movement of the disk head and the disk's rotational latencies constitute significant durations of time, such dynamic scheduling can result in significant disk subsystem throughput reductions particularly under heavy disk traffic. The reductions can be directly attributed to the time wasted by the disk head moving from one end to another and the disk's rotational time. In summary, the deadlines are preferred over a block's physical location.

Traditional scan algorithms, in contrast, re-order the disk request queue such that the block closest to the current head position (in the direction of movement) is accessed next. As a result, a disk request which just arrived can be serviced before another disk request which has been waiting for a long time just because the latter is farther away from the head position. To summarize, the physical block location is favored over timeliness.

The earliest deadline scheduling algorithm and the scan algorithm are therefore at odds with one another. Fortunately, a hybrid scheme which can obtain all the benefits of the earliest deadline scheduling algorithm and at least part of the benefits of the scan algorithm is possible. In priority-driven scheduling, higher priority activities preempt lower priority activities. Since both lower and higher priority activities must be schedulable in an admission-controlled system, higher priority activities typically complete well ahead of their deadlines. In other words, such higher priority activities have a good amount of "slack" in their completion times. Based on this observation, we present a new algorithm we call "Just-in-time" disk scheduling. This algorithm exploits the slack available to higher priority tasks to schedule accesses of other disk blocks which are closer to the current head position.⁵

A brief description of the just-in-time disk scheduling algo-

rithm is as follows. The maximum "slack" available to each disk reservation is computed whenever a new request is admitted (or an existing reservation is deleted). At run-time, if the current slack of higher priority reservations is non-zero, another unreserved (or lower priority reserved) request can be scheduled if closer to the disk head. If slack is stolen, the slack of higher priority reservations is reduced by one. This process is then repeated. If the slack of a high priority reservation goes to zero, it will be serviced independent of its location. The detailed description of the just-in-time algorithm can be found in [27].

4.6. Performance Evaluation

The capability of the disk bandwidth reservation scheme in our resource kernel to satisfy demands on disk bandwidth is illustrated in Figure 4-1. One disk bandwidth reservation of 12 disk blocks every 250 ms was requested in the presence of other unreserved accesses to the disk. As can be seen from the plot, this demand is satisfied by both the earliest deadline scheme and the just-in-time scheme; in fact, both lines are flat and coincide almost completely in the plot. However, the scan algorithm attempts to optimize disk throughput and pays for it by *not* delivering the needed throughput of 12 blocks every 250 ms. As a matter of fact, the bandwidth consumption varies widely.⁶

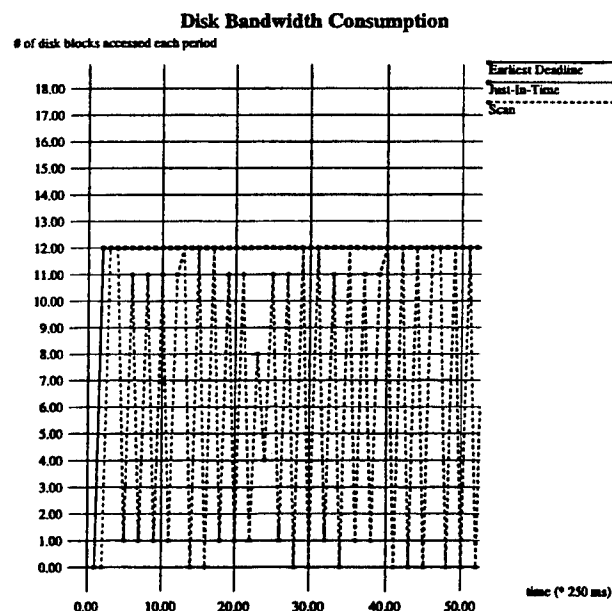


Figure 4-1: Disk Bandwidth consumed (# of disk blocks read) by reserved thread. Earliest deadline and Just-In-Time reservation schemes are flat and coincide almost completely.

We also imposed heavy disk traffic conditions and measured the throughput obtained under the scan and earliest deadline algorithms. This is shown in Table 4-1. As

⁵Such "slack-stealing" has been done in the context of processor scheduling theory in order to provide better response to aperiodic activities [15]. The optimization, cost functions and implementation tradeoffs seem to be different for the processor and the disk, however.

⁶The pattern is more dramatic in a zoomed out view with the x-axis ranging upto 400 periods, but the lines/points are not clearly legible in a relatively small black-and-white graph.

Requested throughput (KB/s)	Throughput with Scan (KB/s)	Throughput for Earliest Deadline Scheduling (KB/s)	Throughput Degradation (%)
1158.6	856.36	764.88	10.68%

Table 4-1: Scan and EDF real-time filesystem throughput comparison

can be seen, the earliest deadline algorithm obtains only about 10% less throughput than the scan algorithm. This is the price to be paid for the predictable and guaranteed disk bandwidth obtained by the earliest deadline algorithm (as shown in Figure 4-1).

4.6.1. Synthetic Workload Behavior with both CPU and Disk Requirements

We next imposed a synthetic workload to determine the completion times of disk access requests, and to study the drop in disk throughput when the Scan policy is replaced with a policy which attempts to satisfy timing constraints in preference to enhancing disk throughput.

As illustrated in Figure 4-2, the real-time workload tested consists of two threads, *Thread 1* and *Thread 1b*. *Thread 1* reads periodically from disk and copies all the data into buffer A, while *Thread 1b* processes data previously stored in buffer B. At the end of the period, there is a buffer switch and the role of both buffers is interchanged. Buffers A and B have the same size. We bound disk bandwidth and CPU reserves to *Thread 1*, and a CPU reserve to *Thread 1b*, and traced the execution in terms of completion times, deadline misses and disk utilization. *Thread 1* makes use of relatively little cpu time and it sleeps till the beginning of the next period to invoke a new read operation. *Thread 1b* processes the data previously stored in the buffer. Both *Thread 1* and *Thread 1b* have a period of 250 ms. Also, *Thread 1* reads 44 KBytes during each of its instances, and has a deadline of 162 ms for completing its reads. Note that this deadline is shorter than its period of 250 ms, forcing a stringent test for the filesystem. *Thread 1b* is offset from *Thread 1* by 162 ms and has a deadline of 88 ms.

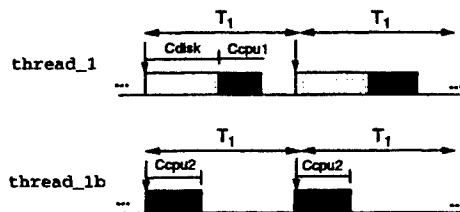


Figure 4-2: Execution patterns of *Thread 1* and *Thread 1b*

Six periodic threads each with a different period (varying from 300 ms to 640 ms) and a different read-access load on the disk (varying from 8 KBytes to 200 KBytes per periodic instance of the thread) were introduced as competing threads without any reserved capacity on either the CPU or

the disk bandwidth. We ran this workload for a duration of 100 seconds and measured the completion times of each periodic instance, and the total disk bandwidth consumed. The completion times are illustrated in Figure 4-3.

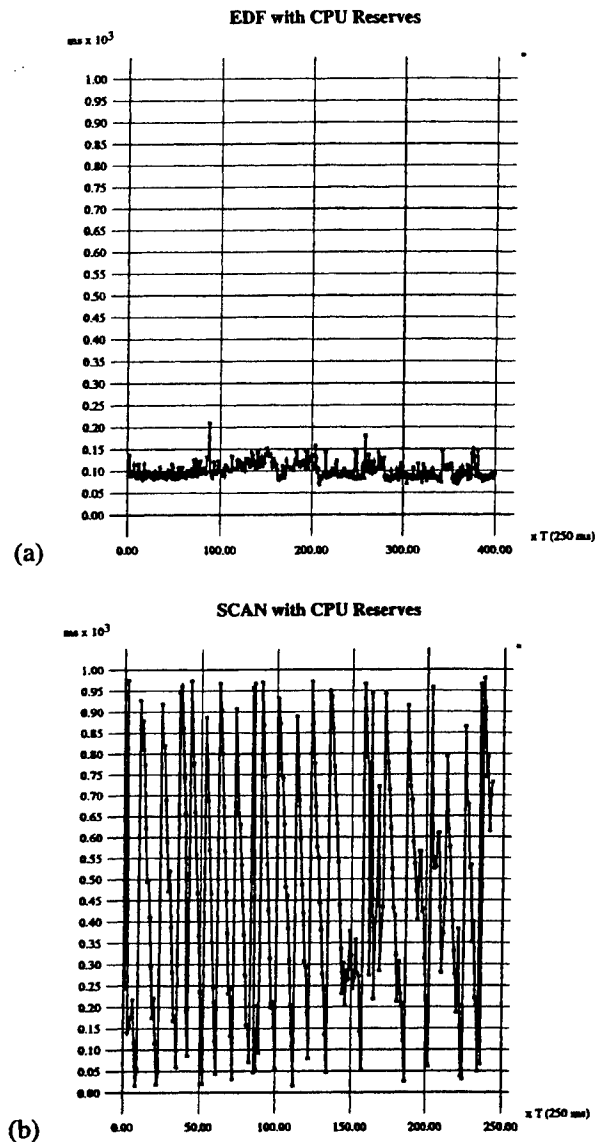


Figure 4-3: *Thread 1* Completion Times

If we use EDF/EDF+JIT without reserving the CPU there are some deadline misses (2 out of 400: periods 88 and 258). In these cases the task finished after 162 ms (but never after the period of 250 ms). These two deadline misses are due to the fact that our filesystem (extension of the Berkeley Fast FileSystem) does not allocate blocks contiguously on disk. So relatively high interblock seek times out of the cylinder group may happen from time to time even with requests for successive blocks within a file from the same thread. This can happen for each 1 MB of filesystem data according to the Berkeley FFS allocation algorithm and can lead to potential deadline misses. Accounting for the worst-case interblock seek times in the admission control test would avoid this problem, but can lead to

extremely low guaranteed disk throughput. Thus, notwithstanding our admission control test, some deadlines can be missed. However, as can be seen from our experiments, the deadline misses are rather infrequent. Conversely, in the Scan case there is no time to run the needed 400 disk accesses and only 248 accesses are completed within the experiment duration. The completion times are nearly always greater than the period itself (> 250 ms) and sometimes much greater. This shows that EDF w/ CPU reserves consistently meets the timeliness constraints of the real-time application accessing the disk.

Disk Throughput: The total disk bandwidth consumed in the above experiment was 16,464 KBytes with the EDF and CPU reserve policy, and 17,750 KBytes with the Scan policy. This represents only a performance throughput loss of 7.25%. In return, however, the timing constraints and periodic bandwidth requirements are satisfied with the EDF/CPU reserves policy, while they are dramatically unsatisfied with the Scan policy.

5. Practical Issues

5.1. Using Different Reservations Together

Consider a video display application which reads a video movie from the local disk and displays it on the screen. The movie is long enough that it does not fit into memory. As a result, subsequent video frames must be read from the disk while frames already read into a double buffer are being played on the screen. In this case, the video display algorithm must not only be scheduled on the CPU (where it can also do decompression or special signal processing) but also obtain guaranteed disk bandwidth to display the movie and its audio track without user-perceptible jitter.

The most straightforward way of approaching this problem is as follows: the application consists of a single thread which binds itself to a processor reservation and a disk bandwidth reservation with the same period, start time and appropriate computation times to satisfy the application's needs. There can be other threads in the system which use other combinations of resources (such as the processor and network bandwidth). Each of these reservations need to satisfy their associated deadlines given by the parameter D . However, it is known that the problem of scheduling concurrent tasks on multiple resources with timeliness constraints is NP-complete [5]. As a result, one faces the dilemma of finding a practical acceptable solution, since finding an optimal solution to the problem is very impractical. We address this problem next.

5.2. Resource Decoupling

Since simultaneous access to multiple resources is the problem we face, a natural solution to the practical dilemma one faces is to try to decouple the use of different resources, which can be used independent of one another. An end-to-end timing constraint problem is normally intractable as a single big problem, and is hence solved as a series of small problems where each problem only spans a single resource. For example, in an audio-conferencing application [23], the first pipeline stage occurs in the sound card which transfers

data to the processor using *periodically self-initiated* DMA or multi-master bus transactions. A 2nd pipeline stage occurs on the processor to transmit the data and the next stage occurs in the network. The end-to-end delay for audio is the sum of the delays encountered in each of the audio pipeline stages. We refer to this strategy where each of the resources involved are scheduled independent of one another as resource decoupling. When resources are decoupled, for example, the completion time test of Section 2 can be applied to each resource independently.

In the audio-conferencing application, the only coupling between the pipeline stages lies actually at the interface between stage 2 and the network (or the network and stage 4).⁷ When the processor is ready to send out packets, the network must be able to transmit them. Memory buffers on the network interface card provide some decoupling by storing packets that the processor is ready to transmit, but the network is not ready to accept yet. We address this problem next.

5.3. Processor Co-Dependency

A phenomenon that we name *processor co-dependency* provides a hint to the solution. Complete resource decoupling seems possible between any two resources if neither of them is the processor. Since the processor is the brain of the system, communications between the network and the disk, for example, must go through the processor. The processor must obtain the network packets and then send them to the disk. In other words, a coupling problem which at first sight is between the disk and the network gets translated into two independent couplings between the disk and the processor, and between the processor and the network. The net result is that as soon as the disk (or the network) demand attention from the processor, the processor must be able to provide it.

5.4. "Immediate" or "System" Reservations

In our resource reservation model, we define the concept of a "system reservation" which is a highest priority reservation which does not get depleted. As a result, any thread or threads bound to a system reservation will be able to execute at the highest priority as soon as possible (subject only to other threads using a system reservation). We also sometimes refer to the system reservation as an "immediate reservation" because of the immediacy of its service. Clearly, the use of "system reserves" must be confined to trusted services only (to satisfy goal G3 of resource kernels), which must be trusted to use them only sparingly for relatively quick transfers of data. The worker threads in the reserved filesystem of Section 4.4 also fall into the category of system reservation users. It must be remembered that the usage of the system reservation will adversely affect new resource requests and must be accounted for in admission tests.

⁷If the network interface card hardware can be configured to be in auto-initiation mode as on the sound card, this coupling problem would disappear as well. This argues for better and more sophisticated support in interface cards and controllers. The trend towards MMX support and "software modems" is unfortunately in the opposite direction!

We measured the time consumed by components of a disk I/O to complete a filesystem block fetch of 4KB: time spent in core filesystem code = 532 μ s, time spent in filesystem overhead (block map queries, etc.) = 171 μ s, time spent in data copies = 131 μ s, time spent in disk reserve overheads (scheduling, updating slack, etc.) = 230 μ s, time spent in I/O = 2550 μ s leading to a total elapsed time of 3082 μ s. The CPU usage for the worker thread in the filesystem is therefore 532 μ s out of 3082 μ s = 17.26%. Since one worker thread can access the disk at any given time, this represents the worst-case processor requirement imposed by the real-time filesystem. However, due to the fact that disk seeks will not be issued continuously in a general system, this number will be lower in practice. Otherwise, for a disk-intensive context, this overhead is likely acceptable.

5.5. Calibrating an Application's Requirements

The computation time C needed for a reservation must be known in order to reserve processor time before it can be requested. It is, however, unknown practically before its actual execution since it heavily depends on a machine platform on which an application program runs. Even on machines with the same CPU and the same clock rate, the execution time may be affected by the presence of cache, the amount of memory, memory and system bus interface chip sets, and other I/O interface cards. Thus, we need to obtain C for the current platform by actually running an application on it. Obtaining C requires the kernel to support precise measurement of the processor time consumed by a certain thread. We now discuss how this can be obtained using only our resource kernel capabilities.

Our resource kernel supports hard reservations and also provides current and accumulated usage on a reservation by a program. The hard reservation ensures that any threads bound to it can only run up to its specified C . The execution time of the application program to be calibrated is then measured as follows. A new hard reservation, named (say) "calibration", is created, and the given application program is bound to it just for the purpose of measuring its execution time. The reservation will get depleted by the running of the application program, get replenished by the resource kernel, and the process will repeat until the application program completes execution. The accumulated usage on the hard reservation "calibration" now yields the execution time of the application program. An advantage of this method is that it is certain that a program can obtain its C even when the system is busy since it is guaranteed to receive a certain amount of processor time for its execution.

5.6. Portability Of Resource Specifications

As mentioned above, the absolute execution time of a program changes from platform to platform depending upon processor speed, etc. As a result, the specification of C in absolute time-units can become inherently not portable. Fortunately, portable time-units are available in the form of the number of clock ticks and the number of instructions executed for a given program segment on the processor. Of these two, the number of clock ticks is perhaps more portable since today's microprocessors contain

on-chip clock counters which can not only provide high-accuracy resolution as well be inherently scalable across chips with lower or higher clock speeds. Similarly, C_i for disk bandwidth reservation can specify the number of disk blocks to be read, or better, the number of bytes to be read. The latter units will also be portable across platforms using different disk block sizes. Implementations of resource kernels must therefore provide convenience functions to translate "portable time-units" on a resource to native absolute time-units.

5.7. Adaptive QoS Management

User-level resource managers can be built on top of a resource kernel to react (or adapt to) to changes in application, system resources and the environment. In distributed real-time applications, such as video conferencing, the change in quality at one end-point typically implies that the other end-point must also adapt its quality correspondingly. Such distributed adaptations must clearly happen at a larger time-scale than single-node resource allocation changes. Similarly, we take the position that user-level application changes happen at a larger time-scale than the decisions made in the resource kernel to dynamically schedule activities on system resources. Such user-level resource managers can also potentially implement more complex resource management policies than the ones used by our resource model.

6. Concluding Remarks

We have presented a resource-centric approach to building real-time kernels, and we call the resulting kernel a resource kernel. The resource kernel provides timely, guaranteed and protected access to resources. We now compare our approach with two related approaches, and summarize our research contributions.

6.1. Resource Kernels and Related Approaches

We now compare the resource kernel notions with the approaches used by operating systems such as Nemesis [29] and Exokernel [7]. Nemesis and our resource kernel approach adopt a similar model of resource specification and allocation, based on the so-called $\{C, T\}$ model originally proposed by Liu and Layland [18]. Nemesis implicitly assumes a deadline of T before which the C units of time must be available. Our resource kernel also supports a deadline shorter than T ⁸. The Nemesis approach to dealing with the problem of priority inversion, a potentially significant stumbling block of multi-tasking real-time systems, is rather unclear. In our resource kernel approach, bounding priority inversion is a key principle of managing interactions between concurrent real-time activities. Priority inversion, where a higher priority request is blocked by a lower priority activity, is unavoidable in the general case (such as critical sections, non-preemptible bus transactions and finite size ATM cells). However, it is imperative that unbounded priority inversion be eliminated, as in the use of semaphores in a priority-driven system [31, 35]. Such

⁸A deadline longer than T is also possible.

durations of priority inversion must be bounded and if possible minimized. Priority inheritance protocols have also been extended to dynamic priority algorithms [3, 9]. In resource kernels, we use priority inheritance in the form of reserve propagation [26] where a blocking thread inherits the scheduling priority of a higher priority reserved thread for the duration of the blocking.

Nemesis advocates the minimization of servers to enable correct "charging" of resource usage to applications. The Nemesis approach is to put 'server code' into client libraries, which would then use critical sections to enforce consistency requirements across multiple clients as necessary. Our resource kernel notions take a neutral stance on the topic of servers in that we (must) support configurations with and without servers. We do so for two fundamental reasons:

1. *Time and space are distinct:* Servers and critical sections executing in client space providing the given service are strictly analogous in a timing predictability sense, and differ only in a spatial organization sense. More precisely, the blocking (or priority version) factor is (almost) the same whether a service is implemented as a client library or within a server thread. Any difference arises only due to spatial overhead factors (primarily due to less context-switching in the case of client library implementations, for example see [24, 23]). This is hardly a fundamental question of functionality or capability. Consider a service *S* (such as a draw-in-window operation) executing in a real-time server like *X*. The server obtains requests from multiple clients. In a real-time system, the requests will be queued up in priority order *and* with support for priority inheritance to avoid unbounded priority inversion problems. If implemented as a client library, the critical section used within the library will use a mutex, which in turn will use a priority queue for waiting threads and support priority inheritance.⁹
2. *Sharing and interactions are in general unavoidable:* Concurrently running applications interact not only because they eventually share the same underlying physical resources, but also because of logical requirements above the physical layer. Shared display, shared files, concurrent access to bank accounts, shared data such as movies and databases are only some examples of these shared logical resources. As a result, critical sections which manage these shared *logical* resources are unavoidable in the context of multi-tasking and multi-threaded systems. Whether these critical sections are organized in client space or in a dedicated server is only a question of convenience and flexibility with the time/space distinctions coming into play. Anyhow, critical sections can be shortened or optimized but in

general cannot be eliminated.¹⁰

Memory implications of using a client library (with a critical section) and a server also need to be considered. When a service is implemented as a server, it is relatively easy (for example) to wire down that server memory for predictable real-time performance. However, if clients used their own libraries (with critical sections), other relatively more complex issues must be addressed. In one case, each client can have its own copy of the library leading to higher memory usage. In contrast, if shared (dynamically linked) libraries are used, memory usage is the same as a server, but one must now be able to ensure that a shared library is wireable. In other words, a finer granularity of memory control becomes necessary.

The Exo-kernel approach advocates that all policy decisions except for access protection reside in user-level programs. However, for real-time systems, the CPU scheduling policy must be centrally managed (at the "root") to ensure that an application group can satisfy its own timing constraints. This global scheduling policy *cannot* be delegated to individual applications. On the other hand, if the CPU resource management policy is deemed to be a temporal protection mechanism that resides in the exo-kernel, the resource kernel notion is actually compatible with the exo-kernel approach as well. Each application can then build its own local scheduler to use its allocated time in a way that it sees fit. However, in practice, we do not expect local schedulers in user space to provide significant added value. Instead, we propose a Quality of Service (QoS) manager running in user space (as a server) on top of the resource kernel [22, 32]. This QoS manager can arbitrate among competing requests when the maximal requests of all applications cannot be satisfied with the available resources.

6.2. Contributions

We have presented the notion of a resource kernel, which provides timely and protected access to machine resources. In this approach geared towards real-time and multimedia operating systems, guaranteed and protected access

- **Uniformity:** a single resource specification scheme can be applied to different time-shared resource types with timeliness control. The scheme can be locally optimized and applied for each resource type.
- **Resource management transparency:** the use of the exact resource management scheme is hidden from the application programs and changed transparently across different implementations. The implementation of the resource management scheme can use, among other things, fixed priority schemes such as rate-monotonic scheduling [18] and deadline-monotonic scheduling [17], dynamic priority schemes such as earliest-deadline-first [18], or processor sharing schemes such as PGPS, virtual clocks or WF²Q [4]. We demonstrate two very different schemes for CPU and disk bandwidth manage-

⁹In the general case of this discussion, one should replace the notion of priority with the notion of 'scheduling attribute' which may be priorities or reserves with the basic concept remaining the same.

¹⁰Lock-free protocols exist but seem to be useful only under limited conditions.

ment even though each uses the same resource specification model.

- **Resource composability:** We show that multiple resource types can be guaranteed at the same time with acceptable performance levels. In specific, reservations of different resource types can be independently created and then composed. We use the technique of *resource decoupling* [36] and management of *processor co-dependency* using higher priority *system reserves* to provide simultaneous access to CPU resource and another resource type simultaneously. We are unaware of other OS work where simultaneous access to two or more resources is addressed.
- **Hard resource reservation:** In this resource allocation scheme, the usage of a resource cannot exceed the allocated amount of the resource even if the resource is idle. While this may sound draconian and wasteful, we expect that this will be a powerful building block for constructing virtual resources, which allow untrusted applications to be built and run in their own resource space with a pre-determined finite effect on other applications at all times.
- **Interactions and Disk bandwidth management:** The resource kernel is able to monitor and control priority inversion arising from the interactions between real-time tasks due to the use of common shared services. By deliberately introducing priority inversion in a controlled fashion, we demonstrate that there is no significant loss of disk subsystem throughput for acceptably substantial ranges of disk traffic while guaranteeing timely access to disk bandwidth for real-time and multimedia applications. This is achieved using a novel *just-in-time* disk scheduling scheme. Guaranteed access to disk bandwidth is obtained at the expense of a relatively small loss in throughput.
- **Flexibility of resource kernels:** Our resource kernel abstractions allow resource usage to be automatically calibrated, and to be portable across different hardware platforms.

6.3. Future Work

Our future work will include exploring network bandwidth reservation in conjunction with processor and disk reservation. Network bandwidth management has many implications in the context of a resource kernel: protocol stack overhead dominates on the CPU. As a result, network bandwidth management translates to both network reservation and CPU management. The times during which both network bandwidth and CPU cycles need to be available seem to be fairly limited, but remain to be verified.

The issue of CPU co-dependency needs to be addressed at greater length. Additional buffer space between different resource types with hardware buffers can also alleviate the problem; this is typical of today's hardware systems with self-triggered DMA on sound cards (such as the SoundBlaster 16), and bus-mastering on multi-master backplanes such as the PCI bus. Finally, distributed resource reservation in networked systems will open up another frontier of work.

Appendix: Admission Control Schemes

1. Resource Specification Notation

Let the set of n reservations requiring processor reservation be denoted as $\tau_1, \tau_2, \dots, \tau_n$. Each reservation τ_i needs to obtain C_i units of time every T_i units of time. In addition, the C_i units of resource time must be available at or before D_i in each periodic interval separated by T_i .

2. Admission Control Using Fixed Priority Policies

The reservations are ordered in descending order of their fixed priorities such that for $i = 1$ to $n-1$, $\text{priority}(\tau_i) < \text{priority}(\tau_{i+1})$.

In mathematical form, a necessary and sufficient condition for the schedulability of a set of periodic tasks using fixed priority scheduling is as follows [14]:

$$\forall i, 1 \leq i \leq n, \quad \min_{0 < t \leq D_i} \left(\sum_{j=1}^i \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right) \leq 1$$

In algorithmic form, the completion time CT_i of a reservation τ_i with a resource allocation can be computed as follows using a recurrence relation [10, 39].

1. Let $w_i^0 := C_i$.
2. Compute $w_i^{k+1} := \sum_{j=1}^{i-1} C_j \left(\left\lceil \frac{w_i^k}{T_j} \right\rceil \right)$.
3. If $w_i^{k+1} > D_i$, $CT_i := \infty$. Skip to Step 6.
4. If $w_i^{k+1} = w_i^k$, $CT_i := w_i^k$. Skip to Step 6.
5. $k := k + 1$. Go to Step 2.
6. If $CT_i \leq D_i$, τ_i meets its deadline.

The completion time test is repeated for all reservations which need to be guaranteed. Even if one reservation will miss its deadline, the admission test will deny the newest incoming request.

3. Admission Control Based on Rate-Monotonic Priority Assignment

The rate-monotonic priority assignment algorithm is an optimal fixed priority algorithm when $D_i = T_i$ [18]. The reservations are ordered in descending order based on their rate-monotonic priorities (i.e., $T_i < T_{i+1}$). The admission control test use the scheme described in Section 2.

4. Admission Control Based on Deadline-Monotonic Priority Assignment

The deadline-monotonic priority assignment algorithm is an optimal fixed priority algorithm when $D_i \leq T_i$ [17]. The reservations are ordered in descending order based on their deadline-monotonic priorities (i.e. $D_i < D_{i+1}$). The admission control test uses the same scheme described in Section 2.

References

- [1] R. Abbott and H. Garcia-Molina. *Scheduling Real-Time Transactions with Disk Resident Data X Server*. Technical Report CS-TR-207-89, Department of Computer Science, Princeton University, February, 1989.

- [2] Baker, T. P. A Stack-Based Resource Allocation Policy for Real-Time Processes. *IEEE Real-Time Systems Symposium*, Dec., 1990.
- [3] Baker, T. Stack-Based Scheduling of Realtime Processes. *Journal of Real-Time Systems* 3(1):67-100, March 1991.
- [4] J. C. R. Bennett and H. Zhang. WF²Q: Worst-case Fair-Weighted Fair-Queueing. In *Proceedings of INFOCOM 96*. March, 1996.
- [5] J. Blazewicz, W. Cellary, R. Slowinski and J. Weglarz. Scheduling under Resource Constraints -- Deterministic Models. In *Annals of Operations Research, Volume 7*. Baltzer Science Publishers, 1986.
- [6] S. J. Daigle and J. K. Strosnider. Disk Scheduling for Multimedia Data Streams. *Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Networking*, 1994.
- [7] D. R. Engler, M. F. Kaashoek and J. O. Toole, Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM Symposium on Operating System Principles*, December, 1995.
- [8] K. Jeffay, D. L. Stone and F. D. Smith. Kernel Support for Live Digital Audio and Video. In *Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 10-21. November, 1991.
- [9] K. Jeffay. Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 89-99. IEEE, December, 1992.
- [10] Joseph, M. and Pandya. Finding Response Times in a Real-Time System. *The Computer Journal (British Computing Society)* 29(5):390-395, October, 1986.
- [11] Khanna, S., Sebree, M., and Zolnowsky, J. Real-Time Scheduling in SunOS 5.0. *The Proceedings of USENIX 92 Winter*:375-390, 1992.
- [12] Klein, M. H., Ralya, T., Pollak, B., Obenza, R. and Harbour, M. G. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9361-9.
- [13] Lehoczky, J. P. and Sha, L. Performance of Real-Time Bus Scheduling Algorithms. *ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1*, May, 1986.
- [14] Lehoczky, J. P., Sha, L. and Ding, Y. The Rate Monotonic Scheduling Algorithm -- Exact Characterization and Average-Case Behavior. *IEEE Real-Time Systems Symposium*, Dec, 1989.
- [15] Lehoczky, J. P., Sha, L., Strosnider, J. K. and Tokuda, H. Fixed Priority Scheduling Theory for Hard Real-Time Systems. *Technical Report, Department of Statistics, Carnegie Mellon University*, 1991.
- [16] Lehoczky, J. P. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proceedings of the IEEE Real-Time Systems Symposium*, Dec., 1990.
- [17] Leung, J. Y., and Whitehead, J. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation* 2(4):237-250, Dec., 1982.
- [18] Liu, C. L. and Layland J. W. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM* 20 (1):46 - 61, 1973.
- [19] Locke, C. D., Vogel, D. R., Lucas, L. Generic Avionics Software Specification. *Technical Report, Software Engineering Institute, Carnegie Mellon University*, 1990.
- [20] S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *The Real-Time Systems Journal* 3:307-336, 1991.
- [21] P. Lougher and D. Shepherd. The Design and Implementation of a Continuous Media Storage Server. *Proceedings of the 3rd International Workshop on Network and Operating System Support for Audio and Video*, November, 1992.
- [22] C. Lee and R. Rajkumar and C. Mercer. Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *the proceedings of Multimedia Japan 96*, April, 1996.
- [23] C. Lee and K. Yoshida and C. Mercer and R. Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. In *the proceedings of IEEE Real-time Technology and Applications Symposium*, June, 1996.
- [24] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244-255. December, 1993.
- [25] C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. May, 1994.
- [26] C. W. Mercer and R. Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. May, 1995.
- [27] A. Molano, K. Juvva and R. Rajkumar. Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *IEEE Real-Time Systems Symposium*. December, 1997.
- [28] Needham, R. and Nakamura, A. An Approach to Real-Time Scheduling: Is it really a problem for multimedia? *The Third International Workshop on Network and Operating System Support for Multimedia*, 1992.
- [29] *Nemesis, the kernel: Overview* Dickson Reed and Robin Fairbairns, Editors, May 20, 1997.
- [30] *IEEE Standard P1003.4 (Real-time extensions to POSIX)* IEEE, 345 East 47th St., New York, NY 10017, 1991.
- [31] Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.
- [32] R. Rajkumar, C. Lee, J. P. Lehoczky and D. P. Siewiorek. A QoS-based Resource Allocation Model. *IEEE Real-Time Systems Symposium*, December, 1997.
- [33] Sha, L., Rajkumar, R. and Lehoczky, J. P. Task Scheduling in Distributed Real-Time Systems. *Proceedings of IEEE Industrial Electronics Conference*, 1987.
- [34] Sha, L., Rajkumar, R. and Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *Technical Report (CMU-CS-87-181)*, Department of Computer Science, CMU, 1987.
- [35] Sha, L., Rajkumar, R. and Lehoczky, J. P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*:1175-1185, September, 1990.
- [36] Sha, L., Rajkumar, R. and Sathaye, S. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE (journal)*, January, 1994.
- [37] Sprunt, H.M.B., Sha, L., and Lehoczky, J.P. Aperiodic Task Scheduling on Hard Real-Time Systems. *The Real-Time Systems Journal*, June, 1989.
- [38] John A. Stankovic. Misconceptions about Real-Time Computing. *Computer* 21(10):10-19, Oct., 1988.
- [39] Tindell, K. *An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks*. Technical Report YCS189, Department of Computer Science, University of York, December, 1992.
- [40] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Operating Systems Design and Implementation*, pages 1-11. November, 1994.

Real-Time Filesystems

Guaranteeing Timing Constraints for Disk Accesses in RT-Mach

Anastasio Molano, Kanaka Juvva and Ragunathan Rajkumar

Real-Time and Multimedia Laboratory¹

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15146

{amolano, kjuvva, raj+}@cs.cmu.edu

Abstract

Traditional real-time systems have largely avoided the use of disks due to their relative slow speeds and their unpredictability. However, many real-time applications including multimedia systems and real-time database applications benefit significantly from the use of disks to store and access real-time data. In this paper, we investigate the problem of obtaining guaranteed timely access to files on a disk in a real-time system. Our study focuses on several aspects of this problem of providing a real-time filesystem. First, we consider the use of two real-time disk scheduling algorithms: earliest deadline scheduling and just-in-time scheduling, a variation of aperiodic servers for the disk. The latter algorithm is designed to improve disk throughput that can be hurt when a real-time scheduling algorithm such as EDF is applied directly. Admission control policies with practically acceptable properties of performance and usability are provided. Next, we design and implement a real-time filesystem on the RT-Mach microkernel-based system running a real-time shell. The new interface we develop is based on RT-Mach's resource reservation paradigm and provides guaranteed and timely access for multiple concurrent applications requiring disk bandwidth with different timing and volume requirements. Finally, we perform a detailed performance evaluation of the real-time filesystem including its raw performance. We show the following positive but rather surprising result: our real-time scheduling filesystem not only provides guaranteed and timely access but also does so at relatively high levels of throughput. Traditional disk scheduling algorithms offer completely unacceptable file access latencies for real-time applications and do so only at slightly higher throughput.

1. Introduction

Real-time and embedded systems are used in many application domains including feedback control systems, (manufacturing) process control, robotics, air traffic control, avionics, target-tracking, real-time object recognition, discrete assembly and vehicle navigation. Many of these systems such as air traffic control tend to be distributed in nature and also need to satisfy end-to-end timing requirements which span processor boundaries. While real-time

applications are many, traditional real-time systems have largely avoided the use of disks. This is in part because they may be relatively slow for some real-time applications. However, many real-time applications can benefit from the use of disks to store and access real-time data (such as real-time database applications). Unfortunately,

- disk accesses can introduce unpredictable latencies, and
- disk access requests must also be managed in conjunction with processor scheduling.

An emerging set of commercially and strategically important real-time applications demand real-time disk accesses. These applications include real-time databases, C³I systems, multimedia applications, on-demand services and intranet servers for in-house training and continuing education.

On the processor side, fixed priority algorithms allow a mix of tasks with different periodicity, and hence the disk subsystem must do too. The disk scheduling problem has not been studied extensively in the general context. We focus on a general computing platform (including desktops) where multiple applications with and without a range of timing constraints and file I/O constraints can be running concurrently. Finally, we extend the processor reservation model in RT-Mach to include guaranteed and timely access to disk bandwidth. In other words, an application can request a portion of the disk bandwidth with pre-specified timing constraints and if the request is accepted by the admission control policy, the application is guaranteed to obtain its requested share of the disk bandwidth on a timely basis.

1.1. Comparison with Related Work

A wealth of resource management schemes and scheduling algorithms exists from which one can draw. Some of our goals (such as resource centrality) are similar to those of Microsoft Research's Rialto kernel among others. The reservation model also has its counterparts in network reservation protocols as used in ATM and RSVP. However, the operating system problem seems more complex in one sense since inherently different resource types must be dealt with, while networks essentially deal with

¹This research was supported by the Defense Advanced Research Projects Agency in part under contract number E30602-97-2-0287 and in part under contract number F30602-96-1-0160. Mr. Molano was funded by a research grant from the Community of Madrid and by the National R&D Program of Spain under contracts TIC96-0982 and TIC97-0438.

one type (namely network packets). Work described in [1, 2, 7, 8] study various aspects of real-time disks and filesystems. The approach used in [1] is a variation of the Scan algorithm in which the SCAN direction changes towards the Earliest Deadline First request only if the deadline is considered to be met. At any scheduling point the request with the Earliest *Feasible* deadline is chosen for service and the disk head scans towards it. Chen et al. [7] propose two algorithms which combine deadline information and disk service time information. They show that these algorithms minimize transaction loss ratio in real-time database applications. Both [1] and [7] are proposed and studied under soft deadline schemes. A scheduling model close to ours is found in [2] but its approach is one of using fixed priority scheduling, minimizing blocking through the use of "chunking" and using a static task set. Also, only simulation studies were carried out. In contrast, we use dynamic priority scheduling, exploit blocking instead of minimizing it and evaluate an implementation within our resource framework.

A multimedia storage server design and implementation is reported in [15]. In this scheme, the disk request queue is divided into two different queues, one for real-time threads and another for non-real-time threads. Both queues were scheduled following C-Scan with the real-time queue taking precedence over the non-real-time queue. The scheme is called a constant-rate access server (CRAS) and accurately reflects the attempt to only support a single rate of service at the disk. However, in a general real-time or multimedia application, many different applications would need to access the disk at different rates. Our scheme employs a different period for each real-time activity, and guarantees a "share" of that period to the activity. We show that the processor scheduling paradigm can indeed be adapted to be useful in the context of disk bandwidth scheduling. Our scheme can be used in conjunction with processor scheduling.

RT-Mach previously offered only traditional non-real-time support for real-time disk scheduling. In other words, disk requests were not served based on priorities or deadlines. As a result, a real-time application such as a video player was forced to read the file completely from disk, store it in memory and then play it back. The playing was treated as the real-time aspect and since strong processor scheduling support was available, many real-time and multimedia applications with concurrent audio and video streams could co-exist and meet their deadlines. The disk access portions, however, continued to be done in non-real-time. In this paper, we add real-time filesystem support to RT-Mach in a way that is completely compatible with its resource reservation model. Our detailed set of experiments presented later in this paper show that guaranteed timely access to files on disk is possible, and that this can be achieved at relatively low cost in terms of disk throughput.

1.2. A Brief Overview of RT-Mach

The primary goal behind the RT-Mach effort [9]² is to

design, develop, demonstrate and distribute an integrated framework that encompasses task scheduling, virtual memory management, synchronization mechanisms, inter-process communications, real-time disk scheduling, network protocol processing and distributed coordination. The framework is intended to ease and facilitate the development of predictable real-time applications. RT-Mach adopts the *resource reservation model* [10, 12], which allows applications to specify their resource demands independent of the scheduling algorithm actually used in the kernel. A real-time socket library provides predictable and efficient protocol processing for networked applications. A schedulability analysis tool and a distributed monitoring facility provide support for the analysis and debugging of distributed real-time applications. Extensive support is available for multimedia devices including full-duplex audio, real-time video capture, mobile networking with networking support.

1.2.1. Resource Kernels

A *resource kernel* [12] is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

The resource kernel gets its name from its resource-centricity and its ability to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage.

1.2.2. RT-Mach as a Resource Kernel

RT-Mach is a resource kernel, and attains its capabilities by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritize them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

Explicit Resource Parameters: The RT-Mach resource reservation model employs the following parameters: computation time C every T time-units for managing the net utilization of a resource, a deadline D for meeting timeliness requirements, a starting time S of the resource allocation, and L , the life-time of the resource allocation. We refer to these parameters, $\{C, T, D, S$ and $L\}$ as explicit parameters of our reservation model. The semantics are

²URL: "http://www.cs.cmu.edu/~rtmach"

simple, well-known in the real-time community and are as follows. Each reservation will be allocated C units of usage time every T units of absolute time. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval. The guarantees start at time S and terminate at time $S + L$.

Implicit Resource Parameter: The resource kernel implicitly derives, tracks and enforces the implicit B parameter for each reservation in the system. B represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its B is unbounded, a reservation cannot meet its deadline. Priority (or reservation) inheritance is applied when a reservation blocks, waiting for a lower priority reservation to release (say) a lock. As we shall see in Section 2.6, this implicit parameter B can also be used to deliberately introduce priority inversion in a controlled fashion to achieve other optimizations.

1.3. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we motivate and describe in detail the disk scheduling algorithms we adopt. We present an aperiodic server-like disk scheduling algorithm called "Just-in-Time" disk scheduling. In Section 3, we present the design and implementation of a real-time filesystem in the RT-Mach operating system environment. In Section 4, we perform a set of detailed experiments to evaluate the timeliness and throughput properties of our real-time filesystem, and compare it with the traditional Scan disk scheduling algorithm. Finally, in Section 5, we conclude with some remarks.

2. Disk Bandwidth Resource Management

Many real-time applications like real-time databases and C3I systems can benefit from having access to disks. Desktop multimedia systems also need to read from (or write to) disk storage relatively large volumes of video and audio data. In addition, these streams represent continuous media streams, and must therefore be processed by the disk subsystem in real-time. In other words, it would be very useful in practice if disk bandwidth can also be guaranteed in addition to managing processor cycles.

In this section, we present a simplistic disk scheduling algorithm based on earliest deadline scheduling. We then improve the algorithm by exploiting "slack" in the reservations to obtain a hybrid of earliest deadline scheduling and a traditional scan algorithm. Our evaluations of these schemes in Section 4 show that guaranteed disk bandwidth reservation can be obtained at only a small loss of system throughput.

2.1. Important Considerations

We now outline some important considerations which need to influence the design of a real-time filesystem:

- *Preemptibility issues:* Once a request is issued to the disk drive, it will not be preempted until it has finished, even if there are higher priority disk requests waiting for service. The time that a higher priority disk request may wait until being serviced is bounded by the longest disk

request, which can still be rather long. The duration of the non-preemption window must ideally be small and perhaps even dynamically adjustable depending on the workload.

- *With Preemption:* by implementing fine-grained accesses to the disk, a higher priority disk request can preempt a lower priority disk request midway through the processing of its larger request. Rather than sending the whole disk request in one SCSI command (for example), one can send smaller disk requests successively with several SCSI commands, so that they can be preempted at smaller intervals. We shall study the impact of the disk block sizes on the disk throughput in Section 4.
- *Heterogeneity of the workload:* Consider very heterogeneous workloads where there are many small requests with deadlines, but they are prevented from execution due to larger low priority disk requests. Examples of such systems are heterogeneous C3I real-time databases.

Consider homogeneous workloads such as multimedia storage servers, where *all* the requests are periodic ones. SCAN-based schemes are the most effective under these considerations since they avoid expensive disk head movements (seeks).

2.2. Filesystem Bandwidth Specification

Our resource specification model for disk bandwidth is identical to that of processor reservation in RT-Mach. In other words, a disk bandwidth reservation must specify a start time S , a processing time C to be obtained in every interval T before a deadline of D . The processing time C can be specified as # of disk blocks (as a portable specification) or in absolute disk bandwidth time in native-platform specification.

In the rest of this paper, we shall use the terms "filesystem reservation", "fs reserves", "filesystem bandwidth reservation" and "disk bandwidth reservation" interchangeably,

2.3. Resource Specification Notation

Let the set of n reservations requiring resource reservation be denoted as $\tau_1, \tau_2, \dots, \tau_n$. Each reservation τ_i needs to obtain C_i units of time every T_i units of time. In addition, the C_i units of resource time must be available at or before D_i in each periodic interval separated by T_i .

2.4. Scheduling Policy

Instances of a disk bandwidth reservation become eligible to execute *every* T_i units (at times $S_i, S_i + T_i, S_i + 2T_i, S_i + 3T_i, \dots$). Consider an instance which arrives at time $S_i + nT_i$. This instance has a deadline of $S_i + nT_i + D_i$. Similarly, all instances of all outstanding disk bandwidth reservations have corresponding deadlines. After each disk block access is completed, the disk scheduler makes another scheduling decision. It picks the next ready reservation instance with the earliest deadline and issues a disk access command corresponding to that instance's next disk access request. If there are no pending requests, the disk remains idle.

2.5. Admission Control

Our simplest disk head scheduling scheme employs the earliest deadline scheduling algorithm (EDF) [6]. Since EDF is a preemptive scheduling algorithm, a higher priority reservation must be able to preempt a lower priority reservation. However, instantaneous preemptions are not possible in disk scheduling. An ongoing disk block access must complete before the next highest priority disk block access request can be issued. This introduces a blocking (priority inversion) factor of a single filesystem block access when $D_i = T_i$ (as per [13]). When $D_i < T_i$, the required earlier completion time ($T_i - D_i$) is added to the blocking factor.

For the *rate-monotonic* [6] and the *deadline-monotonic* [5] fixed priority schemes, the completion time CT_i of a reservation τ_i with a resource allocation can be computed using a recurrence relation [3, 16]. If this completion time is less than the deadline for all reservations including the incoming one, a reservation request can be admitted. Else, it will be rejected.

For the *earliest deadline first* policy, the reservations requiring disk bandwidth are ordered according to rate-monotonic priorities, (i.e. $T_i < T_{i+1}$). The number of disk blocks to be read by an instance of τ_i is denoted as L_i . The admission control test for our disk bandwidth reservation scheme is given by

$$BU_{max} + \sum_{i=1}^n U_i \leq 1.0$$

where, U_i is the utilization of task τ_i given by

$$U_i = \frac{t_{IPC-in} + t_{FSB}L_i + 2t_{Seek} + 2t_{Rot} + t_{IPC-out}}{T_i}$$

t_{FSB} is the time to read (or write) one single file system block given by

$$t_{FSB} = t_{fs-overhead} + \frac{\text{size of FSB in bytes}}{\text{disk BW in bytes/sec}}$$

where,

t_{seek} is the maximum seek time of the disk head,
 t_{rot} is the maximum rotational latency for the disk,
 t_{IPC-in} is the overhead for invoking a read operation,
 $t_{IPC-out}$ is the overhead for the result of the read op, and
 $t_{fs-overhead}$ is the overhead incurred in the filesystem itself.

BU_{max} is the maximum utilization of the priority inversion encountered by all guaranteed reservations, given by

$$BU_{max} = \max \left(\frac{B_1}{T_1}, \frac{B_2}{T_2}, \dots, \frac{B_n}{T_n} \right)$$

and B_i is the priority inversion duration encountered by reservation τ_i given by

$$B_i = t_{FSB} + T_i - D_i$$

It must be noted that we add $t_{Seek} + t_{Rot}$ only twice to each periodic instance of a disk bandwidth reserve, one for

switching the request stream and switching it out once. This seems logical particularly since this is how context-switching on the processor is handled. If all the disk blocks that an application needs are sequentially numbered, this situation is analogous to the processor case. Unfortunately, this may not be sufficient to guarantee disk access deadlines for all applications at all times. Many applications will create files or read file blocks not in sequence but potentially in some (pseudo-)random order. For such applications, this "context-switching factor" is not sufficient in the worst case. However, given that both the maximum seek time and the rotational latency can be fairly large, including them would typically mean that only a small portion of disk requests would be able to pass the admission control scheme. In Section 4, we show that under non-contiguous layout such an assumption does lead to deadlines being missed, but the number of deadlines missed is very small, that it perhaps will not be a concern for many applications. If an application cannot tolerate any deadline miss, then it has to include the context-switching for potentially each disk block it accesses, or use contiguously allocated files. There are three mitigating factors:

- A real-time filesystem can (as ours does) allocate disk blocks as contiguously as possible.
- Utilities (such as *tunefs* on Unix systems, filesystem optimizers) can be used to choose the block allocation policy within a filesystem and/or relocate file blocks so that they are contiguous.
- With contiguous layout, our current admission control test guarantees the requested bandwidth if the test succeeds. The test will succeed for relatively high bandwidths if the periods of the requests are large compared to the seek/latency factor used in the test. For example, suppose one wants to guarantee (say) 200 KBytes per second. Using parameters of the disk we used, one will consume 100% of utilized bandwidth (with overhead) by reserving 20 KBytes from disk every 100 ms. If instead, we transform the periods and reserve 200 KBytes per 1 second, the utilization is just 32.5%. With scaled up periods, more bandwidth can be guaranteed for contiguous files, or guarantees can be provided for non-contiguous files by including additional seek and rotational latencies for each non-contiguous block access. On the other hand, bigger application buffers will be necessary and longer end-to-end delays will be incurred.

2.6. Exploiting 'B': Just-In-Time Disk Scheduling

The earliest deadline disk scheduling blindly picks the next block with the earliest deadline irrespective of the current position of the disk head. Since the physical movement of the disk head and the disk's rotational latencies constitute significant durations of time, such dynamic scheduling can result in significant disk subsystem throughput reductions particularly under heavy disk traffic. The reductions can be directly attributed to the time wasted by the disk head moving from one end to another and the disk's rotational time. In summary, the deadlines are preferred over a block's physical location.

Traditional scan algorithms, in contrast, re-order the disk request queue such that the block closest to the current head position (in the direction of movement) is accessed next. As a result, a disk request which just arrived can be serviced before another disk request which has been waiting for a long time just because the latter is farther away from the head position. To summarize, the physical block location is favored over timeliness.

The earliest deadline scheduling algorithm and the scan algorithm are therefore at odds with one another. Fortunately, a hybrid scheme which can obtain all the benefits of the earliest deadline scheduling algorithm and at least part of the benefits of the scan algorithm is possible. Using the notion of slack-stealing inherent in all aperiodic servers [14], a scheduling algorithm can exploit the slack available to higher priority tasks to schedule accesses of other disk blocks which are closer to the current head position.³

A brief description of the just-in-time disk scheduling algorithm is as follows. The maximum "slack" available to each disk reservation is computed whenever a new request is admitted (or an existing reservation is deleted). At run-time, if the current slack of higher priority reservations is non-zero, another unreserved (or lower priority reserved) request can be scheduled if closer to the disk head. If slack is stolen, the slack of higher priority reservations is reduced by one. This process is then repeated. If the slack of a high priority reservation goes to zero, it will be serviced independent of its location.

2.7. "Just-in-Time" Slack-Stealing Algorithm for Disk Bandwidth

Let $K_i(t)$ denote the maximum "slack" in units of time available for each reserved disk data stream, and let K_i denote the maximum slack available in units of disk blocks. We have,

$$K_i(t) = D_i - T_i \sum_{j=1}^i (U_j)$$

$$K_i = \left\lfloor \frac{K_i(t)}{t_{FSB}} \right\rfloor$$

where U_j and t_{FSB} are as defined earlier.

2.8. The JIT Replenishment Algorithm

Data Structures: The maximum slack K_i of each reservation τ_i is determined at admission control time. At run-time, the current slack k_i of each instance of τ_i is also maintained. The current slack k_i of each new instance of τ_i is initialized to K_i . At any given time, let the set of undepleted reservations be $\{\tau_{\text{valid_reserves}}\}$, and let the set of indices corresponding to the reservations in $\{\tau_{\text{valid_reserves}}\}$ be $\{J\}$.

Usage by an unreserved activity: An unreserved activity can "steal" early access to a disk block from a reservation if $\forall j, j \in \{J\}, k_j > 0$. When such stealing happens, the current slack factors are updated as $\forall j, j \in \{J\}, k_j \leftarrow (k_j - 1)$.

³Such "slack-stealing" has been done in the context of processor scheduling theory in order to provide better response to aperiodic activities [4]. The optimization, cost functions and implementation tradeoffs seem to be different for the processor and the disk, however.

Stealing by a depleted reservation: A depleted reservation τ_i waiting for resource replenishment can steal early access to a disk block from a reserved reservation if $\forall j, j \in \{J\}, k_j > 0$. When such stealing is allowed to occur, the current slack factors are updated as $\forall j, j \in \{J\}, k_j \leftarrow (k_j - 1)$.

Stealing by an undepleted reservation: An undepleted reservation τ_i can also steal early access to disk blocks from a higher priority undepleted reservation if $\forall j, (j \in \{J\}) \wedge (j \leq (i-1)), k_j > 0$. When such stealing is allowed to occur, $\forall j, (j \in \{J\}) \wedge (j \leq (i-1)), k_j \leftarrow (k_j - 1)$.

3. Disk Bandwidth Reservation in Real-Time Mach

In this section, we describe the design and implementation of the real-time filesystem supporting disk bandwidth reservation in Real-Time Mach.

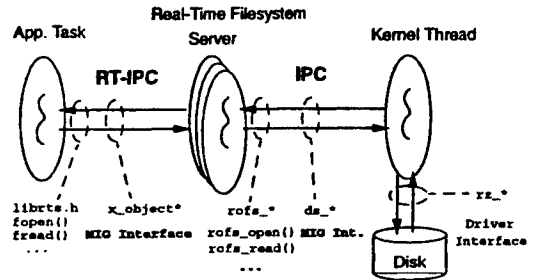


Figure 3-1: Real-Time File System Service Layers

3.1. The Architecture of the Reserved Filesystem

The architecture of our bandwidth-reserved filesystem follows a "traditional" scheme. A *Real-Time File Server* running on top of our resource kernel (based on the RT-Mach microkernel) manages the reserved real-time filesystem. RT-FS has multiple worker threads which receive and process filesystem access requests from real-time clients. Each worker thread stores the incoming request it is processing into a common I/O request queue. The worker thread responsible for issuing the current disk block access waits for its completion. It then awakens, and determines the next request based on the scheduling policy above. If the next request corresponds to another worker thread, that thread is signaled. Else, the worker thread continues to service its remaining disk access requests, if any. Priority inheritance is applied where necessary across the threads to avoid priority inversion problems.

Disk bandwidth reserves have been implemented within RTS (Real-Time server), which runs on top of the RTMach microkernel. RTS supports a process manager, a file system, a device manager, and a simple command interpreter. The most salient feature of RTS is the use of RT-IPC, a realtime interprocess communication mechanism to avoid priority inversion at the server side that includes priority based message queue ordering, establishment of the message buffer size, and control of the handoff policy [11].

The file system runs in the context of realtime worker threads which execute concurrently and attend user requests

sent by means of RT-IPC messages (see Figure 3-1. The file system invokes I/O operations using the RT-Mach device driver interface (`ds_routines()`) implemented as IPC messages to the kernel. The disk driver executes in kernel context, and performs I/O to the disk controller for the actual transfer to the disk. This I/O interface is synchronous in that we wait for the completion of the request before issuing the next one to the disk. The file system has been modified and extended to support the file system reserves mechanism. A new module has been added that maintains file system reserves and implements the interface for creating, terminating and starting them. Additionally it manages the replenishment of file system reserves by means of a replenishment thread handler as it is described below. MIG is the Mach Interface Generator, an Intermediate Data Language for inter-process communications with Mach and RT-Mach.

The file system reserve scheduling mechanism manages all the invoking disk I/O requests, scheduling reserved disk requests under EDF/JIT, and unreserved ones or depleted requests under SCAN. The requests are depleted when they have consumed all their allocated budget (measured in number of disk accesses). The file system accounts for the current resources consumed for each of the pending requests updating their disk usage counter each time they have issued a request to the disk. Within the file system itself, new I/O operation primitives have been added to include the file system reserve control mechanism. The reserved file system maintains two queues of requests, one for reserved requests in reserved mode (with budget different from zero), and other for both unreserved requests and depleted requests (reserved ones which budget have reached zero). The requests queue is protected against concurrent access by means of a mutex and a condition variable that allow the synchronization between all the worker threads executing within the file system concurrently.

The scheduler within the RT filesystem has been implemented as a cooperative scheduler, in the sense that it is executed by the current active thread, which will give control to the next one following the chosen scheduling policy. Worker threads invoking a I/O operation sleep in a condition variable till being signalled as the next request by the active thread currently executing the scheduling algorithm.

The file system is executed in a loop, sending a command to the disk controller each iteration to read/write a file system block. After finishing the I/O operation, the current active thread executes the scheduling algorithm to find out which one should be at this time the next request to execute. It updates its own disk bandwidth reserve budget by decreasing in one unit the number of remaining disk accesses. It checks the queue of unreserved requests to see if some depleted request has been replenished. In this case it removes the depleted request from the queue of unreserved requests and store it in the queue of reserved requests.

It executes the scheduling algorithm (EDF, EDF/JIT, SCAN) choosing the next request based on the current scheme. File system block numbers are used as a reference of physical placement on disk. If the next request is not the

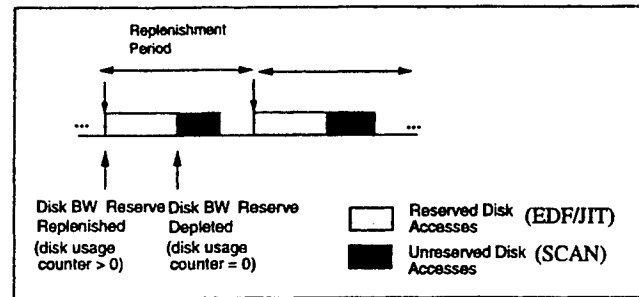


Figure 3-2: Disk BW Reserves Replenishment Scheme

current request then the thread executing the current request signals the thread that will execute the next one, and it is put to sleep after that. The system detects that some reservation has to be depleted when its disk usage counter reaches zero, and it detects when it has to be replenished by obtaining updates from a replenishment handler. A periodic timer and a replenishment timer are allocated to each of the active disk bandwidth reserves.

3.2. The API

A synopsis of the application programming interface we use is presented in Figure 3-3.

```
/* Real-Time File System Services for *
 * disk bandwidth reservation          */

disk_bw_reserve_create(*disk_bw_reserve);
disk_bw_reserve_request(disk_bw_reserve,
    period, start, deadline, bytes_reserved);
disk_bw_reserve_terminate(disk_bw_reserve);
disk_bw_reserve_set_name(disk_bw_reserve, name);
disk_bw_reserve_set_attribute(disk_bw_reserve,
    flavor, new_attr, new_attr_count);
disk_bw_reserve_get_attribute(disk_bw_reserve,
    flavor, old_attr, *old_attr_count);
```

Figure 3-3: Core Application Programming Interface for the Real-Time Filesystem

4. Performance Evaluation

We conducted a series of experiments to study the efficiency of the disk scheduling algorithms and their impact on applications. We start with a performance study of the raw disk and its sensitivity to the size of the data blocks read from the disk. We then summarize our results both quantitatively and qualitatively using two workloads, one synthetic and one real. In the synthetic workload, a single real-time thread uses both the CPU and the disk and therefore needs timeliness guarantees on both. Competing background priority threads accessing the disk continuously attempt to disrupt the timing behavior of this real-time application. Next, we test the real-time filesystem on a video playback application using a QuickTime movie player, which reads a movie file from a disk, processes it and displays the frames on the screen. Heavy competition from non-real-time threads continues in this experiment.

4.1. Disk Parameters

The disk we used for our performance evaluation was a 1 GB drive with a max. seek latency of 24 ms, and a maximum rotational latency of 14 ms. The disk blocks were 512-bytes in size while the filesystem used 4 KB blocks.

The time to access a disk block is on the order of 3 - 6 ms. The RT-Mach microkernel, Version RK97a, was used on Pentium 120 MHz workstations. The RTS environment described in the earlier section was used to support the filesystem utilizing the disk bandwidth reserves. The period of the real-time timer was set up to be 1 ms, a resolution that we found to be sufficient for our experiments.

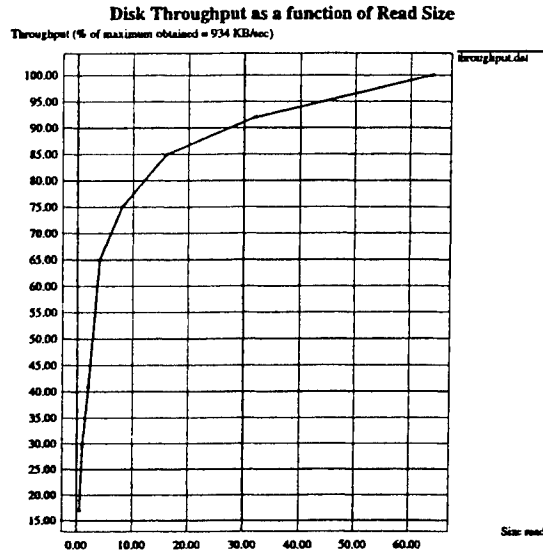


Figure 4-1: The Raw Throughput Obtained from the Disk as a function of data bytes read (in KB)

4.2. Overhead Components of Raw Disk Access

Size (KB)	Filesystem	ds_invoke	IO Transfer	ISR	DS_return
32	1145	36	26414	9	182
16	574	32	14425	9	144
8	336	32	8496	9	130
4	178	31	5111	9	150
2	93	30	3800	8.8	150
1	67	30	3300	8.5	160
0.5	50	30	2350	8	178
Comments	f(Size)	constant	f(size)	constant	constant

Figure 4-2: Overhead Components of Raw Disk Read - μ s

We implemented and tested a Raw and Synchronous interface to the disk under RT-Mach to understand and measure the disk's behavior in terms of the size of the data blocks read from disk and the overhead encountered in various levels of the software. In specific, the interface we built bypassed the file system and accessed blocks on the disk in sequence. Each request must be satisfied before the next one can be submitted. The synchronous interface was then used by the real-time filesystem server to issue disk read requests one block at a time.

We simulated accesses with different granularity and with different layout (contiguous and non-contiguous accesses). The raw throughput obtained from the disk for various

granularities of the data block is plotted in Figure 4-1. Higher the size of the data blocks read, higher is the efficiency obtained by the underlying disk controller. That is, disk throughput increases with the size of an issued read request to the disk. The disk throughput obtained for reading a 64 KB data block was 934 KB/second. The disk throughput obtained for other data block sizes are normalized to this value. As can be seen, the throughput drops off precipitously as the block size drops to 8 KB or below.

The performance overheads in the various software layers in the filesystem and the kernel are listed in Figure 4-2 for the same data block sizes. As can be seen, the invocation of the disk driver (ds_invoke) and its return (ds_return) have constant values as is the processing time for an interrupt service routine (ISR). As can be expected, the transfer time between the disk and the processor is proportional to the size of the data block transferred. Similarly, the filesystem code which receives the data spends time proportional to the block size read.

4.3. Experiment #1

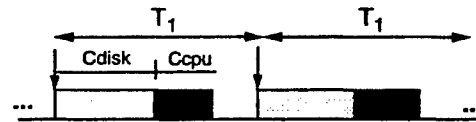


Figure 4-3: Thread execution pattern in Experiment #1

In this experiment, one periodic thread reads from the disk in several configurations: with and without disk bandwidth reserves, with and without cpu reserves, but all in the presence of competition. We ran the experiment in a window time-span of 100 seconds measuring the following parameters: completion time, disk utilization per period, and total disk utilization. The execution of the real-time thread is as shown in Figure 4-3, where T_1 is the thread period ($= 250$ ms), C_{disk} is the time ($= 162$ ms, time to read 48KB) to read disk blocks during each of its periodic intervals and D_{disk} is the deadline for disk access completion ($= 162$ ms also). D_{disk} was deliberately made the same as C_{disk} forcing the admission control test to reach 100% of the utilization factor, thereby forcing a stringent test for the real-time filesystem. C_{cpu} ($= 44$ ms) is the maximum allocated execution time each period of the thread that uses data read using the disk bandwidth reserve. Six periodic threads (with no disk bandwidth reserves) having periods 340 ms, 380 ms, 400 ms, 420 ms, 450 ms, 630 ms were competing for disk accesses attempting to read 60KB, 50KB, 200KB, 100KB, 120KB, 150KB during each of their periodic instances. In addition, an unreserved aperiodic thread was in an infinite loop accessing 4KB for each loop iteration. However all the unreserved threads behave exactly the same as soon as they miss a deadline - they execute as soon as the disk is available as the period has been overrun.

We measured the completion times of the reserved thread

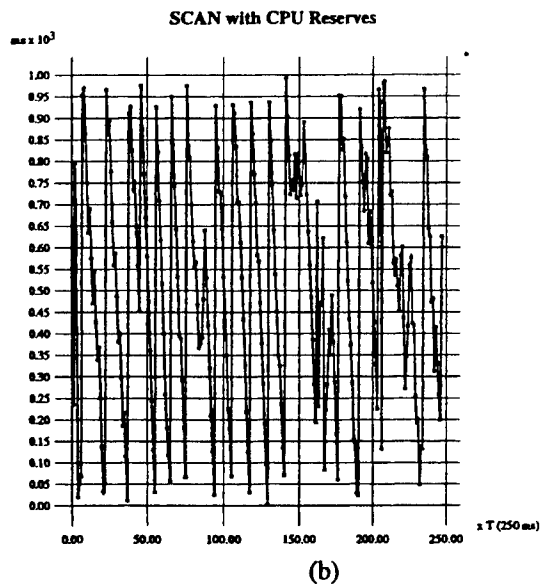
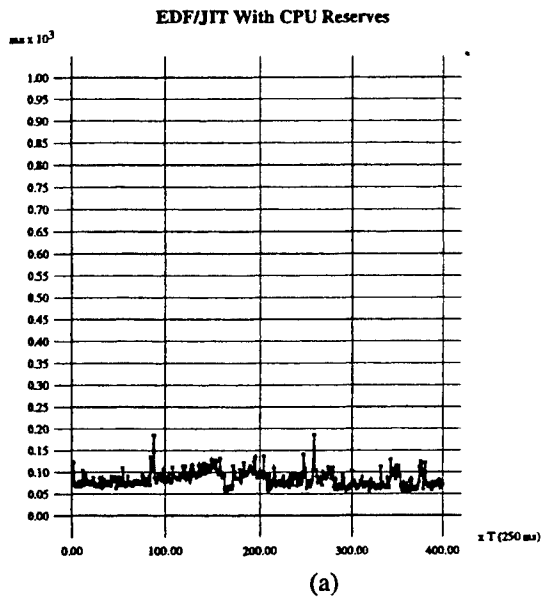


Figure 4-4: Experiment #1: Completion Times

each period (it should be smaller than D_{disk}). Our measurements were conducted with and without CPU reserves for the CPU portion of the real-time thread. The results in either case are qualitatively similar with the CPU reservation case being moderately better. This is because the competing threads had very limited CPU processing needs. Due to paper length limitations, we plot in Figure 4-4 only the results when a CPU reserve is used for the real-time thread. When we use EDF/EDF+JIT, the deadline of 162 ms is missed two times out of 400: the 88th and 258th instances complete in about 185 ms , which is still less than the period boundary of 250 ms . These misses happen because of two reasons. First, as mentioned in Section 2.5, this is because the file being read was not contiguously laid out on the disk, violating an assumption of the admission control policy. We verified this by reading a file as contiguous blocks in which case, *all* deadlines are indeed satisfied. Secondly, files which are longer than 48KB have

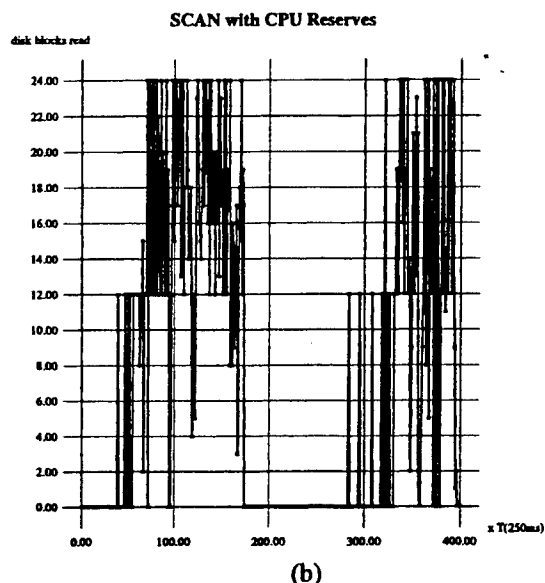
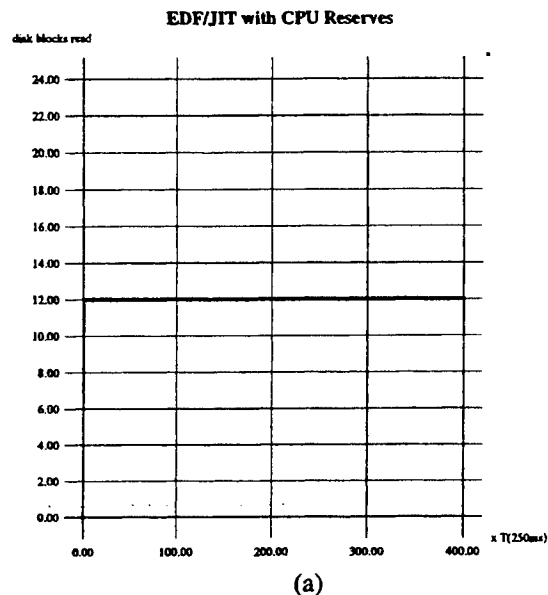


Figure 4-5: Expt. #1: Disk Usage Per Period

block maps (meta-data) which must also be fetched from the disk. This is not accounted for in our current admission control scheme.

In the SCAN case, there is no time to run the needed 400 disk accesses and only 248 access are actually completed during the duration of the experiment. The completion times are nearly always greater than the period itself (> 250 ms) and sometimes much greater. Note that the y-axis scales are very different between the EDF and Scan graphs. This shows that EDF w/ CPU reserves meets the timeliness constraints of the real-time application accessing the disk.

The number of disk blocks read by the real-time thread during each of its periods under the various algorithms are plotted in Figure 4-5. As can be seen, under the EDF schemes, the number of disk blocks read during each period

Thread and the number of blocks read	EDF with CPU Reserves	Scan with CPU Reserves	EDF without CPU Reserves	Scan without CPU Reserves
RT-Thread (reserved)	4788	2975	4800	2980
Thread 1	2100	2267	2100	2681
Thread 2	478	770	510	822
Thread 3	2355	4410	2190	4410
Thread 4	2088	2061	2107	2106
Thread 5	1824	1482	1852	1482
Thread 6	1172	600	1169	650
Looping Thread	2340	3428	2743	3428
Total (blocks read)	17,945	17,993	17,481	18,559
% of Scan	96.69	96.95	94.19	100
% degradation relative to Scan	3.31	3.85	5.81	0

Table 4-1: Expt. #1: Total Disk Usage of each Thread

is exactly 12 (* 4KB/disk block = 48KB), the requested value. In contrast, the Scan algorithm sometimes reads 0 blocks, sometimes 12 and sometimes 24, depending upon the competing stream of requests at any given point in time. However, since it does not and cannot maintain an average of 12 blocks read for the real-time thread, it falls behind for the real-time application.

However, it must be noted that the Scan algorithm was specifically designed to optimize disk throughput. Hence, the question of the relative disk throughput obtained by the Scan and EDF algorithms must be addressed. The disk throughput obtained by each thread under the various schemes is listed in Table 4-1. We obtained the somewhat surprising result that EDF actually performs within 3.5% of the throughput obtained by the Scan algorithm, while satisfying the timing constraints of the application. We shall address the question of whether this throughput difference is always small in Experiment #2.

4.3.1. Experiment #2

Experiment 1 involved synthetic workloads, and therefore its results may or may not correspond to real applications. In addition, that workload assumed a constant load for the real-time task accessing the disk. In practice, a task may also exhibit some stochastic behavior in accessing the disk. We address both of these issues in this experiment. We use a QuickTime movie player and run it within the real-time filesystem context. The player reads blocks from the movie file periodically, processes the data and displays the frames read. The QuickTime decompression algorithm only requires a new frame every 3 or 4 playback periods (it depends on the sequence).

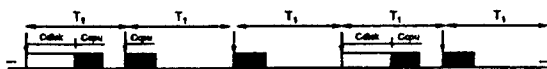
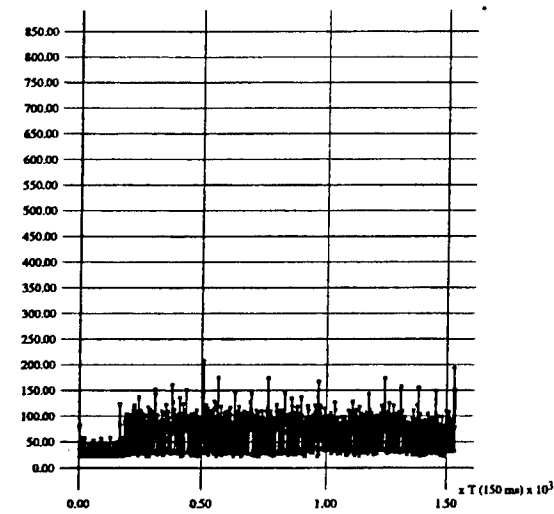


Figure 4-6: Disk accessing pattern of QTPlay_rt

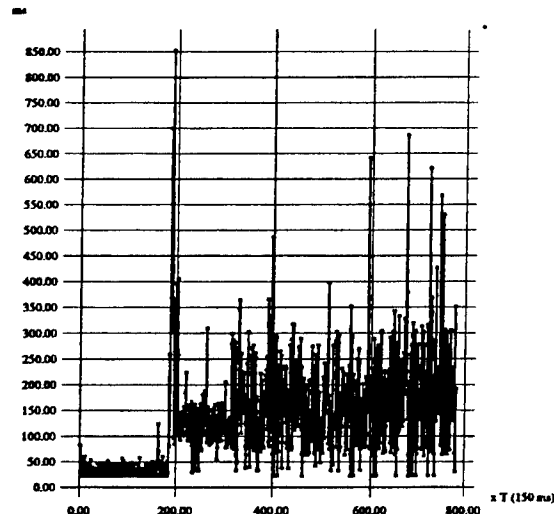
The execution pattern of our QuickTime player application is as shown in Figure 4-6. The video playback thread is a periodic thread that reads from the disk to retrieve the video frames and process them to display the image on the screen. The period T_1 shown is the inverse of the frame rate, C_{disk} is the time to access the disk each period, and C_{cpu} is the execution time need to process a frame each period and display it on the screen.

EDF/JIT with CPU Reserves



(a)

SCAN with CPU Reserves



(b)

Figure 4-7: Expt. #2: QT-Play Completion Times

We ran the experiment under several configurations: with/out disk bandwidth reserves, with/out cpu reserves and under competition in all cases. We ran the experiment in a window time-span of 100 seconds measuring the following parameters: completion time, disk utilization per period, and total disk utilization. When used, the disk bandwidth reserve was for 40KB every 150 ms, with video frame sizes ranging from 19-32 KB. The deadline of the reserve was also 150 ms. The unreserved competition consisted of (a) two periodic threads with periods 180 ms and 380 ms and read 20KB and 100KB respectively during each of their instances, and (b) an aperiodic thread in an infinite loop reading 8KB in every iteration. The video display thread had a period of 50 ms (frame rate = 20). As before, all the unreserved threads execute as soon as the disk is available when they miss an end-of-period deadline. The competition is started after 180 periodic instances of video processing leading to an initial quiescent state.

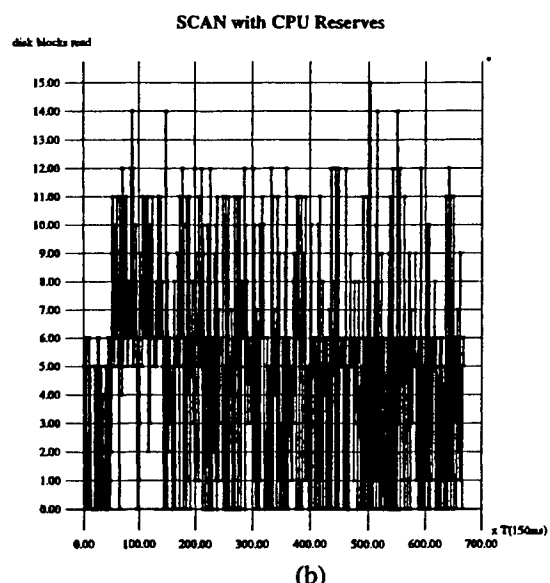
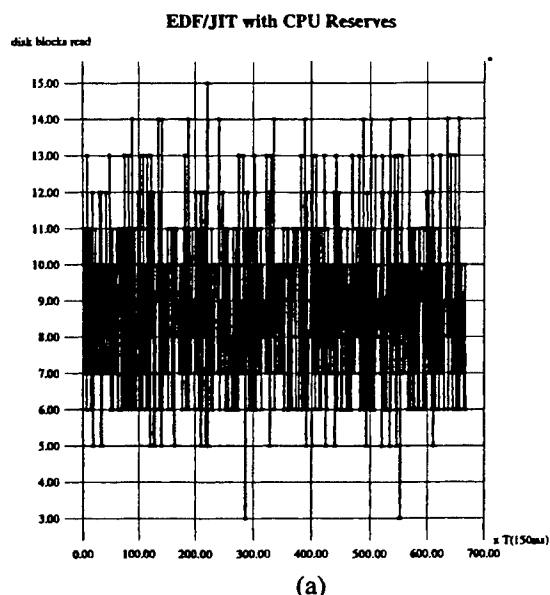


Figure 4-8: Expt. #2: QT-Play Disk Utilization per Period

Test	Scan (total disk blocks read)	EDF/JIT (total disk blocks read)	EDF/JIT Performance Penalty (%) (Scan - EDF) / Scan
1	14951	12521	16.25%
2	13468	11029	18.1%
3	13704	9797	28.5%
4	14879	12601	15.31%
5	15115	13827	8.52%

Table 4-2: Total Throughput Comparison Over Multiple QuickTime Files with Different Disk Locations

We measured the completion times of the video thread for each of its instances. These are plotted in Figure 4-7. As can be seen, under EDF/JIT scheduling, all but a very small number of disk accesses meet their deadline of 150 ms (for the same previous reason of reading a non-contiguous file).

Even those that miss the deadlines miss it by a very small amount. In contrast, under SCAN scheduling, deadlines are consistently missed with some deadlines missed by huge margins (taking up to 850 ms).

The # of disk blocks read by the QTPlay thread under both EDF and Scan is plotted in Figure 4-8. It can be immediately seen from the EDF plot that the QuickTime player does not access the disk every time it runs⁴. The average number of disk blocks read is centered around 9, and is generally above 5. It is also sometimes 0 because it may happen in the sequence that no new block is required in three consecutive periods (each period is 50 ms, and the points are plotted in intervals of 150 ms). In contrast, the Scan algorithm has an average of 6 disk blocks read every instance of the Video thread, with but 0 reads during an unacceptably high number of intervals. This implies that a real-time thread under Scan can fall farther and farther behind since it is using available disk throughput to satisfy the disk requests from other competing unreserved (non-real-time) threads.

4.3.2. Sensitivity of File Location to Disk Throughput

For the above experiment, we found that the total disk throughput under the Scan and EDF/JIT algorithms was 15111 and 10280 disk blocks respectively. In other words, EDF/JIT performs 32% worse than the Scan algorithm in this case. However, Experiment #1 had indicated that EDF/JIT can perform relatively close to Scan in terms of filesystem throughput while the QuickTime player showed that it could be higher. In order to understand the sensitivity, we re-ran Experiment #2 with many movie files at different locations on the disk. These runs are summarized in Table 4-2. We found that depending on the layout of the file in the disk relative to the files accessed by the non-real-time applications, the loss of system throughput could range from 8.5% to 32%. It must be noted that under all these conditions, the EDF/JIT algorithm continues to satisfy disk access deadlines.

The normal optimization strategy of improving filesystem throughput is to pack the disk periodically such that (a) the disk blocks corresponding to a file are close to one another and (b) the blocks are also laid out consecutively. Both of these optimizations will lead to narrower throughput differences between EDF/JIT algorithms and traditional algorithms like Scan. However, EDF/JIT will yield predictable and timely disk accesses for real-time applications, while traditional algorithms do not.

5. Concluding Remarks

Disk storage has been traditionally avoided in real-time systems because of their lack of resistance to vibrations, dirt, size, power, etc., but also because of their slow speeds and unpredictable behavior. However, with today's large disks and advances in packaging technology, the use of disks would be very useful. Desktop and other multimedia systems must necessarily use disks to read and store real-time

⁴The QuickTime movie format yields variable data sizes for each frame played back with the variability depending on the actual movie sequence.

audio and video data. In all these cases, for disks to be usable in a real-time context, disk accesses must be completed on a timely basis. Unfortunately, since disk speeds are limited by physical movements of a disk head, disk throughput is normally enhanced by trying to minimize head movements. This is in general anti-thetic to the real-time requirement, where a late arrival may have the earliest deadline and therefore must be serviced immediately independent of its disk block position. In this paper, we have considered real-time scheduling algorithms that can be used in a general context of concurrent but different types of disk accesses. We presented a "just-in-time" disk scheduling algorithm that attempts to meet timing deadlines while trying to keep system throughput. We have designed and implemented a real-time filesystem in RT-Mach using its resource reservation model. Our performance evaluation on real and synthetic workloads show that our real-time algorithms, and their design and implementation in a practical system yield significant benefits. First, the timing constraints of real-time disk accesses *can* be satisfied. Secondly, contrary to what one might expect, disk throughput does not drop significantly compared to traditional algorithms. The traditional Scan algorithm, for example, yields unacceptable latencies for disk accesses, but exhibits better throughput ranging from 3% to 30% depending on file layouts on the disk.

Our future work on this topic will involve many fronts. If non-contiguous files are used, the admission control scheme can be very pessimistic and lead to extremely low values at which disk bandwidth can be guaranteed. Contiguous files are therefore necessary to obtain more acceptable and predictable disk bandwidth. Hence, we are exploring the performance aspects of contiguous files, and the need for disk space reservation in order to create contiguous files. Finally, we are also studying the problem of co-scheduling processor and disk bandwidth access more extensively.

References

1. R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions with Disk Resident Data X Server. Tech. Rept. CS-TR-207-89, Department of Computer Science, Princeton University, Feb., 1989.
2. S. J. Daigle and J. K. Strosnider. "Disk Scheduling for Multimedia Data Streams". *Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Networking* (1994).
3. Joseph, M. and Pandya. "Finding Response Times in a Real-Time System". *The Computer Journal (British Computing Society)* 29, 5 (October 1986), 390-395.
4. Lehoczky, J. P., Sha, L., Strosnider, J. K. and Tokuda, H. "Fixed Priority Scheduling Theory for Hard Real-Time Systems". *Technical Report, Department of Statistics, Carnegie Mellon University* (1991).
5. Leung, J. Y., and Whitehead, J. "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks". *Performance Evaluation* 2, 4 (Dec. 1982), 237-250.
6. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM* 20 (1) (1973), 46 - 61.
7. S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley. "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems". *The Real-Time Systems Journal* 3 (1991), 307-336.
8. P. Lougher and D. Shepherd. "The Design and Implementation of a Continuous Media Storage Server". *Proceedings of the 3rd International Workshop on Network and Operating System Support for Audio and Video* (November 1992).
9. C. W. Mercer and R. Rajkumar and J. Zelenka. Temporal Protection in Real-Time Operating Systems. *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, May, 1994, pp. 79-83.
10. C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May, 1994.
11. Nakajima, T., Kitayama, T., Arakawa, H. and Tokuda, H. "Integrated Management of Priority Inversion in RT-Mach". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1993).
12. R. Rajkumar, K. Juvva, A. Molano and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. To Appear in the *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, Jan, 1998.
13. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *Technical Report (CMU-CS-87-181), Department of Computer Science, CMU* (1987).
14. Sprunt, H.M.B., Sha, L., and Lehoczky, J.P. "Aperiodic Task Scheduling on Hard Real-Time Systems". *The Real-Time Systems Journal* (June 1989).
15. H. Tezuka and T. Nakajima. Simple Continuous Media Storage Server on Real-Time Mach. *The USENIX 1996 Annual Technical Conference*, San Diego, January, 1996.
16. Tindell, K. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Tech. Rept. YCS189, Department of Computer Science, University of York, December, 1992.

A Resource Allocation Model for QoS Management*

Ragunathan Rajkumar, Chen Lee, John Lehoczy[†], Dan Siewiorek

Department of Computer Science

[†]Department of Statistics

Carnegie Mellon University

Pittsburgh, PA 15213

{raj+, clee, dps}@cs.cmu.edu, [†]jpl@stat.cmu.edu

Abstract

Quality of service (QoS) has been receiving wide attention in recent years in many research communities including networking, multimedia systems, real-time systems and distributed systems. In large distributed systems such as those used in defense systems, on-demand service and inter-networked systems, applications contending for system resources must satisfy timing, reliability and security constraints as well as application-specific quality requirements. Allocating sufficient resources to different applications in order to satisfy various requirements is a fundamental problem in these situations. A basic yet flexible model for performance-driven resource allocations can therefore be useful in making appropriate tradeoffs.

In this paper, we present an analytical model for QoS management in systems which must satisfy application needs along multiple dimensions such as timeliness, reliable delivery schemes, cryptographic security and data quality. We refer to this model as Q-RAM (QoS-based Resource Allocation Model). The model assumes a system with multiple concurrent applications, each of which can operate at different levels of quality based on the system resources available to it. The goal of the model is to be able to allocate resources to the various applications such that the overall system utility is maximized under the constraint that each application can meet its minimum needs. We identify resource profiles of applications which allow such decisions to be made efficiently and in real-time. We also identify application utility functions along different dimensions which are composable to form unique application requirement profiles. We use a video-conferencing system to illustrate the model.

1 Introduction

1.1 Motivation

Many applications can provide better performance and quality of service given a larger share of system resources. For example, feedback control systems can provide better control with higher rates of sampling and control actuation. Multimedia systems using audio and video streams

can provide better audio/video quality at higher resolution and very low latencies. Tracking applications can track objects at higher precision and accuracy if radar tracks are generated and processed at higher frequencies or if better, but more computationally intensive, algorithms are used. Real-time decision-making systems can receive, process and analyze larger amounts of data if more resources are made available. Interactive systems can provide excellent response times to users if more processing and I/O resources are made available.

Applications can therefore seek to improve the quality of delivered services if sufficient resources are available. For example, if encoding/decoding times were not significant, all transmitted data can be encrypted for security/privacy reasons. If spare resources were available, modules can be replicated to assure high availability of critical functionality. Conversely, when resources are tight, applications can still provide lower but acceptable behavior. For instance, a 30 frames/second video rate would be ideal for human viewing, but a smooth 12 fps video rate suffices under many conditions.

Given that applications can operate at high levels of quality or acceptably lower levels of quality based on the resources allocated to them, the following question arises: "How does one allocate resources to those applications when they run concurrently and contend for the same resource types?" This question of resource allocation is traditional in the sense that many papers in the domains of networking, real-time systems and distributed systems have attempted to answer it (e.g. [3]) in their own context. However, we are unaware of any significant work which allows requirements such as timeliness, security and reliable data delivery to be addressed and traded off against each other within the same context. Similarly, much of the QoS work focuses on allocating a single time-shared resource such as network bandwidth. In real-time systems, applications may need to have simultaneous access to multiple resources such as processing cycles, memory, network bandwidth and disk bandwidth, in order to satisfy their needs.

In this paper, we propose the QoS-based Resource Allocation Model (Q-RAM) as an *initial* step in addressing both of these problems:

*This work was supported in part by the Defense Advanced Research Projects Agency under agreements E30602-97-2-0287 and N66001-97-C-8527, and in part by the Office of Naval Research under agreement N00014-92-J-1524.

- satisfying simultaneous requirements along multiple QoS dimensions such as timeliness, cryptography, data quality and reliable packet delivery, and
- having access to multiple resources simultaneously.

We present resource allocation schemes to solve only the first problem dealing with multiple QoS dimensions. Resource allocation schemes in the presence of multiple resources are the subject of ongoing work and are beyond the scope of this paper.

1.2 Related Work

A significant amount of work has been carried out for making resource allocations to satisfy specific application-level requirements. Such work can be classified into various categories. The problem of allocating appropriate resource capacity to achieve a specific level of QoS for an application has been studied in various contexts. For example, [3] studies the problem of how to allocate network packet processing capacity assuming bursty traffic and finite buffers. In [2], the problem of the establishment of real-time communication channels is studied as an admission control problem. The Spring Kernel [15] uses on-line admission control to guarantee essential tasks upon arrival.

Various system-wide schemes have been studied to arbitrate resource allocation among contending applications. In [16], a distributed pool of processors is used to guarantee timeliness for real-time applications using admission control and load-sharing techniques. The Rialto operating system [5] presents a modular OS approach, the goal of which is to maximize the user's perceived utility of the system, instead of maximizing the performance of any particular application. No theoretical basis is provided to maximize system utility. A QoS manager is used in the RT-Mach operating system to allocate resources to application, each of which can operate at any resource allocation point within minimum and maximum thresholds [7]. Applications are ranked according to their semantic importance, and different adjustment policies are used to obtain or negotiate a particular resource allocation.

Once a resource allocation decision has been made, various scheduling schemes are available to ensure that the allocation decisions can be carried out. A CPU resource reservation scheme [11] is used in RT-Mach to guarantee and enforce access to an allocated resource once a resource allocation decision has been made. A large portion of real-time scheduling theory deals with this problem and uses fixed priority schemes [9, 8, 13, 6], dynamic priority schemes [1, 4] or heuristic schemes [17]. The basic requirements of a QoS model in high assurance applications are presented in [18]. It proposes that the QoS attributes of timeliness, precision and accuracy can be used for system specification, instrumentation and evaluation.

The Q-RAM model we propose can be considered to be a generalization of at least two models previously studied in the literature. First, the *imprecise computation*

model proposed by Liu et al. [10] considered the problem of optimally allocating CPU cycles to applications which must satisfy minimum CPU requirements, but can produce better results with additional CPU cycles. The frequency of each application remains constant, while the computation time per instance of an application can be varied. The results were generally assumed to improve linearly with additional resources. Secondly, Seto et al. [14] have studied the problem of allocating CPU cycles optimally to feedback control applications whose control quality improves in concave fashion with higher frequencies of operation. The computation time per instance of an application remains constant.

Our proposed model can be viewed as a combination and broad generalization of these models. First, we allow either the computation time or the frequency of an application to vary. Secondly, and more importantly, we seek to generalize the resource allocation model to support multiple dimensions of quality (timeliness, data quality, reliable packet delivery, security achieved through cryptography, etc.) for each application to support the simultaneous allocation of multiple resource types (CPU and disk bandwidth, for example) for each application. The model in its general form only assumes that an application's quality will not decrease with any increase in resource allocation. We only deal with cryptographic security in this paper and the term 'security' will be used only in that sense.

The rest of this paper is organized as follows. In Section 2, we present our QoS-based Resource Allocation Model (Q-RAM) and illustrate the concepts behind the model using an actual video-conferencing system. In Section 3, we determine optimal resource allocation schemes for single variable QoS constraints. In Section 4, we identify the main considerations of multi-dimensional QoS problems and present optimal and greedy resource allocation for different cases. We also apply Q-RAM to the video-conferencing system and consider schedulability issues. In Section 5, we present our concluding remarks and discuss problems that remain unsolved.

2 Q-RAM: The QoS-based Resource Allocation Model

Q-RAM is based on a dynamic and adaptive application framework with the following characteristics:

- An application may need to satisfy many requirements: timeliness, security, data quality, dependability, etc.
- An application may require access to multiple resource types such as CPU, disk bandwidth, network bandwidth, memory, etc.
- An application requires a certain minimum resource allocation to perform acceptably. It may also improve its performance with larger resource allocations. This improvement in performance is measured by a utility function.

Q-RAM is a model in which resources can be allocated to individual applications with the goal of maximizing a global objective. The model is intended for use with static (off-line) allocation schemes, dynamic (on-line) allocation with admission control schemes, and 'timed' allocations where each resource allocation has a duration of validity associated with it.

2.1 Quality of Service Dimensions

Consider an application which obtains and transmits audio data. The application can use a reliable encoding scheme to tolerate and recover from bit errors during transmission. The data can be made secure by encrypting the transmitted packets. The application may process and transmit the audio data in smaller chunks to meet real-time constraints. The application may also choose to improve audio quality by increasing the size of each audio sample or by increasing its sampling rate. The application may also want to perform one or more of these simultaneously but each option requires the use of additional resources. We refer to these quality aspects such as timeliness, reliability, security and data quality as *QoS dimensions*.

In Q-RAM, we consider a system in which multiple applications, each with its own set of requirements along multiple QoS dimensions, are contending for resources.

- Each application may have a minimum and/or a maximum need along each dimension.
- Each resource allocation adds some utility to the application and the system, with utility monotonically increasing with resource allocation.
- System resources are limited so that the maximal demands of all applications often cannot be satisfied simultaneously.

With the Q-RAM specifications, a resource allocation decision will be made for each application such that an overall system-level objective (called utility) is maximized.

2.2 The Definition of Q-RAM

Q-RAM is defined as follows. The system consists of n applications $\{\tau_1, \tau_2, \dots, \tau_n\}$, $n \geq 1$, and m resources $\{R_1, R_2, \dots, R_m\}$, $m \geq 1$. Each resource R_j has a finite capacity and can be shared, either temporally or spatially. CPU and network bandwidth, for example, would be time-shared resources, while memory would be a spatially shared resource.

Let the portion of resource R_j allocated to application τ_i be denoted by $R_{i,j}$. We enforce $\sum_{i=1}^n R_{i,j} \leq R_j$. Two issues need to be noted in the context of real-time systems in particular:

- *Utilization*: The resource allocation to an application will be in terms of the utilization of a resource. Once a certain utilization has been allocated, an application may either choose its own execution time and period to achieve that utilization or use an appropriate processor-sharing scheme such as weighted fair-sharing.

- *Schedulability*: The constraint $\sum_{i=1}^n R_{i,j} \leq R_j$ implies that a resource can be "fully" consumed. As is well known, this is *not* always true for fixed-priority scheduling algorithms [9] but is true for the earliest deadline scheduling algorithm under ideal conditions. A different maximal resource constraint beyond the scope of this paper must be used to support fixed-priority schemes. For example, see [14].

We now introduce the following definitions:

- The *application utility*, U_i , of an application τ_i is defined to be the value that is accrued by the system when τ_i is allocated $R^i = (R_{i,1}, R_{i,2}, \dots, R_{i,m})$. In other words, $U_i = U_i(R^i)$. U_i is referred to as the *utility function* of τ_i . This utility function defines a surface along which the application can operate based on the resources allocated to it.
- Each application τ_i has a relative importance specified by a weight w_i , $1 \leq i \leq n$.
- The *total system utility* $U(R^1, \dots, R^n)$ is defined to be the sum of the weighted application utility of the applications, i.e. $U(R^1, \dots, R^n) = \sum_{i=1}^n w_i U_i(R^i)$.
- Each application τ_i needs to satisfy requirements along d QoS dimensions $\{Q_1, Q_2, \dots, Q_d\}$, $d \geq 1$.
- The *dimensional resource utility* $U_{i,k} = U_{i,k}(R^i)$ of an application τ_i is defined to be the value that is accrued by the system when τ_i is allocated R^i for use on QoS dimension Q_k , $1 \leq k \leq d$.
- ¹An application, τ_i , has *minimal resource requirements on QoS dimension Q_k* . These minimal requirements are denoted by $R_i^{\min k} = \{R_{i,1}^{\min k}, R_{i,2}^{\min k}, \dots, R_{i,m}^{\min k}\}$ where $R_{i,j}^{\min k} \geq 0$, $0 \leq j \leq m$.
- An application, τ_i , is said to be *feasible* if it is allocated a minimum set of resources on every QoS dimension. We denote the total minimum requirements by $R_i^{\min} = \{R_{i,1}^{\min}, R_{i,2}^{\min}, \dots, R_{i,m}^{\min}\}$ where $R_{i,j}^{\min} = \sum_{k=1}^d R_{i,j}^{\min k}$, $1 \leq j \leq m$.

In this paper, we assume that $m = 1$, i.e. only a single resource is being allocated.

2.3 Assumptions

We make the following assumptions:

- A1. The applications are independent of one another.
- A2. The available system resources are sufficient to meet the minimal resource requirements of each application on all QoS dimensions, $R_i^{\min}$, $1 \leq i \leq n$.
- A3. The utility functions U_i and $U_{i,k}$ are nondecreasing in each of their arguments. In some cases we will assume that these functions are concave and have two continuous derivatives.

¹This aspect of the model is a simplification to be relaxed in future work. In general, multiple resource-tuples can yield a given QoS operating point.

A4. Each application, τ_i , has a weight w_i denoting its relative importance.

We make the following observations concerning these assumptions. First, if assumption A1 does not hold, then the resource allocation methods still apply; however, the schedulability analysis needed to ensure that application timing requirements are met is more complicated. It must take into account phenomena such as the priority inversion that can occur with synchronization protocols.

Second, if assumption A2 does not hold, then the minimal resource requirements cannot be met. If these requirements are not met, then some of the applications must be dropped. One can use a variety of techniques to determine which of the applications should be dropped, or one could even allow some applications to have less than their minimal resource allocations. Although this is a very important issue, it is beyond the scope of this paper.

Third, in view of A4, we can now define a *weighted utility function* for an application as $w_i * U_i$ and then solve the resource allocation problem for those weighted utility functions. Thus, one can remove the weights from the allocation problem. In our subsequent analysis, we use these weighted utilities and drop the weight function.

Note that U_i is not necessarily equal to $\sum_{j=1}^m U_{i,j}$. In other words, the utility obtained by an application τ_i from a resource R_j may not be additive with respect to its utility from another resource. This is because the application may need two or more resources *simultaneously* to achieve a certain utility. For example, an audio-conferencing application may need the CPU resource and the networking bandwidth resource in order to satisfy even a minimal QoS requirement.

2.4 The Objective

The objective of Q-RAM is to make resource allocations to each application such that the total system utility is maximized under the constraint that *every* application is feasible with respect to each QoS dimension. Stated formally, we need to determine

$\{R_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m\}$ such that $R_{i,j} \geq \sum_{k=1}^d R_{i,j}^{min_k}$ and U is maximal among all such possible allocations.

2.5 QoS Considerations in Video-Conferencing

We shall use a video-conferencing system named *RT-Phone* [7] presented in Figure 1 as an example to illustrate Q-RAM. We shall focus primarily on managing resource allocations for the audio stream on a single node. The end-to-end delay encountered by an audio stream as a function of the CPU processing rate and the audio sampling rate is plotted in Figure 2-a. The variable on the x -axis is the periodic interval at which buffered audio packets are obtained from the sound hardware, processed and transmitted over the network. Each plotted line corresponds to a different sampling rate. For shorter

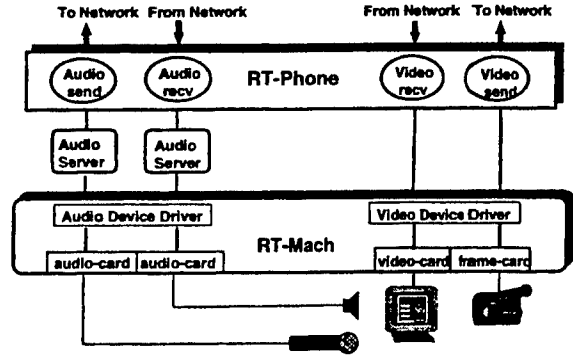


Figure 1: The Architecture of *RT-Phone*.

periods, the end-to-end delay is shorter and vice-versa. This plot also illustrates that with relatively little additional resources, the sampling rate can be increased and improved audio data quality can be obtained. The load imposed on the processor for the data points from Figure 2-a is plotted in Figure 2-b. The y -axis is now the CPU load; as can be seen, for shorter processing periods, the CPU load is high due to higher network packet processing costs (larger number of smaller packets) and higher context switching costs.

The QoS dimensions in this system (as described) are end-to-end delay representing timeliness and audio sampling rate representing data quality. The processing rate and the audio sampling rate can be changed independently of one another, and an increase in either leads to increases in utility of the video-conferencing system. Improvements in end-to-end delay from 200 ms to 50 ms generally tend to be perceived as much higher than improvements from 50 ms to 12.5 ms, i.e. the return on utility diminishes as more and more resources are added. The same applies to the sampling rate. The shapes of the utility functions² corresponding to these QoS dimensions are presented in Figure 2-c.

3 Resource Allocation in Q-RAM

In this section, we derive some basic properties of Q-RAM defined in the previous section.

We start with the simple case of making allocation decisions where there is only a single resource type and a single QoS dimension. We then extend this model to support multiple QoS dimensions. In each case, we state a property that needs to be satisfied for maximizing the total system utility, and/or present an algorithm which can find the optimal (or near-optimal) allocation.

3.1 A Single Resource and A Single QoS Dimension ($m = 1$ and $d = 1$)

Since there is a single resource and quality dimension, we can drop the subscripts associated with them. In this

²The utility assigned to an operating point for an application can be objective or subjective. In feedback control applications, control quality improves with sampling rates, and utility values can often be defined objectively. In multimedia applications, human perceptions saturate beyond a point, and utility values may be subjective. Relative utility values across applications would be based on system and application semantics, and will be, optionally, modifiable by the end-users.

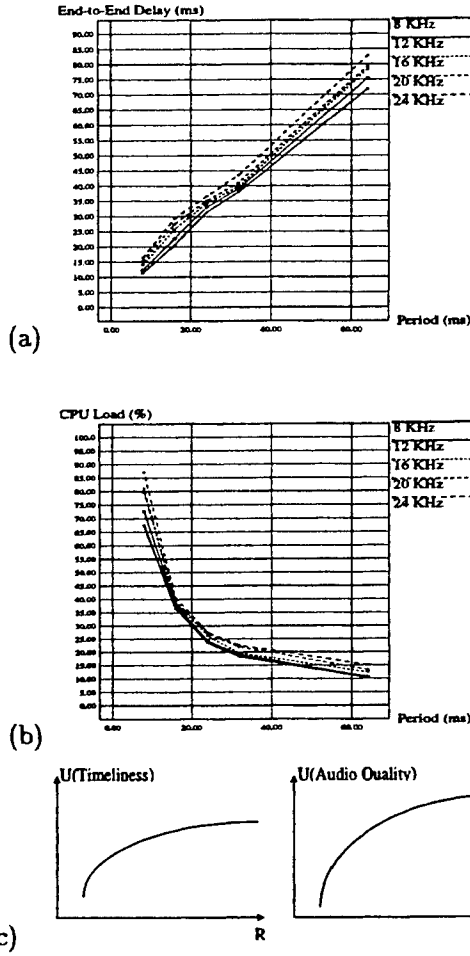


Figure 2: (a) End-to-end audio delay curves as a function of CPU processing rates and audio sampling rates. (b) The CPU load for audio processing as a function of CPU processing rates and audio sampling rates. (c) Audio utility functions for Timeliness and Data quality.

case, we have $U_i = U_i(R)$, $1 \leq i \leq n$, where R is the amount of resource allocated to τ_i . The minimum resource allocation needed to satisfy τ_i is $R_i^{\min}$.

To illustrate our approach, we make the further assumption that the utility functions $U_i = U(R)$ are twice continuously differentiable and concave, that is $\frac{d^2 U_i}{dR^2} = U_i'' \leq 0$ for $R > R_i^{\min}$. By convention, we assume $U_i(R) = 0$ for $0 \leq R \leq R_i^{\min}$.

It is very convenient to transform the resource allocation problem. Since we assume that all minimal application resource requests can be met, we can focus on the allocation of the excess resources available. Consequently, we can, without loss of generality, assume that $R_i^{\min} = 0$, $\forall i = 1$ to n and reduce the quantity of available resources by that amount. In our subsequent analysis, we assume that this transformation has been made and require only that $R_i \geq 0$ and $\sum_{i=1}^n R_i = R$, where R is the remaining quantity of resources left to allocate.

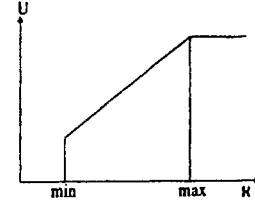


Figure 3: A linear utility function with min and max requirements.

The goal is to determine the values of $R_1, R_2, \dots, R_n$ such that the total system utility, $\sum_{i=1}^n U_i(R_i)$, is maximized subject to the constraint $\sum_{i=1}^n R_i \leq R$. The following theorem provides a necessary condition for an allocation to be optimal.

Theorem 1 A necessary condition for a resource allocation to be optimal is $\forall i, 1 \leq i \leq n, R_i = 0$ or for any $\{i, j\}$ with $R_i > 0$ and $R_j > 0$, $U_i'(R_i) = U_j'(R_j)$.

Proof: The result is a standard conclusion of the Kuhn Tucker theorem (see [12], chapter 5). To understand the intuition behind the results, suppose that for some $i \neq j$, let $R_i > 0$, $R_j > 0$ and $U_i'(R_i) > U_j'(R_j)$.

Since $R_j > 0$, an infinitesimal amount of R can be subtracted from application τ_j and added to application τ_i . Since $U_i'(R_i) > U_j'(R_j)$, the total system utility will increase. This contradicts the assumption that the allocation was optimal. $\square$

Remark: It should be noted that it is possible that all applications except one can receive zero resource allocations³, and this one application consumes all the available resource quantity since the slope of its utility function is the highest.

Remark: If the utility functions were not smooth, then this result requires some modification. To see this, suppose U_1 consists of two line segments with slope s_1 on $[0, L]$, then s_2 on $[L, \infty]$. Now suppose that U_2 is linear with slope s_3 with $s_1 > s_3 > s_2$, while all other utility functions have slopes which are less than s_2 . If the amount of resources available exceeds L , then the optimal allocation will be to give L to the first application, all the rest to the second application and none to any others. This results in a situation of unequal slopes. If, on the other hand, the utility functions are smooth, this cannot happen.

3.1.1 A Special Case of Linear Utility Functions

As a special case, consider the utility curve of Figure 3. The utility curve is linear from $R_i^{\min}$ to a maximum resource requirement $R_i^{\max}$ beyond which it becomes flat. This utility curve is practical in the sense that many non-critical systems such as desktop multimedia applications can gain from its simplicity and resulting efficiencies. We refer to this special class of utility functions as *min-linear-max* functions. The following corollary provides a necessary condition for a resource allocation to be optimal for

³Recall that the resource allocations are normalized and that each application has already been allocated sufficient resources to satisfy its minimum requirement.

min-linear-max utility functions.

Corollary 1 *A necessary condition for a resource allocation to be optimal for min-linear-max utility functions is $\forall i, 1 \leq i \leq n, R_i = 0$, or $R_i = R_i^{max}$, or for any $\{i, j\}$ with $0 < R_i < R_i^{max}$ and $0 < R_j < R_j^{max}$, $U'_i(R_i) = U'_j(R_j)$.*

Proof: The corollary follows from the conditions of Theorem 1, and from the fact that no utility is gained by allocating even an infinitesimal resource to an application τ_i beyond R_i^{max} . $\square$

Remark: It is possible that there exists only one application τ_i which has $0 < R_i < R_i^{max}$, i.e. i, j can refer to the same application in the statement of Corollary 1.

3.1.2 An Algorithm to Determine U_{max}

An algorithm to determine the optimal resource allocation R_i for each application to obtain U_{max} is given below. We assume that each application has already been allocated its minimum resource requirement. By assumption A3, sufficient resources should be available for this allocation. Consequently, we determine the optimal additional allocation to each application, $R_i \geq 0, 1 \leq i \leq n$, subject to $\sum_{i=1}^n R_i \leq R$.

1. Let the current normalized allocation of the resource to τ_i be $R_i, 1 \leq i \leq n$. Let the unallocated quantity of the available resource be R' . Compute $(U'_1(R_1), \dots, U'_n(R_n))$.
2. Identify (a) the subcollection of applications with largest value of $U'_i(R_i)$, (b) the number of applications in that subcollection (denoted by k), and (c) the application (denoted by j) with the second largest value of this quantity if any such application exists. If the largest value of $U'_i(R_i)$ is 0, then stop. No further allocation will increase system utility and spare resources are available.
3. Increase R_i for each of the members of the subcollection so that their values of $U'_i(R_i)$ decrease but continue to be equal until either (i) this value becomes equal to the second largest value or (ii) the additional resources added to this subcollection equal R' . In case (ii), stop as all resources have been optimally allocated.
4. In case (i), one or more new applications should be added to the subcollection. Return to step 1.

4 A Single Resource and Multiple QoS Dimensions ($m = 1$ and $d > 1$)

An application can have multiple QoS dimensions (i.e. $d > 1$). For example, the RT-Phone example has two QoS dimensions, audio data quality (which increases with audio sampling rate) and end-to-end delay (which decreases with increases in processing rate). The resource allocation for systems with multiple quality dimensions depends upon the nature of the relationship between the dimensions themselves. In this section, we classify the relationships between QoS dimensions, discuss their effects and

study the resource allocation problem under various conditions. We provide optimal allocations when possible, and provide a greedy algorithm in another case.

4.1 Relationships between QoS Dimensions

The inter-relationship between QoS dimensions directly impacts the nature of the utility functions. We study two kinds of relationships among QoS dimensions:

Independent dimensions: Two QoS dimensions, Q_a and Q_b , are said to be independent of one another if a quality increase along Q_a (Q_b) does *not* increase the resource demands to achieve the quality level previously achieved along Q_b (Q_a). An example is using different compression schemes on an audio stream but each scheme generates the exact same amount of data. As a result, the processing resources needed to encrypt the data remain the same. If the encryption scheme is changed to consume more resources, the audio compression demands would remain the same. Therefore, security and audio data quality can be considered to be independent QoS dimensions in this system.

Dependent dimensions: A QoS dimension, Q_a , is said to be dependent on another dimension, Q_b , if a change along the dimension Q_b will increase the resource demands to achieve the quality level previously achieved along Q_a . In the RT-Phone system, if the audio sampling rate is increased, the data volume increases and the CPU time needed to process the data increases⁴.

Remark: Two QoS dimensions Q_a and Q_b can both be dependent on a third dimension Q_c . For example, if video quality is improved by increasing the size of the image, both processing capacity and network bandwidth demands would increase. As a result, both timeliness and packet loss QoS dimensions would be affected.

4.2 Dealing with Independent QoS Dimensions

Suppose that the d QoS dimensions are independent of one another. In this case, each QoS dimension offers its own utility to the system and can be varied independent of the other dimensions. In this case, the dimensional utilities of the applications are additive. That is, $U_i = \sum_{k=1}^d U_{i,k}$. The resource allocation problem then is equivalent to the single QoS dimension problem of Section 3.1 with $n * d$ applications $\{\tau'_1, \tau'_2, \dots, \tau'_{n*d}\}$, where $\{\tau'_1, \tau'_2, \dots, \tau'_d\}$ correspond to the d dimensions of τ_1 , $\{\tau'_{d+1}, \tau'_{d+2}, \dots, \tau'_{2d}\}$ correspond to the d dimensions of τ_2 and so on. The optimal resource allocations can now be determined using the algorithm described in Section 3.1.2.

4.3 Dependent QoS Dimensions with Continuous Values

Suppose that one or more QoS dimensions are independent, but the quality on each dimension can be any value within an interval. We now illustrate the general approach using the special case $d = 2$, but the approach

⁴This increase in CPU load is not linear, however, as can be inferred from Figure 2-b.

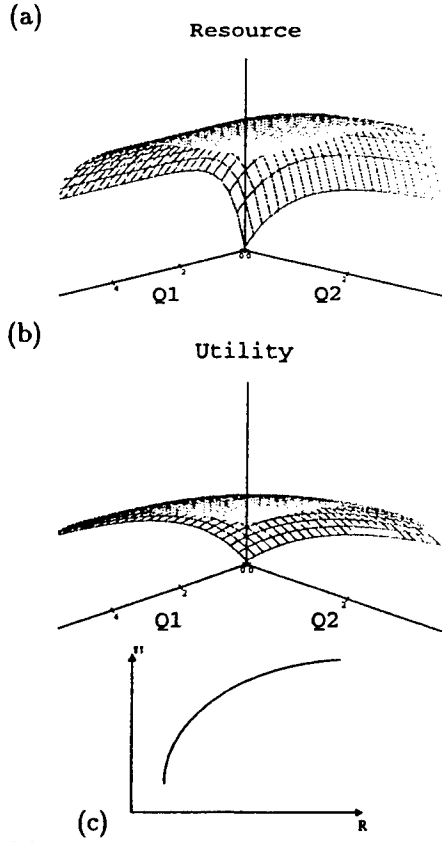


Figure 4: (a) The Resource Consumption Surface for Two QoS dimensions. (b) The Utility Surface as a function of two QoS Dimensions. (c) The final (univariate) utility function for two QoS dimensions and a single resource.

remains the same for $d > 2$. The resource demand for every point p along the QoS dimension Q_1 and every point q along the QoS dimension Q_2 is plotted first. This defines a resource consumption surface along Q_1 and Q_2 , an example of which is provided in Figure 4-a. The utility to the system for any pair $\{p, q\}$ of points along QoS dimensions Q_1 and Q_2 respectively is plotted next. This yields a utility surface, an example of which is illustrated in Figure 4-b. The contours of $R = k$ from the resource consumption surface are then projected to the utility surface. The maximum utility values for each $R = k$ contour projection finally yield a single (maximal) utility function as a function of R . For example, the utility function from the surfaces of Figures 4-a and 4-b yield the shape shown in Figure 4-c. The net result is that the multi-dimensional resource allocation problem gets reduced to the single-QoS dimension problem.

If the resulting univariate utility function is twice continuously differentiable and concave (or min-linear-max), the algorithm of Section 3.1.2 can be applied to obtain the optimal resource allocation.

4.4 Dependent QoS Dimensions With Discrete Options

We now consider special cases, where one of the QoS dimensions is not only dependent on another (base) QoS

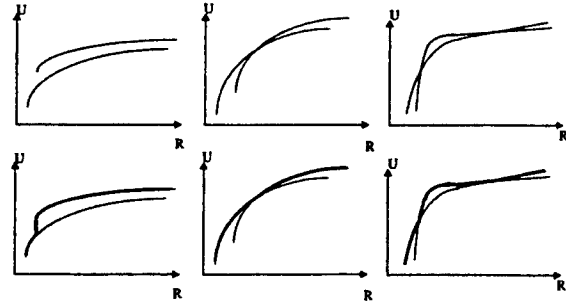


Figure 5: Three sets of U_r , U_e and an aggregate U_a for a dependent binary QoS Dimension.

dimension, but is also discrete in nature. For the sake of illustration, we shall assume that $d = 2$, yielding one independent (base) QoS dimension and one QoS dimension dependent on the former. We shall first consider the binary case where the quality of the dependent dimension is either available or not available. We then consider the case where the quality along the dependent dimension can be any one of multiple discrete values.

4.4.1 Using Dependent Binary QoS Dimensions

Consider again the application sampling microphone input and transmitting an audio stream. In the following, we use the suffixes a , r and e to represent audio, raw audio and encrypted audio respectively. Increasing the audio sampling rate increases the audio quality and the amount of data to be processed. Let R_r be the CPU resource allocated to the processing of this raw data. We have $R_r = g(\text{SamplingRate})$. Assuming that $g(\cdot)$ is monotonic, $U_r = f(R_r)$ has the same shape as Figure 4-c.

Suppose that the audio data will also be encrypted. Now, additional processing per block of audio data (and correspondingly the CPU resource) will be needed. This additional resource consumption scales linearly with the sampling rate. It is reasonable to assume that a constant utility gain Δ is added to the system with encryption. However, since R_r is needed for processing the audio data without encryption, a *larger* value R_e would need to be allocated for encrypting and processing the same amount of audio data. We therefore have $R_e = \Gamma_e * R_r$, where Γ_e is a constant > 1.0 .

The utility function U_e therefore has the form $f(\frac{R_r}{\Gamma_e}) + \Delta$. Therefore, the origin of U_e is offset both vertically and horizontally from U_r . The vertical offset is Δ , and the horizontal offset is $(\Gamma_e - 1) * R_r^{\min}$. Since $\Gamma_e > 1$, the slope of U_e is always smaller than that of U_r . If U_r is continuous and concave, U_e is also continuous and concave.

The aggregate utility function for the audio application is given by $U_a = \max(U_r, U_e)$. Three examples are provided in Figure 5; U_r is the thin line starting lower and more to the left, U_e is the other thin line, and U_a is the bold line. It must be noted that encryption is not possible for resource allocations less than $(R_r^{\min} * \Gamma_e)$. For larger

allocations, encryption is possible but it may or may not yield higher utility. For example, the aggregated application utility function of Figure 5-c yields a higher utility without encryption initially, then with encryption, and then without encryption again. Also, in the general case, U_a will only be piecewise continuous and concave.

4.4.2 Using Dependent 'n-ary' QoS Dimensions

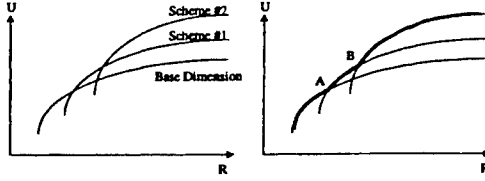


Figure 6: Example of using an n -ary QoS Dimension.

In Section 4.4.1, we assume that a second QoS dimension (namely encryption) is applied as a binary function: it is either available or not available. Such a binary scheme is applicable when only a single scheme (such as encrypting always using a 48-bit public key) is used. However, suppose that more than one scheme is available along this QoS dimension (such as encrypting using a 64-bit key, a 128-bits key, etc. or using a different cryptographic scheme). For convenience, we introduce the following notation.

Notation: Let the QoS dimension Q_k be dependent on another and have $s + 1$ discrete schemes. By convention, we adopt scheme 0 to represent the absence of the dimension. The utility gain constant provided by supporting scheme p , $0 \leq p \leq s$ is denoted by $\Delta_{k,p}$. The resource scaling factor (with respect to the base dimension) for supporting scheme p , $0 \leq p \leq s$ is denoted by $\Gamma_{k,p} \geq 1.0$. The overhead factor of scheme p , $0 \leq p \leq s$ is denoted by $\gamma_{k,p} = (1.0 - \Gamma_{k,p}) \geq 0.0$. By convention, we have $\Delta_{k,0} = 0$, $\Gamma_{k,0} = 1.0$, $\gamma_{k,0} = 0.0$.

Each scheme p , $0 \leq p \leq s$, provides a correspondingly different increase in utility, $\Delta_{k,p}$, while also consuming a different amount of the CPU resource with a different $\Gamma_{k,p}$. The result is a family of utility curves, and the aggregated application utility function is the maximum of these curves. An example family of utility functions for a 3-ary dependent dimension and the resulting aggregated utility function are illustrated in Figure 6.

4.4.3 Linear Dimensional Utility Functions in the Dependent 'n-ary' Case

When min-linear-max dimensional utility functions are used for the independent QoS dimensions, the aggregated application utility function is piecewise linear when one of the QoS dimensions is 'n'-ary in nature. A sample set of the individual dimensional utility functions for two QoS dimensions, one independent and another dependent, and their aggregated application utility function are illustrated in Figure 7. Let the independent (base) dimension be Q_1 and the dependent dimension be Q_2 . We have $R_{min} = 0.1$, $R_{max} = 0.2$ for Q_1 (same as scheme 0 for

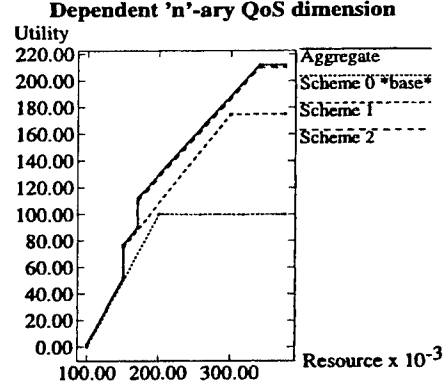


Figure 7: One 3-ary QoS dimension w/ min-linear-max.

Q_2). The corresponding utilities are given as 0 and 100 respectively. For Scheme 1 of the dependent dimension Q_2 , we assume $\Delta_{2,1} = 75$, $\Gamma_1 = 1.5$ yielding $\gamma_1 = 0.5$. Correspondingly, $R_{min} = 0.15$, $R_{max} = 0.3$, and the utilities are given by $(0 + 75 = 75)$ and $(100 + 75 = 175)$. For Scheme 2, assume $\Delta_{2,2} = 110$, $\Gamma_{2,2} = 1.7$, $\gamma_{2,2} = 0.7$. We therefore have $R_{min} = 0.17$, $R_{max} = 0.34$, $U_{min} = 0 + 110 = 110$ and $U_{max} = 100 + 110 = 210$. The aggregate utility function for the application is the maximum of the previous three functions.

It is useful to note that this piecewise-linear application utility function has 3 kinds of discontinuities: *intersecting discontinuities* where dimensional utility lines intersect (point A in Figure 6), *vertical discontinuities* where the maximal dimensional utility line at a point starts above the other lines (at $R = 0.15$ and 0.17 in Figure 7), and *saturation discontinuities* where the maximum QoS point for a dimensional utility line beyond which the utility does not increase (at $R = 0.34$ in Figure 7). We now present a greedy algorithm which determines a near-optimal resource allocation under these conditions.

A greedy algorithm to obtain a good resource allocation R_i for each application in a system with all linear dimensional utility functions is as follows:

1. Assign to each application τ_i its minimum resource requirement R_i^{min} . By assumption A3, sufficient resources should be available for this allocation.
2. Normalize the utility function of each application (by left-shifting and down-shifting the utility curve such that it starts at the origin). Let the total quantity of available resource remaining be R .
3. Let the current normalized allocation of the resource to τ_i be R_i . Let the unallocated quantity of the available resource be R^l .
4. For each application τ_i , $1 \leq i \leq n$, compute the slopes on the application utility curve at R_i , and between the current allocation R_i and any and every discontinuity⁵ in the region $R_i < R \leq R^l$. Let

⁵When there is a vertical discontinuity, pick the higher point.

this set of slopes for τ_i be represented by $\{s_i\}$. The size of this set of slopes is at least one.

5. Let the index of the application with the highest value of the element in $\{s_i\}$, $1 \leq i \leq n$, be p . If there are two or more such applications, pick one at random. Let this largest slope element of τ_p be s_p^{max} . Let the *additional* resource amount that needs to be allocated to τ_p to reach the discontinuity point corresponding to s_p^{max} be r .
6. If $s_p^{max} = 0$, stop. The unallocated resources will *not* increase system utility any further.
7. Allocate an additional (r) to τ_p increasing R_p by that amount. Reduce R^l by this same amount.
8. If $R^l = 0$, stop. Else, go to step 4.

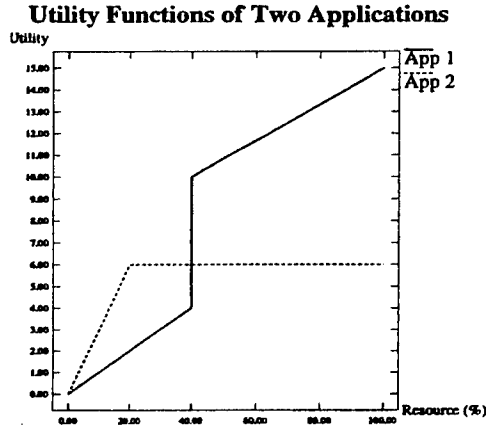


Figure 8: A counter-example of non-optimality of the greedy algorithm of Section 4.4.3).

Remark: We note that the above algorithm, while expected to do well in practice, does *not* always lead to an optimal resource allocation⁶. We now provide a counter-example illustrated in Figure 8 to show that this is indeed the case. Suppose that there are only two applications τ_1 and τ_2 . Let the total amount of resource to be allocated be 0.5. $U_1(0)$ has a linear slope of 10, but a vertical discontinuity occurs at $R_1 = 0.4$ with $U_1(0.4) = 10$. The slope at $U_1(0.4)$ continues to be 10. But, the slope of $U_1(0)$ from 0 to the higher point of the vertical discontinuity, which is at 10, is 25. $U_2(0)$ has a linear slope of 30 and U_2 has a saturation discontinuity at $R_2 = 0.2$ (with $U_2(0.2) = 6$). The above greedy algorithm will first allocate 0.2 units of the resource to τ_2 (since its slope is the highest at a value of 30). The remaining 0.3 units will then be allocated to τ_1 yielding $U_1(0.3) = 3$. The total system utility achieved is therefore $(6+3)=9$. However, an optimal algorithm would allocate $R_1 = 0.4$ and $R_2 = 0.1$ yielding a total system utility of $(10+3) = 13$, which is higher than the utility achieved by the greedy algorithm.

⁶ We are currently developing an algorithm that will find an optimal allocation and replace this greedy (sub-optimal) algorithm.

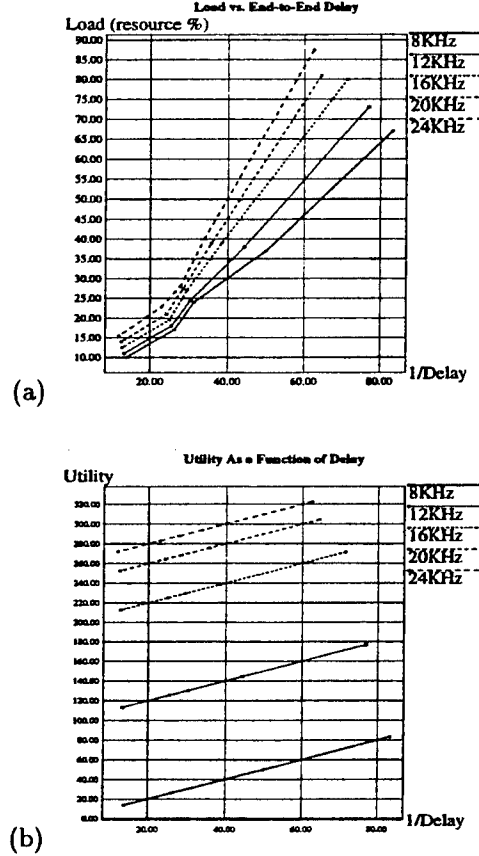


Figure 9: (a) The resource consumption function for RT-Phone. (b) Utility as a function of Timeliness (with a linear model of the value of timeliness).

4.5 Using Q-RAM in the RT-Phone Example

In this section, we apply Q-RAM to the RT-Phone system for the sake of illustration and also show how the real-time constraints can be satisfied. We first generate the resource consumption surface for the QoS dimensions end-to-end delay (represented as $1/\text{delay}$) and audio quality (represented as sampling rate). From Figures 2-a and 2-b, we obtain the surface in Figure 9.a. We now assume that the utility of the timeliness QoS dimension is given by $(1/\text{delay})$. Let us now suppose that the audio quality dimension offers a constant utility *gain* at each sampling rate.

Using $U_g(\cdot)$ to represent the utility gain from a particular sampling rate, let us assume that $U_g(8\text{KHz}) = 0$, $U_g(12\text{KHz}) = 100$, $U_g(16\text{KHz}) = 200$, $U_g(20\text{KHz}) = 240$, $U_g(24\text{KHz}) = 260$ (yielding a tapering-off effect). The total utility at a given sampling rate is given by $U_{\text{delay}} + U_g$. The variation of total utility with end-to-end delay is plotted in Figure 9.b. From the curves of Figure 9, we obtain the (univariate) utility function of Figure 10.

4.5.1 Resource Allocation and Schedulability

Suppose the CPU resource has to be allocated among 10 applications with utility curves similar to those of Figure 10. First, all 10 applications will be allocated their min-

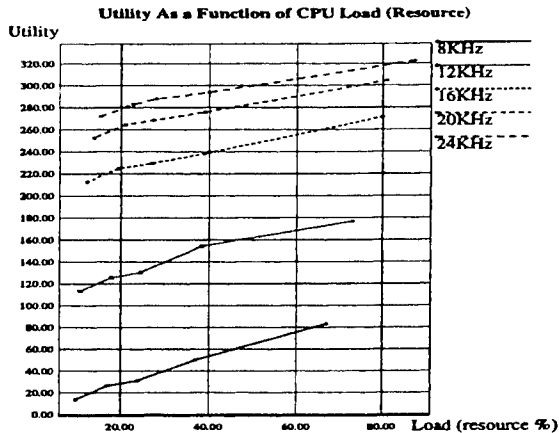


Figure 10: The RT-Phone utility function recommends the use of a 24KHz sampling rate for audio.

imum resource requirements. Next, additional resource allocations will be made only to the application with the highest utility slope. If any CPU cycles remain after that application reaches its maximum requirement, only then would they be allocated to the application with the next higher slope. This is repeated until no more CPU cycles are available. Let the final CPU allocation to application i be R_i . Its corresponding CPU load and processing rate from Figures 2-a and 2-b yield the (computation time, period) pair. These pairs can then be scheduled using the earliest deadline scheduling algorithm.

5 Concluding Remarks

We have presented a QoS-based Resource Allocation Model (Q-RAM) that allows the utility derived from a system to be maximized by making resource allocations such that the different needs of concurrently running applications are satisfied. Each application has a minimal resource requirement, but can adapt its behavior if given more resources and provide additional utility. Each application also needs to satisfy QoS metrics along multiple dimensions such as timeliness, cryptographic security, reliable packet delivery and data quality. Finally, each application may need to obtain access to multiple resource types in order to meet its QoS constraints. We have provided optimal (or near-optimal) resource allocation schemes for applications which need a single resource, but need to satisfy one or more QoS dimensions. A video-conferencing system with timeliness, audio quality and encryption constraints is used as an example to motivate and apply Q-RAM.

We are pursuing several avenues as future work. First, optimal schemes are needed for applications with multiple QoS dimensions (see Section 4.4.3) and for allocation of multiple resources. Second, the underlying OS/kernel for Q-RAM must not only support flexible resource management schemes but also provide feedback to the Q-RAM

manager about available resources and resource consumption by various application threads. The run-time overhead for these actions are also yet to be studied in detail. Finally, Q-RAM is based on single-node systems and needs to be extended to distributed systems.

Acknowledgments

The authors would like to thank other Amaranth project members at Carnegie Mellon University, including Carol Hoover, Pradeep Khosla, Phil Koopman and Lui Sha. The Amaranth project is defining a comprehensive framework for QoS management along multiple quality dimensions, and its goals include the construction of system prototypes and applications. The authors would also like to thank John Wilkes of Hewlett Packard and Tom Lawrence of Rome Air Force Laboratories for insightful discussions on QoS-based resource allocation.

References

- [1] T. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1):67-100, March 1991.
- [2] K. G. Shin D. D. Kandlur and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 1044-1056, Oct 1994.
- [3] R. Guérin, H. Ahmadi, and M. Naghshineh. Equivalent capacity and its application to bandwidth allocation in high-speed networks. *IEEE Journal on Selected Areas in Communications*, September 1991.
- [4] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. Technical report, TR90-038, Department of Computer Science, University of North Carolina at Chapel Hill, November 1989.
- [5] 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.
- [6] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9361-9.
- [7] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *the proceedings of Multimedia Japan 96*, April 1996.
- [8] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhancing aperiodic responsiveness in a hard real-time environment. *IEEE Real-Time System Symposium*, 1987.
- [9] C. L. Liu and Layland J. W. Scheduling algorithms for multiprogramming in a hard real time environment. *JACM*, 20 (1):46 - 61, 1973.
- [10] J. W. S. Liu, K-J Lin, R. Bettati, D. Hull, and A. Yu. *Use of Imprecise Computation to Enhance Dependability of Real-Time Systems*. Kluwer Academic Publishers, 1994.
- [11] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.
- [12] A. L. Peressini, R. E. Sullivan, and Jr. J. J. Uhl. *Convex Programming and the Karish-Kuhn-Tucker conditions*, chapter 5. Springer-Verlag, 1980.
- [13] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.
- [14] D. Seto, J. P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. *IEEE Real-Time System Symposium*, December 1996.
- [15] J. A. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of the Real-Time Systems Symposium*, Dec 1987.
- [16] E. M. Atkins T. F. Abdelzaher and Kang Shin. Qos negotiation in real-time systems and its application to automated flight control. In *The Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1997.
- [17] W. Zhao, K. Ramamritham, and J. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, Aug. 1987.
- [18] T. F. Lawrence. The Quality of Service Model and High Assurance. *Workshop on High Assurance Systems*, July 1997.

Predictable Communication Protocol Processing in Real-Time Mach

Chen Lee, Katsuhiko Yoshida*, Cliff Mercer and Ragunathan Rajkumar

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

{clee,ky2d,cwm,raj+}@cs.cmu.edu

*Visiting Scientist, Nippon Steel Corporation

Abstract

Scheduling of many different kinds of activities takes place in distributed real-time and multimedia systems. It includes scheduling of computations, window services, filesystem management, I/O services and communication protocol processing. In this paper, we investigate the problem of scheduling communication protocol processing in real-time systems. Communication protocol processing takes a relatively substantial amount of time and if not structured correctly, unpredictable priority inversion and undesirable timing behavior can result to applications communicating with other processors but are otherwise scheduled correctly. We describe the protocol processing architecture in the RT-Mach operating system, which allows the timing of protocol processing to be under strict application control. An added benefit is also obtained in the form of higher performance. This scheduling architecture is consistent with the other RT-Mach scheduling mechanisms including fixed priority scheduling and processor reservation. The benefits of this protocol architecture are demonstrated both under synthetic workloads and in a realistic distributed videoconferencing system we have implemented in RT-Mach. End-to-end delays for both audio and video are as predicted even with other threads competing for the CPU and the network.¹

1. Introduction

Distributed real-time and multimedia applications must communicate and coordinate across machine boundaries. Such communications may use a wide range of network communication protocols including UDP/IP, TCP/IP and XTP. Despite the advent of high-bandwidth networks like ATM, Fast Ethernet, Ethernet switching etc., network bandwidth is often considered to be the most serious bottleneck for such network communications. This is certainly true when a large number of nodes use the same network link(s) and each node has to have a chunk of the communication bandwidth. On the other hand, protocol stacks such as UDP/IP and XTP/IP also consume a considerable amount of CPU processing time. When multiple real-time

tasks need to use the network from the same node, as is often the case in distributed real-time and multimedia contexts, the question of how these protocol stacks are structured and processed becomes a critical question for maintaining predictable timing behavior. Some specific questions that arise are:

- *Sender-Related Questions:* If two or more real-time tasks try to send out network packets, what is the level of resource-sharing involved? In general, how are sending of packets by different tasks scheduled?
- *Receiver-Related Questions:* If one or more real-time tasks receive packets from the network, are they processed in FIFO or priority order? In general, how are their protocol processing activities scheduled on network packet reception?
- *Network-Related Questions:* How is network bandwidth allocated and managed?

Most, if not all, commercial protocol stack implementations use a FIFO queueing mechanism. This is clearly detrimental to real-time behavior particularly when extensive support has been added and used to schedule the real-time computations on the CPU. In addition, protocol stacks are often implemented in the kernel, leading to large critical sections when networks packets arrive or depart. Finally, packet arrivals are processed with high (kernel) priority even if the packets are intended for low priority tasks to ensure that as few packets as possible are lost.

In this paper, we address the sender-related and receiver-related questions by defining the requirements of a real-time protocol processing architecture that is "aware" of real-time requirements of tasks sending and receiving network packets. An implementation of an architecture that meets these requirements has been carried out on RT-Mach. We discuss this implementation and evaluate its real-time characteristics under synthetic workloads as well as in the context of a video-conferencing system built on Real-Time Mach.

1.1. An Overview of Real-Time Mach

We now provide a brief overview of the capabilities of Real-Time Mach so as to provide some insight into how the various components of the operating environment fit together with the protocol processing architecture.

¹This work was supported in part by the Office of Naval Research, Naval Research and Development Center, Northrop-Grumman, Philips Labs and Nippon Steel Corporation.

The RT-Mach microkernel supports a wide range of CPU scheduling policies including a fixed priority scheduling policy, earliest deadline first policy and a round-robin policy. One of these policies can be chosen dynamically. RT-Mach also supports a novel scheduling scheme based on processor reservation which serves as a temporal protection barrier between real-time tasks analogous to address space protection between processes [9]. Each *processor reserve* comprises of a requested rate of usage, currently specified as C units of computation time every T units of time. Transparent to user applications, a reserve is assigned by the kernel a rate-monotonic priority based upon this requested usage, and the processor is still scheduled on the basis of fixed priorities². The reservation scheme includes an admission control policy to prevent overload and a mechanism to accurately measure computation time consumed by programs. In addition to measuring computation time usage, the reservation mechanism *enforces* computation time limits reserved by an application thread. Hence, a program which attempts to use more computation time than its processor allocation cannot interfere with the timing behavior of other programs. This is in contrast to pure priority-driven scheduling policies where overruns by higher priority processes can hurt lower priority processes.

In addition to its flexible and novel scheduling policies, RT-Mach supports a real-time inter-process communication mechanism based on priority inheritance (for priority-driven scheduling policies) and reservation propagation (for the reservation-driven scheduling policy). Virtual memory pages (including code, data and/or future allocation) of real-time tasks can be wired down to obtain predictable memory accesses. High-resolution clocks and timers with a resolution of up to 250 ns are supported. An X11-server which supports reserve propagation and shared memory communication is also available. Simpler applications can use a display screen library to access the display frame buffers. A real-time shell (RTS) along with a network protocol server (NPS) provide a compact run-time environment for constructing distributed real-time systems. Video and audio capabilities are also supported to aid in the development of distributed multimedia applications. A complete 4.3 BSD-based environment is available for program development. In addition, a 4.4 BSD-Lites server has been ported to the RT-Mach microkernel by the Helsinki University of Technology.

In this paper, we use both the fixed priority scheduling policy (due to its popular use and support by current standards such as POSIX and Ada95) and the RT-Mach processor reservation policy (due to its better enforcement and abstraction properties) in conjunction with the protocol processing structure.

²This can be easily extended to dynamic priority models such as earliest deadline scheduling due to the transparent nature of the reserve interface seen by applications.

1.2. Organization of the Paper

The rest of this paper is organized as follows. Section 2 discusses some choices for different protocol processing software structures and how they impact the timing behavior of applications. In Section 3, we give a more detailed description of the scheduling structure that we have implemented in RT-Mach, focusing on the features of this mechanism that enable application-level timing control over packet scheduling. In Section 4, we present performance numbers from synthetic workloads which demonstrate the predictable behavior we can achieve. This evaluation focuses on the use of the RT-Mach processor reservation scheme with the real-time protocol processing architecture. In Section 5, we describe a practical 2-way video-conferencing system which transmits duplex audio and video streams. This application has both heavy CPU processing, stringent protocol processing and end-to-end delay requirements, and is an ideal testbed for testing the protocol processing structure described in Sections 2 and 3. The evaluation of this section focuses on fixed-priority scheduling alone. In Section 6, we present our concluding remarks.

2. Real-Time Processing of Communication Protocols

In this section, we look at several different approaches to protocol processing software design, and we identify and discuss the advantages and disadvantages of these approaches.

Most implementations of protocol stacks use a FIFO queueing scheme to process network packets. Hence, even if the processes and threads are scheduled according to real-time scheduling principles, priority inversion exists in the protocol stack. Preemptability is typically very limited as well since many protocol stack implementations are in the kernel and therefore execute at kernel priorities. By applying the known principles of real-time scheduling, protocol processing can be structured in various ways:

1. **Prioritized Processing:** This represents a deceptively simple change and requires only changing the queues from FIFO into priority-based ones. However, this can cause problems on both the sending side and the receiving side. The software structure used for protocol processing in the operating system determines the degree of priority inversion and thus the level of predictability. At one extreme, the 4.3 BSD operating system uses "software interrupt" processing for executing protocols for incoming network packets [6]. This gives protocol processing higher priority than *any* schedulable activity in the system, higher than any system or user processes. Thus, packet protocol processing acts as a kernelized monitor. For fast response to network packets and for high throughput, this is a good design choice, but the problem is that a deluge of low priority data packets can effectively take over the processor for an extended period of time, regardless of the importance of any of the schedulable activities. The system is thus vulnerable to unbounded priority inversion. Sending of large packets by lower priority threads will be processed at kernel priorities causing

problems but to a lesser degree since the maximum number of lower priority sends (and hence blocking) will be limited to a single send.

2. **Shared Communication Protocol Server:** One reasonable alternative is to bring the protocol stack into a separate server (particularly in a microkernel architecture). We can then treat the protocol stack as a shared resource, and then apply the priority inheritance protocol or priority ceiling protocol to it. Problems similar as in approach (1) are possible but to a lesser degree since the priority of the server is under application control.
3. **Processing Using Prioritized Threads:** To prevent the kind of priority inversion from approach (1), it is necessary to associate priorities with packets so that they can be queued and serviced in priority order. This enables preemption of the processing of one low priority packet in favor of a higher priority packet, especially if the computation time required for protocol processing is significantly more than that required for a (thread) context switch. One approach, used in the ARTS real-time kernel, has preemptible threads to shepherd packets through the protocol software [14]. Each thread handles a different packet priority class, and the priority of the thread matched the priority of the packets it handles. For predictable performance, the protocol processing software should be sensitive to packet priority as well as the priority of other activities running on the processor. This approach provides fast response to high priority packets and prevents low priority network activities from interfering with high priority work on the processor. This is similar to the method used in the x-kernel [2], but unlike the x-kernel threads, ARTS protocol processing threads are preemptive.
4. **Application-Level Protocol Processing:** A fourth alternative that we actually chose for use in RT-Mach is to make the protocol stack into a library that resides in application space (in each process). In such a design, individual threads can still preempt one another based on their priorities. As a result, communication protocol processing becomes a local extension of the communicating threads and can be treated as fully preemptive blocks of computation across processes. In a microkernel setting as in RT-Mach, the protocol stack actually can move from the Unix server (which runs as a privileged process on top of the microkernel) to the application level, and additional performance benefits can be accrued since the path is now {kernel to application process} instead of {kernel to Unix to application process}.

2.1. Application-Level Protocol Processing

Coordination between processor scheduling and network packet handling is very important for end-to-end predictability in distributed multimedia systems. Many systems use the notion of priority to support predictability, and one major issue is how *priority inversions* affect the performance of more important activities. Priority inversion occurs when a higher priority activity is forced to wait for a lower priority activity to execute [13, 11]. For example, a priority inversion occurs when a high priority packet goes into a FIFO queue behind a low priority packet. Priority inversion can be a major cause of unpredictable behavior in real-time communication systems [15].

Several principles guide the design of predictable protocol processing software [8]:

1. use packet priority for queueing,
2. schedule protocol processing against other system activities using packet priority,
3. use a preemptive control structure to reduce interference and priority inversion,
4. partition resources such as protocol data structures to reduce interference among priority classes, and
5. limit the context switching overhead of the preemptive control structure.

The multi-threaded protocol software mentioned above enhances the predictability of protocol processing, but at the expense of additional context switching. A protocol processing mechanism implemented for the Mach operating system [7] is amenable to the application of these principles. This user-level library implementation of TCP/IP and UDP/IP was originally done to speed up the fast path in the Mach networking code by reducing the number of IPC's and context switches required to send and receive packets. This design also happens to satisfy our principles for predictable network communication, and with the resource management functionality provided by our reservation mechanism, we achieve predictable end-to-end performance.

3. A Protocol Software Structure for Predictable Real-Time Scheduling

3.1. OS Enforcement and Predictability

To support a predictable communications service, the operating system must cooperate with the network in scheduling networking activities. Two common approaches to building predictable systems are (1) relatively static real-time scheduling for guaranteed service and (2) statistical multiplexing techniques for (mostly) good service and high utilization. Static real-time scheduling approaches typically use priority-driven policies with off-line priority assignments and analyses. They are often based on careful measurement and control of the execution times of each software component in the system. Such approaches are less appropriate for the dynamic, flexible, easy-to-use environment that can be used for both real-time and multimedia environments. Statistical multiplexing, on the other hand, is flexible and better suited to a dynamic environment, but this method requires a fairly large number of activities to realize the benefits of statistical sharing. Many modern operating systems are designed to run only a few concurrent programs on a single microprocessor. On personal workstations, only a few concurrent programs are active at a single time, and on multiprocessors, it is common to think more in terms of allocating processors to applications rather than multiplexing applications on single processors. With so few activities being scheduled, statistical multiplexing does not offer the predictability it might when the numbers are larger.

Our approach is to strike a compromise between static real-time systems and statistical multiplexing. Since resources

are to be shared among only a few activities, we cannot depend on statistical assurances that the resources will be available when they are needed. In RT-Mach, one can use a resource reservation mechanism to ensure resource availability. The reservation mechanism does not preclude resources from being multiplexed among several activities, as long as the resource can be scheduled in such a way that it is available to the reservation holder during the interval of time it is reserved. Some resources are difficult to schedule in this way. Physical pages, for example, cannot easily be multiplexed since the "context switch" to copy out data from a page and copy in new data is quite time-consuming. This argues for physical pages being allocated directly rather than being multiplexed, and reservation in this case means that the physical resources are tied up when reserved and cannot be used by other activities. We call this type of reservation a *dedicated* reservation. Processors, however, can be multiplexed fairly easily; the context switch time is not as large. So reservation for processors means that the processor resource, measured in terms of computation time, must be available at the time the reservation holder needs it, and this type of reservation does not preclude the resource being used by other activities, including background activities. We can think of this as a reservation of capacity rather than a reservation of a discrete resource, and we call it a *scheduled* reservation.

Since reserving discrete resources is a relatively straightforward proposition, we have focused more on how a reservation mechanism for the processor would work. The processor reservation mechanism has four parts: an interface to specify reservation requests, an admission control policy, a scheduling algorithm, and a mechanism to enforce reservations. A more complete description of the design and implementation of this reservation system can be found elsewhere [10].

Suppose instead of processor reservation, a fixed priority scheduling policy is used. In this case, the protocol processing structure can be identical to that with processor reservation, except that priorities must be assigned appropriately by the application(s). In addition, each application must not exceed its specified execution times (for timing guarantees given to lower priority tasks to hold true). In other words, enforcement is absent or is up to the application. With the processor reservation model, if shorter period tasks (and hence higher fixed priority tasks using rate-monotonic priority assignment) execute longer than their specified times, the kernel can suspend them or lower their priorities until the current reservation period expires.

3.2. Mach 3.0 Networking

Networking in the context of the Mach 3.0 UX server [1] is accomplished by calling the 4.3 BSD networking primitives which are handled by the UX server. The UX server interacts directly with the network device drivers to send and receive packets. As shown in Figure 3-1, this makes the

UX server a single point of contention for all activities that are using the network. Unfortunately, the networking code inside the UX server does not support priorities nor does it have well-defined real-time properties. In sum, this software does not satisfy our requirements for prioritization and preemptibility in predictable protocol processing software.

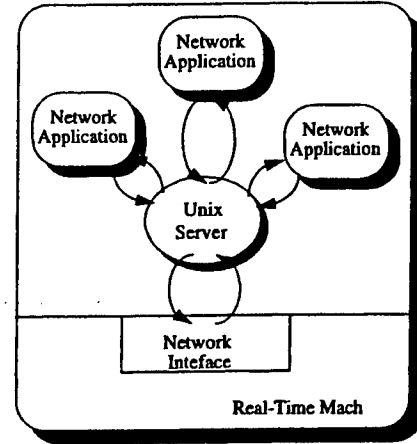


Figure 3-1: Networking with the Unix Server on RT-Mach

Another problem with networking under the UX server of Mach 3.0 is that the interprocess communication (IPC) required between the application and the UX server and between the UX server and the network device drivers adds overhead to network communication. This decreases throughput and increases latency. To alleviate these problems, Maeda and Bershad created a library implementation of TCP/IP and UDP/IP sockets [7]. Their library handles the protocol processing for sending and receiving packets and interacts with the network packet filter [17] and network device drivers directly. The library can be linked in with applications that use the networking calls, so each application can do its own protocol processing in its own scheduling domain (i.e. within its own threads). The library only interacts with the UX server to create and destroy connections and for a few other control operations. The fast path for sending and receiving packets is confined to the library itself (and the device drivers). Figure 3-2 illustrates this networking software structure.

The socket library implementation has multiple threads, internal to the library. Specifically, the threads involved in the protocol processing structure are

1. All socket send operations use the caller's application thread.
2. A `network_thread` receives from the kernel network interface all network packets destined to this application process. All socket receive operations by the application obtain packets received by the `network_thread`.
3. A `network_proxy_thread` receives messages sent by Unix (for use by system calls such as "select" which peek at both socket and file descriptors maintained by the Unix server).
4. A timeout thread is used for timeouts.

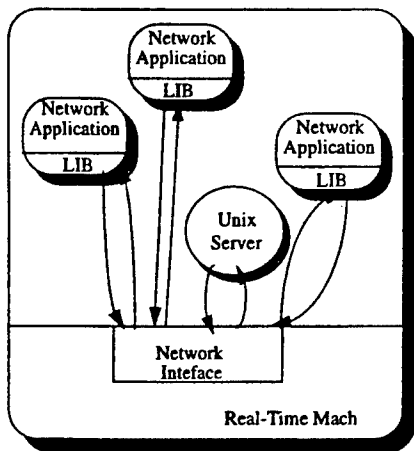


Figure 3-2: Networking with the Socket Library

5. A `pc_sample` thread is used for sampling the PC at roughly periodic intervals for profiling purposes.

The socket library `libsockets` of Maeda and Bershad yields much better performance in terms of throughput and delay than the UX server sockets implementation [7]. Coincidentally, this implementation also satisfies our requirements for effective scheduling of protocol processing. By including the code in a user library, the computation is done by the user thread for sending packets and by the `network_thread` for receiving packets from the network. It is also preemptible since it runs in user mode and shares nothing with other threads in other applications.

3.3. A Real-Time Socket Library with Processor Reservation

We have modified the socket library of Maeda and Bershad to conform to the real-time scheduling model of RT-Mach and to obtain the predictability properties described earlier in this section. Since `libsockets` enables the protocol processing computation to be scheduled as an application-level activity, which can be made preemptible, we can also effectively apply the processor capacity reservation system to programs which do socket-based communication. Compared with a UX server socket implementation, the library partitions the data structures and control paths of all of the networking activities and places them in independent address spaces where they do not interfere with each other. In the UX server, these different activities are forced to share the same queues without the benefit of a priority ordering scheme. In addition, when UX is also used for protocol processing, other UX activities such as file I/O, asynchronous signals, etc. also handled by UX can also interfere with protocol processing. As a result, packets can be delayed as a result of other operating system activities that are not even related to networking.

In our real-time version of the socket library named `libsockets-rt`, these components cannot interfere with each other, and the reservation (or other real-time schedul-

ing) mechanism is free to make decisions about which applications should receive how much computation time and when. The control exercised by the reservation (or real-time) scheduler is not impeded by additional constraints brought on by the sharing of data structures and threads of control.

Conceptually, this socket library structure is not unlike the independently derived design of the real-time publisher/subscriber (RT/PS) inter-process communication model described in [12]. During initialization, both structures talk to a common server (UX for `libsockets` and the `ipc-server` for the RT/PS model). The steady-state operations of sending in the socket library are analogous to steady-state publishing in the RT/PS IPC model. The `network_thread` receiving network packets is slightly different from (but arguably conceptually similar to) the `delivery_manager` in the RT/PS model.³

The following changes are necessary to convert the `libsockets` structure to have controllable and predictable real-time properties in `libsockets-rt` under the RT-Mach reservation policy.

- All threads within `libsockets` must become real-time threads⁴ so that their scheduling attributes can be appropriately controlled.
- For all socket send operations, the calling application thread's processor reservation applies by default under the reservation scheduling policy and no changes are required.
- The `network_thread` must be assigned a processor reservation based on the burstiness and frequency of packets expected from the network. A simple option is to inherit the reservation of the parent thread which initializes `libsockets-rt`. A more complex implementation allows the application to specify a different reserve for use by this thread alone.
- The other threads must be assigned an appropriately small reservation (relatively small computation times with relatively long periods in general).

³The differences between the structures of the socket library and the RT/PS model seem to arise from the fact that the "socket library" only manages local sends and receptions (between the network interface and an application thread), while the RT/PS model deals with transparent communications between processes split across machines. In the latter, multiple copies delivered to the same machine are optimized by sending only one copy to a `delivery_manager` which is then locally sent to all the local recipients. Due to this distributed communications model, in RT/PS, the various `ipc-servers` also need to coordinate with one another. The other major difference in timing semantics is that the RT/PS model has a `notification_thread` which is real-time in nature, while the `network_proxy_thread` which is more limited in semantic scope and not real-time in nature.

⁴In RT-Mach, real-time and non-real-time threads can co-exist, with the real-time threads always having higher priority than the non-real-time threads.

3.4. libsockets-rt with Fixed-Priority Processing

RT-Mach also supports the traditional fixed-priority scheduling scheme and it is desirable that libsockets-rt support this policy too. The changes to libsockets that are required are identical to the changes to support processor reservation except that instead of binding reserves to threads, fixed priorities are assigned to them by the application.

4. Performance Evaluation with Processor Reserves and libsockets-rt

In this section, we evaluate the use of libsockets-rt in the context of the processor capacity reserves supported by RT-Mach. The tests in this section use four different configurations of the RT Mach 3.0 system running on Gateway 2000 i486-66MHz machines. We show the behavior of several task sets using both sockets implemented in the Unix server running on RT-Mach and libsockets-rt under both time-sharing and reservation scheduling policies.

In each of the system configurations, we run several task sets. In the first, we have a single thread which is periodically transmitting several UDP packets (10 packets every 40 ms); this is the activity that is intended to be predictable. This thread has no (substantial) competition from other application programs (other than those normally running under Mach 3.0/UX). We measure the processor usage of this thread which correlates with the number of packets sent, and that is the information that appears in the graphs. In the subsequent task sets, we measure the usage of the same packet transmitting thread, but we introduce competition in the form of several additional non-real-time threads which are doing various kinds of operations. In the second task set, the competition is comprised of 5 compute-bound threads. In the third, the 5 competing threads are making standard I/O calls (stdio) - each stdio call causes IPC messages to be sent back and forth between the application and the UX server. Finally, in the fourth task set, there is a competing low-priority thread sending 10 UDP packets every 40 ms. In the fifth task set, all of these competitive elements are combined.

1. Predictable transmitter with no competition.
2. Predictable transmitter with arithmetic competition (compute-bound).
3. Predictable transmitter with input/output(stdio) competition.
4. Predictable transmitter with background networking competition.
5. Predictable transmitter with all of the above competition.

We refer to the predictable transmitter as the *Net App*. We find that the behavior of *Net App* is affected in different ways, depending on the competition, the CPU scheduling policy and the protocol processing architecture and policies used.

4.1. RT Mach/UX server under time-sharing

In this experiment, we use RT Mach 3.0 with the Unix server providing the networking service to applications. The scheduling policy is Mach time-sharing.

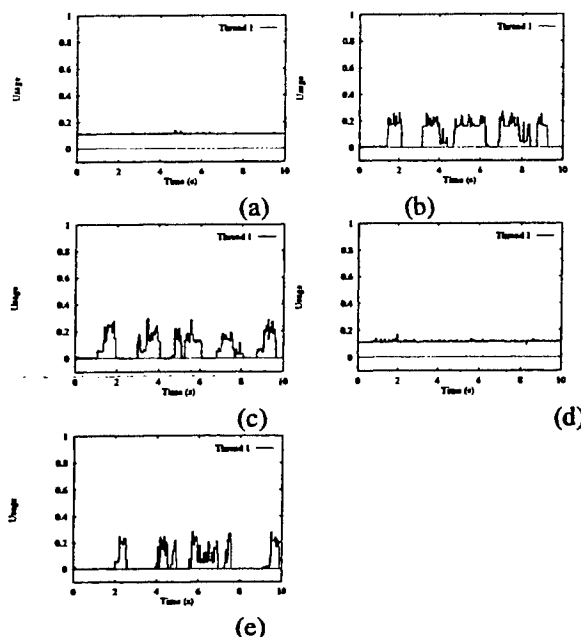


Figure 4-1: Measured Behavior under RT-Mach with Unix sockets, Mach Time-Sharing Policy

In Figure 4-1(a) we see the usage (fraction of the processor capacity) of the *Net App* in isolation. Part (b) of the figure shows the effect of interference from the compute-bound threads. The time-sharing scheduling policy allocates long durations of time to the competition. In Part (c), we see that the stdio competition looks much the same. Part (d) shows that the UDP competition is not very strenuous in terms of computation time, and so the behavior of the *Net App* is fairly predictable, but when we combine all of the types of competition in Part (e), we see that the resulting interference makes the *Net App*'s behavior unpredictable. The interference is substantial; there are periods of up to 1 second where the computation time the *Net App* receives is virtually nil. This is caused by the fact that the Mach time-sharing scheduling algorithm tends to give large durations of computation time to compute-bound programs. Also, the *Net App*'s message processing is done by the UX server which has to do I/O processing for the Stdio Apps and additional message processing for the Bg network application as well.

4.2. RT Mach/libsockets-rt under Time-Sharing

The tasks in this experiment use libsockets-rt and the scheduling policy

used is Mach time-sharing.

Figure 4-2(a) shows the *Net App* in isolation. In parts (b) and (c), we can see that the *Net App* is sensitive to inter-

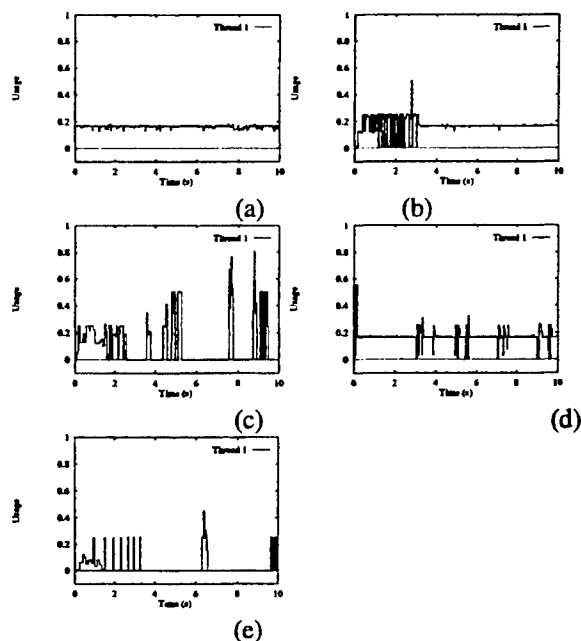


Figure 4-2: Measured Behavior under RT-Mach with libsockets-rt and Mach Time-Sharing

ference from the arithmetic and stdio competition, but it suffers only a little interference from the Bg Net App in part (d). For part (e) where the competition is a mixture of all three types of activity, the interference is severe. Much of this interference comes from the time-sharing scheduling policy sometimes giving preference to the compute-bound threads and sometimes to the I/O-bound threads.

4.3. RT Mach/UX server with Reserves

The tasks in this experiment use the Unix server for network services and the RT-Mach reservation scheduling policy. The point is to demonstrate that simply using reservation scheduling does not solve the problem; the protocol processing architecture plays an important role in achieving predictable behavior.

Figure 4-3(a) shows the *Net App* in isolation, and the behavior is very regular and predictable. The behavior is also (fairly) predictable with arithmetic competition (b), stdio competition (c), and background network competition (d). The combination of these types of competition in Figure 4-3(e), however, reveals the effect of the interaction between the main *Net App*, the Stdio Apps, and the Bg Net App which all share the UX server. Since UX services all of these applications and since it does not have priorities internally, these clients interfere with each other. We can see this reflected in the performance of the *Net App* which is very erratic. This experiment shows that reservation scheduling is not enough to ensure predictability when resources such as the UX server are being shared.

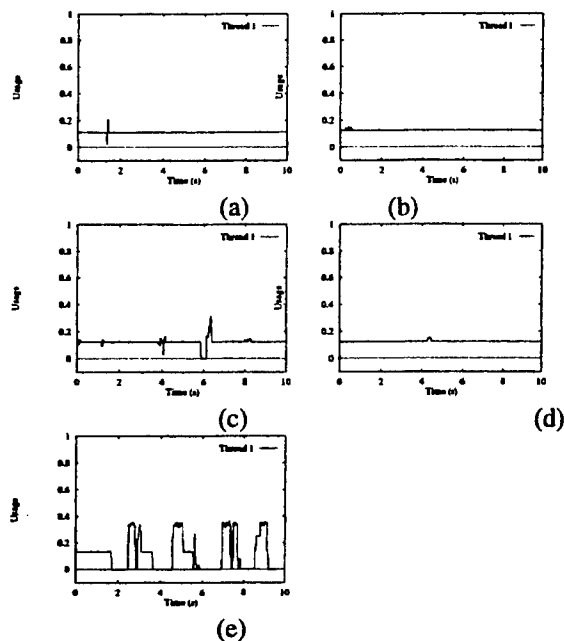


Figure 4-3: Measured Behavior under RT-Mach with UX sockets, RT-Mach Reservations

4.4. RT Mach 3.0/libsockets-rt with reservation scheduling

This final task set uses RT-Mach reservation scheduling and uses libsockets-rt. Thus, it has all the desirable features we discussed in Sections 2 and 3.

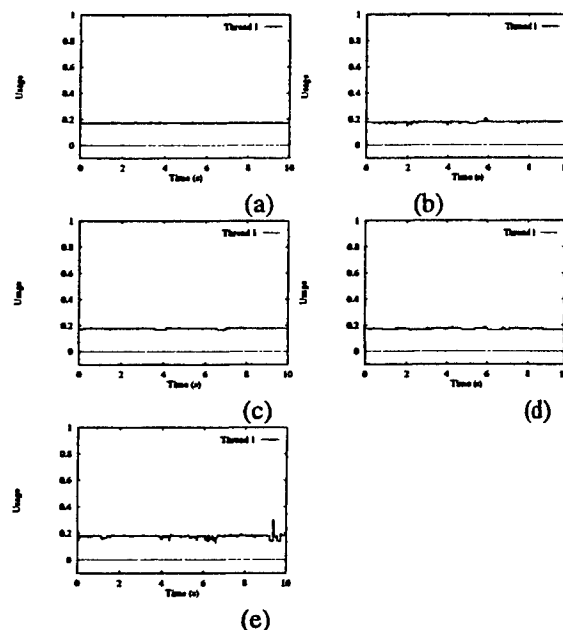


Figure 4-4: Measured Behavior under RT-Mach with libsockets-rt, RT-Mach Reservations

In Figure 4-4(a), we see the *Net App* in isolation. Parts (b), (c), and (d) show that the *Net App* suffers little or no interference from arithmetic competition alone, from stdio com-

petition alone, or background network competition alone. And in Figure 4-4(e), we see that even in the case where all of the various types of competition are combined, this system configuration provides very predictable behavior for the real-time Net App. Although the usage varies a little in this case, the variations are not nearly as damaging as the variations in the previous experiments. These slight variations are due to the unavoidable sharing of low-level system resources such as network interrupt handlers.

5. A Video-Conferencing System with Fixed Priority Scheduling and libsockets-rt

We have extended *RT-Phone* [5], an audio-conferencing system built on *RT-Mach*, to support real-time duplex transfer of video as well, yielding a video-conferencing system. The parties involved in the video conference run on two Intel Pentium 120-MHz PCs with two Pro-Audio Spectrum 16 sound cards for full duplex audio capabilities, and a Matrox Meteor video frame-grabber each. A high-resolution timer card on each machine yields timestamps and time information up to a resolution of 1 μ second. In this paper, we focus on the protocol processing on the CPUs running *RT-Mach* and the actual network delays are considered to be small. The two nodes are connected using a dedicated 10Mbps ethernet. The focus of our experiments is protocol processing and how it affects the end-to-end delay of different real-time streams (audio and video). Hence, we explicitly do not consider synchronization of audio and video streams in the following discussions. For an interesting discussion of audio/video synchronization in an uncontrolled network context, the reader is referred to the work by Jeffay et al. [4]. They have studied the problem of audio/video synchronization in an uncontrolled network.

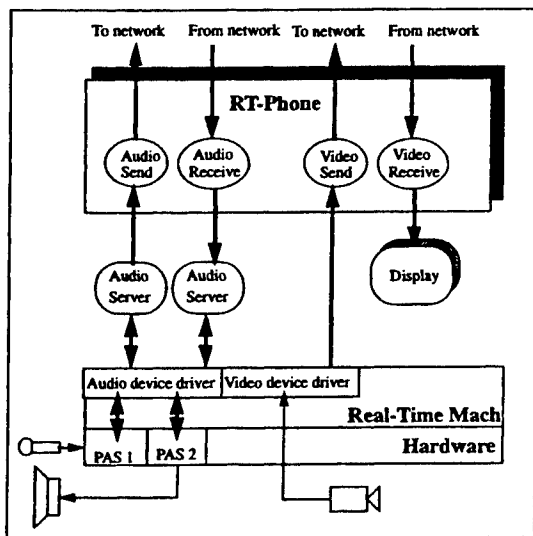


Figure 5-1: RT-Phone Video-Conferencing Support

The configuration of the *RT-Phone* video-conferencing system is illustrated in Figure 5-1. It is in many ways similar

to traditional teleconferencing applications (for example, PictureTel, "CU/see-me" on the Mac, [3, 16]) but it is also quite different in many other ways. For example, a distinct user-level audio server process hides the details of the system's sound card by providing a higher level interface similar to the AudioFile utility. Processor reserves or fixed priority scheduling is used to provide guaranteed timing behavior.

5.1. The Audio End-To-End Delay

The processing pipeline of the audio data from one side to the other side is presented in Figure 5-2. As indicated, let the period T_{audio} represent the time it takes for the sound card to fill its internal buffer and interrupt the CPU. The size of this internal buffer, referred to as an "audio frame" in [4], is application-selectable. Successive interrupts with new blocks of audio data arrive every T_{audio} time-units apart. We assume that the processing and sending of the audio data need to be completed by the arrival of the next block of audio data. In other words, the deadline for processing and sending is T_{audio} . Similarly, on the receiving side, we assume that the audio data after reception must be passed to the audio card for output to the speaker within a duration of T_{audio} time-units.

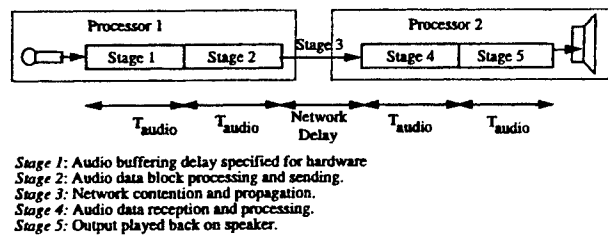


Figure 5-2: The Audio Processing Pipeline

Consider an audio sample obtained by the hardware at the beginning of stage 1. This sample will be played back at the receiver's speaker at the beginning of Stage 5. Hence, the worst-case end-to-end delay for audio is given by $3T_{\text{audio}} + d_{\text{network_audio}}$, where $d_{\text{network_audio}}$ is the worst-case network delay encountered by the audio stream. The deadlines for Stage 2 (the sender) and Stage 4 (the receiver) can be shortened (if need be) to yield correspondingly smaller end-to-end delays constrained by schedulability considerations.

5.2. The Video End-To-End Delay

The pipeline of the video data is presented in Figure 5-3. The audio capture takes T_{audio} units of time to capture T_{audio} units of sound. In contrast, the time to capture a single video frame is smaller than the period at which the frames are displayed. Hence, on the sender side, we capture, process and transmit the video frames every T_{video} time-units. On the receiver side, we require that the video receiver receiver, process and update the display every T_{video} time-units.

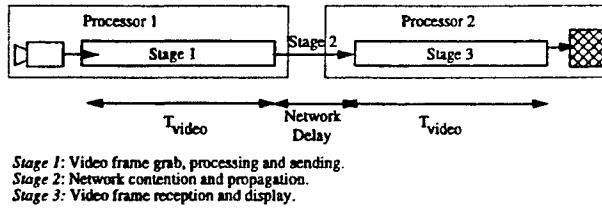


Figure 5-3: The Video Processing Pipeline

Hence, the worst-case end-to-end delay for each video stream is given by $2T_{\text{video}} + d_{\text{network_video}}$, where $d_{\text{network_video}}$ is the worst-case network delay encountered by the video stream. The deadlines for both the sender and the receiver can be shortened to yield a smaller end-to-end delay.

The video sender, video receiver, audio sender and audio receiver on each machine use *libsockets-rt* and therefore their protocol processing is under control of RT-Mach scheduling policies.

5.3. Performance Measurements

We conducted several experiments using *RT-Phone* with the following objectives:

- Check how well *libsockets-rt* prioritizes the protocol processing delays of different real-time activities within RT-Mach.
- Measure the jitter that is visible at the receiving ends of video and audio. If jitter is excessive, period enforcement would become necessary.

For the audio streams, we used a value of 16 ms for T_{audio} with audio sampling rates of 8 KHz, 16 KHz and 24 KHz respectively at a sample size of 8 bits per sample. For the video streams, we used 250 ms, 125 ms and 83.3 ms for T_{video} (corresponding to 4, 8 and 12 frames per second). The frame size is 80x80 pixels at 8 bits/pixel. At this frame rate and resolution, each video stream consumes up to 614 Kbps, and each audio stream consumes up to 192 Kbps (for a net aggregate network bandwidth of up to 1.4 Mbps).

Audio Sampling Rate	Video Frame Rate (fps)	No <i>libsockets-rt</i>		With <i>libsockets-rt</i>	
		No competition (ms)	W/ Competition (ms)	No Competition (ms)	W/ Competition (ms)
8 KHz	4 fps	34	83	26	26
	8 fps	41	87	26	30
	12 fps	38	115	26	26
16 KHz	4 fps	41	71	23	23
	8 fps	38	83	26	26
	12 fps	41	105	26	26
24KHz	4 fps	41	60	26	23
	8 fps	49	86	26	23
	12 fps	45	113	26	26

Table 5-1: The Audio End-To-End Delay with varying frame rate and audio sampling rates.

The end-to-end delay variation of the audio stream is listed in Table 5-1 with and without *libsockets-rt* is used and with and without competition from a low priority network application and a medium priority arithmetic application. As can be seen, the worst-case end-to-end delay for the audio stream is much below the worst-case end-to-end delay of $3T_{\text{audio}}$ when *libsockets* is used independent of the presence of lower priority competition. This is due to the fact that we assign rate-monotonic priorities to these streams, there are no higher priority streams and *libsockets-rt* provides a near-ideal environment. The fact that *libsockets-rt* completely insulates a real-time networking application from other the needs of other networking applications also indicates that almost all of the networking overhead is in the protocol stack and not in the raw device interface (which is still in a common non-preemptive shared kernel in RT-Mach). In contrast, if *libsockets-rt* is not used, and there is no competition, the end-to-end delay is comparable but slightly larger than that with *libsockets*. But in the presence of competition, the end-to-end delay significantly increases the audio end-to-end delay by a factor of greater than 4. This confirms that priority inversion arising from FIFO queueing in the UX server takes a serious toll on end-to-end delays.

Audio Sampling Rate	Video Frame Rate in fps (period)	No <i>libsockets-rt</i>		With <i>libsockets-rt</i>	
		No competition (ms)	W/ Competition (ms)	No Competition (ms)	W/ Competition (ms)
8 KHz	4 (250 ms)	21	30	24	26
	8 (125 ms)	24	40	24	27
	12 (83.3 ms)	26	50	23	26
16 KHz	4 (250 ms)	26	32	26	26
	8 (125 ms)	25	40	26	25
	12 (83.3 ms)	25	50	26	25
24KHz	4 (250 ms)	26	30	25	25
	8 (125 ms)	23	40	25	27
	12 (83.3 ms)	25	51	23	25

Table 5-2: The Video End-To-End Delay with varying video frame rates and audio sampling rates.

The end-to-end delay variation of the video stream is plotted in Table 5-2. Almost identical results as obtained for audio are obtained in this case in that the video end-to-end delay is much better than its worst-case latency⁵ and is not affected by the presence of lower priority competition. With *libsockets-rt*, the video streams experience some jitter with a standard deviation of around 6 ms. Without *libsockets-rt*, the jitter has a standard deviation ranging from 14 ms to 25 ms due to the unpredictability of conflicts.

⁵The measured video end-to-end latency did not include display time and the actual delay should be correspondingly longer.

6. Conclusion

The two major system components essential for ensuring end-to-end predictability in distributed real-time and multimedia applications are the protocol processing software structure and network bandwidth management. The protocol processing software structure must exhibit the features necessary for good real-time performance: prioritized scheduling and preemptibility. We have implemented a protocol processing structure in RT-Mach that satisfies these requirements. The structure can be used equally well with a fixed priority policy or RT-Mach's processor reservation scheduling policy. Under the fixed priority scheduling policy, it is up to the application to ensure that higher priority tasks do not overuse the CPU. Under the processor reservation model, the kernel can monitor and enforce the maximum usage of all threads including the threads interacting with the network interface. These schemes address the need for the CPU scheduling policies to coordinate with scheduable protocol structures to obtain predictable end-to-end delays.

While the performance figures we obtain for both synthetic workloads and realistic multimedia applications look very promising, we continue to evaluate the performance of the network protocol processing structure under different kinds of network load scenarios, scheduling policies and other protocol implementations. For completeness, the protocol processing structures must also be integrated with the schemes used for allocating and using network bandwidth. Future work will also address this issue.

Acknowledgements

This paper is partly based on the Technical Report, "Predictable Operating System Protocol Processing", CMU-CS-94-165, by C. Mercer, J. Zelenka and R. Rajkumar. We also like to thank Chris Maeda for help with his socket library and Jim Zelenka for his initial port of the library to RT-Mach.

References

1. D. Golub, R. W. Dean, A. Forin and R. F. Rashid. Unix as an Application Program. Proceedings of Summer 1990 USENIX Conference, June, 1990, pp. .
2. Hutchinson, N. C. and Peterson, L. L. "The x-Kernel: An Architecture for Implementing Network Protocols". *IEEE Transactions on Software Engineering* 17, 1 (Jan. 1991), 64-76.
3. K. Jeffay, D. L. Stone and F. D. Smith. Kernel Support for Live Digital Audio and Video. Proceedings of the Second International Workshop on Network and Operating System Support for Digital Audio and Video, Nov., 1991, pp. 10-21.
4. K. Jeffay and D. L. Stone. Adaptive, Best-Effort Delivery of Digital Audio and Video Across Packet-Switched Networks. Video Abstract in the Proceedings of ACM Multimedia 94, Oct, 1994.

5. Lee, C., Rajkumar, R. and Mercer, C. W. "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach". *Multimedia Japan* (March 1996).
6. Leffler, S. J. and McKusick, M. K. and Karels, M. J. and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
7. C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, Dec., 1993, pp. 244-255.
8. C. W. Mercer and H. Tokuda. An Evaluation of Priority Consistency in Protocol Architectures. Proceedings of the IEEE 16th Conference on Local Computer Networks, Oct., 1991, pp. 386-398.
9. C. W. Mercer and R. Rajkumar and J. Zelenka. Temporal Protection in Real-Time Operating Systems. Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, May, 1994, pp. 79-83.
10. C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS), May, 1994, pp. 90-99.
11. R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
12. Rajkumar, R., Gagliardi, M. and Sha, L. "The Real-Time Publisher/Subscriber Communication for Inter-Process Communication in Distributed Real-Time Systems". *The First IEEE Real-time Technology and Applications Symposium* (May 1995).
13. Sha, L. and Goodenough, J. B. "Real-Time Scheduling Theory and Ada". *Computer* (May 1990).
14. Tokuda, H. and Mercer, C. W. "ARTS: A Distributed Real-Time Kernel". *ACM Operating Systems Review* 23, 3 (July 1989), 29-53.
15. Tokuda, H., Mercer, C. W., Ishikawa, Y. and Marchok, T. E. Priority Inversions in Real-Time Communication. Proceedings of 10th IEEE Real-Time Systems Symposium, Dec., 1989.
16. H. M. Vin, P. T. Zellweger, D. C. Swinehart and P. V. Rangan. "Multimedia Conferencing in the Etherphone Environment". *IEEE Computer* 24, 10 (Oct. 1991), 69-79.
17. M. Yuhara, B. N. Bershad, C. Maeda and J. E. B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. Proceedings of the 1994 Winter USENIX Conference, Jan., 1994, pp. .

Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach

Chen Lee, Ragunathan Rajkumar and Cliff Mercer

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

USA

{clee,raj+,cwm@cs.cmu.edu}

Abstract

The RT-Mach microkernel supports a *processor reserve* abstraction which permits threads to specify their CPU resource requirements. If admitted by the kernel, it guarantees that the requested CPU demand is available to the requestor. We designed this kernel-supported mechanism to be relatively simple based on the microkernel notion that user-level policies can use this simple mechanism to build more complex and powerful schemes. In this paper, we focus on the needs of such user-level policies in the form of a dynamic Quality of Service server.

We seek three goals: (a) explore the necessity, sufficiency, power and flexibility of the kernel-supported reserve mechanism (b) dynamic management of application quality in real-time and multimedia applications, and (c) investigate our ability to predict and achieve end-to-end application delays in realistic distributed real-time and multimedia applications. We use a two-pronged approach to accomplish our goals. First, we apply the processor reserve abstraction in a user-level dynamic quality of service server. A QOS server can allow applications to dynamically adapt in real-time based on system load, user input or application requirements. Second, we apply the dynamic QOS control capabilities to a distributed multimedia application whose threads have to interact and coordinate with each other within and across processor boundaries. A new notion called *continuous thread of control* is introduced to assist in bundling processor reserves. Our experiments show that we can indeed predict and achieve end-to-end delays in a distributed multimedia application. A summary of lessons learned and additional functionality needed is also provided.

Keywords: Real-Time Mach, processor reservation, audio conferencing, reserve bundling, end-to-end latencies.

1. Introduction

The need for real-time resource management is a key element which explicitly distinguishes multimedia applications from other traditional applications. The Real-Time Mach kernel [19] supports many primitives for constructing predictable and dynamic real-time applications. These primitives include a wide choice of real-time scheduling policies (including fixed priority scheduling, rate-monotonic

scheduling and earliest deadline scheduling), periodic and aperiodic real-time threads with the notion of deadlines and handlers for deadline misses, real-time synchronization primitives [12] and real-time inter-process communication primitives which support priority inheritance [15], virtual memory wiring, a real-time shell and a network protocol server, a processor reserve abstraction, an integrated toolset for schedulability analysis and real-time monitoring of scheduling decisions [2], and real-time X11 windowing services [10]. In this paper, we build upon the processor reserve abstraction to provide dynamic quality of service support, and to predict and achieve end-to-end delays in distributed multimedia applications.

1.1. Processor Reserves in Real-Time Mach

The Real-Time Mach microkernel supports an abstraction called *processor capacity reserves* [9] which allows application threads to specify their CPU requirements in terms of their timing constraints:

- The kernel performs admission control using a specification that consists of required CPU usage per specified interval (such as 10 *ms* every 50 *ms*, or 20 *ms* every 60 *ms*).
- The kernel scheduler schedules application threads such that these timing constraints are satisfied.
- The kernel allows multiple threads (possibly from different tasks) to be bound to the same reserve.
- The kernel enforces that application threads bound to a reserve specification cannot disrupt the timing behavior of other applications. This is achieved by ensuring that only the reserve specification is guaranteed by the kernel, and *beyond the* stated usage level, the kernel can suspend or execute the demanding application threads at low priority. Such kernel enforcement of each reserve provides a temporal protection barrier between applications, a temporal domain analogue to the address space protection of processes in the spatial domain [9].
- Real-time applications with reserves and non-real-time applications which do not need guarantees reserves can co-exist comfortably. Applications without reserves are implicitly bound by the kernel to a *default reserve*, which is scheduled only where there are no other threads bound to reserves ready to run.

- System calls are available to provide feedback to applications about the recent usage and guaranteed information of various reserves. Since the kernel must monitor the execution time of threads to enforce reserves, it provides a built-in framework for measuring the amount of time spent by threads, overhead and CPU idleness. The CPU idleness is charged to a kernel-defined *idle reserve*.
- Reserve parameters can be dynamically adjusted subject to the admission control policy. The timing behavior of reserved applications can therefore be changed dynamically.

These features of the processor reserve abstraction were intended to provide a simple yet powerful set of mechanisms for use by real-time and multimedia applications in a context where non-real-time applications may also co-exist. This abstraction has been tested previously in stand-alone applications such as a video viewer which reads a file into memory in non-real-time (background) mode and then displays frames in real-time mode.

1.2. Related Work

Software developers have written many multimedia applications that run acceptably on general purpose operating systems as long as no other programs compete for system resources [1, 5, 14, 18]. With additional real-time operating system support to carefully manage operating system resources, these applications could run even with competition.

Many researchers in the distributed real-time multimedia community have turned their attention to end-to-end performance guarantees which ultimately include network communication and end-system resource management (including, but not limited to, CPU, network protocol stack, memory, file system or server). Much research has been done on the networking issues, [4], [7] [22] to name a few, in which specific assumptions are made about the end-points of the distributed computation. As that work matures, attention is turning to issues of end-system control as well [21].

Jeffay et al. [6] employ hard real-time scheduling theory in a specialized micro-kernel to address timing issues related to different stages of live video and audio processing. Robin et al. [16] designed a system based on Chorus [3] micro-kernel that addresses both the network and end host QOS control. The system uses an earliest-deadline-first scheduling policy and a time-line-based admission test for "guaranteed class" threads.

Nahrstedt and Smith [11] used AIX for their telerobotic application and showed that the AIX real-time priorities are not enough to control protocol task behavior when used for implementation of rate-monotonic or deadline-based scheduling, unless severe restrictions are made which includes only one user allowed, one multimedia application

running on the RS/6000, application/transport protocols implemented in a single user process with real-time priority etc. "Only with these restrictions satisfied can we map rate-monotonic scheduling onto the real-time priority scheme of AIX to provide (approximate) predictability for guaranteed services" [11].

1.3. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we outline the objectives of a dynamic QOS server on top of RT-Mach and describe its architecture. In Section 3, we describe a real distributed multimedia application called *RT-Phone*. In Section 4, we apply the reserve and QOS abstractions to *RT-Phone*. We also derive the predicted end-to-end delay that must be guaranteed by the use of reserves. A new concept named *continuous thread of control* is introduced to *bundle* many threads to a single reserve, and a multi-phase protocol to coordinate two QOS servers is presented. We also provide performance numbers from *RT-Phone* to demonstrate that we do satisfy the predicted end-to-end application delay. Finally, Section 5 presents a summary of the lessons that we learned based on *RT-Phone* and our experiments.

2. Dynamic Quality of Service Control

The relatively simple mechanisms associated with the reserve mechanism are designed to be basic building blocks that can be used to construct more powerful user-level schemes. Real-time and multimedia applications on RT-Mach cannot be assumed to be static in nature. Thus, we must allow a dynamic mix of concurrent real-time and multimedia applications whose timing requirements not only vary widely but can also change during run-time. Instead of each application designing its own scheme(s) to use the kernel reserve support, a consistent framework that lays out the ground rules for cooperation and coordination between various applications is highly desirable. Based on this motivation, we propose a dynamic user-level QOS server with two specific requirements. First, it must be able to meet the timing requirements of each application independent of the behavior of other applications. Second, it must be able to let an application dynamically adapt its own behavior based on its own internal requirements, user input and/or the total load on the system.

2.1. Quality Management and its Implications to Systems Support

We summarize our approach to quality management as follows. The "quality" of an application can ultimately be defined only by the application in concern. For example, one recording application may emphasize the reception of all incoming audio packets, another interactive application may emphasize lower jitter and yet another may emphasize very low end-to-end delay. An application-independent kernel can therefore cannot institute direct support for all possible notions of quality. However, application quality requirements can and in practice are eventually translated to

the demands that they place on system resources. From a kernel's perspective, therefore, we strive to support flexible and powerful mechanisms which can support variations in the resource requirements of various applications, and also dynamic quality changes that may be required of individual applications.

The key theme behind our resource management approach is not unlike that behind the reserve abstraction. We aim to provide a 2-way mechanism between the application and the system layer wherein the application can flexibly specify its requirements to the system layer, and in turn, the system layer can provide accurate and dynamic feedback on the state of the application's resources individually and with respect to the other applications that are co-resident. The QOS server we describe next is based on this primary theme.

2.2. The QOS Server

The architecture of the QOS server is presented in Figure 2-1. Our implementation currently assumes that applications are cooperative rather than malicious in nature¹.

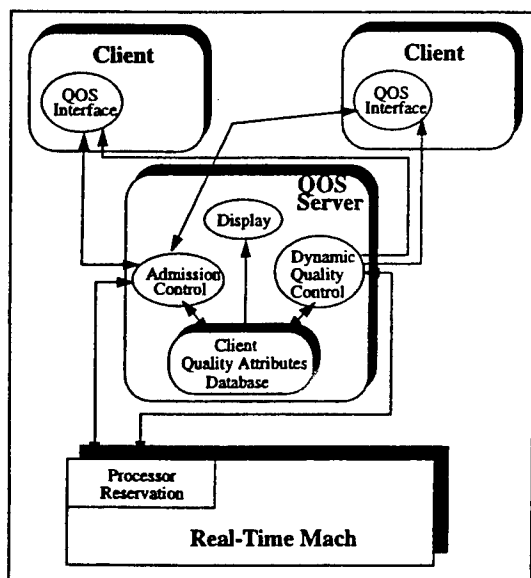


Figure 2-1: The Architecture of the Dynamic QOS Server.

The QOS server is based on the following complementary aspects:

- *QOS attributes* are used by application threads to specify their acceptable quality levels to the QOS server.
- An application can choose from one of many *quality adjustment policies* that the QOS server can use on it if the quality guarantee given to the application has to be

upgraded or downgraded. In other words, this determines how an application wants to be treated when its current resource allocation has to be altered. Included in this policy choice is an application *quality adjustment priority* which determines the order by which the QOS server selects the beneficiary (victim) of a server decision to upgrade (downgrade) the quality of one or more applications.

- One of many *admission control adjustment policies* is used by the QOS server to admit as many new application requests as possible without violating the needs of currently registered applications.
- One of many *overrun control policies* is used by the QOS to satisfy the needs of registered applications whose current CPU demand exceeds their actual allocation. This is done by the QOS server exploiting the slack that may be available from applications not using their current resource allocation.

These components of the QOS server are described in greater detail next.

2.3. QOS Attributes

Application threads can submit requests to the QOS server for resource allocation specifying *QOS attributes*. The kernel reservation mechanism requires a fixed (worst-case) computation time and a fixed period, but the QOS server relaxes this constraint to be more flexible and dynamic. QOS attributes include the minimum, maximum and most-desired levels of the computation time as well as the minimum, maximum and most-desired levels of the period. In other words, if an application chooses a range of acceptable computation times (or periods), the QOS server is free to pick a legal value in this range. The QOS server tries to provide to each application the maximum quality value in this range, but will not go below the minimum requested value (once an application is admitted).

2.4. Quality Adjustment

A quality adjustment policy for each request indicates how the QOS server should pick from the possibly wide range of possibilities for the 2-tuple {computation time, period} specified in an application's QOS attribute:

- An *Adjust-Computation-Time-First* policy adjusts the computation time first keeping the period constant (later adjusting the period if needed). A *Keep-Period-Constant* policy is obtained by specifying the minimum, maximum and desired values of the period to be the same. This option can be utilized by applications which can pick from different algorithms to process their data (e.g. choose a simple but less accurate algorithm versus a complex but more accurate algorithm), or pick different paths of the same algorithm based on the available time (e.g. choose not to decode some blocks in JPEG decoding of a video frame).
- An *Adjust-Period-First* policy adjusts the period first

¹This assumption can be eliminated by forcing the QOS server to be the sole interface to the kernel's reservation mechanisms, similar to the user-level UX server on Mach.

keeping the computation time constant (later adjusting the computation time if needed). A *Keep-Computation-Time-Constant* policy is obtained by the application by specifying the minimum, maximum and desired values of the computation time to be the same. This policy can be useful for applications which can tolerate changes to their rate of execution but not their computation time per instance (e.g., capture a video frame and compress it using JPEG but the rate of capture can be varied).

- A *Keep-Reservation-Constant* policy is obtained by specifying the minimum, maximum and desired values of the computation time to be the same, and similarly for the period.
- A *Step-wise Adjustment* policy is available for the computation time (period) in which the computation time (period) will be adjusted only in discrete steps, the size of which is specified by the application.
- A *Negotiation Policy* tells the QOS server to notify the client when an adjustment needs to be made (along with information about the maximum available reservation at the time) and the client can negotiate the computation time and reservation parameters of the new adjustment. This policy leaves the actual choice of resource allocation parameters to the application and can therefore be used by those applications which cannot be satisfied by any of the other policies. This policy is particularly useful for real-time applications such as feedback control, where controllers may only be defined in the application for some specific values of the period².

The chosen quality adjustment policy for an application is applied when the QOS server decides to alter its resource allocation. Such decisions are made when the server exercises its admission control policy and dynamic quality control policy. These are discussed in subsequent sections.

In addition to the above quality adjustment policies, a *quality adjustment priority* associated with each reserve specifies the global priority at which the application's quality will be adjusted. This means that if adjustments to reservations allocated to applications were to be made by the QOS server, an application with a lower adjustment priority would be upgraded after (and downgraded before) an application with a higher adjustment priority.

An application with reserves can submit a dynamic on-line request to change its prior status, requesting a higher allocation and/or changing its quality adjustment policy. The request is not guaranteed to be honored, however. This allows a QOS client to change its behavior dynamically based on user input (or changing internal application requirements).

²A feedback control application, for example, can use a PD controller at some pre-defined high frequencies, a PID controller at some pre-defined medium frequencies, and a fuzzy/adaptive controller at relatively low frequencies.

2.5. Admission Control Policies

The QOS server tries to provide the maximum reservation available in the system based on the application requirement and the system load at the time of request. The reservation parameters of existing QOS clients may be modified if necessary to admit a new client. An admission control policy allows the QOS server to decide whether to accept a new request and at what level of quality. Six different *admission control policies* are available:

- *New-Minimum, Existing-Minimum*: Both incoming and current applications can be pushed down to their minimum acceptable quality levels.
- *New-Minimum, Existing-Desired*: The incoming application can be pushed down to its minimum quality level but current applications must not be pushed farther below their desired levels.
- *New-Minimum, Existing-Actual*: The incoming application can be pushed down to its minimum quality level but current applications must not be pushed farther below their current allocation levels.
- The remaining 3 policies, *New-Desired / Existing-Minimum*, *New-Desired / Existing-Desired* and *New-Desired / Existing-Actual*, are counterparts to the above 3 but the incoming request must be accepted at its desired quality level.

2.6. Dynamic Quality Control Policies

The QOS server polls the kernel at periodic intervals (currently every 5 seconds) to determine how much of each guaranteed reservation is being under-used (signaling an under-run), and how many threads have their usage exceeding their reservations (signaling an over-run which is executed in background mode and charged to a *default reserve* in the current version of RT-Mach). Based on this information, the QOS server may decide and notify clients that their reservations are being reduced or increased so that they can adapt their behavior. One of four different *overrun control policies* can be chosen. These policies are similar in nature to the admission control policies discussed above.

2.7. The QOS Server Threads

As illustrated in Figure 2-1, the QOS server implementation consists of 3 threads. The *admission control thread* determines if new QOS requests can be granted. The *dynamic quality control thread* is the one which periodically checks the kernel for status information regarding the reserve usage of various registered applications. The *display thread* is an X-client which graphically depicts the granted reservations to QOS clients, the actual usage of reservations, the quality adjustment policy for each QOS client, and the admission and overrun policies of the QOS server itself.

2.8. The Client Interface to the QOS Server

QOS clients access the QOS server using library calls to register/unregister their presence, as well as to request and alter their quality attributes. The QOS server notifies each QOS client of any changes in resource allocation using a communication port registered specified by the application at the time of registration.

3. The RT-Phone Teleconferencing Application

In order to study the necessity, sufficiency, power and performance of the reservation and QOS mechanisms, we implemented a distributed teleconferencing application across the network. We describe this application named *RT-Phone* below followed by a description of the application architecture. Our QOS experiments with the application will be described in the next section.

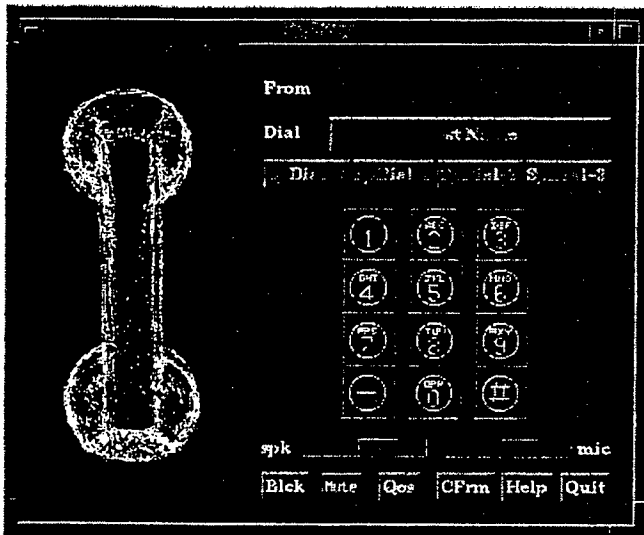


Figure 3-1: The *RT-Phone* user interface on starting up.

3.1. RT-Phone

RT-Phone presents a telephone-pad-like Motif-based graphical interface running on top of RT-Mach. The initial state when *RT-Phone* starts is presented in Figure 3-1. A "caller" initiates communication with a remote workstation running *RT-Phone* by first clicking on the "handset" as illustrated in Figure 3-2. The "dialer", corresponding to "Hostname" and other "speed-dial buttons", becomes enabled when the handset is "off the hook". The caller then specifies a destination host (using a speed button or by specifying a specific remote hostname in a dialog box), and a connection request is sent to the remote machine. This connection request is displayed on a "caller-id" window as shown on the top right window in Figure 3-3. The callee completes establishment of the two-way connection by clicking on his/her "handset" button. At this point, a 2-way audio conversation can take place on the network. A 2-way real-time video stream can also be transmitted when the

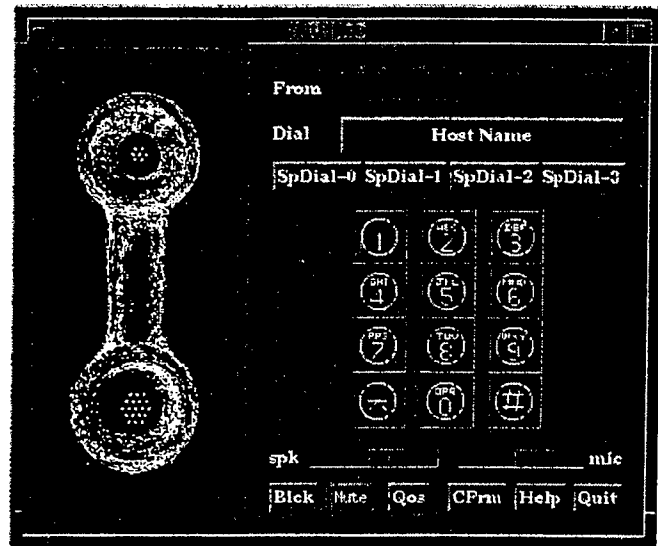


Figure 3-2: The *RT-Phone* user interface in the "off-the-hook/ready-to-dial" state.

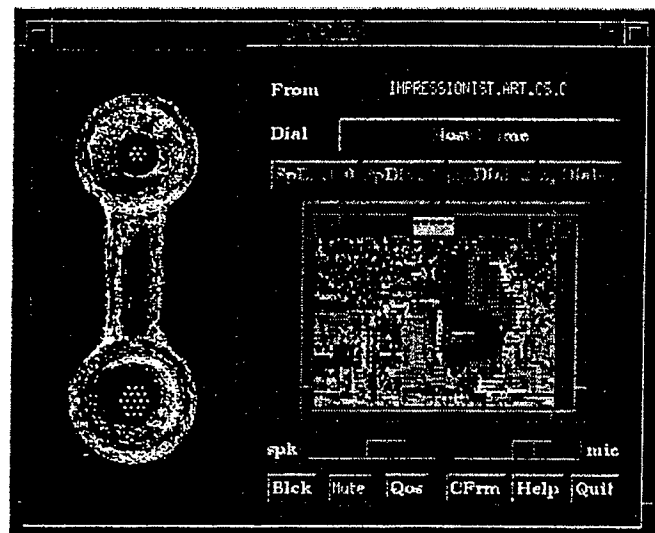


Figure 3-3: The *RT-Phone* user interface after a network phone connection is established.

connection gets established, but in this paper we shall confine our discussion to the duplex audio streams only.

Controls are available for the user to set the volume level of the speaker and/or the microphone, to block incoming connection requests from specific hosts, mute microphone input, to clear the "caller-id" field, and to modify the quality of the audio streams transmitted (this will be discussed in more detail in Section 4.5). These features are generally only a subset of more sophisticated tools such as the Ether-Phone system [20], and were custom-designed only to exercise relevant portions of our QOS and reservation mechanisms.

3.2. The Architecture of the RT-Phone Application

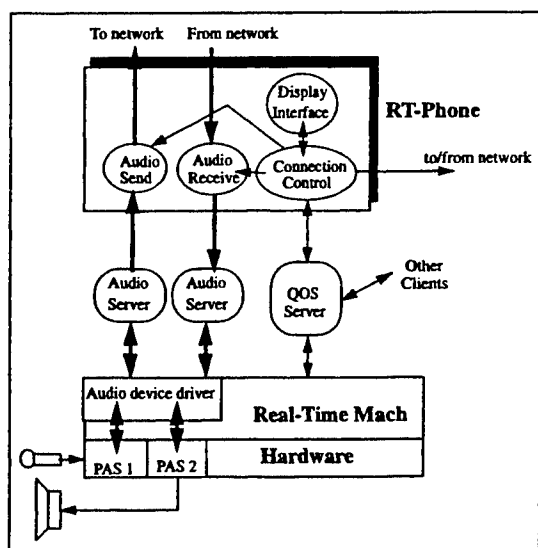


Figure 3-4: The Architecture of *RT-Phone* at each end-point.

The architecture of *RT-Phone* at each node is presented in Figure 3-4. It is in many ways similar to traditional teleconferencing applications (for example, PictureTel, "cu/c-me" on the Mac, [6, 20]) but it is also quite different in many other ways. For example, a distinct user-level audio server process hides the details of the system's sound card by providing a higher level interface³. Processor reserves provide a mutual temporal barrier between *RT-Phone* and other applications. In addition, the same processor reserve is applied across task boundaries. Finally, we allow the quality of the audio streams to be modified dynamically.

This application runs on Real-Time Mach on two 66Mhz 486DX processors networked using Ethernet. The dark directional arrows of Figure 3-4 represent the data flow in the application. A *User-Level Audio Server (ULAS)* [13] provides a high-level programming interface to audio cards, by hiding the details and device driver calls for the specific sound card in use. For example, in *RT-Phone*, we use SoundBlaster-compatible Pro-Audio Spectrum 16 (PAS) audio cards. This ULAS interface allows the choice of an audio sampling rate, the choice between 8-bit samples and 16-bit samples and the choice of a buffer size, which is used to store audio samples until ready for input and output. Three threads in the audio-server deal with control of these configuration parameters, audio data input from the microphone, and audio data output to the speaker respectively.

Although the PAS card supports bi-directional audio (cap-

ture and output), it cannot do both simultaneously (in duplex mode). Hence, the user-level audio-servers can only do either audio input or output at any given time. We therefore use two PAS cards and two audio servers, one for continuous audio sampling and another for continuous audio output respectively. A periodic interrupt based on the sampling rate and buffer size triggers the input audio server to copy the buffer into its requesting client, an *audio-send* thread within *RT-Phone*. A shared memory buffer is used between the audio server and the *RT-Phone* thread. The *audio-send* thread then transmits the data to the remote callee across the network. An *audio-recv* thread receives the audio data and outputs it to its speaker through the use of the output audio-server⁴. A mirror image of the same streams transmits audio from the callee to the caller.

4. Reserves, QOS and End-to-End Delays in *RT-Phone*

In this section, we derive our predicted end-to-end delay of the *RT-Phone* application described in the previous section. We also describe the application of the reserve and dynamic quality schemes to *RT-Phone*. Finally, we provide performance measurements of *RT-Phone* which show that we indeed satisfy the predicted end-to-end delays.

4.1. The Quality Parameters of *RT-Phone*

There are five quality parameters associated with *RT-Phone*: end-to-end delay, jitter, audio sampling rate, audio sample size, and audio drop rate. In this paper, we focus on making the audio drop rate being 0 by allocating and guaranteeing necessary resources, and satisfying an acceptable end-to-end delay requirement for different values of audio sampling rate, sample size and the value of T in the pipeline stage. Based on requirements used in the telephony domain, we require that all end-to-end delays be less than 100 ms.

4.2. The Guaranteed End-To-End Delay

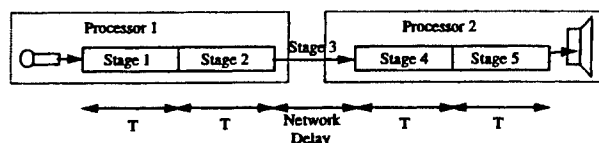


Figure 4-1: The Audio Stream Pipeline Stages in the End-To-End Delay

The pipeline stages involved in the audio transmit path of Figure 3-4 are illustrated in Figure 4-1. It consists of 5 stages:

- Stage 1 represents the filling of its DMA buffer by the

³The AudioFile utility supports a similar server as well.

⁴Jitter control is a critical issue to be dealt with in this context, but is beyond the scope of this paper.

(PAS) sound card. This delay is determined by the parameters chosen for the sampling rate, sample size and buffer size on the sound card.

- Stage 2 represents the "processing" of the data (receive audio data from server and protocol processing for transmit).
- Stage 3 represents the network propagation delay.
- Stage 4 represents the processing on the receiver side (protocol processing for receive and send to PAS card through the audio server).
- Finally, stage 5 is the actual playing back of the audio samples by the hardware.

If T represents the hardware buffering delay for stage 1 (and the same for stage 5), we let stages 2 and 4 have the worst-case timing constraint of T . This is done by assigning reserves to these processor with the reserve period T such that the kernel can guarantee that the processing in these two stages is completed within T units. The end-to-end delay for an audio sample is therefore given by

$$\text{worst-case end-to-end delay} = 3T + \text{network propagation delay}$$

It must be noted that the delay component $3T$ is *not* $4T$. Consider an audio sample which is sampled by the hardware in Stage 1. This sample at offset t from the beginning of the stage will be played at the same offset t in stage 5. The number of T stages inbetween these two points is 3 and not 4.

Since network communications is not a focus of this paper, we use a dedicated network which results in negligible propagation delay - e.g. a typical 256 bytes output every 16 ms at 16KHz sampling with 1 byte/sample takes 0.2 ms at 10 Mbps to transmit but take 16 ms to collect.

4.3. Bundling of Threads to a Reserve

The audio-send thread is normally blocked waiting for data arrival from the audio-server. Consider the flow of control from the arrival of the hardware "buffer full" interrupt through the user-level audio-server to the audio-send thread. Logically, a single consecutive action takes place from the audio server which makes data available to the audio-send thread, which then processes it and transmits it across the network. The timing constraint is that this reception from hardware and succeeding transmission must complete by the interval T .

We define the notion of a *continuous thread of control* as comprising those segments of code where (a) the flow of control is sequential and (b) a strict precedence constraint exists between the segments. The former means that no two segments can run concurrently. The latter means that a segment can start only after, and soon after, its previous segment (if any) has completed. These segments of code can transcend task boundaries but not resource boundaries such as a CPU. Under this definition, a normal OS thread

is always a continuous thread of control since the flow of control is sequential - multiple portions of the thread cannot be active at the same time.

This notion of a continuous thread of control is rather useful in the context of a processor reserve which can be bound to multiple threads simultaneously, as stated in Section 1. All OS threads (or segments) comprising a continuous thread of control typically have a single timing constraint from the first segment to its last segment. A single processor reserve may then be usefully bound to all OS threads comprising the continuous thread of control, and the reserve can be used to satisfy the timing constraint of that continuous thread of control.

Based on this notion, the input thread in the audio-server and the audio-send thread are bound to the same reserve with a period of T . Recall that T is determined by the sound card configuration parameters sampling rate, sample size and buffer size. Similarly, the audio-receive thread and the output thread in the audio-server can be bound to another reserve with a period of T . Our reserve admission control in RT-Mach uses results from real-time scheduling theory [8] which can guarantee the timing constraints of tasks with different periods and computation times. Hence, the reserve period for one direction need *not* be the same as the reserve period for the opposite direction. In other words, the quality of the two output streams at the two speaker devices can be completely independent of one another.

4.4. Performance Measurements

We measured the end-to-end delays actually encountered in *RT-Phone* to validate that the predicted delays are indeed achievable by the application of reserves and the dynamic QOS server. The measurement scheme and the actual measurements are provided below.

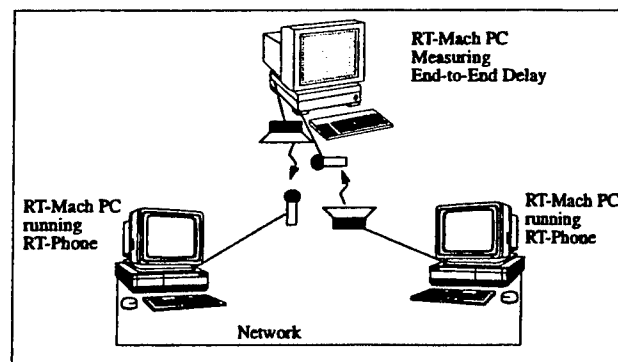


Figure 4-2: Measurement of Audio End-to-End Delay

RT-Phone was run between two machines on a network as illustrated in Figure 4-2. The end-to-end delay measurements were made using a third machine. The measuring machine generated a sharp sound pulse on one machine running *RT-Phone* and then recorded the source pulse and the transmitted output from the destination machine running

RT-Phone. By measuring the timestamp of the generated sound and the timestamp at which the transmitted pulse is heard, the end-to-end delay from the users's true reception is determined to a very high degree of accuracy (within a few *ms*). Note that the audio stream in the direction opposite to the direction being measured is active while the measurement is being done but this is not illustrated in the figure.

Figures 4-3 and 4-4 present measured performance numbers from *RT-Phone* running on Real-Time Mach. Figure 4-3 presents the variation of the end-to-end delays between the two conversation endpoints as the reservation period used for processing the audio samples is varied. This is repeated for various audio sampling rates. Figure 4-4 presents the change in the CPU load (on i486DX 66MHz PCs) as the reservation period and the sampling frequencies are varied. As can be expected, when the reservation period increases, the end-to-end delay increases proportionally. It can also be seen that the worst-case end-to-end delay bound of $3 \times (\text{Reservation Period})$ is satisfied. In fact, the plotted numbers represent the *best-case* end-to-end delay given by the sum of

- The delay to fill up the DMA buffer at the audio source,
- The processing time of the audio data buffer by the input audio server and the audio send thread,
- The network propagation time,
- The processing time of the received audio data by the audio receive thread and the output audio receiver, and in addition,
- any preemption time from the audio stream for the opposite direction.

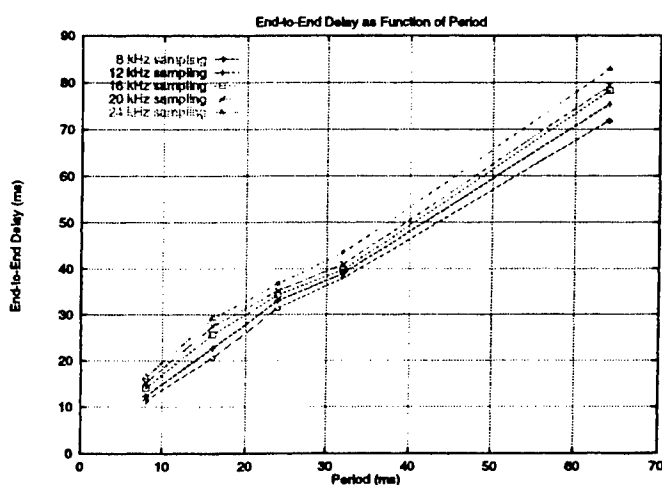


Figure 4-3: The End-To-End Delay w/ varying reservation periods and audio sampling rates.

As the audio sampling rate increases, the processing times increase slightly resulting in a corresponding increase in the end-to-end delay. However, the most significant overhead

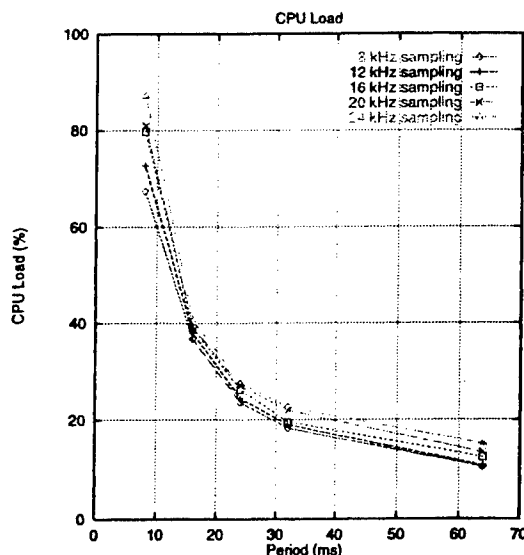


Figure 4-4: The CPU Load with different reservation periods and audio sampling rates.

for threads with small periods is packet sending and receiving overhead. In addition, for small reservation periods, timers are set and reset frequently resulting in higher overhead. Finally, context-switching overhead increases with sampling rate.

One can draw the following conclusions from the performance numbers. First, for a given reservation period in use, the audio sampling rate has very little impact on both the end-to-end delay and the CPU load. For example, when the reservation period used is 24 *ms*, the end-to-end delay is 38 *ms* and 43 *ms* for 8 KHz sampling and 24 KHz sampling respectively, and the total CPU load increases from 24% to 28%. This testifies to the fact that data processing delays (including communication packet processing) dominate data sizes. Secondly, the CPU load increases non-linearly with shorter reservation periods as seen in Figure 4-4, indicating that the much shorter end-to-end delays (below the knee of the curve around 25 *ms*) are obtained at a disproportionately higher cost. One would like to be towards the left-end of Figure 4-3 and the right-end of Figure 4-4. A reasonable compromise for the current hardware configuration lies around a reservation period of 24 *ms*.

4.5. QOS Interface for Changing Quality Parameters Dynamically

RT-Phone also has a user-interface to change the quality of the received audio stream (from the remote participant) and is illustrated in Figure 4-5. When such a quality change is requested, the QOS manager of both the local and remote nodes are contacted to modify the reservations they need for the new quality setting. If granted, the audio-server is requested to change to the new settings. The sender must upgrade quality only after the receiver upgrades its reservation (and be ready for the added incoming traffic). The

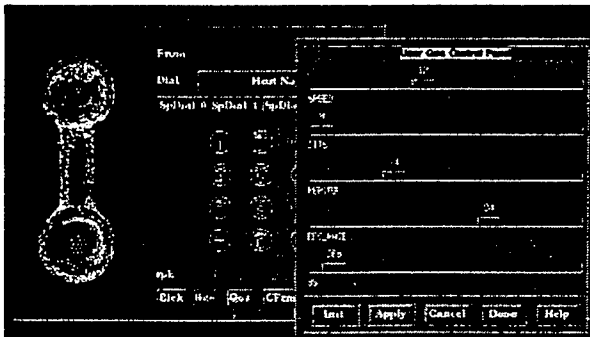


Figure 4-5: The interface to set system quality parameters.

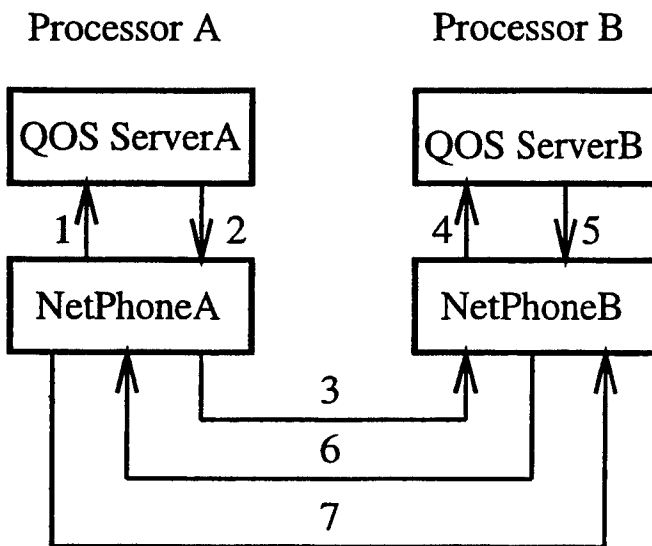


Figure 4-6: The Multi-Phase Protocol to dynamically adjust QoS during teleconferencing

actual quality change must be effected by a multi-phase protocol illustrated in Figure 4-6. Similarly, if the sender degrades quality, the receiver must downgrade its reservation only after the sender does.

Many subtleties underlie such dynamic transitions of network traffic and processing requirements. For example, it is possible that the old resource allocation and the new allocation are completely schedulable (i.e. all their timing constraints can be satisfied with these resources) independently, but timing constraints can be violated during the transition [17]. While this may not be a major concern in non-critical multimedia applications, this loss of schedulability must not lead to all subsequent deadlines to be missed or synchronicity to be lost (say between independently arriving audio and video streams). Of critical importance are the times at which changes to reservation come into effect, the times at which changes to audio card set-

tings come into effect and the need for continued synchronization between the sender and receiver.

5. Lessons Learned and Future Work

We have learnt a number of lessons based on our experiences with the *RT-Phone* application in the context of support for processor reservation and dynamic QOS control in *RT-Mach*. The current processor reserve mechanism seems to provide a sufficiently general means that can support both dynamic real-time and multimedia applications. However, the programming abstraction may have to be raised with "middleware libraries" which map application-level QOS parameters (such as sampling rate) to kernel-level/QOS-server-level parameters (such as computation time and period). The reserve enforcement already yields a built-in framework for measuring computation time (which is dependent on the actual hardware being used). The period must be determined based on the end-to-end delays of the application. More precise control over when a reservation change will be effected would be very desirable. In addition, mechanisms to synchronize the "start times" of reserves which are active in different processors may be critical for networked applications particularly when network load can be variant.

Our future work includes extending the reserve abstraction to address the above requirements and to other resources such as filesystems. A more general set of protocols is needed to coordinate changes in reserves across processors for networked applications. We are currently looking at an audio-mixing server to combine multiple incoming audio streams to a single output device, and its complexity in terms of reservation requirements is much higher compared to an audio-server with a single client. In addition, we are yet to test our QOS framework with multiple clients each with a different kind of dynamic quality change requirements.

Acknowledgements

The authors would like to thank Bryan Hixon, Manish Modh and William Nagy for their first prototype of the QOS server on *RT-Mach*.

References

1. S. R. Ahuja, J. R. Ensor and D. N. Horn. The Rapport Multimedia Conferencing System. Proceedings of the Conference on Office Information Systems, March, 1988, pp. 1-8.
2. Rajkumar, R. "The Advanced Real-time Monitor: User Manual". *Real-Time Mach Project, Carnegie Mellon University* (November 1994).
3. A. Bricker, M. Gien, M. Guillemon, J. Lipskis, D. Orr and M. Rozier. "Architectural Issues in Microkernel-based Operating Systems: the CHORUS Experience". *Computer Communication* 14, 6 (July 1991), 347-357.

4. D. Ferrari and D. Verma. "A Scheme for Real-Time Channel Establishment in Wide Area Networks". 8, 3 (April 1990).
5. H. Gajewska, J. Kistler, M. S. Manasse and D. D. Redell. Argo: A System for Distributed Collaboration. Proceedings of the Second ACM International Conference on Multimedia, Oct., 1994, pp. 433-440.
6. K. Jeffay, D. L. Stone and F. D. Smith. "Kernel Support for Live Digital Audio and Video". (Nov. 1991), 10-21.
7. J. Kurose. "Open Issues and challenges in Providing Quality of Service Guarantees in High-Speed Networks". *ACM Computer Comm. Review* 23, 1 (January), 6-15.
8. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM* 20 (1) (1973), 46 - 61.
9. Mercer, C. W., Savage, S. and Tokuda, H. "Processor Capacity Reserves: OS Support for Multimedia Applications". *IEEE International Conference on Multimedia Computing Systems* (May 1994).
10. C. W. Mercer and R. Rajkumar. Design and Evaluation of a Real-Time X Server. Tech. Rept., ART Project, Carnegie Mellon University, Oct., 1995.
11. K. Nahrstedt and J. Smith. "The QOS Broker". *IEEE Multimedia Spring* (1995), 53-67.
12. Nakajima, T., Kitayama, T., Arakawa, H. and Tokuda, H. "Integrated Management of Priority Inversion in RT-Mach". to appear in *IEEE Real-Time Systems Symposium* (December 1993).
13. Zelenka, J. "The Audio Server: User Manual". *Real-Time Mach Project, Carnegie Mellon University* (November 1994).
14. Adobe Premiere for Macintosh. 1993.
15. Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.
16. P. Robin, G. Coulson, A. Campbell, G. Blair and M. Papathomas. Implementing a OoS Controlled ATM Based Communications System in Chorus. Technical Report MPG-94-05, Dept. of Computing, Lancaster University, March, 1994.
17. Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K. "Mode Changes in a Prioritized Preemptive Scheduling Environment". *The Real-Time Systems Journal* (December 1989).
18. D. B. Terry and D. C. Swinehart. "Managing Stored Voice in the Etherphone System". *ACM Transactions on Computer Systems* 6, 1 (Feb. 1988), 3-27. Also available in Xerox PARC report CSL-89-2, May, 1989.
19. Tokuda, H. The Real-Time Mach Operating System. Carnegie Mellon University, Pittsburgh, PA, September, 1991.
20. H. M. Vin and P. T. Zellweger and D. C. Swinehart and P. V. Rangan. "Multimedia Conferencing in the Etherphone Environment". *IEEE Computer* 24, 10 (Oct. 1991), 69-79.
21. A. Vogel, B. Kerherve, G. Bochmann and J. Gecsei. "Distributed Multimedia and QOS: A Survey". *IEEE Multimedia* (1995).
22. L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala. "RSVP: A New Resource Reservation Protocol". *IEEE Network* (September 1993).

Operating System Resource Reservation for Real-Time and Multimedia Applications

Clifford W. Mercer

June 1997
CMU-CS-97-155

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

*Submitted in partial fulfillment of the requirements for
a degree of Doctor of Philosophy in Computer Science*

Thesis Committee:

Elmootazbellah N. Elnozahy, Co-chair
Ragunathan Rajkumar, Co-chair
Mahadev Satyanarayanan
Hideyuki Tokuda
Kevin Jeffay, University of North Carolina-Chapel Hill

Copyright ©1997 Clifford W. Mercer

This research was supported in part by a National Science Foundation Graduate Fellowship, in part by Bell Communications Research, in part by U. S. Naval Research and Development, in part by the Office of Naval Research, in part by Northrop-Grumman, in part by Philips Research, and in part by Loral Federal Systems. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Bell Communications Research, NRaD, ONR, Northrop-Grumman, Philips Research, or Loral Federal Systems.

Keywords: Resource reservation, Real-time systems, Operating systems, Multimedia applications, Scheduling, Resource management, Real-Time Mach



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

*Operating System Resource Reservation
for Real-Time and Multimedia Applications*

CLIFFORD MERCER

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

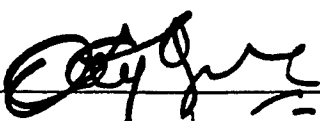
ACCEPTED:



THESIS COMMITTEE CHAIR

6-24-97

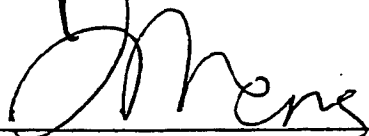
DATE



THESIS COMMITTEE CHAIR

6/30/97

DATE

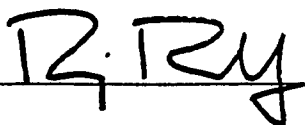


DEPARTMENT HEAD

8/7/97

DATE

APPROVED:



DEAN

8/10/97

DATE

For Franke

Abstract

Increases in processor speeds and the availability of audio and video devices for personal computers have encouraged the development of interactive multimedia applications for teleconferencing and digital audio/video presentation among others. These applications have stringent timing constraints, and traditional operating systems are not well suited to satisfying such constraints. On the other hand, hard real-time systems that can meet these constraints are typically static and inflexible.

This dissertation presents an enforced operating system resource reservation model for the design and implementation of predictable real-time programs. Applications can reserve resources based on their timing constraints, and an enforcement mechanism ensures that they do not overrun their reservations. Thus, reserves isolate real-time applications from the temporal properties of other real-time (and non-real-time) applications just as virtual memory systems isolate applications from memory accesses by other applications. In addition, reserves are first class objects that are separated from control abstractions such as processes or threads. Therefore reserves can be passed between applications, and this model extends naturally to distributed systems.

Reserves support the development of hard real-time and soft real-time programs, and programming techniques based on reserves illustrate how to use them effectively. An implementation of processor reserves in Real-Time Mach shows that reserved multimedia applications can achieve predictable real-time performance.

Acknowledgments

I would like to express my appreciation to my advisors, Mootaz Elnozahy and Raj Rajkumar, for their guidance, technical expertise, and time which they so generously shared with me during my doctoral studies. Thanks also to M. Satyanarayanan, Hide Tokuda, and Kevin Jeffay for their insights, critical comments, and encouragement. I am especially indebted to Hide Tokuda who was my mentor for several years before I entered the graduate program at CMU. Hide started the Real-Time Mach Project, which provided the research environment for this work, and Raj took over the project in the final years of my thesis work. I am indebted to both of them for their support.

I would like to thank Jim Morris for his guidance and assistance. He helped me better understand how to do systems research and served as a role model for me. Thanks go to Roger Dannenberg for the opportunity to work with him and learn from him on several interesting computer music projects (and for the occasional trumpet duet and brass ensemble session).

A special thanks to Joan Maddamma who looked out for me in many ways during my years at CMU. From the time I was an undergraduate through my years as a graduate student, Joan has been a very special friend.

Many other people who I worked with over the years had a positive influence on me and on the work contained in this dissertation. I would like to thank Stefan Savage and Jim Zelenka for their help with pieces of the implementation of reserves in Real-Time Mach and for being good friends and colleagues. And I would like to thank Chen Lee, Prithvi Rao, Andrei Ghetie, Yutaka Ishikawa, Tatsuo Nakajima, Takuro Kitayama, Dan Katcher, Kevin Kettler, John Sasinowski, and Saurav Chatterjee for being part of the invigorating academic environment that makes CMU so rewarding.

I would like to thank my father, mother, and sister for their moral support during my years away at school.

Finally, I want to thank my wife, Franke, who supported me wholeheartedly in the pursuit of my graduate studies. Without her, this work would not have been possible. Thanks also to my daughters, Caroline and Leigh, for their love and patience with me during the final years of the thesis work.

Table of Contents

1	Introduction.....	71
1.1	Motivation	71
1.2	Background.....	72
1.2.1	Programming real-time applications	72
1.2.2	Resource management problems	73
1.3	Resource reserves	75
1.4	Contributions	77
1.5	Overview of the dissertation.....	77
2	Background and Motivation	79
2.1	Real-time and multimedia application requirements	79
2.1.1	Timing characteristics	79
2.1.2	Criticality	83
2.2	Applications timing requirements	86
2.3	Quality of service management	88
2.3.1	QOS background	88
2.3.2	Mapping QOS parameters	89
2.3.3	QOS negotiation.....	90
2.4	System design approaches	91
2.4.1	Specialized hardware	91
2.4.2	Time-sharing systems.....	91
2.4.3	Real-time operating systems	92
2.4.4	Soft real-time system support.....	93
2.5	Reserve abstraction.....	94

3 Reserve Model	95
3.1 Reserve abstraction	95
3.1.1 Reservation guarantee	95
3.1.2 Scheduling frameworks	96
3.1.3 Styles of programming with reserves	96
3.2 Basic reserves	97
3.2.1 Reservation specification	98
3.2.2 Admission Control	99
3.2.3 Scheduling	99
3.2.4 Enforcement	100
3.3 Reserve propagation	103
3.4 Example scheduling frameworks	104
3.4.1 Rate monotonic	105
3.4.2 Deadline monotonic	106
3.5 Basic reserve types	108
3.5.1 Processor	109
3.5.2 Physical memory	110
3.5.3 Network bandwidth	111
3.6 Reserve management	112
3.6.1 Default reserves	112
3.6.2 Composite reserves	112
3.6.3 Reserve inheritance	113
3.7 Chapter summary	114
4 Programming with Reserves	115
4.1 Overview	115
4.2 Using reserves in application design	116
4.2.1 Program structure	116
4.2.2 Reservations for periodic computations	118
4.2.3 Localized reserve allocation	119
4.2.4 Activity-based reserve allocation	122
4.2.5 Coordinating multiple resources	123
4.3 Sizing Reservations	128
4.3.1 Determining resources required	129
4.3.2 Determining initial reservation levels	130
4.3.3 Measuring performance	131
4.3.4 Adapting	137
4.4 Chapter summary	141

5 Implementation	143
5.1 Overview	143
5.2 Reserves in RT-Mach	144
5.2.1 Attributes and basic operations	144
5.2.2 Reservation requests and admission control	146
5.2.3 Scheduling	147
5.2.4 Usage measurement and enforcement	148
5.2.5 Reserve propagation	149
5.3 Applications	151
5.3.1 QuickTime video player	151
5.3.2 MPEG decoder	152
5.3.3 X Server	153
5.4 Reserved network protocol processing	155
5.4.1 Software interrupt vs. preemptive threads	155
5.4.2 Mach 3.0 networking	156
5.4.3 Reserved protocol processing	158
5.5 QOS manager	158
5.5.1 Information sources	158
5.5.2 Admission control	160
5.5.3 Extensions	160
5.6 Tools	161
5.6.1 Reserve monitor	161
5.6.2 Usage monitor	163
5.7 Chapter summary	164
6 Experimental Evaluation	167
6.1 Overview	167
6.2 Predictability	169
6.2.1 Independent synthetic workloads	171
6.2.2 Client/server synthetic workloads	177
6.2.3 QTPlay/X Server	185
6.2.4 mpeg_play/X Server	192
6.2.5 Protocol processing workloads	194
6.3 Scheduling cost	201
6.3.1 Measured aggregate scheduling cost	202
6.3.2 Individual operations	203
6.4 Chapter summary	206

7 Related Work	207
7.1 System Implementation	207
7.1.1 Multimedia support	207
7.1.2 QOS architectures	209
7.1.3 Networking	210
7.2 Scheduling theory	211
7.3 Applications	212
7.3.1 Adaptive applications	212
7.3.2 Tools	212
8 Conclusion	215
8.1 Contributions	215
8.1.1 Resource reservation abstraction	215
8.1.2 Implementation	216
8.1.3 Experimental evaluation	216
8.2 Future directions	217
Bibliography	219

List of Figures

2-1	Schematic of a Time Constrained Computation	80
2-2	Periodic Task	81
2-3	Aperiodic, Predictable Task	81
2-4	Aperiodic, Unpredictable Task	82
2-5	Preemptible Task	82
2-6	Non-preemptible Task	83
2-7	Hard (Catastrophic) Deadline Value Function	83
2-8	Hard Deadline Value Function	84
2-9	Ramped Hard Deadline Value Function	84
2-10	Soft Deadline Value Function	85
2-11	Non-real-time Value Function	85
2-12	Periodic Playback Computations	86
2-13	Playback Application Computing Ahead	87
2-14	Interactive Application with Limited Workahead	87
2-15	Levels of QOS Specification	89
2-16	QOS Levels with QOS Manager	90
3-1	Periodic Scheduling Framework	99
3-2	Enforcement Illustration	100
3-3	Enforcement Timers	102
3-4	Reserve Propagation	103
3-5	Deadline Monotonic Scheduling Framework	107
3-6	Resources and Basic Reserves	108
3-7	A Composite Reserve	113
3-8	Reserve Inheritance	114
4-1	Call Graph for Frame Generation and Display	117
4-2	Thread and Reserve Out-of-Phase	118
4-3	Call Graph with Separate Client and Server Reserves	119
4-4	Switching Reserve from Client to Server	120
4-5	Client Requirement with Intermediate Deadline	121
4-6	Call Graph with One Reserve for All Nodes	122

4-7	Call Graph for Video Playback	124
4-8	Call Graph for Video Playback with Reserves	125
4-9	Synchronization Problem with Multiple Resources	126
4-10	Multiple Resources Used with Intermediate Deadlines	127
4-11	Video Playback with Better Reserve/Computation Mapping	128
4-12	Resource Demand Constant and Reserved	132
4-13	Resource Demand Occasionally Exceeds Reservation	133
4-14	Exceedingly Demanding Computation Aborted	134
4-15	Computation Impinges on Following Computation	134
4-16	Computation Impinges on Subsequent Computations	135
4-17	Average Demand Exceeds Reservation	136
4-18	Resource Demand Smaller than Reservation	136
4-19	Measurements of Multiple Resources	137
5-1	System Components	143
5-2	QTPlay Outline	151
5-3	mpeg_play Outline	152
5-4	Networking with the UX Server	156
5-5	Networking with the Socket Library	157
5-6	Resource Management Schematic	159
5-7	rmon Main View	161
5-8	rmon Detail Views	162
5-9	Modifying a Reservation	164
6-1	Compute-Bound Periodic Task with No Competition	169
6-2	Compute-Bound Periodic Task with Competition	170
6-3	Experiment 1 Results	174
6-4	Experiment 2 Results	175
6-5	Experiment 3 Results	176
6-6	Experiment 4 Results	180
6-7	Experiment 5 Results	181
6-8	Experiment 6 Results	182
6-9	Experiment 7 Results	183
6-10	Experiment 8 Results	184
6-11	Software Configuration	186
6-12	Experiment 9 Results	188
6-13	Experiment 10 Results	189
6-14	Experiment 11 Results	190
6-15	Experiment 12 Results	190
6-16	Experiment 13 Results	191
6-17	Experiment 14 Results	193
6-18	Experiment 15 Results	197
6-19	Experiment 16 Results	199
6-20	Experiment 17 Results	200
6-21	Experiment 18 Results	201
6-22	Scheduling Cost	203

List of Tables

6-1	Summary of Testbed Platforms	168
6-2	Experiment 1 Parameters	172
6-3	Example Parameters for Experiment 2	172
6-4	Example Parameters for Experiment 3	173
6-5	Experiment 4 Parameters	178
6-6	Experiment 6 Parameters	178
6-7	Experiment 7 Parameters	179
6-8	Experiment 8 Parameters	179
6-9	Experiment 9 Parameters	186
6-10	Experiment 10 Parameters	187
6-11	Experiment 11 Parameters	187
6-12	Experiment 12 Parameters	187
6-13	Experiment 13 Parameters	188
6-14	Experiment 15 Parameters	195
6-15	Experiment 16 Parameters	195
6-16	Experiment 17 Parameters	196
6-17	Experiment 18 Parameters	196
6-18	Reserve Switch	204
6-19	Replenishment Timer	205
6-20	Checkpoint Cost	205

Chapter 1

Introduction

This dissertation presents the design, implementation, and experimental analysis of a model for operating system resource reservation. The reservation system supports predictable performance in real-time and multimedia applications, enabling them to meet their timing requirements, and facilitating adaptive resource management. This approach is suitable for real-time programming problems that arise in personal computers and workstations where users may want to run real-time multimedia applications or other real-time programs. The approach is also applicable to embedded system design where better resource reservation abstractions at the system level aid in the design, debugging, and maintenance of such systems.

1.1 Motivation

Recent increases in processor speed and network bandwidth combined with the wide availability of digital audio and video devices have enabled a plethora of multimedia applications and services. Examples of these include audio/video presentation and playback, audio/video phone and conferencing, persistent multimedia data storage services, telephone answering and call management, speech processing, and virtual reality applications.

Stringent timing constraints and large volumes of data characterize these applications. Existing operating systems are not designed to support such applications, especially when real-time multimedia applications execute alongside a non-real-time workload. A key requirement of systems for multimedia applications is predictability, and this means that possible contention for resources must be identified and managed to ensure the timeliness of multimedia data processing and delivery. Although it is possible to manage contention for system resources in an *ad hoc* manner based on the specific requirements of a particular class of applications, this dissertation describes a more structured approach to managing contention based on:

- A reservation model that provides an abstraction for resource capacity

reservation and a system-level mechanism for scheduling resources.

- A higher-level layer for implementing resource management policy using the system-level mechanism.
- Programming techniques for structuring applications in a way that can take full advantage of the resource reservation model.

The reservation mechanism and allocation policy provide abstractions that relieve the system designer from relying on complicated high-level application information to make low-level scheduling decisions. And the programming techniques facilitate the programming of real-time applications that meet their timing constraints.

1.2 Background

Real-time system designers must take timing constraints into account when developing real-time applications and the systems to support them. The programming techniques and resource management policies that have been developed for real-time systems typically apply *ad hoc* solutions for each application area. Several issues and problems arise which can be addressed with appropriate system abstractions.

1.2.1 Programming real-time applications

In applications with timing requirements, the software must be designed to satisfy the timing constraints. The user-level servers and system services used by such applications must be designed with timing constraints in mind. The resource management policies underlying those system services must also be designed to support applications with timing constraints.

Traditionally in real-time system design, application programmers use small, simple operating systems that provide fixed priority processor scheduling, priority queueing for various system resources such as semaphores and mailboxes, and perhaps priority inheritance protocols. The application programmers must then build many of their own real-time system services such as database management systems, file systems, and network communication. These programmers must carefully schedule the various applications running on the system and manage contention for the processor and other system resources. In doing the design and scheduling, the computational requirements of applications must be carefully measured and characterized, and resource sharing must be carefully planned. Applications are therefore very sensitive to the misbehavior of other applications. For example, if a high-priority real-time application has a bug that sends it into an infinite loop, the effect on other applications and the system as a whole would be devastating. The errant application would take over the processor and would not relinquish it, forcing a system crash.

This extreme sensitivity among applications is the result of a lack of suitable system abstractions for effectively managing shared resources in real-time systems. Many abstractions exist in real-time scheduling theories, but typically the assumptions of the theoretical results are implicitly embedded in real systems. An example of an assumption that many

theories require is that the worst-case execution times (WCET) for computations in real-time applications are known at system design time. Most systems designed using analysis techniques that have this assumption do not explicitly check or enforce worst-case execution times for computations. Appropriate system abstractions would explicitly bring the assumptions into the actual system where they could be checked and, in the best case, even enforced.

1.2.2 Resource management problems

Suitable system abstractions could effectively address several problems that arise in real-time system. These problems fall into three broad areas: resource management policies that can satisfy timing constraints, mechanisms to support the policies, and analysis techniques based on the available mechanisms.

Many systems do not provide scheduling policies that directly support real-time resource management. For processor scheduling, most systems provide either fixed priority or deadline scheduling policies. Both of these policies lack complete information about real-time requirements and therefore do not address important problems such as how priority should be assigned or how to prevent overload. The following issues arise in the design of resource management policies:

- **Priority assignment problem:** Simple priority schedulers are hard to use, especially if there are many activities with timing constraints. If there is no global repository of knowledge about the timing constraints of different activities in different applications, there is no basis for deciding what the priority ordering of the activities should be.
- **Overload problem:** To prevent overload, the scheduling policy requires information about real-time constraints such as the computational requirements and frequency of execution. Even if the designer can express the resource and timing requirements for real-time applications, a system with no admission control policy cannot protect itself from overload.
- **Flexibility requirement:** The timing constraints and resource requirements for dynamic real-time applications may change dynamically during execution. The scheduling algorithm must support efficient adjustment of requirements.

To effectively schedule real-time applications such that applications cannot monopolize system resources requires some usage measurement and enforcement mechanisms. Information gleaned through these mechanisms can be used in the scheduling policy to make decisions about how priorities or deadlines should be dynamically adjusted to reflect the requirements and usage patterns of applications. The problems that motivate the use of these mechanisms are described briefly below:

- **Enforcement problem:** An application that specifies resource and timing requirements may accidentally or deliberately attempt to exceed its

stated requirements. This may interfere with the satisfaction of timing constraints for other reserved activities, and it may cause starvation among unreserved activities.

- **Measurement problem:** Enforcement of resource requirements means that resource usage of each application must be accurately measured and compared to the allocation that has been made on its behalf. If the activity uses external modules, servers, and system services, the measurement must include that usage as well.
- **Coordination problem:** Many time-constrained activities are composed of multiple sub-activities implemented by other modules, user-level servers, or system services. Allocating resources for a single activity that spans multiple modules, possibly in different memory protection domains, is complicated. It is necessary, however, to be able to place timing requirements on the overall activity and to track the resource usage for the overall activity.

Other problems deal with higher level issues of how to analyze system behavior given more sophisticated abstractions and mechanisms for scheduling and resource management. The following issues arise in this context:

- **Distributed real-time scheduling problem:** An activity that uses external modules and services may require the use of multiple resources within certain time constraints. Coordinating usage across multiple resources to meet timing constraints is a hard problem.
- **QOS management problem:** Real-time applications may dynamically change their quality of service (QOS) requirements. In a system with many such applications, a high-level resource manager (sometimes called a QOS manager) is needed to resolve conflicts and negotiate between applications.

In traditional real-time systems design, many of these problems are avoided in the design phase by careful measurement and analysis of simple computations and their resource requirements and by ad hoc scheduling techniques. This approach results in inflexible systems that are difficult to maintain [43,68]. In particular, supporting dynamic real-time activities with timing constraints and resource requirements that may change freely at run-time stretches the traditional approach beyond its limits.

Recent work in real-time systems addresses some of these problems. Several systems (e.g. [3,53,113,124,125]) allow specification of timing requirements instead of forcing the programmer to determine an appropriate priority assignment. A few systems have on-line admission control policies [3,78,113], but many others use off-line analysis [53,124,125]. Still others have no admission control at all [19,21,90]. Some limited flexibility requirements for hard real-time have been addressed recently [122]. This work focuses on mode changes, which are radical but infrequent changes in the task sets of a real-time system. For example, the real-time system in an airplane might experience a mode change after takeoff as it switches from the ground-based task set to the air-based task set.

Very few software systems address the measurement and enforcement problems with respect to resource usage although most systems can detect and react to missed deadlines [43,125]. In networks, the notion of enforcement or policing is much more common [33,98,128]. The work on priority inheritance protocols [8,16,51,86,97,108] addresses some aspects of the coordination problem.

The distributed real-time scheduling problem is an active area of research [38]. The QOS problem is another active area of research. Some operating system research in this area focuses on best effort approaches [21] although other research emphasizes guarantees [46,53,78,81,126].

1.3 Resource reserves

This dissertation defines a resource reservation model that provides an operating system abstraction called a reserve. Reserves explicitly represent reservations on resources such as processors, memory pages, disks, and network devices. In particular, reserves support

- specification of reservation parameters,
- admission control,
- scheduling based on timing constraints and usage requirements,
- reservation enforcement,
- reserve propagation in the RPC mechanism,
- flexible binding of threads to reserves.

Reserves prevent applications from over-running their allowed resource usage and interfering with other reserved activities or starving unreserved activities. Applications reserve capacity on the resources they need to carry out their computations. For example, an application can reserve 10 ms of computation time on a processor for every 100 ms of real-time. The application then binds to the reserve, and the processor scheduler uses the information associated with the reserve to control the scheduling of the application. The system also performs an admission control test before granting the reservation to make sure that the resource can support the reservation being requested. The enforcement mechanism ensures that an application does not use more than its reserved time if doing so would interfere with other reserved activities.

The reservation parameters associated with an application's reserves are not necessarily fixed for the lifetime of the application. A dynamic real-time application must be prepared to change its behavior and timing requirements based on changing requirements of users and possibly the changing availability of resources. A user may want to change the frame rate on a video player or change the resolution, and the application must be ready to adjust its resource reservation levels appropriately. Likewise, reserves must support an operation that modifies the reservation parameters, subject to the admission control policy.

Reserves are first class objects in the operating system; a reserve is associated with a particular thread (or process) by explicitly binding the thread to the reserve. This allows an

application to reserve all of the resources it will need for its computation, including resources that will be needed by various modules, servers, and system services it intends to invoke. The application can then pass references for its resource reserves to modules or servers along with the operation invocation. The module or server can bind its thread to this reserve when performing the operation, and it can then take advantage of the resources that have been reserved by the client. Having modules and servers charge a caller's reserve for work done on behalf of the caller also maintains a consistent view of the resource usage that is being consumed on behalf of that client.

The reserve model presented in this dissertation addresses the problems identified in the previous section. The scheduling policies embedded in the reserve model address the priority assignment problem and the overload problem while remaining flexible in terms of accommodating dynamic changes in application resource requirements as discussed below.

- **Priority assignment problem:** Reserves avoid the priority assignment problem by accepting reservation specifications that include the timing constraints and usage requirements.
- **Overload problem:** The admission control policy of the reservation mechanism prevents overload. This is possible since the scheduling policy has information about both the computational resource requirements and the timing constraints (such as period of invocation) for each application.
- **Flexibility requirement:** Changes in reservation parameters can be made at any time, subject to the admission control policy.

The reserve model makes use of several system mechanisms that support the abstraction. These mechanisms provide information about resource usage and that coordinate the consumption of resources for an activity that crosses memory protection boundaries as follows:

- **Enforcement problem:** Reservation enforcement isolates reserved activities from undue interference from other reserved activities.
- **Measurement problem:** The flexibility in binding reserves makes it possible to accumulate resource usage charges for an activity even when work is done by external modules, servers, or system services.
- **Coordination problem:** The reserve model supports reserve propagation which includes a "priority" inheritance mechanism and which takes advantage of the flexibility in binding reserves to threads.

The reserve model provides an abstraction that can be used for distributed real-time scheduling and QOS management. The approaches to these problems are described briefly below:

- **Distributed real-time scheduling problem:** The reserve model supports reservations for remote resources, and pipeline-style software architectures are supported. This is not optimal, but future work on this problem could take advantage of the reserve model as a base.

- **QOS management problem:** A high-level QOS management layer can use reserves to carry out resource allocation policy decisions. The QOS managers can carry out their allocation decisions by manipulating reservation parameters for the applications being managed.

1.4 Contributions

This dissertation describes and analyzes a resource reservation solution to the problem of supporting predictable execution of real-time and multimedia applications with specific quality of service parameters. It shows that:

Resource reserves, an enforced operating system resource reservation abstraction, effectively supports real-time and multimedia applications. Reserves allow the software designer to specify timing requirements on resources required, thus providing a method for guaranteeing deadlines in real-time applications. The reservation abstraction accommodates non-real-time applications as well as real-time applications.

This dissertation defines an enforced resource reservation model called reserves and then describes programming techniques for developing applications using reserves. It presents an implementation of one type of resource reserves, processor reserves, in Real-Time Mach along with several real-time applications that use reserves to satisfy their timing constraints. These applications included a suite of synthetic benchmark programs, a QuickTime video player, an MPEG player, and a version of the X server. Experiments with these applications showed that reserves can indeed provide predictable behavior for real-time applications even with competition from other real-time and non-real-time applications.

Reserves are a tool that real-time system designers can use to raise the level of abstraction for resource scheduling in real systems. This allows applications' timing requirements to be defined and guaranteed in isolation with the system providing high-level guarantees that resources will be available when needed by each real-time application.

1.5 Overview of the dissertation

Chapter 2 describes the background and motivation for the dissertation in more detail. Chapter 3 describes the reserve model, and Chapter 4 discusses techniques for structuring programs to best take advantage of reserves.

Chapter 5 describes the implementation of processor reserves in Real-Time Mach, a quality of service (QOS) manager, and several reserved applications and servers. Chapter 6 presents an experimental evaluation that explores the predictability achieved by using reserves and the overhead involved in providing that predictability.

Chapter 7 discusses related work, and Chapter 8 summarizes the contributions of the dissertation and presents the conclusions and future directions.

Chapter 2

Background and Motivation

This chapter discusses the requirements of real-time and multimedia applications and describes some of the problems system designers encounter in attempting to support such applications. The case is made for an operating system resource reservation approach to the problem.

2.1 Real-time and multimedia application requirements

Real-time applications require not only that computations result in logically correct answers, but that the answers are available within certain timing constraints. A logically correct answer that arrives late is considered incorrect in a real-time system [114]. Many multimedia applications have this property that late computations are useless. For example, if a video frame or audio sample arrives after the time at which it was to be displayed or played, it is no longer useful.

This section gives an overview of different kinds of timing constraints and criticality characteristics of real-time and multimedia applications. The models described here represent a compendium of the models that researchers have addressed in the literature.

2.1.1 Timing characteristics

In general, a task in a real-time system has timing constraints that specify when the computation may begin, how long it executes, and the deadline for the completion. Figure 2-1 illustrates a computation schematically. This generic model is common in the operations research literature [20]. The computation has a ready time, r , at which the computation becomes available for scheduling. At some point after the ready time, the computation will be scheduled and start processing for a total duration of C . The duration between the ready time and the start of processing is enclosed in a white box. This indicates that the task is available for scheduling but has not started yet. The black box represents the computation that completes at time E . A deadline, d , may be associated with the computation as well, and

the goal is to complete the computation before the deadline. In Figure 2-1, a thick vertical line represents the deadline.

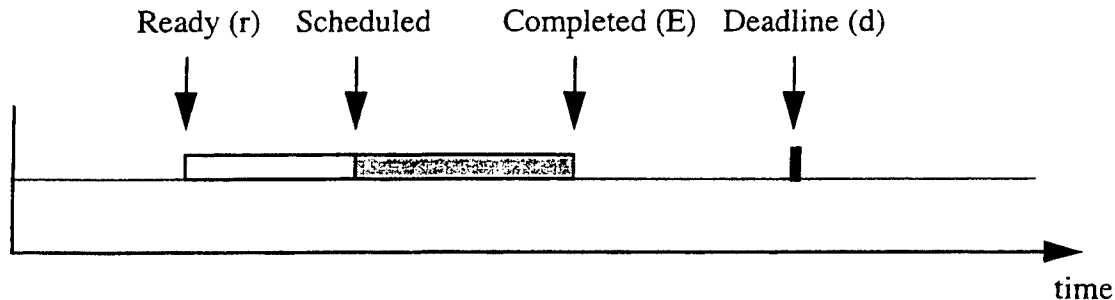


Figure 2-1: Schematic of a Time Constrained Computation

The ready time of a computation may arise from a clock event, an external interrupt, or a software event generated by some other computation. The ready event may be an instance of a periodic computation where the same computation is activated periodically. The ready event may be aperiodic but predictable, or it may be unpredictable. The computation time may be fixed or it may be variable or unpredictable. The computation itself may be preemptible, or it may form a non-preemptible critical region. The deadline is usually some fixed duration after the ready time, but the implications of missing a deadline may vary. Hard real-time computations take the deadline to be a *hard* deadline where the computation must be complete by the deadline time. Alternatively, the deadline may just be a recommendation or preference for completion of the computation, a *soft* deadline.

Since a computation may be periodic, we must sometimes distinguish between the overall activity and the periodically occurring computations. We call the overall activity a *task*, and we refer to an instantiation or individually scheduled computation of the task as a *job*; thus a task is a stream of jobs. The jobs of a particular task are considered to be identical for the purposes of scheduling although variations can be indicated by a variable or stochastic computation time. We will use the word task to mean both the stream of instantiations and an individual instantiation when such usage is clear from the context.

A periodic task has ready times for the task instantiations separated by a fixed period. Periodic tasks are the main focus of the original rate monotonic scheduling work [67] and the many extensions that have followed [63,108]. Figure 2-2 shows a periodic task. The figure shows four instantiations, each with an associated ready time, r_i . The ready times are separated by the period τ . The computation time is represented by the black box, and the preceding white box represents the time when the task is ready by has not yet been scheduled to execute. In this example, the computation time is constant across task instantiations, and the deadline is at the end of the period.

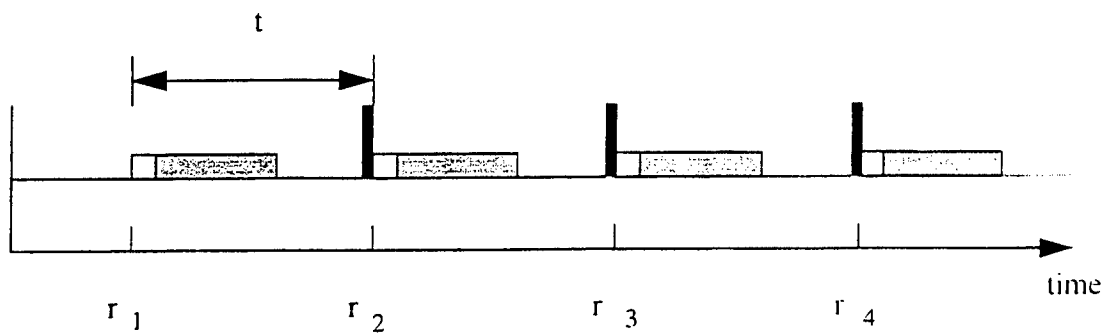


Figure 2-2: Periodic Task

Aperiodic tasks are more difficult to specify. Some aperiodic tasks are predictable to a certain extent; it may be possible to predict the arrival of instantiations of an aperiodic task within some scheduling horizon of h time units. Figure 2-3 shows an aperiodic task with a scheduling horizon of duration h from the current time. This kind of timing requirement is used in computer music, for example [5,25]. Within this window of h time units, the ready times of instantiations of the task are known, but beyond the horizon, nothing is known of the behavior. The computation time is assumed to be constant across instantiations in the single task, and the deadlines are left unspecified.

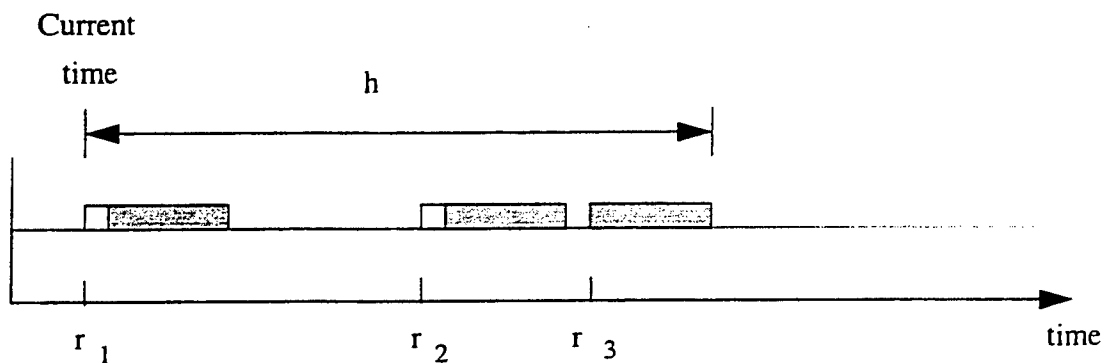


Figure 2-3: Aperiodic, Predictable Task

Another class of aperiodic tasks is almost completely unpredictable. It is common, however, to associate a minimum interarrival time for the instantiations of these unpredictable aperiodic tasks. Much work has been done on scheduling aperiodic tasks with soft deadlines [120] and on aperiodic tasks with hard deadlines, which are known as *sporadic* tasks [52,111]. Figure 2-4 illustrates an aperiodic task where the arrivals are unpredictable.

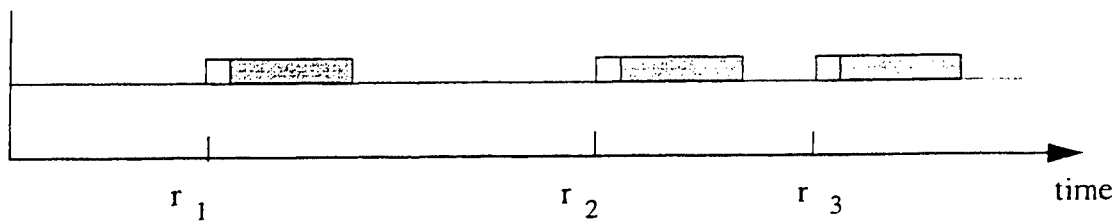


Figure 2-4: Aperiodic, Unpredictable Task

The computation time is another dimension along which tasks may vary. The computation time may be fixed or merely bounded in duration. The computation could also be described by a statistical distribution, but that case is much harder to analyze.

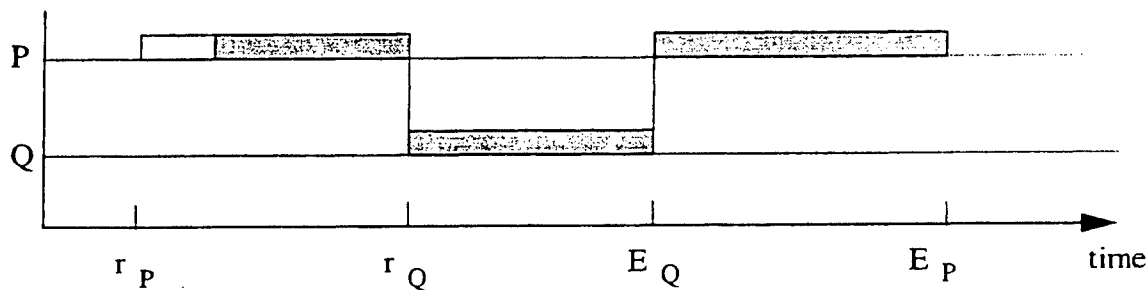


Figure 2-5: Preemptible Task

Another characteristic of the computation is its preemptibility. It may be completely preemptible (that is preemptible at any point) or it may be non-preemptible. Or it may be preemptible but with one or more non-preemptible critical regions during which scheduling events are not allowed (possibly during system calls for example). Different assumptions are made to achieve different results [67,72,82], and in particular, much work has been done on handling non-preemptible critical regions [8,16,51,97,108]. Figure 2-5 shows an example of a preemptible task, P, and its interaction with another task, Q. For this example, we assume that P is preemptible and has a lower priority than Q. P becomes ready at time r_P and begins to execute immediately. At time r_Q , Q becomes ready, and since Q has priority over P, Q preempts the ongoing execution of P. After Q completes, the execution of P resumes.

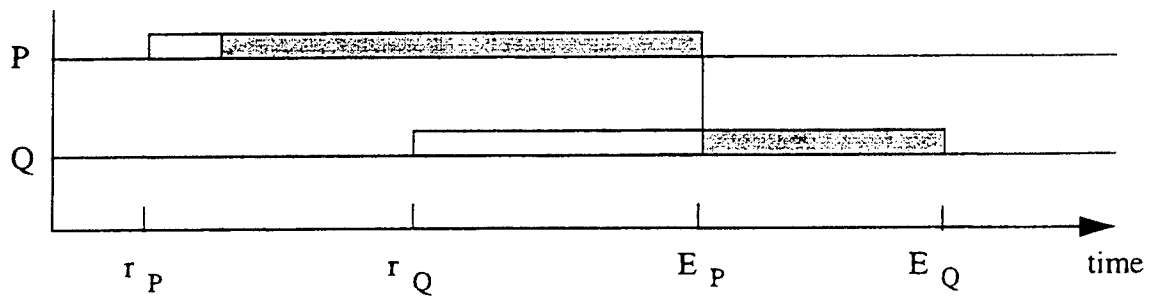


Figure 2-6: Non-preemptible Task

Figure 2-6 illustrates a similar case where the computation of P is non-preemptible and where Q has priority over P. P becomes ready at time r_P and begins to execute. Q becomes ready at time r_Q , but even though Q has priority over P, P cannot be preempted, and Q must wait until the execution of P completes. After P is finished, Q can begin execution.

2.1.2 Criticality

Deadlines may be classified as hard or soft. We can describe various types of deadlines by means of a value function. Value functions have been used for scheduling [15,55], but here they are used for purposes of exposition. A value function is a function of time that indicates the value that completion of the task would contribute to the overall value of the system. For example, Figure 2-7 shows the value function of a task that has a hard deadline: the value drops off to negative infinity at $t = d$. The task becomes ready at time r , and its deadline is d . If the task is completed at time t where $r \leq t \leq d$, then the system receives some value, V . On the other hand, if the task completes after d , the value is negative infinity, a catastrophic failure.

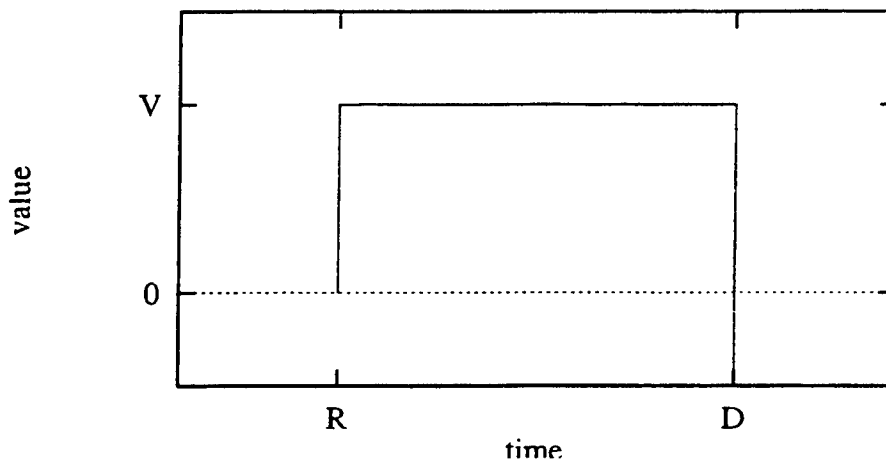


Figure 2-7: Hard (Catastrophic) Deadline Value Function

The result of missing a hard deadline may not be catastrophic. Figure 2-8 shows a case where completion of a task would have some value until the deadline d when the value of completion of the task goes to zero. This indicates that the system will receive no benefit from completing the computation after d , and so the task should be aborted, freeing any resources it holds. In contrast to the previous case, the system can continue to operate, achieving a positive value even if this particular task is aborted and makes no contribution to that value.

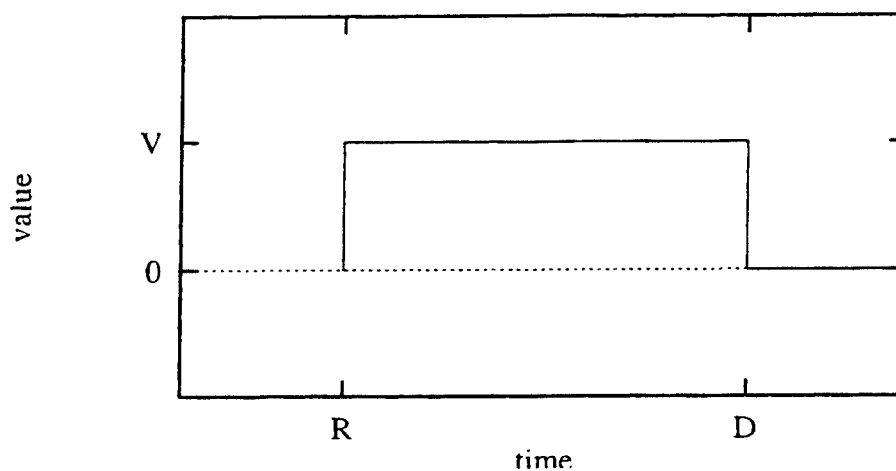


Figure 2-8: Hard Deadline Value Function

Other variations on the idea of hard deadline might include a value function that ramps up to the deadline as illustrated in Figure 2-9. And depending on where the ramp starts, this type of value function can specify tasks that must be executed within very narrow intervals of time.

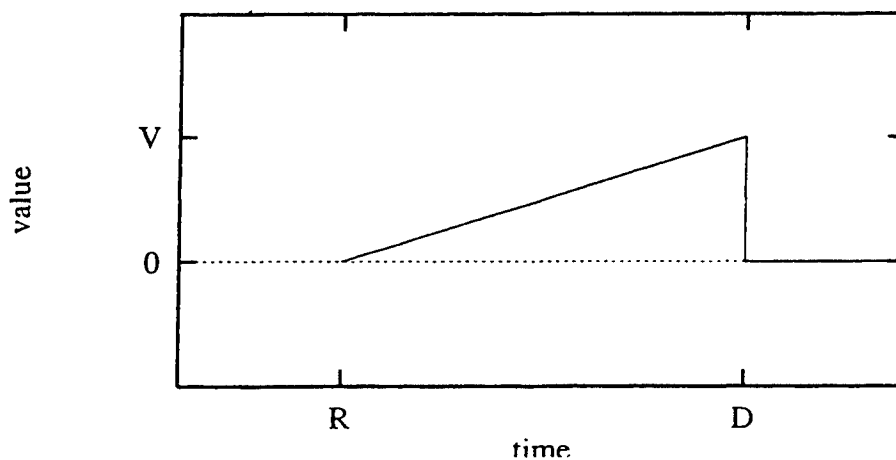


Figure 2-9: Ramped Hard Deadline Value Function

The concept of a soft deadline is illustrated in Figure 2-10 where the value function goes gradually to zero after the deadline. In this case, there is some value to the system in completing the task after the deadline, and the task should not be aborted right away as in the case of the hard deadline.

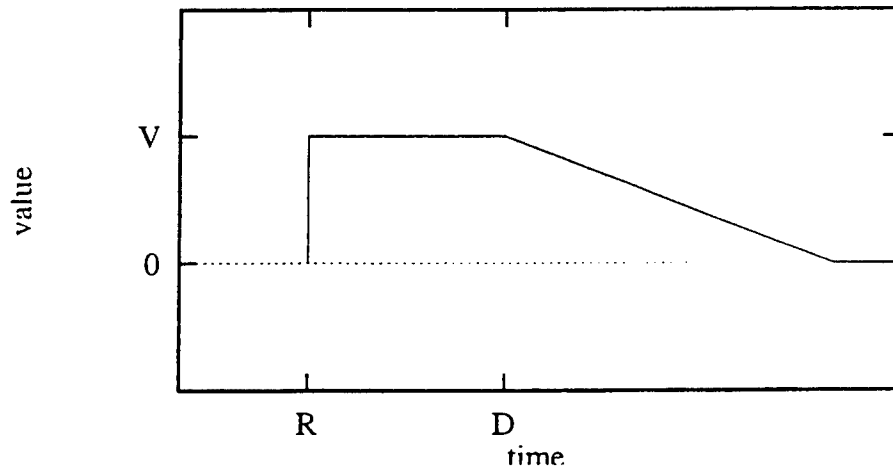


Figure 2-10: Soft Deadline Value Function

A non-real-time task might be described by the value function shown in Figure 2-11. In this case, completion of the task always has a positive value associated with it. This indicates no explicit timing constraint, although in practice, few of us are willing to wait indefinitely for a computation to complete.

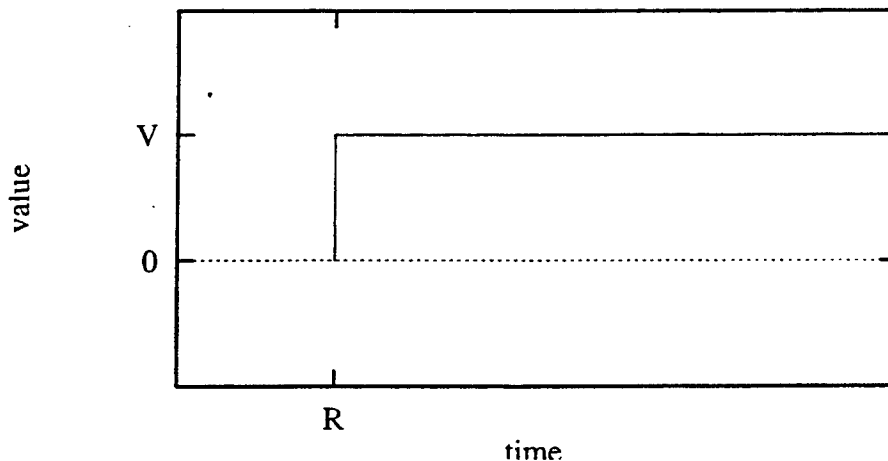


Figure 2-11: Non-real-time Value Function

2.2 Applications timing requirements

Many multimedia applications have real-time constraints. Even simple playback applications have real-time constraints that must be satisfied. For example, an audio player application might repeatedly read audio data from a file on disk and then enqueue the data for the audio device. Figure 2-12 illustrates the periodic computational requirements of such a playback application. The activity of the player is illustrated on the line labeled “P”, and the activity of the device is represented on the line labeled “D”. In each period, the applications reads, processes, and enqueues the data to a device. At the end of each period, the device reads the data out of its buffer, performs D/A conversion, and the analog signals goes to a speaker.

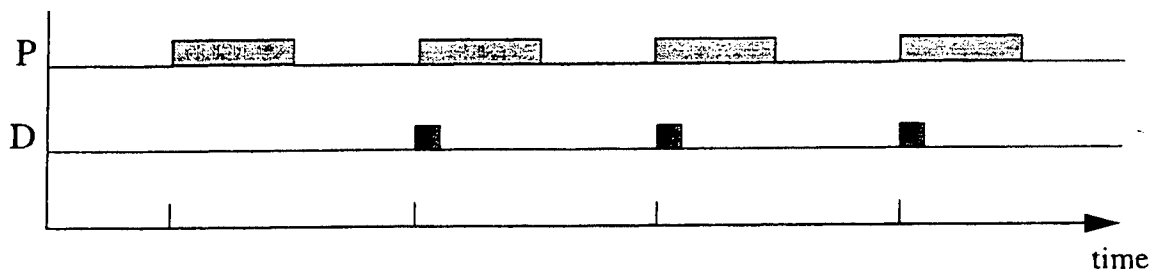


Figure 2-12: Periodic Playback Computations

A potential problem is that the computation of the audio player may be delayed so much that the device buffer empties and there are no samples for the device to convert to an analog signal. For a playback application, one solution is to introduce a large buffer and allow the playback application to execute for many periods to build up a large buffer of data ready to be played by the device.

Figure 2-13 shows a player with execution history shown on the line labeled “P”. The player buffers a number of data blocks for a device; the size of the data is indicated in the area labeled “B” between P and D. The device consumes data from the buffer, and the activity of the device is indicated on the line labeled “D”. Even if the audio player is delayed for a period or two, there will still be plenty of data in the buffer and the device will not run out of data blocks. The player can catch up with the processing that was delayed.

Another potential problem is that if some other activity on the machine is very active and manages to deprive the audio player of the resources (like processor time) for a very long time, then audio playback will be noticeably disturbed. This kind of intense competing activity can be avoided, and with large buffers audio playback will be quite smooth.

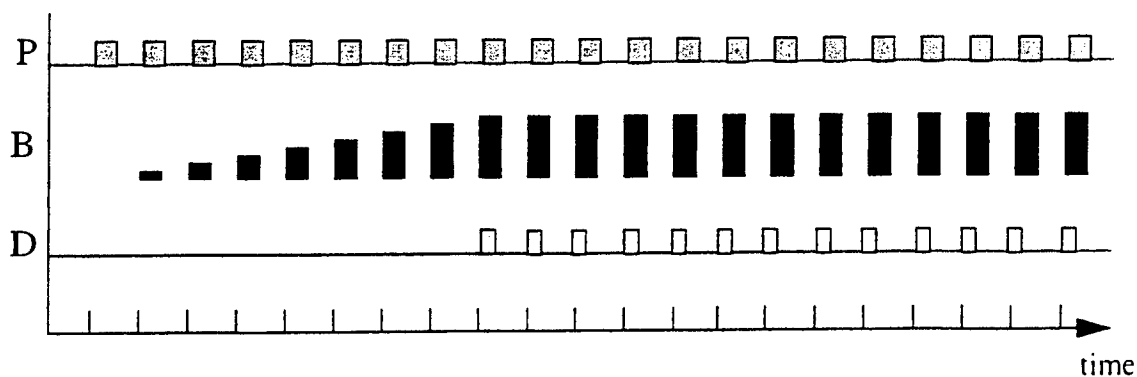


Figure 2-13: Playback Application Computing Ahead

Interactive applications cannot afford to use large buffers to smooth variations in scheduling delay. The delay introduced by large buffers is often too great to satisfy stringent end-to-end delay bounds in interactive applications like video teleconferencing. Instead, the systems must support applications that can ensure that bounds on the scheduling delay are observed. Thus the variations can be reduced and less buffering is required.

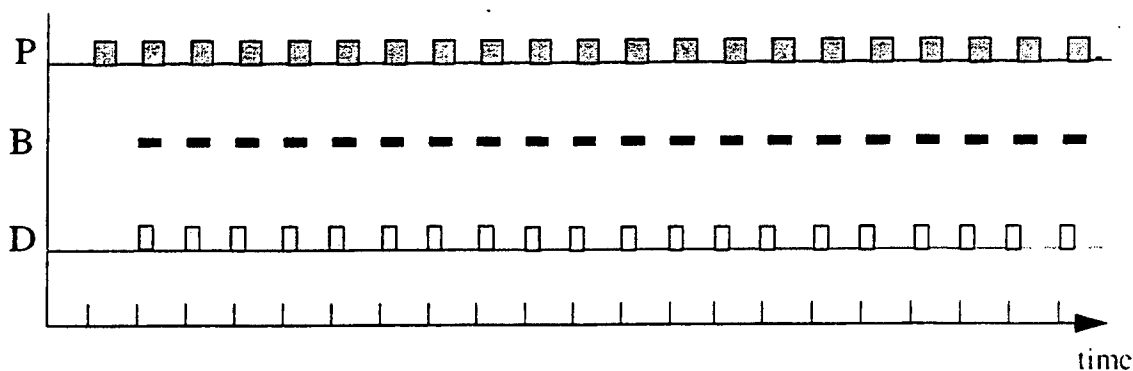


Figure 2-14: Interactive Application with Limited Workahead

Figure 2-14 shows the case where an interactive application can buffer the data generated in one period, but since there is a delay constraint, the application cannot afford to compute ahead several periods as in the previous case. The buffer must remain small. In order to make sure that the buffer is never empty when the device goes to get the next block of data, the application must make sure that the computations to enqueue the next block of data are done in time. This means it must also make sure that resources it will need from the system are available in time as well.

Other multimedia applications have timing constraints that differ from the periodic timing of the stream-oriented applications described thus far. For example, computer music applications involve performing computations to generate musical notes and other musical events. The number of notes that need to be generated at any point in time may vary widely depending on what the music calls for, so there is little periodic structure to describe the computations to be executed.

Silence-suppressed audio presents a similar problem. For audio conversations, it is not always necessary to transmit data for the silent portions of the audio streams in a two-way conversation. Again, the computational requirements become aperiodic when the amount and timing of data depend on speech patterns.

There are other applications, such as compressed video players, where a periodic execution pattern exists but where the computation time required within the periods varies. Compressed video data contains frames whose size varies according to the compression algorithm and the characteristics of the scene and how fast scenes are changing in the video source.

2.3 Quality of service management

Many multimedia applications have timing requirements and other quality of service (QOS) parameters that represent the user's desires and expectations for the performance of the applications. The complexity of providing for these timing requirements at the system level is exacerbated by the fact that the user may change those timing requirements at any time during the execution of the applications; and of course the user may create and terminate multimedia applications at any time.

2.3.1 QOS background

In recent years, researchers in the computer networking and in the telecommunications communities have been working on ways to express the QOS requirements for multimedia applications. Some of this work dealt with human perception requirements for various media [30,117], and other work focused especially on parameters and QOS architectures that are important in the context of scheduling traffic on a network [14,88]. This work can be considered an extension of the earlier work done in the telephone companies to characterize quality of service for telephone customers [100].

As researchers gained more experience with the idea of building networks that could provide quality of service levels suitable for different types of multimedia traffic, the question of how to ensure quality of service levels for end-to-end applications arose. Achieving that goal means QOS requirements must be supported in the operating system as well as the network. This observation was an initial motivation for the work described in this dissertation [76], and other system designers have started to focus on this aspect as well [3,21,46,53,83,102,126].

2.3.2 Mapping QOS parameters

In dealing with QOS management it is important to realize that there are different types of QOS parameters for different levels of the system. Applications must interact with the user in terms of user-level QOS parameters. For video, these user-level parameters might include frame resolution (width and height of each frame), number of bits/pixel, frames per second, maximum delay, and maximum jitter. For audio the user-level parameters might include sample rate, sample size, maximum delay, and maximum jitter. These are the types of parameters that might be meaningful to the user of a multimedia application. Or it might be preferable to offer QOS levels with names like: “worst”, “fair”, “good”, “better”, and “best” to simplify the interface. It would then be up to the application to translate these abstract QOS specifications to frame rates and sample rates.

Once the user-level QOS parameters are determined, they have to be mapped into system-level QOS parameters that would be meaningful for system-level resource management mechanisms. These system-level QOS parameters would describe how much time is needed on various resources. They depend on the user-level QOS parameters and on detailed computations that the application performs on data elements in the media stream.

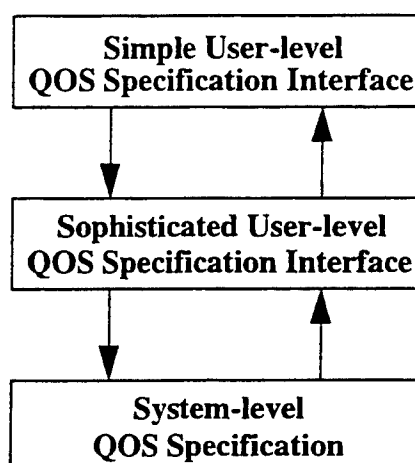


Figure 2-15: Levels of QOS Specification

Figure 2-15 summarizes these levels of QOS specification. The arrows in the figure indicate that there are mappings from one level to the next lower level and also that there are inverse mappings that come into play as well.

To allow the user to specify the QOS parameters desired at the highest level, the application must be able to map from user-level QOS parameters to system-level QOS parameters. The system-level parameters are required for the application to be able to ask for the resources it will need to execute. If the resources are unavailable, the system-level resource management mechanism should be able to communicate the fact that those parameters cannot be guaranteed. It should then initiate a negotiation to arrive at a set of system-level

parameters that can be supported by the system. The inverse mapping to user-level QOS parameters should yield a QOS specification that can be tolerated by the user. Thus, the inverse mapping from system-level to user-level QOS parameters is just as important as the forward mapping.

2.3.3 QOS negotiation

The user's QOS requirements may sometimes conflict with resource usage limitations, and therefore the QOS layer may need to negotiate user requirements to resolve such conflicts. This negotiation process may be needed throughout the lifetime of certain applications since user-level QOS requirements may change over the course of execution.

The approach advocated in this dissertation for handling the complexity of a dynamic execution environment (where programs may have changing real-time requirements) is to divide the problem into two parts. One part is the dynamic negotiation of resource allocation. The second part is the resource reservation and scheduling based on the allocation. These two parts can be addressed with a layering of functionality in the system where a system-level QOS management layer handles the resource allocation policy decisions and a low-level operating system resource reservation mechanism handles the details of dynamic scheduling and usage enforcement.

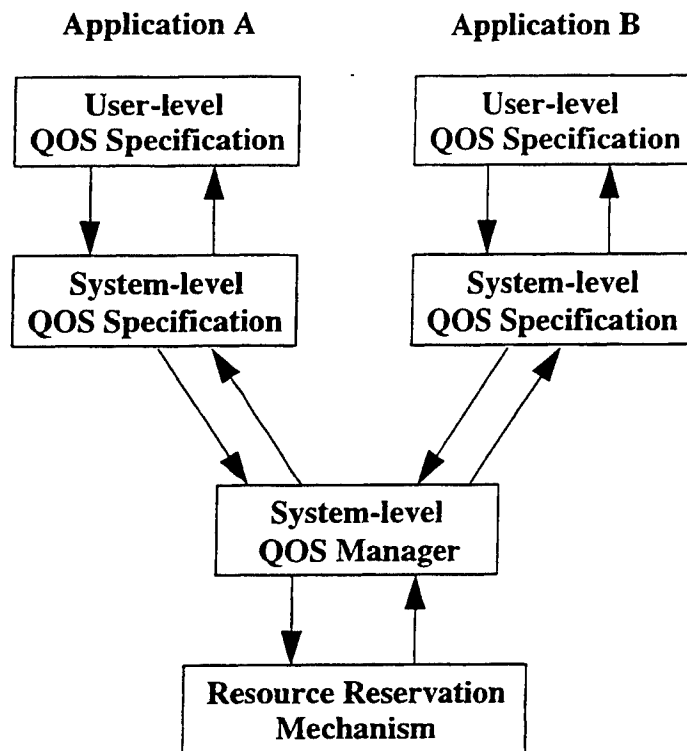


Figure 2-16: QOS Levels with QOS Manager

An illustration of the basic outline of the QOS management system appears in Figure 2-16. Each application has user-level QOS specifications. The applications map the user-level QOS parameters in to system-level QOS parameters and then negotiate with the system-level QOS manager to determine a mutually acceptable set of system-level parameters. The operating system contains a resource reservation mechanism which is used by the system-level QOS manager to actually allocate the resource capacity, schedule appropriately, and enforce the resource reservations.

The QOS management layer makes policy decisions about where resource capacity in the system should be allocated. To do this, it will depend on input from applications as they make their system-level QOS parameters known. The QOS manager may take input from user preferences expressed in the form of rules about which applications are more important than other applications, and it may take input from user interface tools designed to help the user manage resource allocation in the system. The QOS manager may also coordinate with other QOS managers on remote hosts for setting up distributed multimedia applications that require resources on several different hosts.

2.4 System design approaches

Several approaches have been used in the past to support interactive and playback types of multimedia applications. These range from hoping for the best, to dedicating expensive resources, to system support for real-time programming.

2.4.1 Specialized hardware

One approach to supporting real-time multimedia applications is to dedicate hardware to the tasks that must be performed in real-time. This relieves any contention for resources. As an example, Pandora's Box [44] was an early attempt to support multimedia applications in the context of a desktop workstation. The box contained six transputers, each one dedicated to a particular activity or class of activity including audio processing, storage/disk management, video processing, decompression, network communication, and bus management. This box was connected to a Sun workstation, to a network, and to a monitor. It coordinated graphical display from the workstation and video streams from the network or other devices, combining them and displaying the result on the monitor. The system allowed researchers to learn much about programming multimedia applications, what kinds of applications were useful, and user interface issues. However, the cost of the box was very high, and the complexity of programming the box itself was also great.

2.4.2 Time-sharing systems

A number of multimedia applications are available for personal computers and desktop workstations that run more sophisticated operating systems like UNIX and Windows NT. These systems use time-sharing scheduling policies that are not particularly well suited for meeting the timing constraints of multimedia applications.

Time-sharing schedulers are tuned to provide fair allocation of resources among users which are considered equally important. These schedulers also look at whether a process is compute-bound or not, and they depress the priority of compute-bound processes in favor of interactive (or I/O-bound) applications which can benefit from better response times.

This kind of scheduling behavior works well in mainframe systems, but may work against multimedia applications. For example, a video application that performs a filtering computation on video frames may look compute-bound to a time-sharing scheduler and may therefore get a low priority compared to network and I/O activity on the machine. This may occur even if the video application is the most important activity to the user.

Consider a teleconferencing application that displays several video streams on the screen at the same time. A fair time-sharing scheduling algorithm would give each of these streams an equal share of the processor, resulting in the same frame rate for all of the video displays. This might not be appropriate for the particular application. The user might want more control over where resources are focused, perhaps to show a higher frame rate for the person in the conference who has the floor.

In many operating systems such as UNIX [62], VMS [80] and Windows NT [24], the time-sharing scheduling policy is augmented with a fixed priority extension. The extension is usually in the form of a range of fixed “real-time” priorities. With fixed priorities, it is possible to exercise more control over how the processor is scheduled, but there are other problems. Many of these issues arise in the context of real-time operating systems as well, where fixed priority scheduling is commonly used. The next section addresses some of the difficulties of real-time programming with fixed priority schedulers.

2.4.3 Real-time operating systems

Much work in recent years focuses on how to apply real-time systems techniques to multimedia systems and applications. This includes work directed at methods for using technology available in commercial real-time operating systems as well as efforts to build research prototype operating systems.

2.4.3.1 Commercial real-time systems

Most commercial real-time operating systems support fixed priority (FP) scheduling of processes [99,104]. FP schedulers while useful for real-time scheduling, cannot by themselves support multimedia applications.

For example, fixed priority schedulers have no mechanism for dealing with overflow situations. In general, real-time operating systems do not have the mechanisms for deciding whether enough resources are available in a system to run a new application; they do not have support in the system for admission control. Furthermore, there are no mechanisms for detecting and dealing with overload situations when too many applications are allowed to run. These issues of load management are typically addressed in an ad hoc manner by subsystems specific to particular applications.

Even if a system user could determine that a particular collection of applications could run successfully on a system, the problem of determining what priority assignment should be used remains unsolved. An application designer may use multiple threads or processes in the implementation of the program, and that designer will undoubtedly know enough about the processes' computational requirements, timing constraints, and precedence relationships to assign priorities in a reasonable way. However, when running several such applications developed by different people on the same system, the question arises: How should priorities be assigned to processes in different applications in a way that will result in correct timing behavior? Without global knowledge of all processes and their timing constraints, assigning priorities appropriately is exceedingly difficult.

In practice, commercial operating systems are used mainly in embedded applications where designers carefully measure the resource requirements and coordinate scheduling based on a scheduling analysis of the specific task set designed for the application [68]. The system designers have global knowledge about resource requirements, and they use that information in the scheduling analysis to generate a priority assignment. This makes the systems rigid and difficult to maintain. Much more flexibility would be desirable.

2.4.3.2 Research prototype real-time systems

Several research prototype operating systems have applied results from real-time scheduling theory to multimedia applications [3,53,126]. The DASH kernel [3] used an admission control algorithm based on a timeline and then used earliest deadline scheduling to actually sequence the tasks. Other systems used analyses from real-time scheduling theory to guarantee timing constraints for applications. For example, YARTOS [53] uses algorithms for scheduling sporadic tasks [52] to guarantee timing constraints. In order to guarantee performance, the computational requirements of the applications must be measured and analyzed along with the timing constraints such as delay bounds. Then the application can be run with the expectation that timing constraints will be satisfied.

In RT-Mach [125], much of the work on support for multimedia applications (other than the work described in this dissertation) used the traditional rate monotonic scheduling algorithm. As with YARTOS, the computational requirements of applications were measured in advance and analyzed to ensure that timing constraints would be met. The applications could then run successfully on the system. Much of the work on RT-Mach centered on high-performance real-time threads packages [92] and QOS managers [127].

These systems took a careful approach to analyzing and guaranteeing timing requirements for multimedia applications based on real-time scheduling analyses. They also incorporated advanced real-time system features such as priority inheritance protocols [108] and inheritance protocols for deadline scheduling [16]. These features are essential for supporting strong real-time guarantees among programs that share data and interact in other ways.

2.4.4 Soft real-time system support

Several multimedia systems have used scheduling algorithms like earliest deadline first to make the system more sensitive to timing without necessarily guaranteeing that timing

constraints will be met. For example, a system based on Chorus [103] proposed using deadline scheduling with no admission control [21]. Other algorithms such as lottery scheduling [133] attempt to support multimedia applications using proportional sharing of resources without real-time delay guarantees. A deterministic version of the approach called stride scheduling [134] was proposed to better support multimedia applications.

These types of systems are able to be more sensitive than time-sharing systems to the timing constraints of multimedia applications, but without effective admission control, overload cannot be prevented.

2.5 Reserve abstraction

The reserve abstraction described in this dissertation addresses several of the key problems raised above. The abstraction provides a framework for reasoning about resource reservation in an operating system. Other research efforts have focused on reservation of different resources in isolation. A framework to unify various reservation algorithms is needed.

The reserve abstraction gives resource reservation first-class status in the operating system. Reserves can be allocated independent of any particular process, and references to reserves can be passed around and bound to different processes as appropriate. For example, a real-time client might pass information about its timing constraints to a server to ensure expedited service. This is in contrast to the approach where timing constraints and resource usage requirements are associated directly with processes which makes it difficult or impossible to have one process temporarily take on the timing constraints of another process.

The key feature of the reserve abstraction is the enforcement mechanism. This prevents applications from overrunning their reservations if that would interfere with the timing requirements of other reserved activities.

The reservation framework and first-class status of reserves provide the power and flexibility to deal with the problems that arise in real-time system design and practical resource management. And the enforcement mechanism guaranteed proper resource scheduling. In combination, these aspects of the reserve abstraction offer an effective solution to the general real-time programming problem.

Chapter 3

Reserve Model

This chapter gives a definition of reserves on resources, which form the basis of the reservation model. Reserves provide a framework for integrating admission control, scheduling, and usage enforcement. Issues in reserve management are also addressed.

3.1 Reserve abstraction

The reservation model defines the concept of a *reserve* against a particular operating system *resource*. A reserve is a first class object that represents a part of the capacity or a quantity of the resource that is set aside for a computation which presents that reserve along with a request to use the resource. The word “capacity” is used in a broad sense here: reserving a portion of the capacity of a resource means that a thread will have access to the resource subject to some detailed reservation parameters, and the parameters are specific to the resource and the reservation system implementation. As an example, a reserve might specify that 30 ms of computation time on the processor are reserved out of every 100 ms of real time.

Reserving resource capacity implies that the resource can be multiplexed among several computations, and the model focuses on multiplexed resources such as processors and network bandwidth. Other types of resources such as physical memory pages and buffers are not amenable to extremely fine-grained multiplexing, and these are referred to as discrete resources. In this model, discrete resources are reserved on a per unit basis and the reservation dedicates the resources units indefinitely rather than implying a multiplexed usage of capacity over time.

3.1.1 Reservation guarantee

A key requirement in offering a resource reserve abstraction in a system is that the reserved resource capacity be available, subject to the reservation parameters, to a computation which presents the reserve and requests the resource. If the system cannot guarantee

that the resource capacity will be available as promised, the usefulness of the system is limited. Therefore, the enforcement of resource reservations is of critical importance. Enforcement is important not only to protect against malicious users, which may present a problem in systems where resources are shared by many, but also to relieve individual applications from the burden of ensuring that their own performance is strictly predictable and controlled. The system should tolerate applications which may try to use more than their reservation allows, isolating unrelated applications from this kind of behavior. This kind of *temporal isolation* is similar in concept to the isolation provided by a virtual memory system, which allows a process to try to access memory locations as it wishes, intervening only when a memory access is not allowed. In no case should a virtual memory system allow a process to access the memory of another process's protected memory, and likewise in no case should a reservation system allow a thread to impinge on the reserved resource capacity of another thread.

Thus, the system guarantees that resource capacity, given by the reservation parameters, will be available to the thread that has a reserve. However, it is up to the application programmer to make sure that the thread is in a position to take advantage of the resource capacity when it is made available. The reservation system itself makes no claims that a particular application will meet its timing constraints. For example, if an application blocks indefinitely waiting for a message, it may not be in a position to take advantage of resource capacity when it is available. For an application to have predictable real-time performance, it must have the proper resource reserves, and it must be able to use those resource reserves in a way that satisfies the timing constraints of the application.

3.1.2 Scheduling frameworks

The reserve abstraction can accommodate different frameworks for admission control, scheduling, and enforcement. Most of the features of reserves, specifically the operations available in the reserve abstraction, remain the same even if the scheduling framework changes. The primary differences in the interface to a different framework are the specification of the reservation request parameters for admission control and the resource usage statistics available from the enforcement mechanism.

3.1.3 Styles of programming with reserves

Reserves can be used in two different styles of real-time programming. Reserves support strict hard real-time applications and can equally well support more flexible soft real-time applications. The primary distinction between hard and soft real-time programming in the discussion of the reservation model is:

- whether resource usage requirements are carefully measured and specified in exact detail guaranteeing performance before the program is actually deployed (hard real-time), or
- whether resource usage requirements and resource capacity reservations are dynamically adjusted based on run-time usage measurements instead of being matched exactly during the design phase.

In either case, the resource reserves guarantee the requests that are admitted to the system, and whether or not those reserves are used for hard real-time programming or soft real-time programming depends on how the applications themselves use the reserves.

Another distinction in real-time programming using reserves is whether resources are reserved locally for each thread or whether they are reserved globally for an *activity* that may span multiple threads in different protection domains and even on different machines. In the activity-based model of using reserves, the originator of an activity acquires resource reserves for the activity and then passes those reserves along with any requests made to various servers. Using this model, accounting for resource usage across clients and servers is simplified, and the negotiation of quality of service parameters can be simplified as well. The reserves for each request come in with the request, and the server charges resource usage to those reserves when servicing the corresponding request. The responsibility of getting the appropriate resource reserves falls to the original client.

To summarize, several features of reserves are useful for both hard and soft real-time programming:

1. Take care of global admission control decisions, relieving the designer of doing global scheduling analysis.
2. Schedule threads on resources according to their reserves.
3. Accumulate usage information that could be useful during development, especially for adaptation in soft real-time applications.
4. Prevent interference from other real-time applications and non-real-time applications and activities that may be competing for the same resources.
5. And finally, reserves can serve either to separate the resource allocation and management of modules from each other or to integrate the resource allocation and management of modules, allowing reservations to span multiple threads and protection domains of a single activity.

3.2 Basic reserves

The *basic reserve* provides an abstraction for capacity on a particular resource. Note that this statement about basic reserves *does not guarantee that applications will meet their timing requirements*. The only guarantee is that *capacity will be reserved and available to be used*. We will explore the issue of what guarantees can be made at a higher level about the behavior of applications that use reserves.

The reserve itself is an operating system abstraction that is orthogonal to control structures like threads. A thread may bind to a resource reserve in which case the scheduler will use the information in the reserve when making scheduling decisions about the thread. Multiple threads may be bound to a single reserve, but typically a reserve will have only one thread bound to it at a time. The scheduler will always find an associated reserve, although sometimes that reserve will be a *default reserve* which has no actual reservation and just

serves to accumulate the resource usage of all threads that make unreserved use of the resource.

The specification of the reservation depends on the type of resource. Multiplexed resource reservations include information about the amount of work to be done per period of real time. They may also include a delay requirement. This parameter would specify the maximum amount of time after the beginning of each period the thread will have to wait before getting its reserved time on the resource. Discrete resources reservations just specify a count of the units of discrete resource required; they include no notion of time.

Despite the differences between multiplexed and discrete resource reservations, they share the same basic structure. They both require:

1. a reservation specification interface,
2. an admission control policy,
3. a scheduling policy, and
4. an enforcement mechanism.

For discrete reserves like memory pages and network buffers, the reservation specification gives a number of units of resource being requested. The admission control policy for discrete resources would just check the availability of the requested number of units of resource. Since discrete reserves are by definition not multiplexed, they require no scheduling.

It is important to note that the basic reserve abstraction is independent of the admission control and scheduling policies used. For multiplexed resources, reserves provide a framework to request resource capacity reservations, do admission control, schedule computations, and enforce capacity reservations. The choice of admission control and scheduling policies will impact the way reservation requests are specified and the way the enforcement mechanism tracks their behavior, but the framework is general enough to accommodate different policies. The following sections illustrate the reserve model in terms of a periodic scheduling framework.

3.2.1 Reservation specification

The reservation system must provide a way for applications to specify the resource capacity they would like to reserve. The form of the specification differs from resource to resource, and different admission control and scheduling policies may require different reservation specification parameters. In the most general sense, reservations specify a duration of usage with some time constraints be available, used, and replenished by some specific regimen.

As an example of the kind of parameters that might appear in a specification, a resource reservation may specify an amount of time to be spent on the resource per period of real time, and it may specify a start time for the periods as well. For example, a reservation request might specify 30 ms every 100 ms starting at 1:00pm.

Figure 3-1 illustrates how the reserved time might be consumed in a simple computation time per period of real time framework for reserves. The reserved computation time is available to be used during each reservation period. The computation time is guaranteed to be available at some point during the period; it is not guaranteed to be in any particular place such as the front of the period or the end of the period.

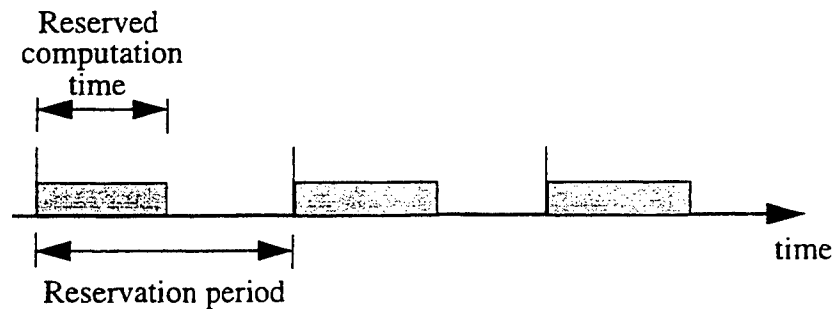


Figure 3-1: Periodic Scheduling Framework

There are many different scheduling policies and scheduling analysis techniques that could be used to provide a reserve abstraction, each of which would require a corresponding admission control test, scheduling policy, and enforcement mechanism.

3.2.2 Admission Control

An admission control policy associated with each resource decides which reservation requests for that resource can be admitted and which must be denied. It makes this decision based on the parameters provided in the reservation request and information about the other reservations that have already been granted for that resource. The admission control policy necessarily depends on the scheduling policy in order to do an admission control analysis.

3.2.3 Scheduling

The scheduling algorithm for a resource makes decisions about the order in which threads receive time on a resource. The scheduler looks at the reserve owned by each thread that is ready to run, and uses information in the reserve to determine which thread will get access to the resource. The algorithm supports the decision made by the admission control policy.

The scheduler must also coordinate with the enforcement mechanism to make sure that it does not try to schedule threads associated with a reserve that has already used its reserved resource time for a particular reservation period. Thus a reserve which still has time left on its reservation is in “reserved mode” and one that has run out of time is temporarily in “unreserved mode.” This represents a significant departure from other real-time scheduling algorithms, which generally assume that the resource usage requirements of application are accurately characterized and need not be enforced [67].

3.2.4 Enforcement

The reservation system must ensure that processes do not use more than their reserved capacity or reserved units of a resource. Enforcing reservations on discrete resources is straightforward; the system ensures that a resource dedicated to one process is not re-allocated to another process. Enforcing multiplexed reservations requires the system to keep accurate usage numbers that describe how much capacity has been consumed against each reservation. If a thread attempts to use some capacity beyond its reservation, the system must recognize this and actively prevent the process from consuming any additional capacity in reserved mode (consuming additional capacity in an unreserved mode may be allowed.)

If for some reason the reserved time on a resource is not used by the owner of a reserve in a given reservation period, that allocation of time is lost to the owner. The owner may not be in a position to use the resource if it is blocked waiting for some other resource to become available or for synchronization or communication with another computation. The resource will not necessarily be idle for that amount of time since the scheduling policy is free to allow an unreserved computation to use the resource (as long as the unreserved computation can be preempted to allow the reserve owner to use the resource). This implies that a computation may not save up reserved time (by not using it) and then use it all at once in a burst. The allocation of resource time is available during each period, but cannot be carried over past the end of the period.

On the other hand, if the thread that owns a reserve consumes the entire reserved allocation for a reservation period and attempts to continue executing, the thread will compete with the other unreserved computations for the resource under whatever policy the resource scheduler uses for unreserved computations. Thus a reserve may be in *reserved mode* where it still has some resource usage allocation left for the current reservation period, or it may temporarily be in *unreserved mode* where the allocation for the current reservation period is depleted (until the next reservation period).

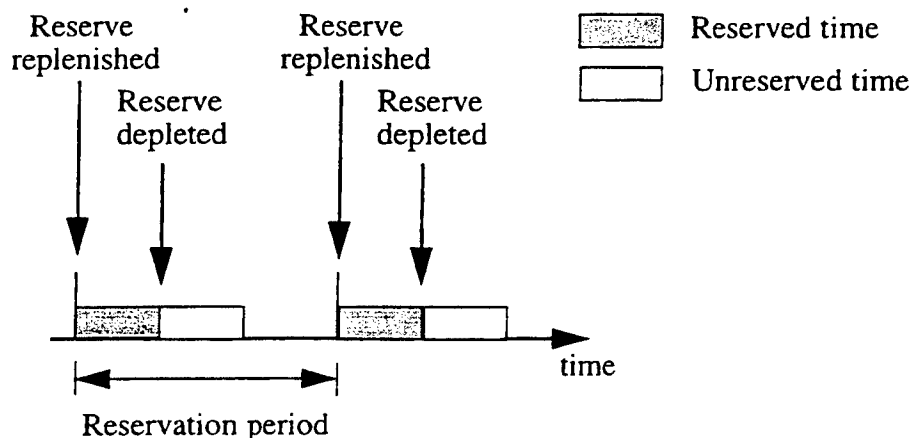


Figure 3-2: Enforcement Illustration

Figure 3-2 shows the reserved time made available on a particular resource for a single reservation. At the beginning of each reservation period, the allocation of reserved time is replenished, and the thread that has this reservation uses the resource in "reserved mode." After the reserved time allocated for that period is depleted, the enforcement mechanism generates a scheduler event to indicate that the reserved time has been consumed for that period. The scheduler is responsible for using that information in making scheduling decisions. In the figure, the thread continues to use the resource in "unreserved mode," consuming more time on the resource at the discretion of the scheduler. The execution history shown in the figure is based on the assumption that no other threads compete for the resource and that the policy lets it run in timing-sharing mode after its reservation has expired, and so the thread can get time in unreserved mode. At the beginning of the next reservation period, the reserve is replenished and the thread can run in reserved mode again. The main point of this figure is that the enforcement mechanism tracks resource usage and raises this "reserve depleted" event for the scheduler. The scheduler can then use this information in making future scheduling decisions. For example, it can give other reserved activities preference or allow unreserved time-sharing activities to use the resource.

Three important issues arise in the design of the enforcement mechanism:

1. how to accurately accumulate resource usage,
2. how to notice that a thread has depleted its reserve for a resource,
3. how to know when to replenish the allocation of a reserve.

To accurately accumulate the resource usage for each reserve, the system records the time during each reserve switch. A reserve switch occurs in two cases: when a thread context switch is performed or when a thread is changing the reserve against which it will charge its computation time. In the case of a thread context switch, the reserve switch records the current time, cancels the overrun timer (which signals the reserve depletion event as described below), computes the time the old thread ran, and adds that time to the accumulated usage of the old thread. The reserve switch mechanism then saves the current time for use later in computing the computation time of the new thread and sets the overrun timer. A reserve switch triggered by a thread changing the reserve to which it wants to charge its computation time works the same way, the only difference is how the reserve switch is triggered. Figure 3-3 illustrates how timers are used in enforcement. In the execution history on the timeline, it shows the reserved activity of interest in a solid pattern and some other activity (associated with other reserves) in a striped. The reserve switches (also context switches in this example) between these activities occur where the striped boxes and the stippled boxes meet.

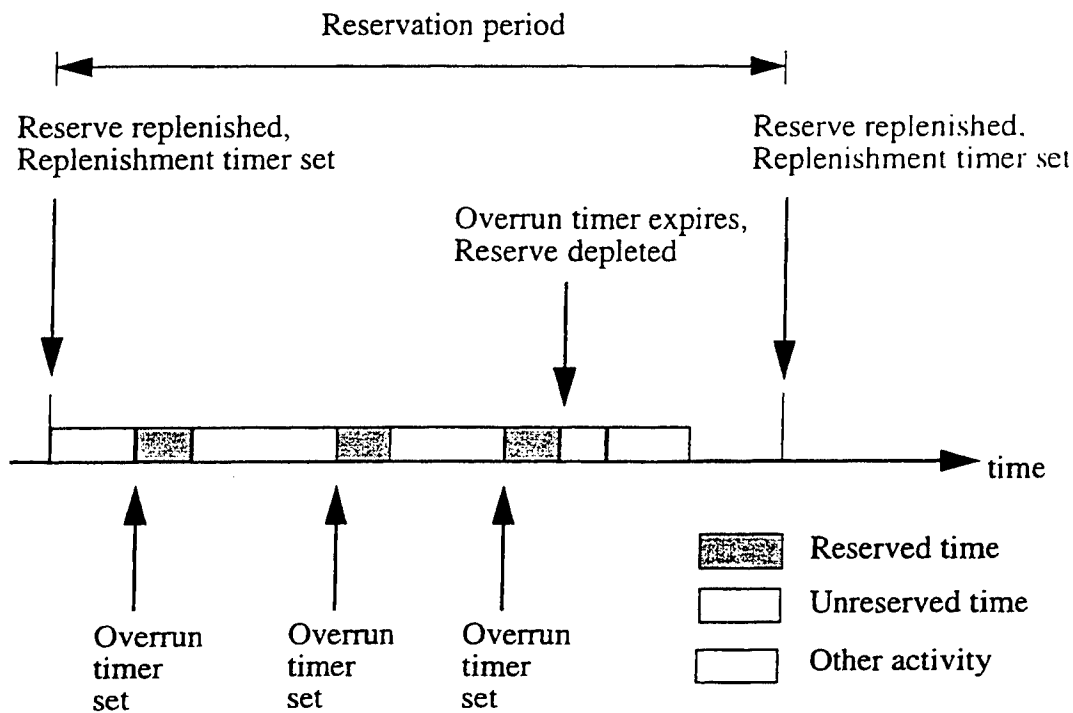


Figure 3-3: Enforcement Timers

The overrun timer is set during the reserve switch to expire at the end of the new thread's remaining reserved computation time or at the end of its reservation period, whichever is earlier. If the new thread is preempted before its reserved computation time is completed, the overrun timer will be cancelled. If the new thread consumes all of its reserved time, the overrun timer expires, and the scheduler is called to take some action based on that event. Figure 3-3 shows where the overrun timer is set for the reserve of interest; the overrun timer may also be set for the other activity if it is reserved, but that is not shown in the figure. The overrun timer in the figure does not actually expire until the last time it is set.

The handle replenishment, each reserve has a replenishment timer which is started at the reserve start time and which expires periodically at each reservation period. The replenishment timer records the usage accumulated on the corresponding reserve at the time of the reservation period and records that along with the current time as a "usage checkpoint." Then the timer handler changes the state of the reserve to reflect a new allocation of resource usage and resets the periodic replenishment timer. This replenishment model corresponds to a deferrable server [120] replenishment scheme; other replenishment methods are described and analyzed by Sprunt [111,112]. Figure 3-3 illustrates where the replenishment timer is set relative to the reservation period.

3.3 Reserve propagation

One important feature of the reserve model is that reserves can be passed from clients to servers, enabling the server to take advantage of the resources the client reserves for its entire computation. Passing reserves also enables the server to charge the resource usage it consumes to the appropriate client, preserving system-wide consistent usage accounting. This is called *reserve propagation*.

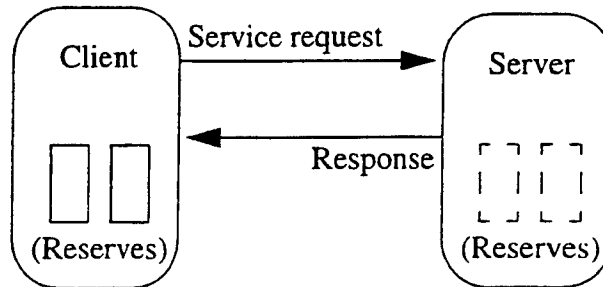


Figure 3-4: Reserve Propagation

Figure 3-4 illustrates a client/server interaction with reserve propagation. Assume that the client acquires resource reservations sufficient for the computation that it will perform locally as well as the computations to be done by servers on the client's behalf. In the figure, these reserves appear in the clients as two small rectangles. The interaction is a straightforward remote procedure call from the client to the server. For simplicity, assume that the client sends an RPC request to the server and waits for the reply. The server processes the request and sends an RPC reply, and then the client receives the reply.

When the client send the service request, it also sends references to the reserves that it has allocated. These reserves are to be used by the server as it processes the client's request. Thus, the server must start charging its resource usage to the client's reserves when it starts processing that request, and it must stop charging to those reserves when it finishes with the request and sends back the response.

Ideally, a server would schedule service requests to execute in the same order that the computations would execute if the clients could do them instead of the server. For example, the processor scheduler orders ready threads based on the processor reserve parameters. This ordering can be seen as a sort of "priority" ordering among those activities. If a thread makes a request of a server, the server should take the "priority" in the scheduler's ordering while it is servicing that request. Then the fact that a client relies on a server for some computation does not affect the progress of its computation with respect to other threads.

To help the server achieve this ideal, the RPC mechanism should propagate the reserve "priority", as represented by the reserve and its reservation parameters, of the client to the server. The queue of service requests for the server must be maintained in reserve "priority"

order, and a kind of “priority inheritance” must be used to prevent priority inversion in the access to the server.

On the receipt of a new service request, his “reserve priority inheritance” mechanism enqueues the request in the priority queue, and then it checks to see if the thread that will service the requests is waiting for new requests or servicing a previous request. If the thread is busy and the “priority” of the new request is greater than the “priority” of the currently processing request, the RPC mechanism sets the priority of the thread so the priority of the new request. It does not, however, change the reserve that the thread is charging against. When the server thread finishes the previous request and receives the new request, it keeps the priority of that new request (which it inherited before) and also begins charging to the reserves associated with the new request. When the request is finished, the server thread stops charging against the client’s reserves.

The “priority” inheritance mechanism described here, which is referred to as “reserve propagation”, differs from traditional priority inheritance in two ways:

1. reserve propagation specifies how a server should change the reserves it charges to based on the client it is servicing whereas traditional priority inheritance has no notion of charging to a reserve or account,
2. the “priority” of the server may change during the course of the request processing if the reserved resources are depleted during that time whereas with traditional priority inheritance, the priority is fixed.

The fact that a server’s “priority” may drop during request processing complicates reserve management and reserve “priority” inheritance. When a reserve is depleted, the scheduler must determine whether the thread charging against the reservation inherited the reserve “priority” or not. If not, the thread’s order in the ready queue may change. If so, the scheduler must determine from the threads pending request queue what the appropriate reserve “priority” for the thread should be, given the change in the state of the reserve that was depleted.

Reserve propagation from client to server is not mandatory. The next chapter discusses different programming models where this is useful and where it is not. Briefly, reserve propagation is useful when the system is organized such that an application allocates the resource reserves it will need for all of its activities and passes those reserves around to the servers it invokes to do work on its behalf. In this model, applications need not negotiate quality of service parameters explicitly with these servers. The scenario where reserve propagation is not that useful is where the system is organized such that applications negotiate quality of service explicitly with all of its servers.

3.4 Example scheduling frameworks

Many different admission control and scheduling policies could be used to support the reserve abstraction. For example, reserves based on rate monotonic scheduling [67] would be able to guarantee the availability of a certain amount of time on a resource per period of real time. For pure rate monotonic scheduling, the delay associated with receiving the com-

putation time in each period would be the length of the period itself. For deadline monotonic scheduling [64], the delay bound could be shortened. The following sections discuss these frameworks and others in more detail as they apply to the reserve abstraction.

3.4.1 Rate monotonic

The rate monotonic (RM) priority assignment of Liu and Layland [67] can guarantee that a certain amount of computation time will be available for a reserve for each period of real time with a delay bound equal to the length of the period. The discussion of rate monotonic scheduling uses the word “task” to denote the series of instances of a computation; with reserves, it is understood that this computation represents available resource capacity and not necessarily a complete program. Under rate monotonic scheduling, higher priority is assigned to the higher frequency programs. The rate monotonic scheduling analysis yields a basis for a processor reservation admission policy.

3.4.1.1 Reservation parameters

Reservation parameters for the simplest form of rate monotonic based reserves include computation time and reservation period: A start time is also useful for controlling the phase of the periodic reservations. This allows the programmer to synchronize the availability of the reserved computation time with a periodic program.

3.4.1.2 Admission control decision

Let n be the number of periodic tasks and denote the computation time and period of program i by C_i and T_i , respectively. Liu and Layland proved that all of the tasks would successfully meet their deadlines and compute at their associated rates if

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

When n is large, $n(2^{1/n} - 1) = \ln 2 \approx 0.69$. This bound is pessimistic: it is possible for task sets which do not satisfy the inequality to successfully meet their deadlines, but this cannot be determined from the Liu and Layland analysis.

An admission control policy follows naturally from this analysis. To keep track of the current reservations, the system must remember the utilization of the tasks that have reserved computation time, and the total reservation is the sum of these individual utilizations. A simple admission control policy is to admit a new reservation request if the sum of its utilization and the total previous reservations is less than 0.69. Such a policy would leave a lot of computation time that could not be reserved. One possibility is to use that time for unreserved background computations. Another possibility is to use the exact analysis of Lehoczky *et al.* [63] to determine whether a specific collection of reservations can be scheduled successfully, although the exact analysis is more expensive than the simple, pessimistic analysis above. In their work, Lehoczky *et al.* also gave an analysis showing that on aver-

age, task sets can be scheduled up to 88% utilization. So in most cases, this unreservable computation time is only 10-12% rather than 31%.

It should be noted that the rate monotonic scheduling algorithm was analyzed under simplifying conditions. Liu and Layland [67] made the following assumptions to enable their analysis:

- arrivals are periodic, and the computation during one period must finish by the end of the period (its deadline) to allow the next computation to start.
- the computation time of each program during each period is constant.
- computations are preemptive with zero context switch time, and
- computations are independent; i.e. computations do not synchronize or communicate with other computations.

In the context of the reserve abstraction, this means that rate monotonic scheduling can be used to guarantee that resource capacity is available to the applications. However, applications that have precedence constraints with other applications may not be in a position to use the available resources.

3.4.1.3 Scheduling

Reserved mode activities get precedence over unreserved. Among reserved mode activities, smaller period gets precedence over larger. Among unreserved activities, some time-sharing algorithm may be in effect.

3.4.1.4 Enforcement

Accumulate usage in each period. Update usage (determined using accurate measurement techniques) at each context switch. Set a timer for the currently running activity to expire at the end of its reserved usage. Set another timer for each reservation period.

3.4.2 Deadline monotonic

The deadline monotonic scheduling (DM) algorithm [7,64,66] is closely related to the original rate monotonic (RM) algorithm [67]. DM has the same kind of periodic scheduling frame as RM; the difference is that with DM, there is an additional parameter called the deadline specifies the duration of time by which the computation released at the beginning of the period must be completed. For the original version of RM, this deadline is assumed to be the end of the period, when the next instantiation of the computation will be released. For DM, this deadline is specified explicitly.

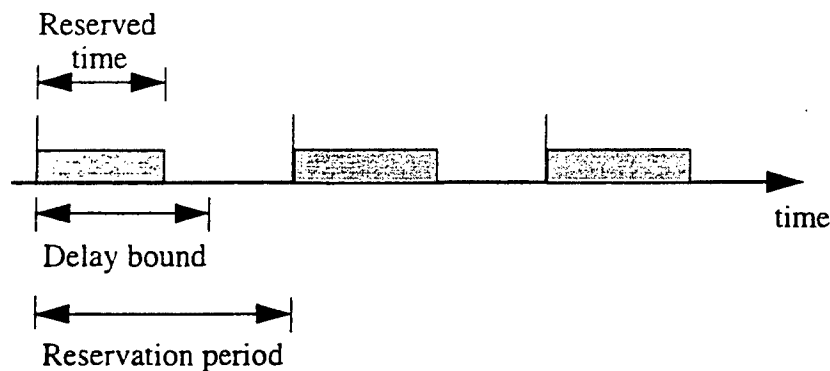


Figure 3-5: Deadline Monotonic Scheduling Framework

Figure 3-5 illustrates the periodic scheduling framework of DM along with the additional deadline parameters that does not appear in the original RM algorithm. The deadline in this case is before the end of the period, but it could also be after the beginning of the next period (in which case there would be multiple instantiations of the computation, started in different periods, at one time).

3.4.2.1 Reservation parameters

The reservation parameters for DM are the same as RM with the addition of the deadline parameter. As an example, a reservation request may specify 30 ms on the resource be reserved for every 100 ms with the delay constraint that the 30 ms must be available within 50 ms of the beginning of each 100 ms period. As for rate monotonic scheduling, a start time parameter is useful for synchronizing reservations with periodic threads and with other reservations.

3.4.2.2 Admission control decision and scheduling

Schedulability analysis tests for DM are given by Lehoczky [64] and by Audsley et al. [7]. These tests are quite a bit more complicated than the simple schedulability bound test for RM, involving systems of equations that have to be checked. Even so, the schedulability tests provide suitable admission control decisions for a reservation mechanism based on DM.

Scheduling is based on the deadline monotonic algorithm that assigns fixed priorities to tasks based on the deadline value. Shorter deadlines are assigned higher fixed priorities than longer deadlines. As with the rate monotonic algorithm, the reservation mechanism distinguishes between reserved mode activities and unreserved mode activities. At the beginning

of any given reservation period, an activity with a reservation is in reserved mode until it consumes all of the reserved time for that period. It is then changed to unreserved mode. The scheduler services reserved mode activities first, in order of their deadline values. If there are no reserved mode activities, unreserved activities are scheduled.

3.4.2.3 Enforcement

The enforcement mechanism accumulates usage in reserved mode until one of the following occurs: the resource usage reserved for that period is consumed or the deadline time has passed. In either case, the activity is changed from reserved mode to unreserved mode where it can compete with time-sharing activities for the resource.

3.5 Basic reserve types

Operating systems manage many different kinds of resources that system and user programs may use to do their work. The most important examples are processors, physical memory, buffers, and network bandwidth.

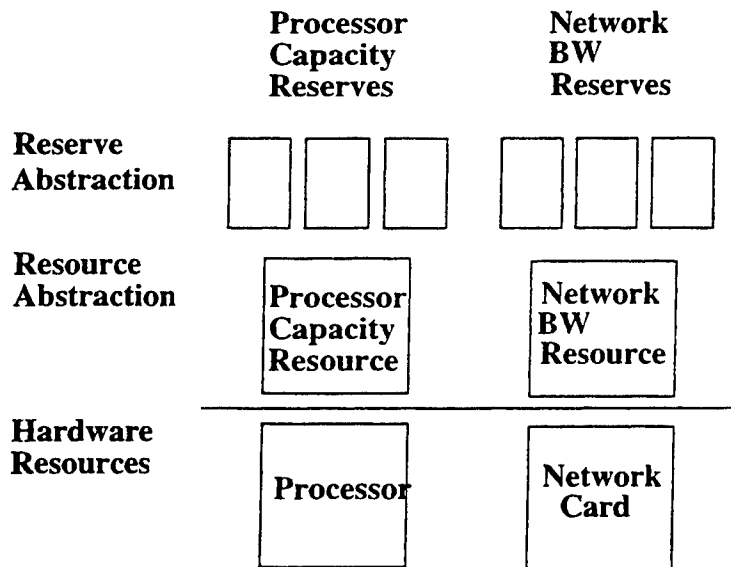


Figure 3-6: Resources and Basic Reserves

Many of these resources must be managed in the reservation system, so we define *basic reserves*, which are used to reserve and control the usage of different types of system resources. Each basic reserve type is associated with a resource type in the system. Figure 3-6 illustrates the relationship between the operating system resources and the basic reserves.

3.5.1 Processor

Processor capacity reserves represent reserved time on a processor. Reserve requests specify capacity in terms of time that will be reserved on the processor, rather than in terms of instructions that will be executed or any similar measure of processor usage. The requests may specify other information, depending on the admission control and scheduling policy in effect. The discussion in the following sections assumes a deadline monotonic scheduling framework where the reservation request specifies the amount of time to reserve on the processor, a period at which the allocation will be replenished, and a delay bound. These sections will cover these topics in more detail and will additionally discuss issues in enforcement, blocking time, and usage monitoring.

3.5.1.1 Units of work

Processor reserves deal with allocating real time on a processor rather allocating a sequence of instructions. The reason for this is that reserving instructions would be too difficult. It would require knowing the exact sequence of instructions to be used with the reservation, fixing the exact sequence for accuracy (to avoid cache effects, etc.), allocating a time slot on the processor to execute those instructions, etc.

Several pitfalls complicate the use of this reservation model that is based on time spent on the processor. For example, DMA can impact the amount of work that gets done in a certain amount of time spent on the processor. Cache effects can introduce variance in the amount of work per time on the processor. Processor pipeline flushing at context switches decreases the amount of work done during a fixed time on the processor. These are all second order effects, but their impact should be accurately characterized.

Processor reserves leave it to the individual applications and other higher-level software to make an appropriate mapping between the computational requirements of the applications to the appropriate reservation specification. For hard real-time applications, accurately characterizing processor requirements is very important. For soft real-time applications, an adaptive approach is the key to dealing with the fact that reservations are for time on the processor rather than work done by the processor. These applications can look at their own behavior and make adjustments as necessary.

3.5.1.2 Admission control and scheduling

For processor reserves in a rate monotonic scheduling framework, a reservation request consists of three parameters: a computation time, a period, and a start time. The admission control and scheduling policies described here are based on rate monotonic scheduling [67] as described above.

3.5.1.3 Enforcement

The enforcement mechanism for processor reserves must keep track of the processor usage for each reserve so that a scheduling event can be raised at the point where the reserves allocation has been depleted for a given reservation period. The usage measure-

ment task is complicated by the fact that a thread charging to a particular reserve may be preempted, and so at each thread context switch, the usage numbers must be updated to reflect the usage since the last context switch.

Accurately accumulating resource usage

To accurately accumulate resource usage in the face of preemptive use of the resource, the system must, at each context switch, compute the usage since the last context switch. This can be achieved by recording the start time of the computation (at the last context switch) and then computing the difference between the time at the current context switch and the time of the last context switch. This is the time the last thread was using the processor resource, and this time is added into the usage accumulator for that thread's reserve. Thus the accumulator keeps an accurate account of the resource usage charged to it.

Noticing reserve depletion

The enforcement mechanism must be able to notice when the reserve of the currently executing thread becomes depleted. To do this, the system at each context switch computes the longest contiguous time the thread is entitled to execute on its reserve, and it sets a timer for that time. If a context switch occurs before the timer expires, the accumulators are updated and the timer is set for the next thread to execute. If the timer expires while the thread is executing, the system updates the accumulators, marks the reserve as "inactive", and calls on the scheduler to make some decision based on the new state of that reserve.

Replenishing a reserve's allocation

Each reserve must have its allocation replenished at the beginning of each reservation period so that the time on the resource is available if it is needed during that period. To do this, the system uses a periodic timer for each reserve which is set to expire at the beginning of that reserve's reservation period. When the timer expires, the state of the reserve is updated to reflect a full allocation of resource usage for the upcoming period.

3.5.2 Physical memory

A physical memory reserve represents a collection of physical memory pages. Physical pages are discrete resources, so they support simple discrete reservations. The more interesting question is how the owner of a page reserve uses this collection of physical pages. Basically, pages can be locked down or paged in and out, and they can be prefetched or demand paged. A small application which could fit into its page reserve would benefit from prefetching its image into the page reserve and locking down the pages. A larger application might benefit from prefetching and locking down some (more frequently used) pages while keeping other physical pages available for less frequently used logical pages to be paged in and out. The advantage of using physical page reserves for these larger applications is in the increased control reserve give the application over traditional time-sharing demand paging replacement policies. With physical page reserves, the owner of a page reserve will at least be isolated from competition for pages with other threads in the operating system.

3.5.2.1 Admission control and scheduling

The admission control policy for this discrete resource is as follows: if there are enough free physical pages to satisfy a reservation request, then the reserve is granted; otherwise the reservation request is denied. The physical page reserve contains the number of pages that are reserved, and these pages are completely free and ready to be used by the thread that owns the reserve. The system may want to keep some number of physical pages as "unreservable" pages to allow time-sharing threads enough resources to make progress.

There is no scheduling of the use of pages by the reserve mechanism.

3.5.2.2 Enforcement

The enforcement of reservations for this discrete resource is relatively straightforward: a thread that has a physical memory reserve can use pages in its own memory pool and can also use pages from the time-sharing free page pool. Thus a thread using a physical memory reserve is assured of having at least the reserved number of pages available and possibly more. At no time will the pool of pages in the reserve fall below the reserved number.

3.5.3 Network bandwidth

Reserves for network bandwidth represent reservations for time on the network device. The system must include a mechanism for identifying the reserve to be used for incoming network packets. These reserves will typically be closely coordinated with processor capacity reserves and with bandwidth reservation supported by the network. The operating system will control the amount of outgoing traffic for each session (or virtual channel), and it will ideally coordinate with a network reservation system to limit the amount of incoming traffic for each session.

3.5.3.1 Units of work

The unit of work for a network bandwidth reserve is the transfer of a number of packets of a particular size (which will probably be constant, the MTU). Servicing of single packets is certainly non-preemptive, and it should also be possible to bundle multiple packets into non-preemptive work units.

3.5.3.2 Admission control and scheduling

The reservation specification for net bandwidth reserves includes a reserved time per period of real time, and possibly an indication of expected blocking time.

Timeline or rate monotonic scheduling frameworks among others would be appropriate for net bandwidth reserves. Several important issues relate to the non-preemptive nature of the work unit. Ideally, the expected blocking time would be used in the admission control policy and scheduling algorithm.

3.5.3.3 Enforcement

Accurate measurements of usage time can be computed between requests. This information can then be used in the enforcement mechanism and for input into scheduling policy decisions.

3.6 Reserve management

3.6.1 Default reserves

Default reserves exist in the system to simplify the implementation of the reservation mechanism by providing “reserves” for non-real-time programs to charge usage against. These default reserves do not actually represent reserved resources, but they do accumulate usage for all activities that have created their own reserves or had reserves created for them.

For example, new threads are assigned to run under the default processor capacity reserve when they are created. Thus a thread will charge its time to this global reserve until it acquires a reserve of its own.

3.6.2 Composite reserves

Having many types of reserves allows flexibility in specifying resource requirements to the system and in allocating resources, but the job of managing those resource reservations at the user level becomes more involved. For example, a multimedia application, such as a video player, might reserve resources for several constituent activities. It might reserve some processor capacity for the module which reads audio and video data from the disk and passes the data to an audio server and a display server. It might also reserve processor capacity for a control module which provides fast response to interactive control commands from the user. The player part might also reserve physical memory and message queue buffers at the file system manager. Each of these reservations has an associated reserve, and we would like to be able to collect a subset of these reserves under a single name to avoid having to refer to them individually.

Grouping related reservations together helps alleviate this complexity. The model allows reservations for different types of resources, and the situation arises where a program has reserves for several different resources. Since it has to present the appropriate reserve handle to be able to use a resource, a way of grouping all of the reserves under one handle would make it easier for the program to identify its reservations to the system and to the servers it invokes.

A *composite reserve* groups reserves for different resource types under a single handle. A composite reserve has the following properties:

- it will contain a number of basic reserves,
- it may contain only basic reserves (no composite reserves),

- it may contain at most one reserve for each basic resource type

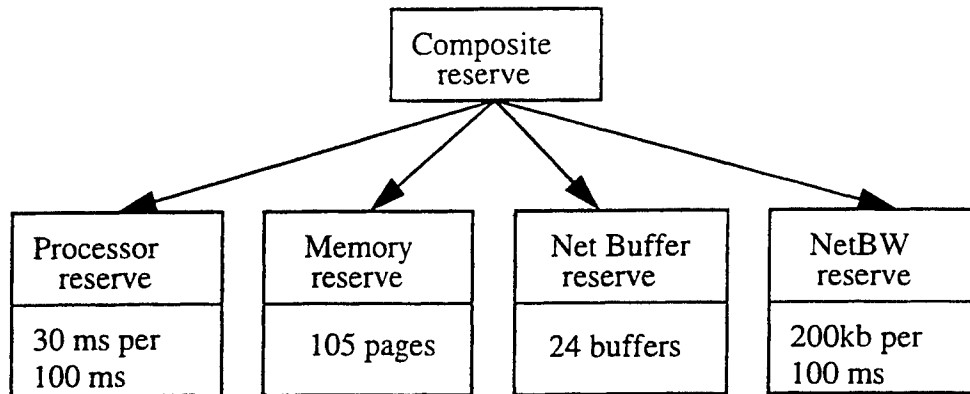


Figure 3-7: A Composite Reserve

Figure 3-7 shows the relationship between a composite reserve and its constituent basic reserves. In the video player example above, we might collect all of the reserves to be used by the player part (processor, physical memory, and message queue buffers) into a composite reserve. Then the system could use this reserve to reference the collection of resources reserved for the player. To charge computation time to the player, the system would take this reserve and look for the processor reserve under it.

3.6.3 Reserve inheritance

When a process creates a child process, the reserve of the parent is passed to the child, and the child runs against the resources reserved in the inherited reserve. This feature provides a way to allocate resources for non-real-time activities that create large process trees (like “make”). Reserve inheritance is appropriate for the automatic propagation of reserves for non-real-time programs, but real-time programs should generally configure their resources reserves explicitly. Figure 3-8 shows the difference between a process P whose children do not inherit its resource reserves and another process Q whose children and other descendants do inherit its resource reserves.

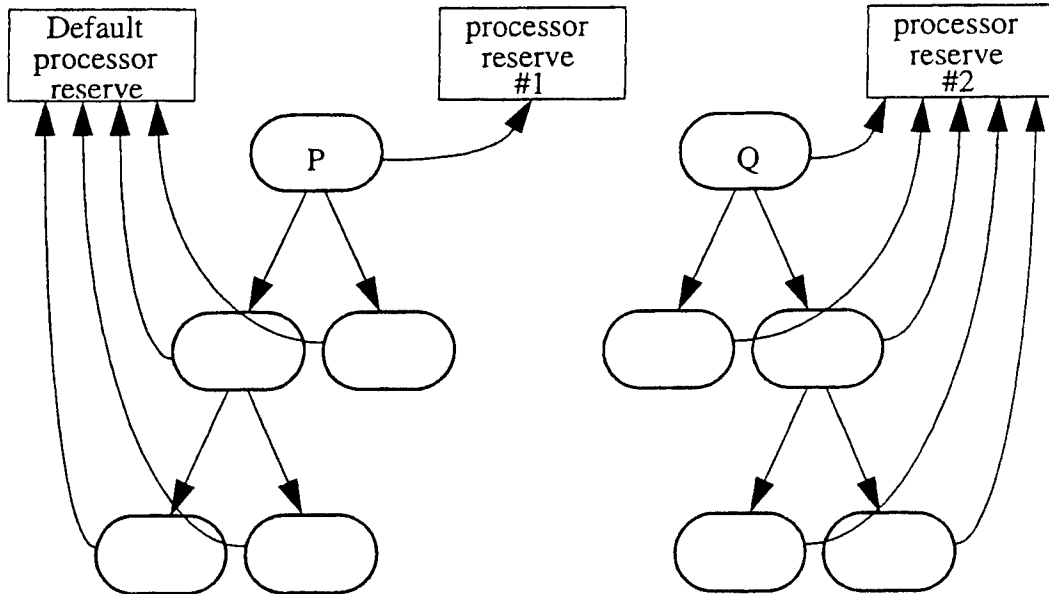


Figure 3-8: Reserve Inheritance

3.7 Chapter summary

This chapter described the basic reserve abstraction including reservation specification, admission control, scheduling, and enforcement. The idea of reserve propagation where a client hands reserves to a server to which it sends a request for service is shown to be a powerful mechanism for making reservations on a per-activity basis (rather than a per-thread basis). Several different scheduling frameworks which could be used in an operating system supporting the reserve abstraction were described, and the chapter discussed several different types of basic reserves for various resources such as: processor time, physical memory, and network bandwidth. A section on reserve management described default reserves, composite reserves, and reserve inheritance which address some practical issues in using reserves in a real system.

Chapter 4

Programming with Reserves

This chapter describes how to write programs that take advantage of resource reserves to satisfy their timing constraints. It focuses on three main issues: How should reserves be used in an application given that it uses various modules, external servers, and system services in the course of its computation? And also: How should the reservation parameters, particularly the reserved resource usage parameter, be initially chosen? How should they be adjusted given that applications must support different platforms and may have computational requirements that depend on changing input data?

4.1 Overview

This chapter describes the major issues involved in programming with reserves including the design decisions and tradeoffs that a programmer must make. The specific issues addressed are:

- How to structure programs to take advantage of reserves.
- How to map reserves onto a program's structure.
- How reservation parameters should be sized.
- How adaptive programs should adjust reservation levels.

One can think of a program as a graph of computational nodes, and each computational node has a reserve associated with it. Determining exactly what reserves should be allocated, what their reservation parameters should be, and how reserves should be associated with these computations involves design decisions that impact the program structure.

For example, the programmer must decide whether applications that depend on each other will explicitly negotiate timing requirements among themselves for the specific services they provide to each other. The alternative is to allocate resource reserves for their combined activity and then pass those reserves along as the abstract "activity" passes from

one to the other. In the first case, the partitioning of requirements and the explicit specification of timing requirements for each computation in each application creates a great deal of bookkeeping that has to be done. In the latter case, the requirements summarize the entire activity without specifying each detail along the way. As long as each phase of the activity adheres to a few rules such as not introducing unnecessary delays into the overall activity, the same high-level timing requirements can be satisfied without excessive dissection of the programs.

Another design decision addressed here relates to the flexibility of applications that use reserves. Hard real-time applications would typically specify fixed reservation parameters. Adaptive programs might be able to monitor their resource usage and adjust reservation parameters to fit their behavior over time. They might even be able to select different algorithms with different semantics and different performance characteristics to tune their computation time.

Finally, this chapter addresses the issue of programming with multiple resources. This requires applications to be broken into sub-computations at points where different resources are required. Coordinating resource reservations on multiple resources to satisfy end-to-end timing constraints requires careful design. Two approaches using reserves are described.

4.2 Using reserves in application design

This section focuses on the structure of applications and how reserves fit into that structure. Programs are considered to comprise one or more concurrent activities. Each activity might have a thread associated with it, and each activity has a call graph describing the sub-routines that are called by each subroutine. The call graph is extended to include calls to external servers or system services made by each subroutine.

The following sections describe these extended call graphs and address the coordination of reserves between reserved modules, reserved clients and servers, and reserved operating system services.

4.2.1 Program structure

To understand the timing constraints and resource requirements, one must consider the structure of application code. This section describes how an application might be divided into separate activities. It describes how a periodically executed computation in an activity might be broken down into sub-computations by splitting computations at procedure calls, remote procedure calls (RPCs), and system traps. The result is like a call graph that includes "calls" to servers and to the operating system.

For the purpose of this analysis, an activity is defined to be an abstract thread of control that starts out in a process and moves in and out of user-level servers and the operating system as calls are made to those servers and the system. This is similar to threads traversing objects in Clouds [26]. Such an abstract thread model corresponds to a synchronous programming style, which is in contrast to an asynchronous programming style where activities

are essentially “forked” by making asynchronous service requests to external servers or kernel primitives. For example, consider an animation application that synthetically generates animation frames in real time. The application consists of two activities: one to generate and display animation frames and one to process user interface events such as requests to resize the animation window.

Each of these activities may call modules in the same address space, make RPCs to servers, or make system calls. By this definition, when a (synchronous) RPC is made to a server, the activity “moves” to the server for the duration of the server’s computation and then the activity returns to the client when the call returns. If the server were to call another server synchronously, the activity would move to the second server for the duration of the call. The same is true of a system call. When a system call is made, the activity “moves” to the operating system and returns when the call returns.

An activity may be periodic. For example, consider the frame generation activity of the animation application. Suppose this activity originates in a subroutine (called `process_frame`) that is invoked periodically every 33 ms to process and display frames. Now suppose `process_frame` calls `generate_frame` and `display_frame`, which eventually performs an RPC to a window system server that accesses the frame buffer. Figure 4-1 shows an example call graph rooted at the function `process_frame`.

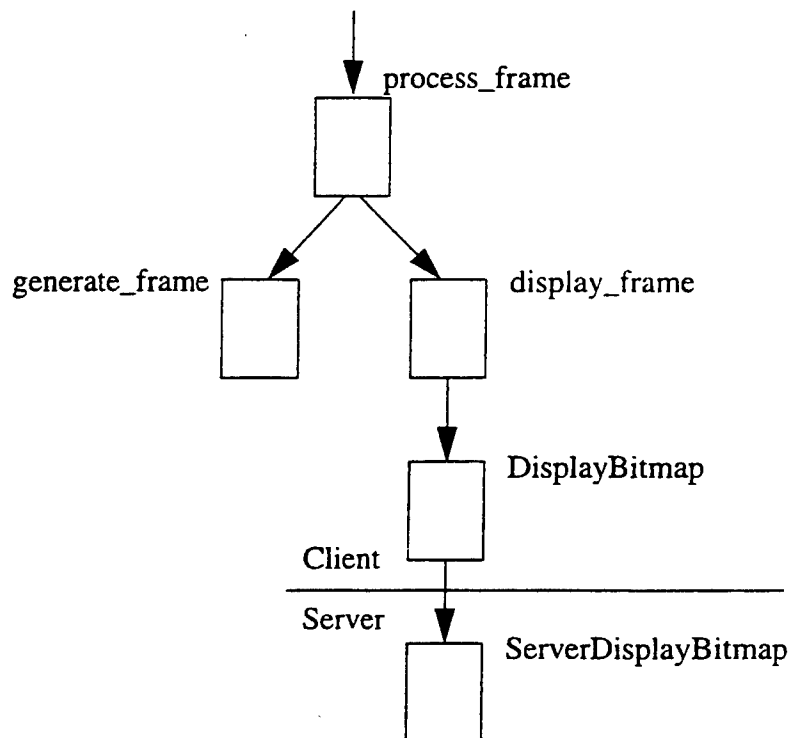


Figure 4-1: Call Graph for Frame Generation and Display

This call graph includes an RPC from the client to the server. The DisplayBitmap subroutine makes the RPC in the client, and the ServerDisplayBitmap subroutine in the server continues the activity. Thus, this graph captures all of the sub-computations of the animation activity.

4.2.2 Reservations for periodic computations

Given that the `process_frame` subroutine shown in Figure 4-1 is invoked periodically, the thread would “release” the computation periodically by using a while loop with a delay primitive or by setting a period attribute in the case of RT-Mach’s periodic threads [125]. Thus, the activity has an initial release time and a period parameter. To associate a processor reserve with this activity requires that a reserve be allocated with a start time and period that corresponds to that of the `process_frame` activity. It is possible to bind a periodic thread that attempts to execute its computation every 40 ms to a reserve that has a reservation period of 50 ms. This is not recommended, however, because the resources would not necessarily be available when the activity was released.

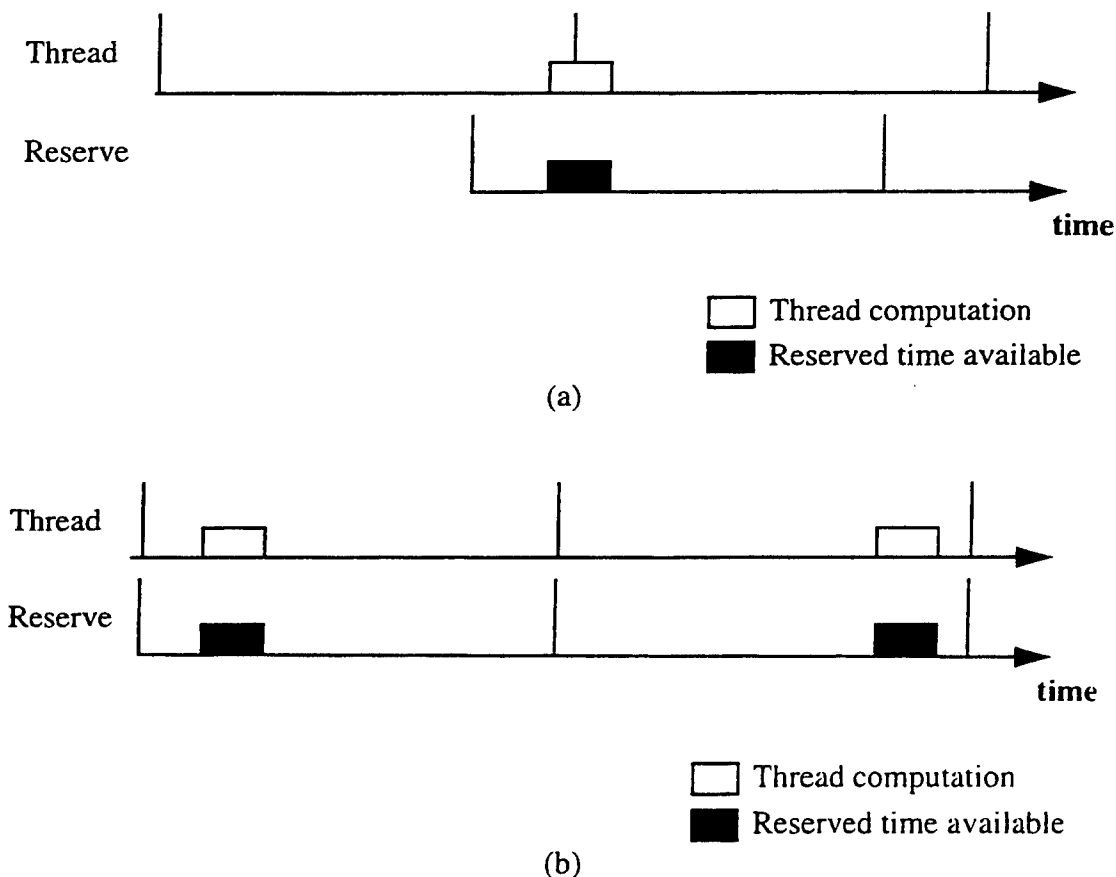


Figure 4-2: Thread and Reserve Out-of-Phase

Figure 4-2(a) illustrates a case where the thread period is not aligned with the reserve period, resulting in undesirable delays. In part (a) of the figure, the reserved computation time is not available until near the end of the thread's period. Thus the thread cannot start running until the very end of its period, and it misses its deadline at the end of the period. The problem is that the availability of the reserved time did not match the availability of the thread. Figure 4-2(b) shows the case where the thread period and reservation period are synchronized. This means that the thread will be ready when the reserved computation time is available, and the reserve guarantees that the reserved computation time will be available by the end of that period, so the thread is assured of being able to complete.

4.2.3 Localized reserve allocation

Consider the resources required in each node of the call graph in Figure 4-1. Assume that `generate_frame` requires only the processor. For nodes under `display_frame`, assume the frame buffer is mapped into the window system server's address space and that the processor is the only resource required.

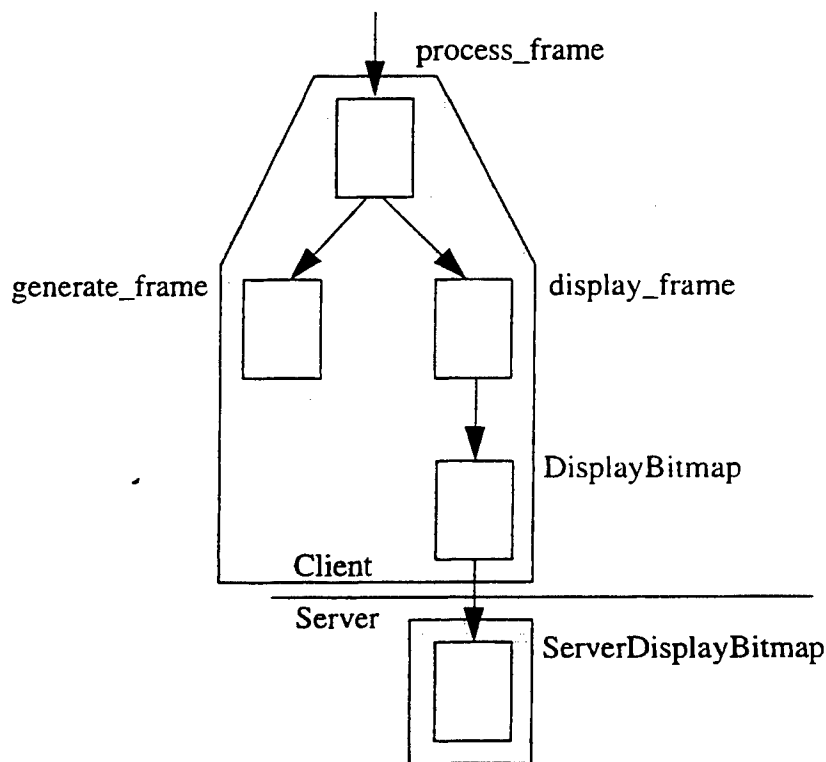


Figure 4-3: Call Graph with Separate Client and Server Reserves

With these assumptions, one approach to allocating reserves for the sub-computations would be to allocate a processor reserve for all of the nodes in the animation application and another processor reserve for the nodes in the window system server. Thus, the server would have a reserve allocated for each of the clients holding open connections to it. This approach

is necessary in the case where the server resides on a remote machine, but it may be preferred even when both utilize the same processor. Figure 4-3 illustrates this approach.

The RPC from the client to the server implies a switch from the client's reserve to the server's reserve. With this approach, the traditional "priority" inheritance mechanism would not be useful because the fact that the server has a reserve allocated internally for the animation client defines the "priority" for the client's request in the server and the client's "priority" does not get propagated. Another mechanism to associate the animation applications RPC request with the appropriate reserve inside the server would be very useful. Such a mechanism might take the "priority" of the server-allocated reserve to be associated with the animation client and apply it to the thread that will handle the client's request. This is a kind of "priority" inheritance where the server's thread gets the priority of the reserve it allocated for a client instead of getting the priority of the client itself.

Since the server must allocate the reserve for its computation on behalf of a client, it must know what the reservation parameters should be for the reserve. This approach requires the client and server to enter into a dialogue to allow the client to explicitly request a server-specific QOS level, meaning a certain pattern of server operations to be called with certain timing constraints. The server must then map the requested QOS requirements to system resource requirements and decide whether it can acquire the reserves to support that activity. All of this negotiation must be explicit, and that means a client/server interface for negotiating server-specific QOS requirements must exist. Further, the server must have the machinery to map those QOS requirements to system resource requirements.

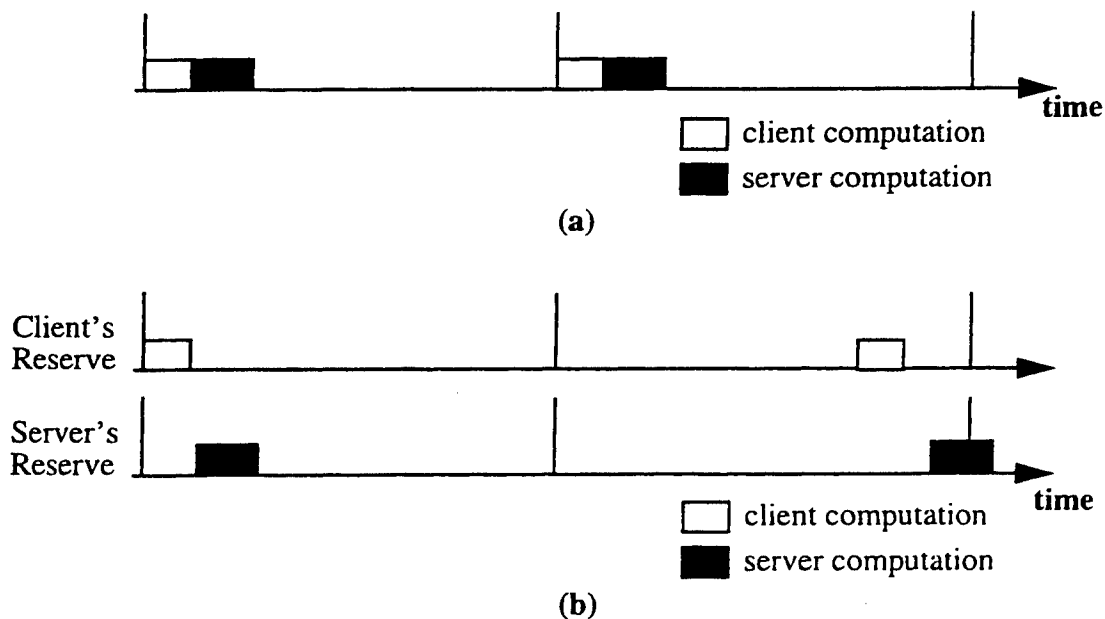


Figure 4-4: Switching Reserve from Client to Server

Another issue is that the timing parameters of the two reserves must be carefully coordinated for the call to be executed smoothly. Essentially, the client's call to the server means that the computation in the server becomes ready, and its reserve must provide the resources for it to execute in a timely fashion after it is ready. The sequence of client computation followed by server computation is illustrated in Figure 4-4(a).

The server's computation time might be available immediately after the call is made, as in the first period of execution history shown in Figure 4-4(b), in which case the deadline for the combined activity is met. But as shown in the second period of the execution history in Figure 4-4(b), the client's computation time may be available very late in its period. The server's computation time may be available earlier in its period but not available so close to the end. It is guaranteed to be available sometime in the period, but not at any particular time. Thus the activity could miss its deadline.

Introducing an intermediate deadline for the client's computation could solve this synchronization problem. Figure 4-5(a) shows the usage pattern of the client and server with an intermediate deadline for the client.

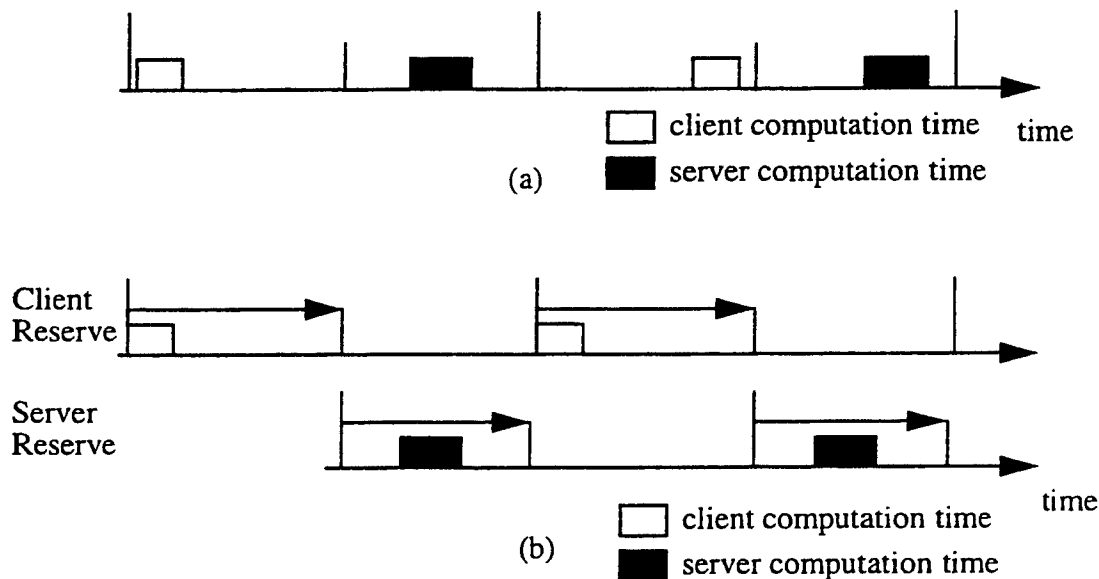


Figure 4-5: Client Requirement with Intermediate Deadline

Using reserves with deadline parameters, Figure 4-5(b) shows how the reserves can be allocated such that the client reserve has its computation time available at the beginning of the client's period with an intermediate deadline halfway through the period. The server's start time is at that intermediate deadline, and it has a deadline that corresponds to the end of the client's overall period. Thus, both activities are guaranteed to synchronize and complete by the overall deadline as desired.

All of the explicit handling of QOS requirements and resource requirements and the careful synchronization of interactions between reserves makes programming clients and servers much more complex. While this may be necessary for designing complex hard real-time systems, for soft real-time systems and less complex hard real-time systems, the approach where resource requirements for clients and servers are folded into one reserve may be better.

4.2.4 Activity-based reserve allocation

Another approach would be to allocate a single processor reserve for all of the nodes in the entire call graph: Figure 4-6 illustrates this approach. Of course, if the server is running on a remote host, this approach may not be feasible since it is not clear how a single processor reserve could be made to represent processor time on two different processors.

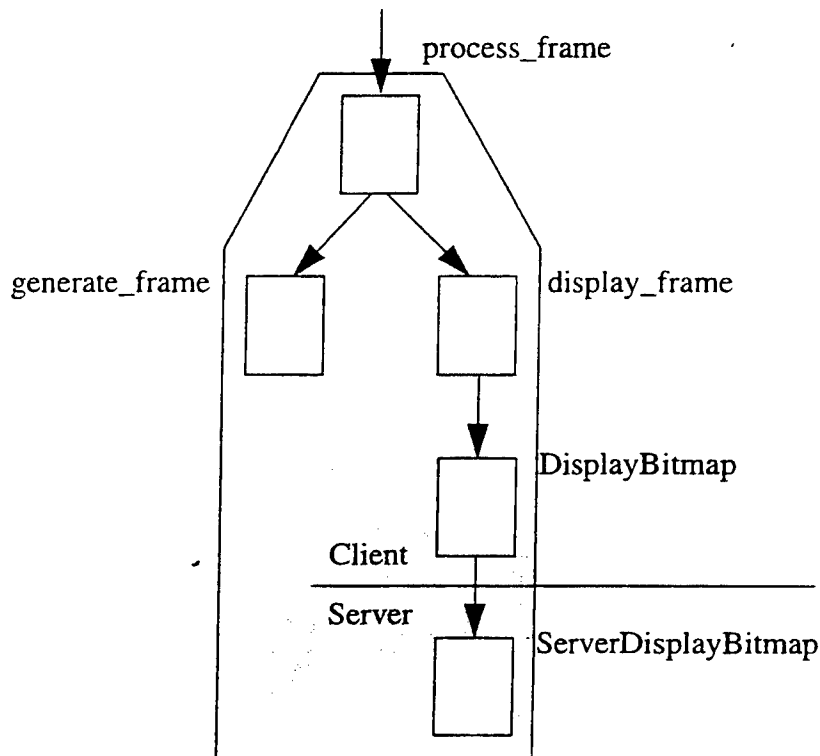


Figure 4-6: Call Graph with One Reserve for All Nodes

Even if the animation application and the server are on the same host running on the same processor, there is a problem that must be addressed in this approach: the server may not be ready to service the RPC call at the time it is issued. In fact, there is a potential “priority” inversion problem associated with such an RPC (where “priority” refers to the ordering of reserved activities by the scheduler rather than an integer priority for a thread). If the

RPC arrives in the server's queue at a time when the server is servicing another client, the server's thread will be bound to the reserve associated with the other client. If that other client is processor-poor, the reserved time may run out during the service, and the server may experience some scheduling delay. If the other client has a reserve that happens to get its processor time very late in its period, there may be a significant delay until the server, running with that other client's reserve, can finish the on-going operation.

To limit the delay the animation application experiences waiting for the server to handle its RPC, a "priority" inheritance protocol [108] must be employed. If the animation client's reserve would be sequenced by the reserve scheduler before the other client's reserve, the server which is using that other client's reserve would be sequenced as if it were using the animation client's reserve. However, for consistency of the usage measurement, it will still charge its usage to the other client's reserve. Then when that service is finished, the server will bind to the reserve of the animation server and that completes the propagation of the animation application's reserve to the server.

For this reserve passing to work best, the RPC should be synchronous, meaning that the client should wait for the result after making the call to the server. With a synchronous RPC, either the client or the server will be charging against the client's reserve whereas with an asynchronous RPC where the client does not wait for the result from the server, both the client and server may be charging against a single reserve at the same time. This is not catastrophic, but it may result in complicated interactions between the client and server.

This approach implies that the client application must request reservation parameters that include the computation time that will be consumed by the nodes residing in the server. This can be done by having the client discover the requirements empirically during runtime, by having the server explicitly provide its resource requirements, or by determining the requirements at design time (this issue is discussed in detail in the next section).

The important point here is that the client and server need not explicitly exchange information about resource requirements if the client allocates the reserve and passes it to the server. In particular, a great deal of complexity can be avoided if the client/server interface does not need to be able to support a complex negotiation of requirements. For legacy systems, this means that existing interfaces need not be radically modified, the only change being the mechanism for passing reserves from client to server.

4.2.5 Coordinating multiple resources

This section describes an issue that arises when an application uses multiple different kinds of resources in different sub-computations. Consider an audio/video player application that reads data stored on a disk and then outputs an audio stream and displays video frames. The player could be structured as three activities: audio playback, video playback, and user interface. The video playback activity would be periodic, reading and displaying a frame every 30 ms.

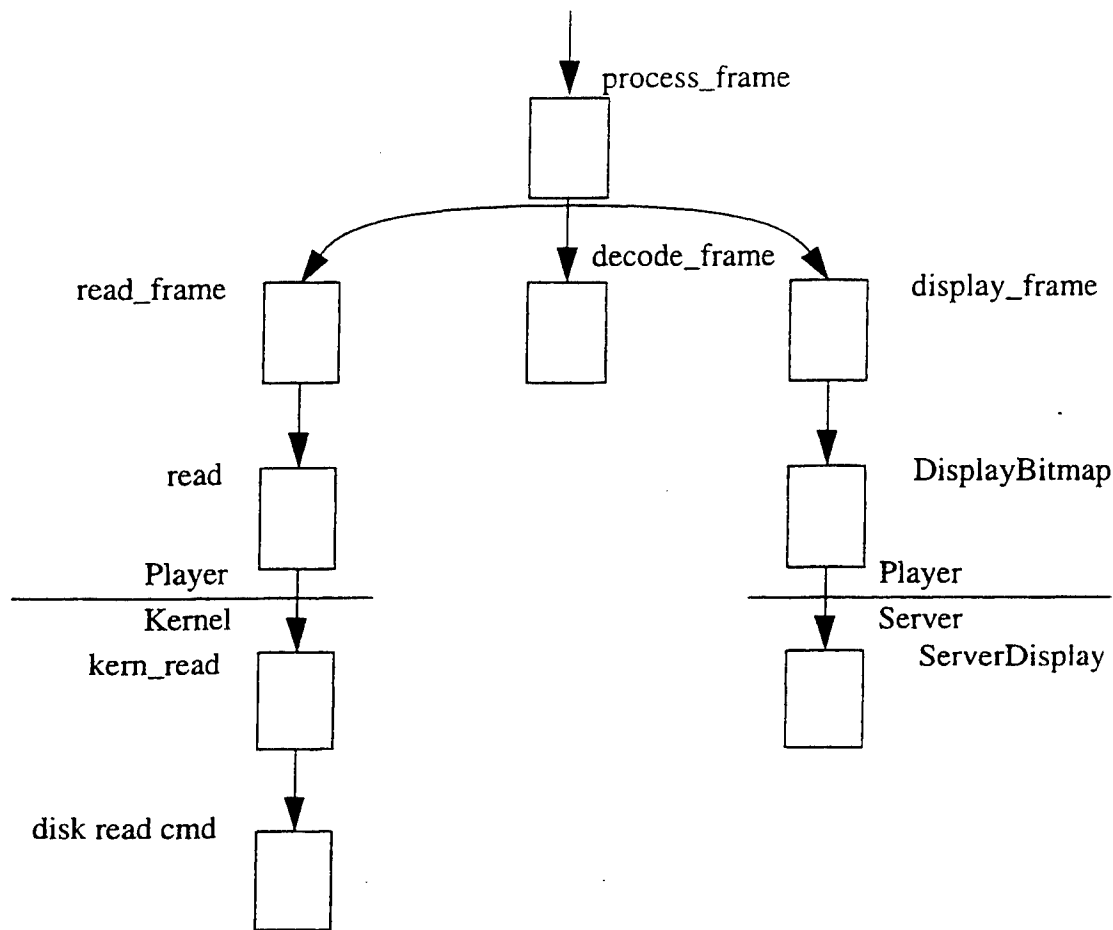


Figure 4-7: Call Graph for Video Playback

Figure 4-7 shows a possible call graph for the video playback activity. The graph is rooted at a subroutine called `process_frame`, which calls three more subroutines: `read_frame`, `decode_frame`, and finally `display_frame`. In the `read_frame` routine, calls are made until eventually the program makes a system call and traps into the kernel where more subroutines are called until finally a device command is issued to read data from the disk. This call graph introduces another type of call, referred to as a device command, which is used in addition to the original three types of calls (procedure call, RPC, and system call). The `decode_frame` routine converts the video frame data to a form that is suitable for display. The last call is to `display_frame` which is the root for a sequence of calls resulting in an RPC to a window system server which makes additional subroutine calls and finally accesses the frame buffer.

The resource requirements for this call graph include disk access as well as processor time, so a disk reserve is allocated and bound to the disk read node. The other nodes require

only the processor, and one approach is to allocate a single processor reserve for all of those. This reserve allocation and binding approach is illustrated in Figure 4-8

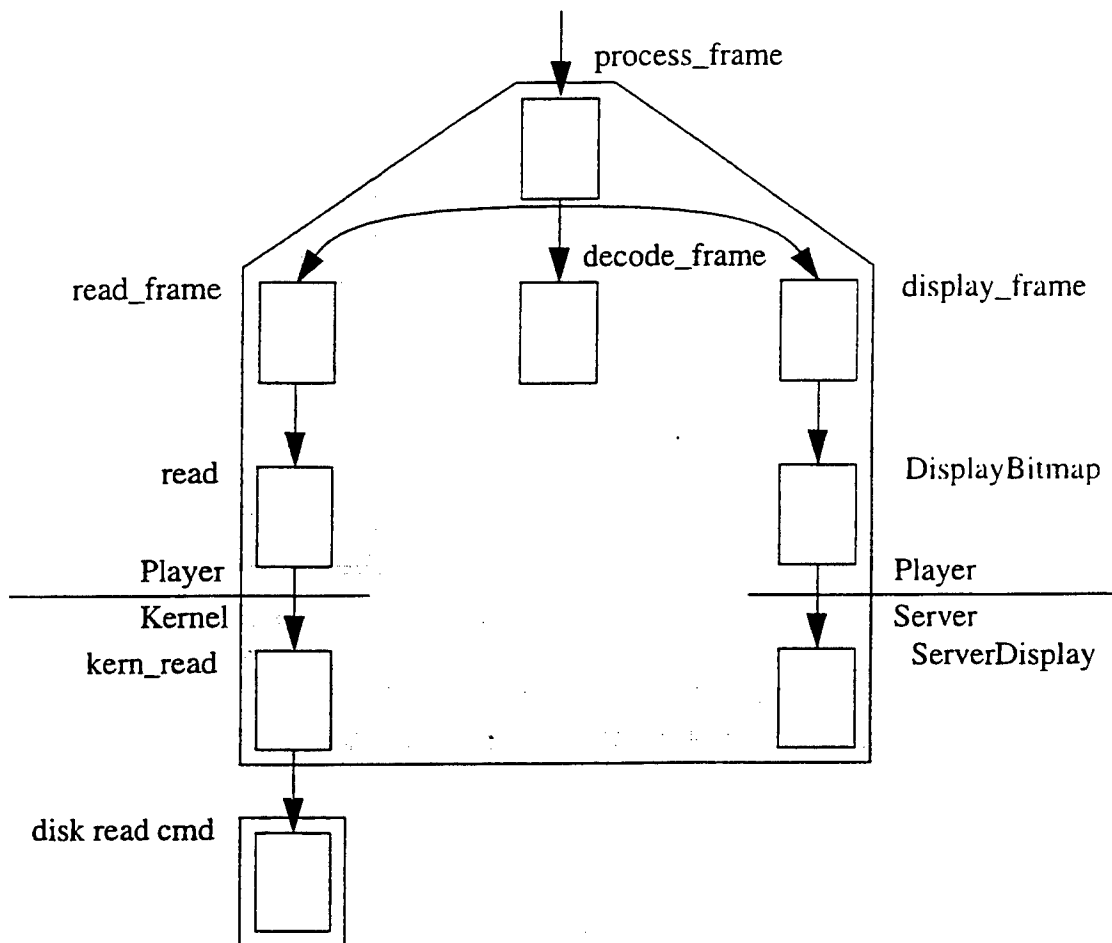


Figure 4-8: Call Graph for Video Playback with Reserves

There is a subtle problem, however, that is related to the synchronization problem between and client and server with localized reserve allocation. The resource usage pattern for the `process_frame` activity is the following:

- The processor is needed for all the nodes up to where the read command is issued to the disk,
- the disk is required for that read command node,
- all the nodes after that require only the processor.

This resource usage pattern is illustrated in Figure 4-9(a); in every period, the computation first has a processor requirement, then a disk requirement, and then another processor

requirement. If a processor reserve and a disk reserve are allocated, the execution pattern may look like the pattern shown in the first period of Figure 4-9(b).

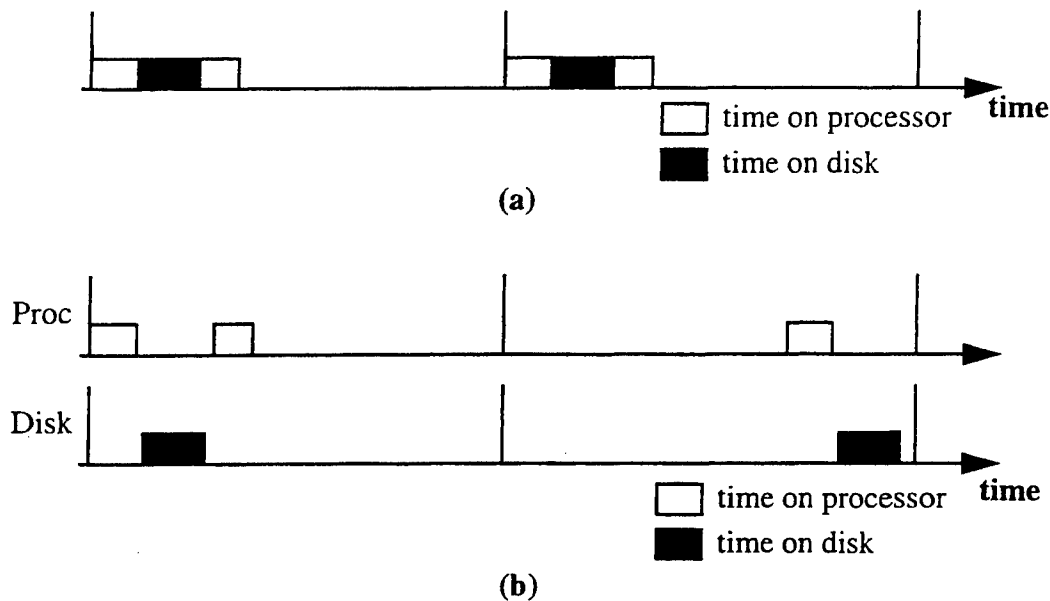


Figure 4-9: Synchronization Problem with Multiple Resources

However, the only guarantee associated with the processor reserve is that the processor time reserved will be available by the end of the period. If that time happens to only be available at the very end of the period, the execution pattern might look like the one in the second 33 ms period of Figure 4-9(b). In that second period, the leading processor requirement is serviced too late, and by the time the disk activity is finished, there is no more time left in the period for the second half of the processor requirement, and the deadline is missed. Worse still, if the reserved disk usage is only available at the beginning of the disk reservation period, the activity will be delayed into the next reservation period and certainly miss a deadline and possibly miss the following deadline as well.

One way to solve this problem is to introduce intermediate deadlines in different stages of the computation to separate sub-computations that use different kinds of resources. For example, Figure 4-10(a) illustrates the usage requirements and new intermediate deadlines for the video playback activity.

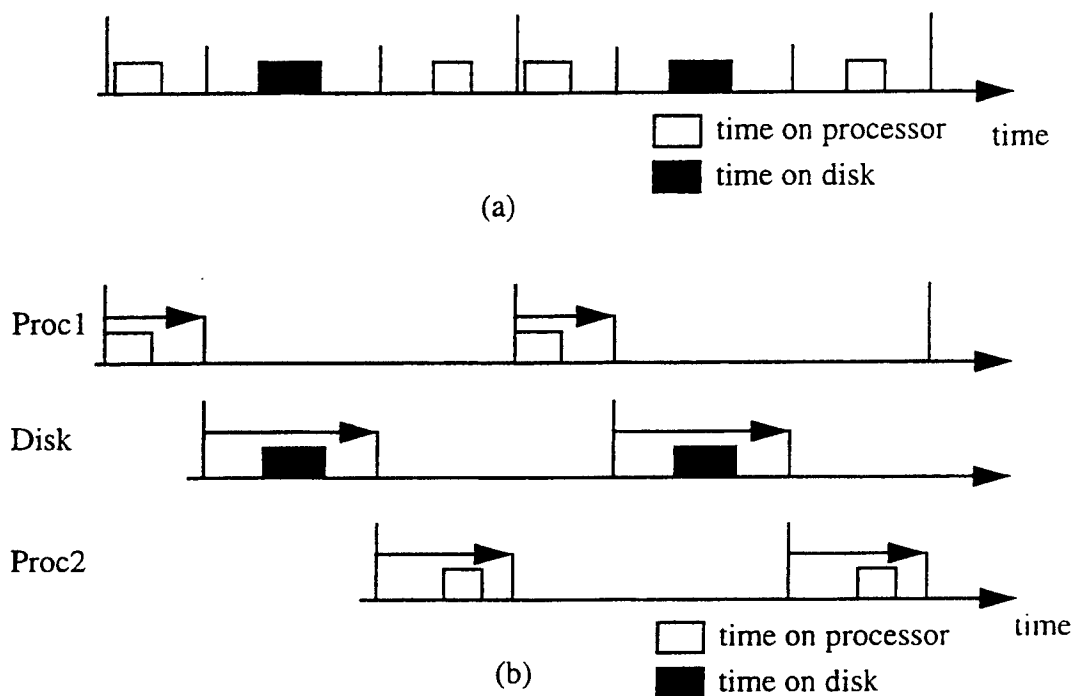


Figure 4-10: Multiple Resources Used with Intermediate Deadlines

A reserve is allocated for each of the three phases of the computation: the leading processor requirements, the disk requirement, and the final processor requirement. Since the end-to-end timing requirement or deadline is divided up into intermediate deadlines for performing the three phases of the overall computation, the reserves that are associated with the phases must have deadline parameters. Figure 4-10(b) shows a timeline for each reservation and how the usage is timed in the three reserves.

So with this approach, two processor reserves (labeled Proc1 and Proc2 in Figure 4-10) and one disk reserve are allocated. The call graph with this reserve allocation and binding appears in Figure 4-11.

This example points out two major factors that influence how reserved computations should be structured and how reserves should be bound to the sub-computations. One factor is the temporal sequence of the resource requirements. Generally speaking, a node in the graph that requires a resource different from its parent acts as a delimiter for grouping computations that can use the same reserve. To minimize the number of reserves required, the application programmer should minimize the number of times computations must switch between required resources.

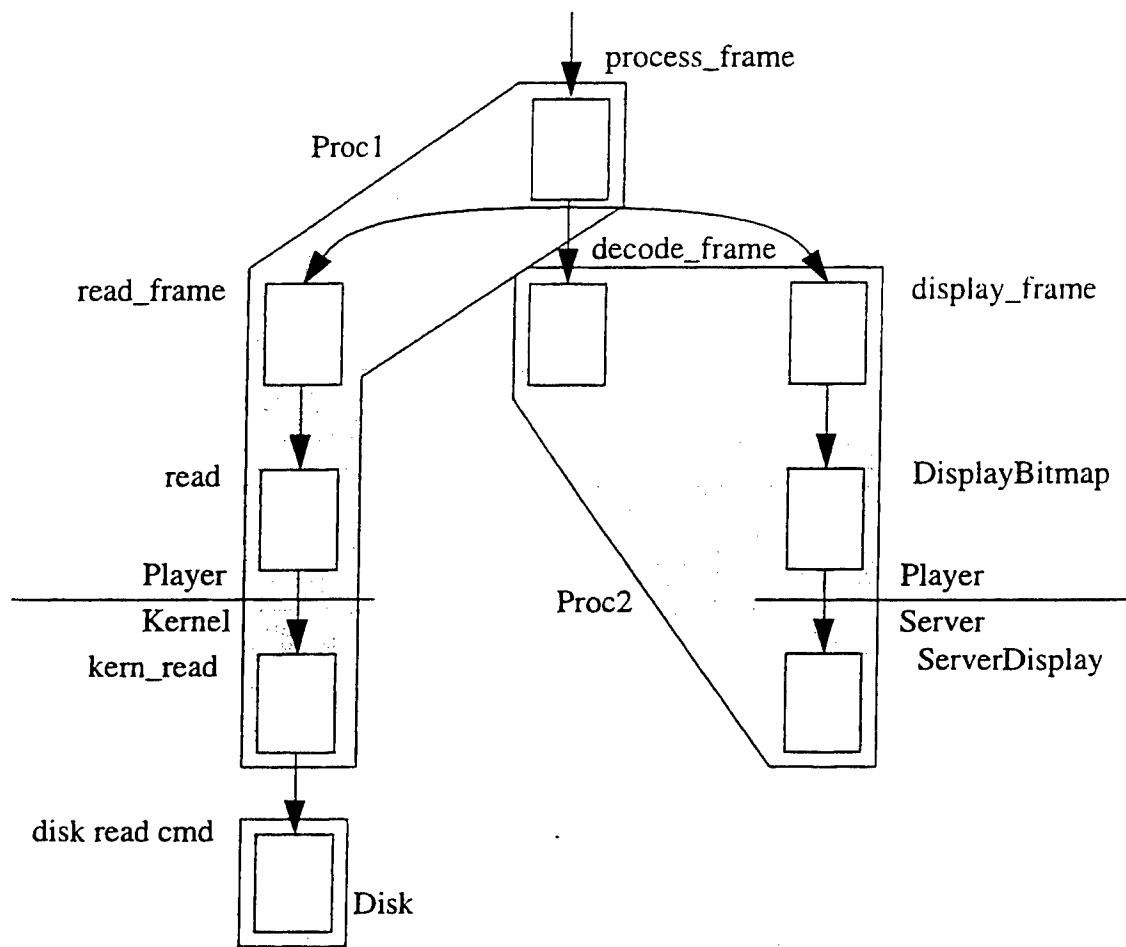


Figure 4-11: Video Playback with Better Reserve/Computation Mapping

The second factor is the spatial organization of the nodes in the system. When a call crosses to another address space or to the operating system, a decision has to be made about whether to switch reserves at that point or not. The system designer has more control over that choice, as described above.

4.3 Sizing Reservations

To use resource reserves, an application must specify appropriate reservation parameters. For hard real-time applications, the reservation parameters would be determined *a priori* by the system designer. For dynamic real-time applications, external agents such as a QOS Manager may suggest or require different reservation parameters during the course of the application's execution. In the dynamic framework, it is important for the application to be aware of the resources required to do the work it needs to do and to be flexible in terms of

its timing requirements (how often or under what delay bounds it does the work). In general, two important questions must be addressed:

- How can the resource usage requirements of an application be determined, especially given that the application may be used on different machine types and system configurations?
- How should an application adjust its resource reservations using information about previous performance?

This section deals with the determination of initial resource reservations and adjustment of reservation levels for dynamic periodic activities.

4.3.1 Determining resources required

The first problem to be addressed is how to determine which resources will be needed during the course of the computation and how to determine the initial reservation levels for the various resources required by an activity.

The resources that are needed during the course of a computation will depend on what external services are used by the computation. The list of resources for the computation will be the union of all the resource lists for the transitive closure of external services used by the computation. It is therefore very important for services to be named so that each service can name those services that it uses. And in turn, each service must name those resources that it uses. Then it is possible to find the resources used by the transitive closure of services the application uses.

A potential problem is that different functions offered by a service may use different resources. If a client uses only one function offered by the service, it should reserve only the resources needed for that function rather than the complete list of resources needed by every function the service offers. In this case, it may be useful to consider the resource lists required for "sub-services" or subsets of operations of the service where the subsets are defined to use similar sets of resources.

In any case, the method for ascertaining required resources should be flexible, efficient, and easy to use. Ideally, the system would help to determine the list of resources during an initialization phase of each application. Each time the system encountered an application that required a particular resource but had no corresponding reserve, it would add a reserve of the appropriate type with no reservation parameters to the reserve tree bound to the application. After the initialization phase, the application would have references to the resource that were required by its component computations, even those resources used by servers that were called on its behalf.

Other approaches to determining resources required could be used as well including the following.

- The list of resources could be obtained by sending a query to every server to be used in the computation and having the server provide a list of resources it requires and a list of services it uses. The transitive closure of

required resources collected during this query would be accurate as of the server connection time. This method does not require that the system and (possibly more static) documentation remain synchronized with respect to specification of resources used. Upgrades for various system software modules could be made without having to issue new resource list documentation. Also, servers would be free to determine which resources, among many possibilities, would be best to serve that connection given the state of the system and its load at the time the connection was requested. Thus an additional degree of freedom is allowed the servers.

- The list could be found using a database where each server registers the list of resources as well as other services it requires. This makes it possible to write applications that automatically determine the transitive closure of services used and resources required, even if some of those services and resource types did not even exist at the time the application was developed and compiled. One important requirement to make this dynamic method work is that service names and resource types not be hard-coded. Instead, a program should be able to handle and manipulate new service names and resource names with no recompilation.
- The list of resources required by various user-provided services and system services could be static, long-lived, and well documented. The programmer must manually look up all the services and find the transitive closure of services used and then the union of the resource lists of all those services. The major problem with this approach is that the slightest changes to the software for the services may change the list of services used and the resource list, thus making the lists in the manual obsolete and making all the programs written to the specification of the manual obsolete.

4.3.2 Determining initial reservation levels

Once the programmer knows what resources are needed by an activity, she must set up the reserves for those resources and request reservations. Requesting reservations requires that reservation parameters be provided. In the reserve model, a reservation request has parameters for resource time to be reserved and for a reservation period. In many cases, the reservation period will be the same as the period of the activity. This will sometimes be derived directly from user-level quality of service requirements (such as frame rate), and sometimes it will be derived indirectly from user-level requirements. For example, the rate for handling audio packets might depend on the audio sampling frequency, the packet size, and perhaps the system overhead per packet. The resource usage time is more difficult to ascertain. It depends on the platform, the system software, and the data being processed among other things.

One way to get a reasonable estimate as to what the resource usage requirements might be for a given instantiation of an application involves measuring the actual computation that forms the main focus of the application. With one run through a periodic activity, for exam-

ple, the application could get a fairly good estimate of future computation times using the reservation mechanism's usage measurement features. Another variation on this approach is to use a simple computation to gauge the speed of the machine and/or system architecture, and then use a characterization of the real application's computation expressed in terms of the simple computation to estimate the appropriate reservation level. For example, if the application first ran a SPECint benchmark and knew how much the reserved computation needed in terms of SPECint benchmarks, it could derive the estimate directly.

The following methods could also be used to determine the initial reservation level:

- An application could store in a persistent preferences database some information about reservation levels used in previous instantiations of the application. This information would be a good guess as to what reservation levels should be procured, and it might be possible to maintain a small database to map prior experience with different QOS parameters to reservation parameters. This approach might get much more complicated as more QOS parameters, reservation parameters, and target system architectures are used.
- The initial reservation level could be set to zero or some other relatively small value that is known to be smaller than the actual reservation level, though unknown, that will be required. This approach requires the mechanisms for reservation level adaptation to quickly acquire the feedback on usage that is necessary to set a reasonable reservation level where desired quality of service parameters can be achieved. Initially, the desired QOS parameters will not be achieved and they may never be achieved. These are the major drawbacks of this no-knowledge approach.
- An alternative approach to the zero level initial reservation is to take the maximum reservation level available on the resources at the time the reservation is requested. This has the advantage of having the highest chance of meeting the desired QOS parameters for the application, but the disadvantage is that resource capacity may be unnecessarily tied up and unavailable to other applications requesting reservations. This situation would persist until the adaptation mechanism had the chance to evaluate the situation and make the proper adjustments to the reservation levels.

4.3.3 Measuring performance

An adaptive reserved application should keep track of the resource usage required to perform its computation at each repetition to decide if it has more resource capacity reserved than it needs or if it has too little resource capacity to do its work during each period. It should also keep track of the real-time delay incurred during each repetition of the computation to determine whether the computation was completed within the period or not.

The application can easily measure the real-time delay of a computation by taking a timestamp at the beginning and at the end of the computation. Measuring the resource

capacity usage for a computation, however, is more involved, requiring support from the operating system. This support must be more accurate than the traditional logical clock for processor time provided to processes in most operating systems. Logical clocks usually take usage measurements by sampling at clock interrupts to find which process is running and incrementing that process's logical clock as if it had been executing for the entire period. Statistical sampling of this kind, which is inherently inaccurate for short-term measurements, will not provide an application with the clear picture of short-term behavior. Such knowledge of short-term behavior is needed to be able to make suitable adjustments to the reservation parameters.

For the kind of accuracy required for measurements of resource capacity usage, the system must accumulate usage associated with reserves at each context switch. In this context, reserves act as *abstract thread* logical clocks rather than *process* logical clocks. And since the reservation system manages capacity usage for resources other than just the processor, the system must keep usage accumulators for all types of resources, and these must be updated at each context switch on the appropriate resource.

Reserve usage measurements will indicate how an application's actual behavior is related to its reservation. Several possible patterns of behavior are described in the next sections.

4.3.3.1 Balanced applications

An application is balanced with respect to its reservation if the resource usage in each period is fairly constant and the reservation level is at this constant value. (It may be impossible for resource usage to be completely constant for some interesting resources such as processors.)

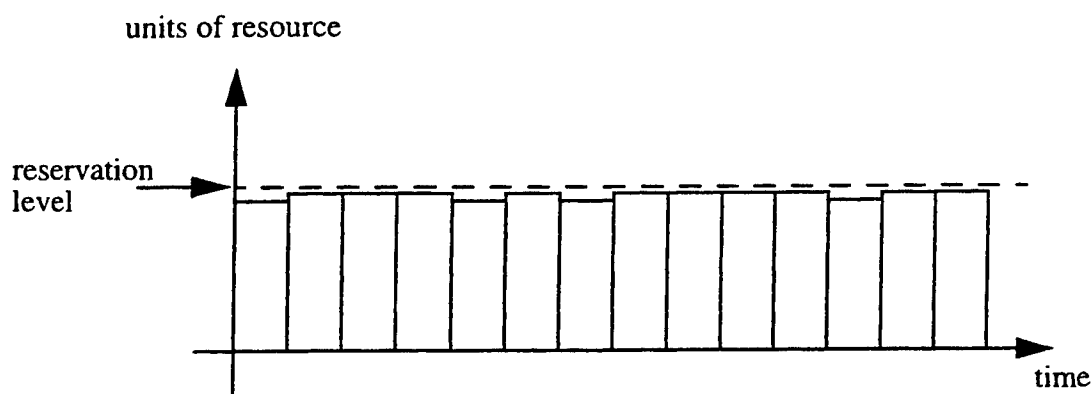


Figure 4-12: Resource Demand Constant and Reserved

Figure 4-12 illustrates a computation's demand on a particular resource over time. Time is on the x-axis, and it is divided into intervals equal to the reservation period. The y-axis is units of resource, e.g. time spent on the processor executing instructions, bytes transmitted,

etc. Within each reservation period, the number of units of resource consumed by the computation is measured, and the height of the bar in that interval is the number of units consumed.

In the figure, the number of units of resource required in each reservation period is nearly constant, and the reservation level is slightly more than this constant demand. Therefore, the demand is satisfied by the reservation, and the computation will have the resources to be able to execute completely in each period.

4.3.3.2 Under-reserved applications

An application is under-reserved with regard to a particular resource if its resource usage requirement is greater than its reservation. Two cases are distinguished:

1. worst-case (maximum) resource usage requirement for the computation is greater than the reservation but the average resource usage requirement is less than the reservation, and
2. the average resource usage requirement is greater than the reservation (implying that the worst-case resource usage requirement is also greater than the reservation).

In the first case, the average resource usage requirement is less than the reservation, so over the long term, the application will be able to keep up with its work requirement. The problem is that since the worst-case resource usage requirement is larger than the reservation, the completion of the worst-case computation may be delayed and this may delay or otherwise affect the computations in subsequent periods. If the worst-case computation occurs very infrequently, its negative affects on the overall performance of the application can be minimized or ignored. A human viewer may not even notice an occasional dropped frame during video playback. If the worst-case computation occurs frequently, it may be more difficult to ignore; many dropped video frames would certainly be noticed.

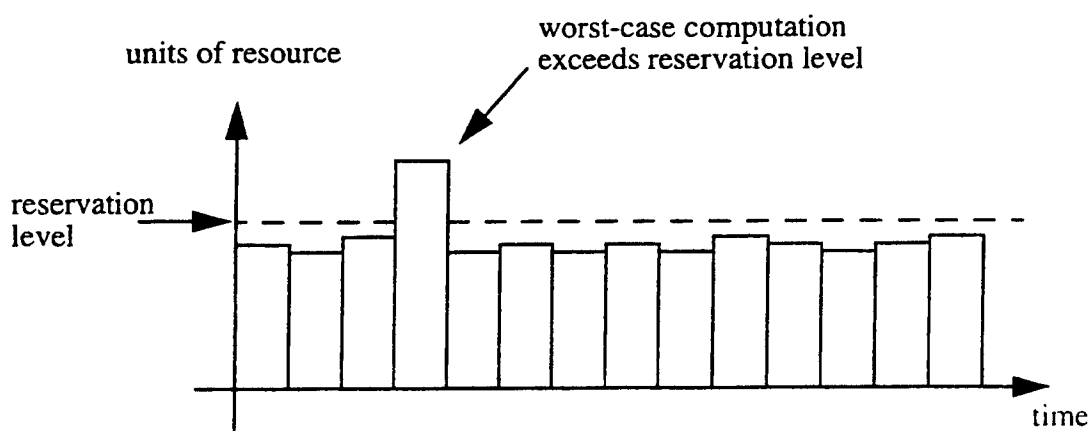


Figure 4-13: Resource Demand Occasionally Exceeds Reservation

Figure 4-13 illustrates this slightly underreserved case. The resource usage requirement in most reservation periods is less than the reservation level for this particular resource. In these periods, the computation will have the resources available to complete. However, there is one period in the illustration (the 4th) in which the resource usage requirement is larger than the reservation. Depending on the system's policy for treating this case, the computation may happen to be completed (using idle time), it may be aborted, or it may extend into the next reservation period, interfering with the completion of the computation which would normally execute in that period.

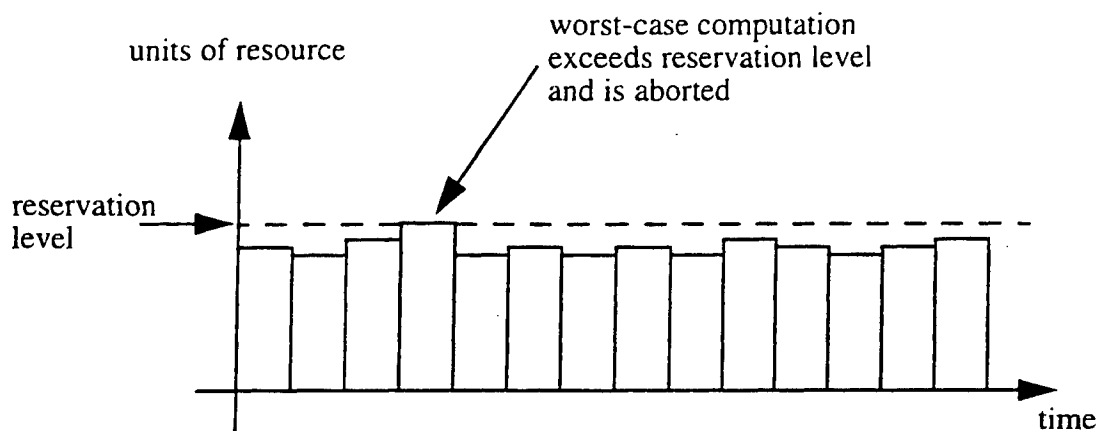


Figure 4-14: Exceedingly Demanding Computation Aborted

Figure 4-14 shows how the usage pattern would look if the computation in the 4th reservation period were aborted. Note that the computations in the subsequent periods are not affected by the aborted computation.

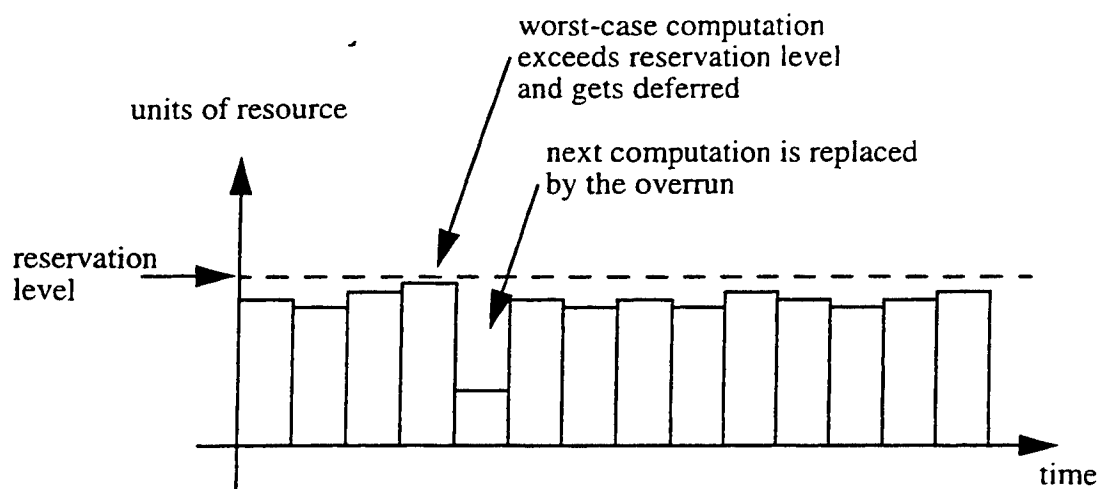


Figure 4-15: Computation Impinges on Following Computation

Figure 4-15 shows the case where the computation in the 4th reservation period extends into the next reservation period and prevents the next computation from being initiated. Computations following that are left undisturbed.

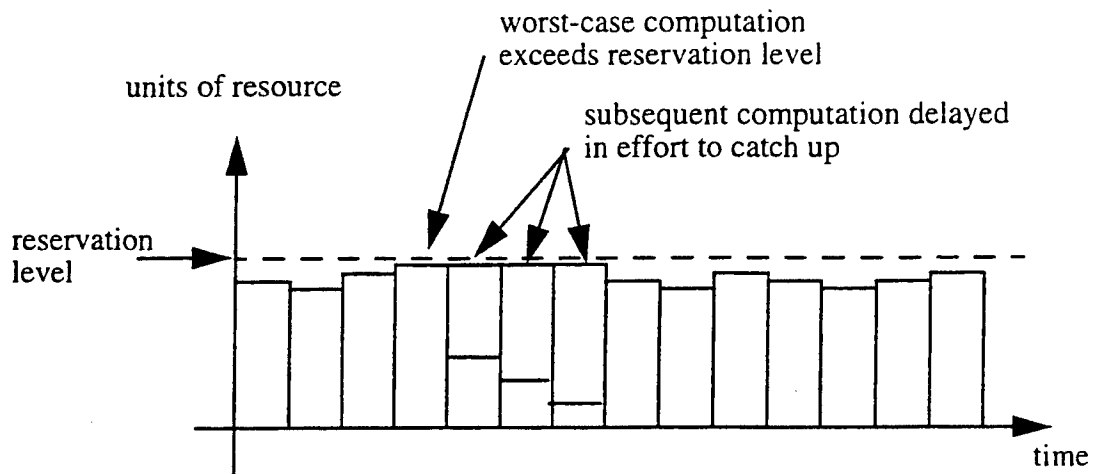


Figure 4-16: Computation Impinges on Subsequent Computations

In Figure 4-16 the 4th computation overruns its reservation period and consumes part of the next period. The computation associated with that period is initiated after the previous computation is completed (as opposed to the previous case where this computation was not initiated). But since the 5th computation is initiated later than usual, it also overruns its reservation period and is deferred to the next period. This cascading effect continues until there is enough (normally) unused but reserved units of resource to make up for the original overrun.

In cases where the average resource usage requirement is more than the reservation, the activity will never be able to accommodate all of the computations which overrun, and it would be necessary to shed some of the load by aborting some computations or by not initiating some computations. In either case, the attempted overruns would occur frequently and have a potentially damaging effect on overall application behavior. In a video player, for example, this would mean that many frames get dropped.

Figure 4-17 illustrates a possible pattern of demand that has the average demand greater than the reservation level. The computations in several reservation periods require more than the reservation for that period. Many of the computations will have to be aborted if there is no idle time available beyond the reserved level.

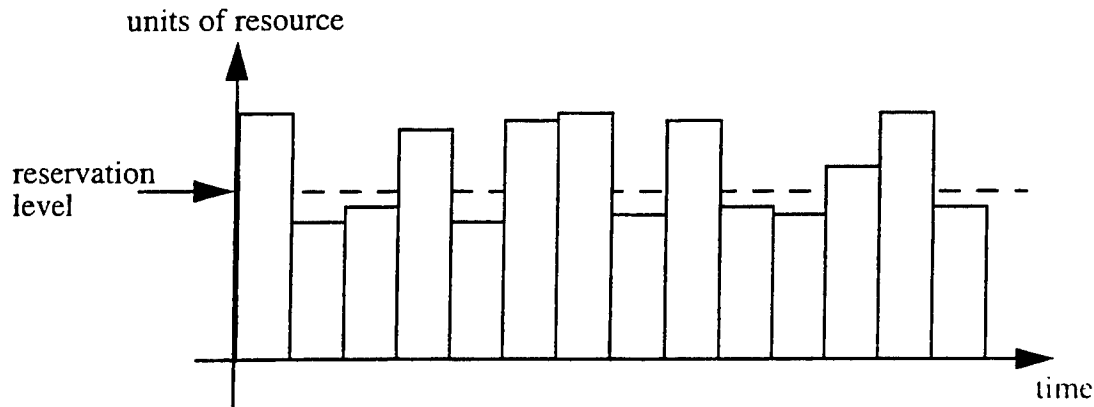


Figure 4-17: Average Demand Exceeds Reservation

4.3.3.3 Over-reserved applications

An application is over-reserved if the resource usage in each period is (much) smaller than the reservation level.

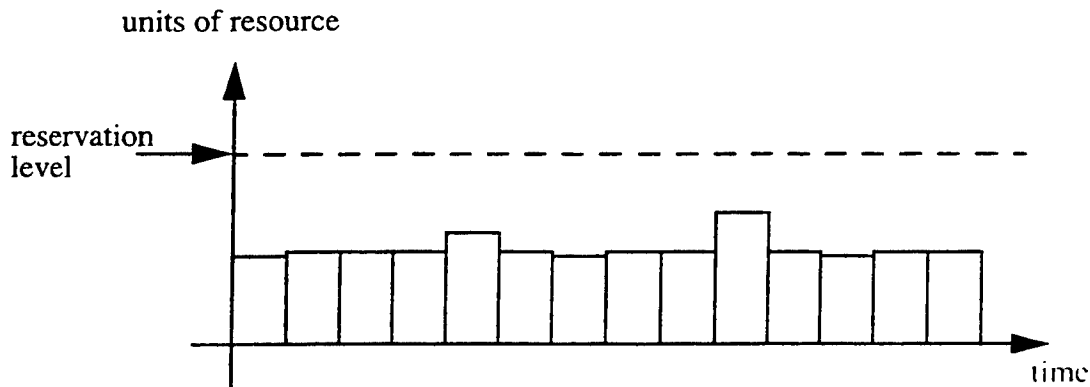


Figure 4-18: Resource Demand Smaller than Reservation

Figure 4-18 illustrates a case where the usage on a particular resource is much smaller than the reservation in all of the reservation periods. Here the computation is never in any danger of overrunning into the next period.

4.3.3.4 Multiple resources

When there are multiple resources involved in each computation, the measurements of usage compared to reservation level for each reserve will be different. One resource may be over-reserved while all of the others are under-reserved, or perhaps more commonly, one

resource may be under-reserved (representing a bottleneck) while all of the other resources are over-reserved for the activity.

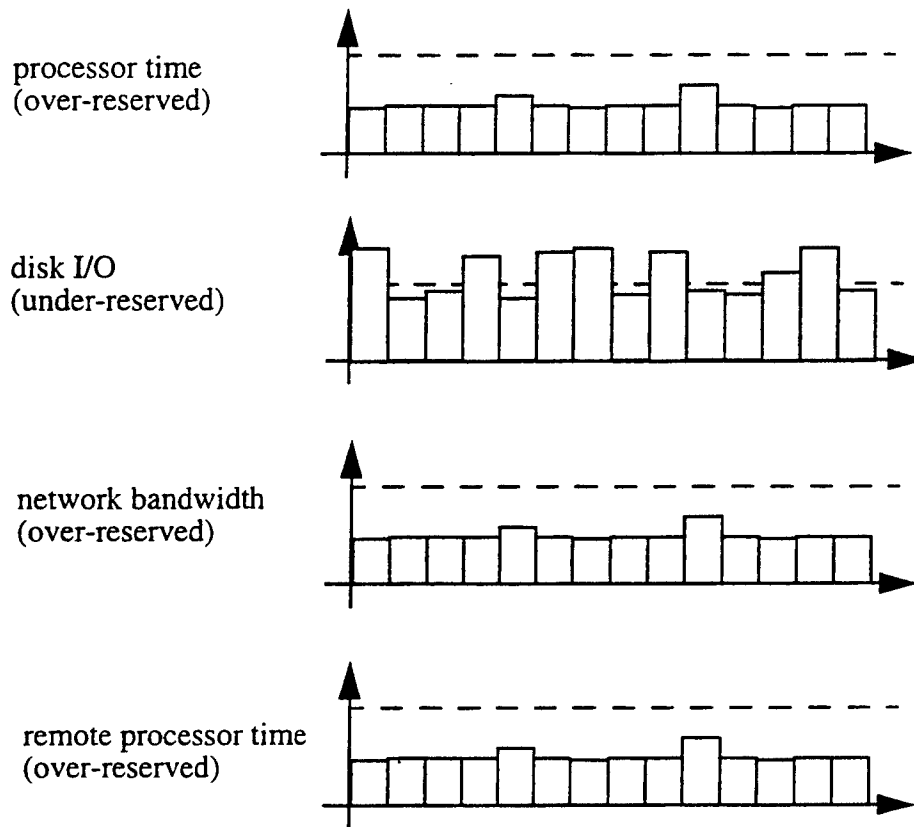


Figure 4-19: Measurements of Multiple Resources

Figure 4-19 illustrates a case where there are multiple resources involved in a single activity. They all have the same reservation period, but the demands placed on various resources are different. In this case, the disk I/O is under-reserved while the local processor usage, the network bandwidth, and the remote processor usage are all over-reserved.

Once an application has measurements of usage for the various resources it requires for its computation, it can begin to make decisions about how to modify its own behavior or modify its own resource reservation levels to achieve better performance or better efficiency.

4.3.4 Adapting

Reservation parameters can be changed dynamically as the user, the application itself, or a central quality of service manager determines that new reservation parameters would be preferable. Applications adapt based on the influences of various external entities, but once a resource reservation is made, the system ensures that the resources are available.

Adaptive applications can measure their own performance by mapping measured system resource performance metrics to application-specific performance indicators. This application-specific information can then be used along with application-specific performance objectives to:

1. modify the computation being done by the application (to change resource requirements),
2. modify the reservation level for resources being used by the application, or
3. do nothing.

The performance measurement interval could be comparable to the period of the repetitive computation, but it would be more efficient if the adaptation interval were an order of magnitude larger than the activity period (e.g. the adaptation might occur every 500 ms for an activity with a 50 ms period).

4.3.4.1 Modifying an application's computation

One way that an application might react to the fact that its resource usage is different from its reservation is to change its behavior so that its usage more closely matches its reservation (leaving the reservation unchanged). The actual mechanisms for modifying behavior in an application are fairly straightforward. An application which is meant to modify its own behavior must have different behaviors available (i.e. different algorithms implemented internally). It must be able to tell which algorithm it should use depending on the format of an incoming or outgoing data stream, on the resources such as network bandwidth or computational power that are available to it, or on the limitations of other software with which it must interact. For example, an MPEG video decoder could use different decoding or dithering algorithms depending on the resources available. If the decoder were taking the MPEG data stream from a server in real time, it might be able to negotiate MPEG encoding parameters with the server and have the server place new parameters in the data stream. Thus, adaptive applications are constrained by:

1. the algorithms they have coded,
2. the data formats they are using,
3. the data formats and data rates that other components of the pipeline can handle.
and
4. the resources that are available to it.

Typically, the application would contain a collection of algorithms that could be ordered based on processor requirements, network bandwidth requirements, etc. Thus, once an adaptive application decided to increase or decrease a reservation on a particular resource, it could determine which algorithms could satisfy that constraint. It is important to distinguish between two kinds of behavioral adaptations:

1. local changes in algorithm and
2. global changes.

An example of a local change would be a change in the number of bits actually being decoded by the receiver of a bit stream (assuming the bit stream was encoded using a hierarchical encoding scheme). A global change would involve not only the receiver but the sender. It would require a way for the receiver to request a change in the format or number of bits being sent as well as requiring the receiver to recognize that the format of the bit stream changed.

As an example of what the code structure for an adaptive application would be, consider a video player. The basic control structure of the player is a loop that reads some data (from a disk, network or some other source), decodes the data, displays the data, and evaluates its performance.

```
while (1)
    read data for a frame
    decode data
    display frame
    evaluate resource usage
```

It is the evaluation part of this loop which will look at the resources that are being expended over time to play the video frames and decide whether the amount of work being performed should be increased, decreased, or remain the same. If it decides the work should be *locally* increased or decreased, it may change some state in the player to indicate how the data should be decoded (by looking at more or fewer bits of the data). If the evaluation phase decides that the work should be *globally* changed, it may initiate negotiations with the source of the data stream to try to increase or decrease the bandwidth of the bitstream. This negotiation may or may not change the state of the player itself, but the changes to the characteristics of the bit stream and the point in the bitstream where the change takes place should be clearly identified in the player and should be recognized in decoding the data. Thus, if and when a change in the format of the bitstream occurs, the player will be able to make the appropriate changes in decoding the stream.

So the software structure of the adaptive player, with a little more detail filled in, becomes:

```
while (1)
    read data
    check for control data
    switch based on bitstream format
        switch based on local decoding state
            decode data
    display frame
    evaluate performance
```

The application first reads the data and then checks either for control information embedded in the data or for some kind of synchronization point which is known, through information communicated via an external channel, to imply a change in data format. Then the proper algorithm is chosen to decode the data based on the format of the bitstream and on the local decoding state. Once the data is decoded, it is displayed (possibly using different algorithms indicated by the player state), and finally the performance is evaluated. To reduce overhead, the performance may not be evaluated during every iteration of the loop.

This example shows how a player might change its behavior and thus its performance characteristics based on decisions about local algorithms and global changes in data streams.

4.3.4.2 Adjusting reservation levels

Another way an application might react to noticing a difference between its usage and reservation is to change the reservation (without modifying its computation). We will examine several cases, possible behavior modifications, and their effects on delay, efficiency, and total reservation.

Under-reserved applications

As indicated in the section on measuring resource usage, an application is under-reserved if its resource usage requirements are greater than its reservation level. There are a couple of ways to change the reservation parameters to accommodate this situation:

1. increase only the units of reserved resource usage
2. increase both the units of reserved resource usage and the reservation period.

By increasing the reserved resource usage to match the computation's requirements, the application can ensure that the resources will be available to the computation, and the computation can be invoked just as often as before. The delay experienced by each computation will be decreased since there will be fewer overrun situations to cause delays, but the overall reservation level is increased. This means there is less resource capacity available for other applications, or when resource reservations are really tight, it may be impossible to increase the reservation level at all.

If the application increases the units of reserved resource and the reservation period proportionally, there will be enough reserved resource capacity in each reservation period to service the computation. And since the reserved amount and the reservation period are increased proportionally, the overall reservation level is not increased. This also implies that the computation is requested less often to correspond with the longer reservation period since requesting it just as often would not reduce the overall workload (without a load-shedding mechanism coming into play).

Over-reserved applications

Over-reserved applications are those which have a reservation level that is greater than the actual resource demand. This situation is inefficient since the application has more resource capacity reserved than it expects to use, and that reserved capacity could be used to ensure predictable performance for other applications that will actually use the resource.

The simplest action to take in this case is to reduce the units of resource reserved in each period to a value that is closer to the actual resource requirement. It is possible to increase the reservation period without increasing the period of the computations to decrease the reservation level, but that would have other undesirable effects on the timing of the program, such as increasing the delay for some computations.

4.4 Chapter summary

This chapter describes how programs should be structured to take advantage of reserves for predictable real-time performance. For hard real-time applications, information about the resources used by and timing requirements of each program must be known at design time and must be used in planning reserve allocation. So a localized way of using reserves might be appropriate. With local reserves, each program allocates its own reserves based on its requirements and the requirements of other programs that depend on it. For dynamic soft real-time systems, a global method for reserve allocation where an activity allocates the resources for all of its constituents including external servers and operating system services might be more appropriate. This makes it easier to monitor and control the usage of the entire activity rather than just localized parts of it. These recommendations are not cast in stone; the choice of whether to use localized or global reserve allocation ultimately rests with the system designer.

The discussion also dealt with methods for determining resources required by an application, reservation parameters appropriate for an application, and adaptive methods for adjusting reservation parameters or behavior based on performance history. In hard real-time systems, many of these questions must be answered at design time, and there is less flexibility in adaptation strategies. Soft real-time systems, however, have a great deal of flexibility and can take advantage of some of the techniques described in this chapter.

Chapter 5

Implementation

This chapter describes an implementation of processor reserves done using the Real-Time Mach operating system. It discusses applications that were modified to use processor reserves, network protocol processing software modified to use reserves, a QOS manager for negotiating resource allocation with applications, and tools for reserve monitoring.

5.1 Overview

This chapter describes the implementation of processor reserves in RT-Mach as well as several other components of the system. It also covers some applications that were modified or designed and implemented to use processor reserves. Figure 5-1 shows the various components and gives an indication of their relationship.

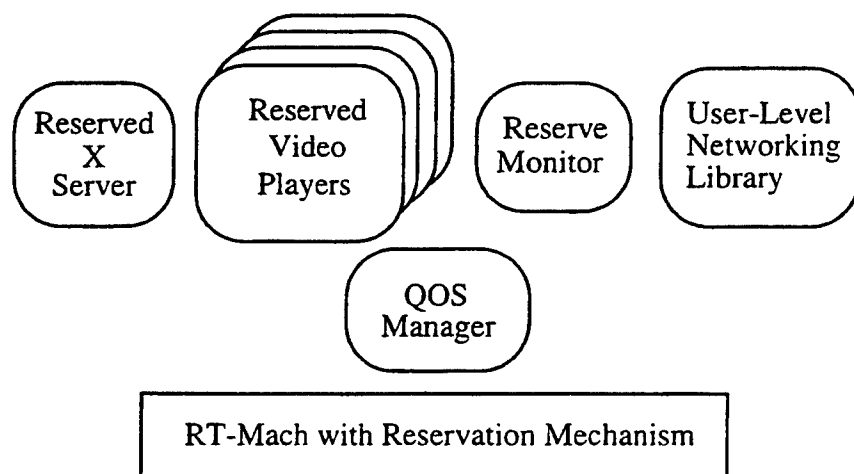


Figure 5-1: System Components

First there is the implementation of processor reserves in the RT-Mach kernel which is the basis for the rest of the implementation work. Reserves were implemented as a new kernel abstraction with operations for create/terminate, requesting reservation parameters, binding threads to reserves, and extracting usage information about reserves.

Several real applications were modified to use processor reserves including: a Quick-Time video player developed at CMU called QTPlay, an MPEG decoder called mpeg_play [93], and a version of the X Server [34].

A version of the user-level socket library [70] was modified to use reserves as well. This socket implementation supports predictable performance for applications that send and receive network packets.

A QOS manager was implemented to allow for more sophisticated negotiation of reserve parameters than that provided by the kernel mechanism itself. The QOS manager interacts with applications to try to balance resource usage and negotiate with applications when conflicts arise in the resource reservation requests.

Tools were implemented to help manage reserves and to monitor resource allocations and measure usage. The rmon application is a reserve monitor that provides a graphical user interface for reserves. It displays the reservation levels and usage in recent history, and it also allows the user to change the reservation parameters from the graphical interface.

5.2 Reserves in RT-Mach

The implementation of processor reserves in RT-Mach involved adding the new reserve abstraction, implementing the operations on reserves, creating a new scheduler, and adding code for accurate usage measurement. In addition to the new scheduler, the reserve implementation also required modifications in the RT-Mach priority inheritance mechanisms to support reserve inheritance and reserve propagation.

The reserve abstraction in RT-Mach is managed much like the other abstractions like hosts, processor sets, tasks, threads, etc. that originated in Mach 3.0 [11]. These types of resources in Mach are referenced by ports, which are used as capabilities.

In RT-Mach, processor reserves are allocated from processor sets. In the uniprocessor version, there is only one processor set, so all reserves originate from this processor set.

5.2.1 Attributes and basic operations

Abstractions like tasks and threads offer basic operations such as create, destroy, get attributes, and set attributes. The basic operations on reserves are as follows:

reserve_create(out reserve) Creates a new processor reserve and returns it as an out parameter.

reserve_terminate(reserve) Terminates the given reserve, making its reserved capacity available for other requests.

reserve_set_attribute(reserve, attr_name, attr_value, attr_value_size) Set the value of an attribute of the reserve.

reserve_get_attribute(reserve, attr_name, out attr_value, out attr_value_size)
Get the value of an attribute of the reserve.

processor_set_reserves(processor_set, out reserve_list) Returns the list of reserves associated with the given processor set.

The get attribute and set attribute operations give the programmer access to some of the attributes of reserves. The externally visible attributes that reserves have appear in the following list. The data types for the attributes are given in parentheses after the attribute names; "int" is an integer, "timespec_t" specifies a time value, "mach_reserve_name_t" is a fixed length string, and "boolean_t" is a boolean flag.

name (mach_reserve_name_t) A symbolic name for the reserve.

ckpt_total (timespec_t) The cumulative total usage measured at the at last period boundary.

ckpt_time (timespec_t) The absolute time of last period boundary (when the ckpt_total value was recorded).

accum_total (timespec_t) Cumulative total usage at the current time (usually updated when accessed).

accum_time (timespec_t) The time that the total usage was last updated.

used (timespec_t) Usage charged against the reserve so far in current period.

next_period (timespec_t) The absolute time of the next period boundary (end of the current period).

period (timespec_t) The duration of the reservation period.

computation (timespec_t) The reserved computation time.

recent_checkpoint_position (int) The position in the recent_checkpoints array for the next item to be written. The array is a circular buffer.

ncheckpoints (int) The number of how many checkpoint entries currently in the array.

recent_checkpoints[MAX_CHECKPOINT_COUNT] (timespec_t) The usage values for the recent checkpoints.

recent_checkpoint_times[MAX_CHECKPOINT_COUNT] (timespec_t) The times at which corresponding checkpoint usage values were recorded.

A "checkpoint" occurs at each period boundary for each reserve. At that time, the accumulated usage is recorded along with the absolute time of the period boundary. This information can be used later by applications or monitoring tools that need information about how much usage was charged against a reserve in a particular reservation period.

There are a few other reserve attributes that are only used internally. They form part of the scheduling state for a reserve and are not available through the get attribute operation. These internal attributes are:

reserved (boolean_t) The bit that indicates whether the reserve is in “reserved mode” or “unreserved mode”.

wait_replenish (boolean_t) An internal flag indicating that the reserve has a reservation that has been depleted for the current period. The reserve is awaiting replenishment.

start (timespec_t) The absolute time at which the first period for the reserve started.

5.2.2 Reservation requests and admission control

When initially created, a reserve does not have an associated resource reservation. Getting a resource reservation for the reserve requires an additional call. The following operation allows a programmer to specify reservation parameters and request a reservation. This call is used when there is no reservation associated with the reserve, or if the programmer wishes to request a reservation with parameters that are different from the reservation associated with the reserve.

reserve_request(reserve, reservation_parameters) Requests a resource reservation to be allocated to the reserve. The caller provides the reservation parameters. Reservation parameters include desired reserved time per period, the period itself, and start time for reservation to take effect.

This operation is used to request a reservation with certain parameters. If there was previously no reservation associated with the reserve and the reservation request succeeds, then the operation returns “success” and the reservation is granted for that reserve. If the reservation request fails, the operation returns the error, and the reserve is left without a reservation.

If the reserve already had a reservation at the time the request call was made, the behavior is as follows. If the new reservation request is granted, the new reservation parameters will be associated with the reserve, and the old reservation will be freed in this process. If the new reservation request fails, the old reservation parameters remain in effect; that is, the old reservation will not be freed if the new reservation request cannot be granted.

The request operation invokes the admission control policy to determine whether the new reservation request can be accommodated given the collection of other reservations that have already been accepted for the resource. The RT-Mach implementation uses an admission test based on rate monotonic analysis, but the decision is somewhat optimistic in that it uses a utilization bound of 90% for testing for schedulability. This is based on the analysis of average schedulable bound [63], which says that for a randomly generated task set the schedulable bound is 88% on average.

5.2.3 Scheduling

The scheduler in RT-Mach was structured so as to make it easy to develop and incorporate scheduling policies. The scheduler uses a well-defined interface for scheduler operations, and a function pointer table in the scheduler contains the operations for the scheduling policy in effect. The scheduling policy can be changed dynamically by putting the ready threads in a policy-independent queue, changing the pointers in the function table to refer to the operations for the new scheduling policy, and then transferring the ready threads into a policy specific queue.

Several scheduling policies, including the "Reserves" scheduler, are supported in RT-Mach [85]. The scheduling policies in RT-Mach are associated with "processor sets," and in the case of a uniprocessor, there is only one processor set in the system. The operations to get and set attributes of a processor set are used to query or set a scheduling policy.

processor_set_get_attribute(processor_set, attr_name, out attr_value) To get the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". The operation sets the "attr_value" to reflect the currently active scheduling policy.

processor_set_set_attribute(processor_set, attr_name, attr_value) To set the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". A value such as SCHED_POLICY_RESERVES is given for the "attr_value".

Several scheduling policies are implemented in RT-Mach. The original Mach time-sharing policy is available, as are several varieties of fixed priority and rate monotonic policies. Earliest deadline scheduling is available, and round robin scheduling is supported for experimental purposes. Reserve-based scheduling is also a policy option. The policies that are available in the MK83j version of RT-Mach are:

1. Mach Time-sharing - Original time-sharing policy.
2. Fixed Priority/RR - The fixed priority/round robin policy services threads in order of a fixed priority associated with each thread. Within a priority class, threads are scheduled round robin with a quantum.
3. Fixed Priority/FIFO - The fixed priority/FIFO scheduler uses fixed priorities as well, but within a priority class, threads are scheduled using a FIFO discipline (with no quantum).
4. Rate Monotonic - Rate monotonic scheduling based on the periods given to periodic threads.
5. Deadline Monotonic - Deadline monotonic scheduling based on the deadlines given to periodic threads.
6. Earliest Deadline First - Schedules based on the deadline information of threads.
7. Round Robin - Simple round robin scheduling (with a time quantum).

8. Reserves - Scheduling policy for the reservation system.

The “Reserves” scheduling policy uses one queue for “reserved mode” threads, which are listed in order from smallest reservation period to largest. It uses an additional table of queues for implementing a multi-level feedback queue for time-sharing or “unreserved mode” threads. The reserved mode threads are scheduled first, and when there are no more reserved mode threads, the scheduler services the unreserved mode threads. In order to prevent starvation of unreserved mode threads, the reservation parameters are limited. In the implementation, a reservation cannot have a period larger than one second. This ensures that no reserved computation time can be greater than 0.9 second, so in the worst case, reserved activities can hold the processor continuously for no longer than 1.8 seconds before time-sharing programs get a chance to use the processor.

5.2.4 Usage measurement and enforcement

The scheduler for the reservation mechanism requires very accurate usage measurement so that the system can keep track of how the resource usage of each activity relates to its reservation (if any). In particular, reserved activities must be prevented from over-running their reservations and interfering with other reserved and unreserved activities.

To accumulate very accurate usage measurements, the system has code in the context switch from an old thread to a new thread that does the following:

1. takes a timestamp from a high-resolution free-running clock.
2. computes the duration of time the old thread was running and charges that usage against the reserve associated with the old thread.
3. stores the timestamp for doing the same computation later for the new thread.

Those actions mean that threads get charged for the amount of time they spent on the processor rather than getting charged an estimate of the time they spent. This timestamp method much more accurate than the method used for accumulating usage in many time-sharing systems where the process running at the time of a clock tick is charged for the duration of the clock tick (whether it was running the whole time or not).

The reserve abstraction has several operations that provide the programmer with access to usage information in the reserve. The usage-related operations are:

reserve_get_checkpoint(reserve, out_checkpoint_total, out_checkpoint_time, out_accum_total, out_accum_time) Get reserve’s checkpoint information, taken from the last reservation period boundary.

reserve_get_attribute(reserve, attr_name, out_attr_value, out_attr_value_size)
The **reserve_get_attribute** operation can be used with “attr_name” set to **RESERVE_RECENT_CHECKPOINTS** to get an array of the recent checkpoint values for the last several period boundaries.

In addition to the accurate usage measurement, the enforcement mechanism uses timers to keep threads from over-running their reserved computation time and to replenish the reserved time for a reserve appropriately. Two kinds of timers are used for doing these things: the overrun timer and replenishment timers.

The overrun timer is set at each context switch, and it is set for the maximum time the new thread could run before over-running its reserved computation time for its current reservation period. If the thread is still running when the timer expires, the system will update the reserve to show that the activity used all of its time for that period, and it will change the activity to unreserved mode. Then the scheduler will get an opportunity to reevaluate ready threads, and it may decide to switch to another thread.

If the current time is close to the end of the reservation period for the new thread and the reserved time is longer than the difference between the current time and the end of the reservation period, the overrun timer is set to expire at the end of the reservation period. If the timer goes off at that point, the reserve will be replenished and the activity will again be eligible to run.

The second kind of timer is the replenishment timer. Each reserve has a replenishment timer that is initially set at the reserve's start time. The timer is set to expire at the end of the reservation period (or the beginning of the new reservation period). When a replenishment timer expires, the system changes the state of the reserve to reflect that it has a new allocation of its reserved time for the next reservation period. The reserve is set to reserved mode, and the replenishment timer is set to expire again at the end of the new reservation period.

5.2.5 Reserve propagation

One of the key features of the reserve abstraction is that reserves can be bound to threads as appropriate for particular applications. This feature is useful in the situation where an application initiates a reserved activity that may invoke services of server processes locally or even on remote machines. When invoking a server, an application can make its reserves available for the server to use in its computations. The server can then take advantage of having the resources available, and the time it takes to perform the computation on behalf of the client can be charged to the client's reserves. In this way, reserves provide a method for consistently measuring resource usage of entire activities, even if threads in different protection domains cooperate on behalf of the broader activity.

The following operations are related to the binding of reserves to threads:

thread_set_current_reserve(thread, reserve) Each thread has a current reserve and a base reserve. The value of the current reserve may be the result of a reserve propagation, but it is not necessarily permanent. It may eventually revert to the base reserve. This primitive sets the current reserve of a thread.

thread_restore_base_reserve(thread) Makes the base reserve the current reserve for the given thread.

thread_set_reserve(thread, reserve) Set the reserve of a thread.

thread_get_reserve(thread, out reserve) Get a thread's reserve.

In addition to the binding operations, the priority inheritance mechanism of RT-Mach [59,86] aids in ensuring bounded delay for access to servers and mutexes. In the context of reserve scheduling, "priority inheritance" means "reserve inheritance" in the following sense. Interpreting "priority" in the broadest sense, one could think of a thread's reserve information and time-sharing priority information as combining to yield a total ordering for values of these fields. The scheduler schedules threads based on this total ordering from highest "priority" to lowest:

1. threads that have the "reserved" bit set are ordered with smaller reservation period having higher "priority" than larger reservation periods.
2. threads with the "reserved" bit cleared are ordered according to their time-sharing priority, which is a field of each thread (not a field of the reserve).

This concept of "priority" comes into play in the priority inheritance mechanism in RT-Mach. As an example, consider a single-threaded server with several clients. When a client makes a call to the server, the server takes on the "priority" of the client (done in the priority inheritance mechanism) and binds its own thread to the client's reserve to charge its time to it (using the bind operation).

If during this service time a second client with a higher "priority" makes a request, the IPC mechanism enqueues the new request for the server. It then calls on the priority inheritance mechanism to change the "priority" of the server to that of the newly enqueued client (thus limiting the duration of the "priority" inversion). The server continues to charge time to its first client's reserve, however, so that reserve will reflect the true resource usage required for the computation. After the service is finished, the server stops charging against the first client's reserve, picks up the request from the second client, and starts charging against the second client's reserve. The server continues to execute under the "priority" of the second client.

Priority inheritance for reserved activities presents an additional complication beyond what fixed priority inheritance mechanisms must face. In particular, with reserves (and with other dynamic priority disciplines), the "priority" the server takes may change during the service. For example, if the server executes for longer than the reserved time of its client's reserve, the reserve will be degraded into unreserved mode, and the "priority" thus changes. In the implementation, the priority inheritance mechanism is informed when this happens so that it can set the priority of the server to the appropriate value given the list of clients waiting for that server. For example, if a server uses all the reserved computation time for a particular client it would normally have its reserve downgraded and its "priority" decreased. However, if another reserved client is waiting for the server, the server will inherit the "priority" of that client so as to avoid a priority inversion.

5.3 Applications

A number of applications were modified to use reserves to show that real applications could actually achieve predictable behavior using the reservation system. A QuickTime video player and an MPEG decoder were modified to use reserves, and a version of the X Server [34] was modified to cooperate with reserved applications to provide predictable window system services.

5.3.1 QuickTime video player

A QuickTime video player, called QTPlay, was implemented at CMU. The player can display JPEG encoded video as well as raw, unencoded video. The player was modified to use reserves to achieve predictable performance.

QTPlay avoids interactions with system components that have not been modified to support predictable performance via reserves, such as the UX server. It loads a short clip of video into memory during initialization to avoid interaction with the UX server during playback. For experiments, the player loops over the clip for the duration of the test. Figure 5-2 summarizes the structure of the reserved QTPlay application.

```
load short video clip
allocate reserve with command-line parameters
create periodic threads
bind thread to reserve
...
while not done
    save start timestamp
    display a frame
    save end timestamp
...
dump timestamps to a file
```

Figure 5-2: QTPlay Outline

At initialization, QTPlay reserves time on the processor and binds the periodic thread responsible for frame processing to the reserve. The start time of the periodic thread and the start time of the reservation are synchronized so that when the thread becomes ready at the beginning of each period, the allocation of processor time will be available as well. The other reservation parameters, in particular the reserved computation time and the reservation period, are given as command-line arguments. They are typically determined by measurements made prior to the execution of the player. This particular application does not dynamically discover the appropriate reservation parameters nor does adjust the reservations after execution begins.

The player uses a version of the X Window System library, Xlib, that was modified to cooperate with a reserve-enabled X Server. This library passes a reference to the thread's

reserve when the player opens the connection to the X server. The X server then uses the reserve for operations requested by the player (such as DisplayBitmap).

In each period, the thread displays a frame of the video and then saves the start time and completion time for the frame in a buffer in memory. Just before the player exits, it dumps the contents of this timestamp buffer to a file for subsequent analysis.

5.3.2 MPEG decoder

The Berkeley MPEG decoder [93] was modified to use processor reserves in RT-Mach. This version of `mpeg_play` reserves processor capacity during its initialization and periodically evaluates its performance and makes adjustments to its processor reservation and timing constraints as necessary.

The original Berkeley MPEG decoder works by repeatedly reading MPEG encoded macroblocks from an input stream, transforming them, and displaying the frames. The underlying `mpeg` library has some features for managing the timing of frames, but the simple player that is provided to demonstrate the use of the library displays frames as fast as possible without attempting to regulate their timing.

A number of changes and extensions to the MPEG player were required to enable predictable performance and to take advantage of the timing features of RT-Mach as well as the processor reservation mechanism. Figure 5-3 summarizes the code structure of the modified version of `mpeg_play`.

```
load short video clip
allocate reserve with command-line parameters
create periodic thread
bind thread to reserve
...
while not done
    save start timestamp
    display a frame
    save end timestamp
    if frame_number mod 30 == 0 then
        evaluate usage
        adjust reservation parameters and/or algorithm
```

Figure 5-3: `mpeg_play` Outline

As with the QTPlay, `mpeg_play` prefetches a short clip of video into memory to avoid interacting with the file system during runtime. The frames are decoded and displayed by a periodic thread that has the period desired for video playback, typically 33 ms.

During the initialization of the modified MPEG player, it requests a processor reservation based on an estimate of the computation time and the length of the period. Since the computation time may vary on different hardware platforms and different MPEG data

streams, it is very difficult to get an accurate estimate before running the application, and this player tunes its reservation parameters as it executes.

For each frame, the player records the time the computation was started, the time it ended, and the amount of processor time it used during its execution (taken from the usage information in the processor reserve). It periodically computes statistics on these numbers for the recent periods to find out how much computation time was required for each frame and whether the delay for the computation is excessive. This information is used to decide what adjustments need to be made (if any). In its evaluation, `mpeg_play` distinguishes three cases:

1. reservation level okay, do nothing.
2. reservation level too low but some capacity is available to be reserved. increase reserved computation time while keeping the same period.
3. reservation level too low and no additional capacity is available to be reserved, increase the reserved computation time to the desired amount and increase the reservation period proportionally.

The modified version of `mpeg_play` incorporates some basic adaptive techniques, but it could be extended in a number of directions to improve its flexibility and performance. The decoding and display phases of the player should be decoupled to allow the variation in decoding time to be masked by buffering with the application. Incremental decoding techniques would yield several options for how much computation to do for each frame, and changing the dithering algorithm dynamically would increase flexibility as well.

5.3.3 X Server

Each real-time X client acquires a processor reserve and charges its own execution time against that reserve as well as providing the reserve to the X Server so that the Server can charge service time done on behalf of that client to the appropriate reserve. We have modified a version of the X Server to order service requests according to their timing constraints and to charge service time to the client for which the service is performed. Basically, the server should mimic as closely as possible the behavior that would be observed if each client could do its own graphical display within the context of its own address space and scheduling domain. Thus the modified X Server has the following properties:

1. The Server ensures that the activities of real-time clients are isolated from unwanted interference from non-real-time X clients by ordering all request from real-time clients down through non-real-time clients and servicing them in that order.
2. The Server itself is isolated from unwanted interference from non-real-time applications (even applications which are not X clients) by virtue of the processor reservation mechanism. The reservation system ensures that, while the X Server is running under a client's reservation, the resource capacity associated with that reservation is available to the X

Server.

3. Other real-time applications that are not X clients are isolated from unwanted interference from the X Server and its clients if they use the reservation system. This would not be true if the X Server were just assigned a "high priority" or if it over-reserved resources.

The goal of this work is to achieve predictable performance for real-time applications that make use of the graphical display services provided by the X Window System. "Predictable performance" means that real-time applications will be scheduled based on their timing requirements, and their graphical display requests serviced by the window system will be scheduled to meet the timing requirements. Thus, the abstract activity for each real-time client, consisting of the computations within each client application and the associated computations within the window system, should suffer only bounded delays due to other real-time and non-real-time applications sharing the window system.

The processor capacity reserve mechanism provides this kind of timing isolation for independent programs which do not communicate or synchronize with each other. However, when applications share a single software resource such as the X Server, the same kind of timing isolation provided by reserves must be extended into the Server's computations. To provide this isolation and bounded delay, the following is required of the server:

1. Requests from different clients that queue up in the server should be serviced in the order that the clients would be serviced if they were doing the work themselves and being scheduled by the processor reservation mechanism. In other words, the server should handle requests in order of client "priority" (where client priority refers to an implied ordering among clients defined by the reservation system).
2. Computation performed in the server on behalf of a client should enjoy resources, such as processor capacity, that have been reserved for that client. The server should execute at the priority of the client whose request it is servicing. Likewise, the resource usage for such a computation should be charged to the client's reservation, so that a client is prevented from getting more than its reserved time by sending some work to the server and then doing other work locally.
3. Priority inversion should be minimized (and unbounded priority inversion completely avoided) in servicing the clients' requests. Thus if the X Server is occupied with a request from a client when another request comes in from a higher priority client, the server should inherit the priority of the newly arrived client.

These requirements have many implications for the coding of the server. The idea that the server should mimic the behavior of individual threads performing the same computations places some restrictions on how the server can be designed. Also, each of the three specific requirements listed above has some additional implications for the coding of the server.

First we address the desire to have the server behave as the individuals would. This is conceptually easy to achieve by thinking of spawning a new thread for each client's request as it arrives at the server, binding the thread with the reservation or priority of its client, and then allowing the threads to be scheduled by the processor reservation system based on that information. Unfortunately, spawning a potentially large number of new threads is expensive, and while there exists a version of the X Server that is multi-threaded [110], the one on which this work is based has only a single thread. With a single-threaded server, we try to mimic the desired behavior by satisfying the three requirements listed above as follows:

1. Client requests are enqueued in the server in priority order.
2. At the beginning of the computation for each service request, the server takes on the resource allocation persona of the client, enjoying the resource reservations of the client and charging usage against the client's reserve.
3. The RT-IPC [59] mechanism handles priority inheritance to minimize the effects of priority inversion.

These are the modifications made to the X Server to provide predictable performance. However, there are some problems with the X Server that interfere with real-time applications and which are very difficult if not impossible to fix. Several of these problems are addressed in the development of a window system intended for real-time performance [105]. Briefly, the problems are:

1. The X Protocol supports a "grab server" operation which blocks out all other operations for an unbounded period of time.
2. The X library batches requests for higher throughput. This can increase the delay of single operations as multiple operations are combined into one.

Despite these hindrances to 100% guaranteed real-time performance, the modified X server can provide good real-time behavior for typical multimedia applications such as video players.

5.4 Reserved network protocol processing

A predictable network service depends on how the protocol processing for network packets is handled as well as how these activities are scheduled. This section examines several different approaches to protocol processing software design and discusses the advantages and disadvantages of these approaches.

5.4.1 Software interrupt vs. preemptive threads

Traditionally, protocol processing software has been designed to take packets from the network interface and immediately begin processing them at high priority. For example, 4.3

BSD protocol processing is done at a “software interrupt level” which executes at a higher priority than any schedulable activities in the system (like processes) but at a lower priority than hardware interrupts [62]. Unfortunately, network packets associated with a low priority activity may flood the protocol processing software and execute while higher priority processes are delayed. This is an example of priority inversion [48,75].

To prevent this kind of priority inversion, it is necessary to associate priorities with packets so that they can be queued and serviced in priority order. It may also be helpful to be able to preempt the processing of one low priority packet in favor of a higher priority packet, especially if the computation time required for protocol processing is significantly more than that required for a context switch. One approach, used in the ARTS real-time kernel, has preemptible threads to shepherd packets through the protocol software [124]. This is similar to the method used in the *x*-kernel [45], but unlike the *x*-kernel threads, ARTS protocol processing threads were preemptive. This approach provides fast response to high priority packets and prevents low priority network activities from interfering with high priority work on the processor.

5.4.2 Mach 3.0 networking

Networking in the context of the Mach 3.0 UX server [36] is accomplished by calling the 4.3 BSD networking primitives, which are handled by the UX server. The UX server interacts directly with the network device drivers to send and receive packets.

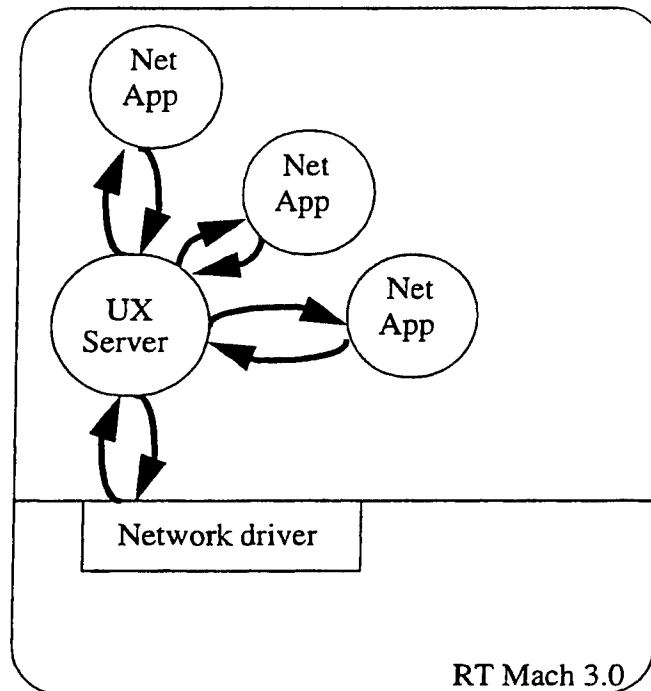


Figure 5-4: Networking with the UX Server

As shown in Figure 5-4, this makes the UX server a single point of contention for all activities that are using the network. Unfortunately, the networking code inside the UX server does not support priority. So this software does not satisfy the requirements for priority and preemptibility in predictable protocol processing software.

Another problem with networking under the UX server of Mach 3.0 is that the interprocess communication (IPC) required between the application and the UX server and between the UX server and the network device drivers adds overhead to network communication. This decreases throughput and increases latency. To alleviate these problems, Maeda and Bershad created a library implementation of TCP/IP and UDP/IP sockets [70]. Their library handles the protocol processing for sending and receiving packets and interacts with the network packet filter [139] and network device drivers directly. The library can be linked in with applications that use the networking calls, so each application can do its own protocol processing in its own scheduling domain (i.e. within its own threads). The library only interacts with the UX server to create and destroy connections and for a few other control operations. The fast path for sending and receiving packets is confined to the library itself (and the device drivers). Figure 5-5 illustrates their networking software structure.

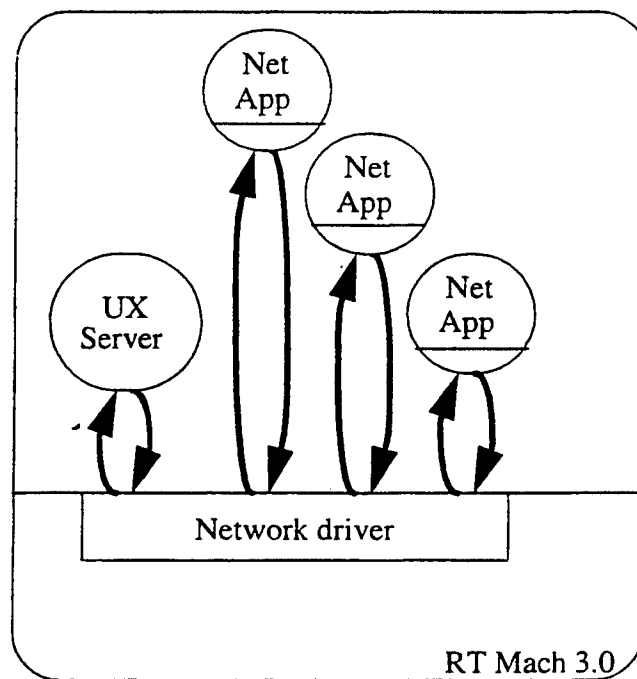


Figure 5-5: Networking with the Socket Library

Maeda and Bershad report that their socket library yields much better performance in terms of throughput and delay than the UX server sockets implementation [70]. Coincidentally, their implementation also satisfies the requirements for effective real-time scheduling

of protocol processing. By including the code in a user library, the computation is done by the user thread at the user's priority. It is also preemptible since it runs in user mode and shares nothing with other threads in other applications.

5.4.3 Reserved protocol processing

Since the socket library enables the protocol processing computation to be scheduled under the priority of the application and since it is also preemptible, the processor reservation system can be applied to programs which do socket-based communication [77]. Compared with a UX server socket implementation, the library partitions the data structures and control paths of all of the networking activities and places them in independent address spaces where they do not to interfere with each other. In the UX server, these different activities are forced to share the same queues without the benefit of a priority ordering scheme. Other activities such as file I/O, asynchronous signals, etc. may interfere with the protocol processing, thus delaying packets as a result of other operating system activity that is not even related to networking.

In the socket library, these components do not interfere with each other, so the reservation mechanism is free to make decisions about which applications should receive computation time and when. The control exercised by the reservation scheduler is not impeded by additional constraints brought on by the sharing of data structures and threads of control. Applications that use the socket library with the reservation mechanism should therefore achieve very predictable networking behavior.

5.5 QOS manager

A QOS manager was implemented to provide a central point for resource allocation decisions. It exports an interface that allows the application programmer to create and terminate reserves, to request a reservation at a specific desired level, and to set preferences for the minimum reservation level. If the reserved load becomes high and the server has difficulty granting minimum reservation levels for new requests, the server begins to downgrade some of the previously granted reservations to their minimum levels in order to admit the new reservation request at its minimum level.

In general, information about resource allocation requirements may come from a variety of sources and may change over time. Resource allocation information can come from applications themselves which may request resource and negotiate if the request cannot be satisfied immediately. It can come from static user preferences about which applications should be more resources under what circumstances. And it can come from various user interface elements designed to bring resource management decisions to the console user.

5.5.1 Information sources

The QOS manager uses the information it gathers to make policy decisions about how to allocate resources to various activities. The information may come from user preferences

files, applications themselves, and graphical resource management tools. Figure 5-6 summarizes the information flow associated with the QOS manager.

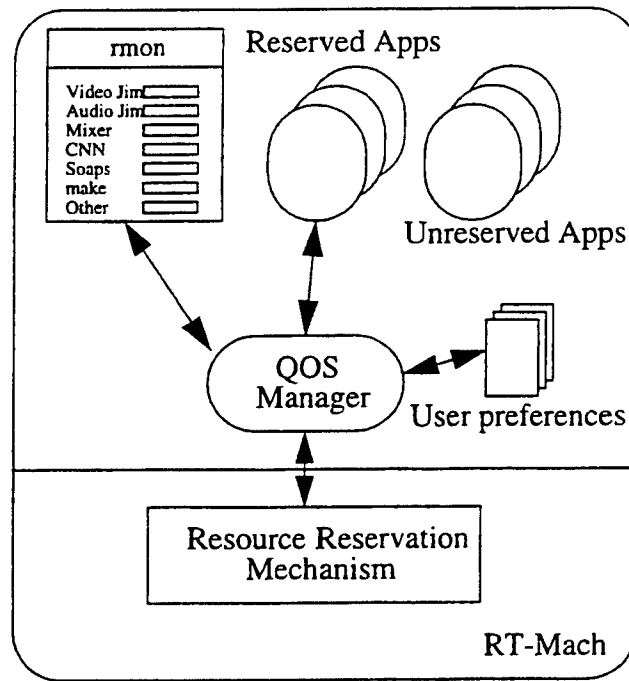


Figure 5-6: Resource Management Schematic

The static user preferences used by a QOS manager might come from a configuration file located in the user's home directory or in a system default directory. Such a file could contain arbitrarily sophisticated rules for the QOS manager to use in making allocation policy decisions. For example, the file might contain rules to indicate how the user's focus should affect resource allocation. It might have rules to determine which applications are more important (e.g. specifying that audio/video applications are more important than file transfer). There might be rules about how temporal properties indicate which applications are more important (e.g. giving recently created applications preference over older applications). And there might be rules about how past usage should affect future reservation.

The dynamic user preferences might come from the applications themselves, from a separate tool, or from some mechanism associated with a window manager. In any case, cues given by the user, which can be picked up in the user interface, are very important to the policy decisions that must be made about where to allocate resource capacity. These cues can be explicit, where the user makes certain gestures to change the resource capacity allocations of various activities. Or the cues can be implicit, as in the case of a window manager

which notices which window has the user's focus (based on the position of the mouse pointer) and passes this information along to the QOS manager.

Information about the recent resource usage of various activities might be used to determine what the future resource reservation levels should be for those activities. For example, an audio player receiving transmissions over the network might become quiet due to long-lasting lull at the sender. When this happens, it may be appropriate to notice the lack of resource usage in the associated reserve and temporarily scale down the reservation level in order to free up more reservable capacity for other activities. A QOS manager with this feature would undoubtedly also provide a mechanism for such dormant applications to come back to life at their original reservation level once they become active again.

5.5.2 Admission control

The admission control policy of the QOS manager must be coordinated with the admission control of the system. The reservation system has an admission control policy that allows it to enforce reservations and keep itself internally consistent with respect to resource allocation and enforcement. The QOS manager must have a version of the same admission control policy so that it can evaluation reservation requests that it gets and look at more sophisticated issues such as how different requests it gets can be combined or changed to fit together better.

This design was chosen because it keeps the admission control test of the kernel simple and fast while allowing arbitrarily sophisticated admission control decisions and negotiations to be carried out in user-level QOS managers. It would be possible to combine the two policies but there are drawbacks to that approach. If the sophisticated policy with negotiation were implemented in the kernel, the system would become more complicated, slower, and less flexible. If the kernel depended on user-level admission control for its own consistency, it would be vulnerable to errors in the user-level QOS managers.

5.5.3 Extensions

The QOS manager reacts to new reservation requests that strain the available resource capacity by trying to free up resource capacity from among previously reserved activities, subject to the limitations that those activities allow as expressed by their minimum reservation levels. This policy could be extended to accommodate information about which activities should be downgraded first, whether new minimums could be negotiated with activities to free up even more capacity, or whether the activities requesting reservations should be denied to keep the previously reserved activities at their current reservation levels [79]. Another extension might upgrade reservations to the old desired values once reservable resource capacity became plentiful.

5.6 Tools

Two tools were developed in the course of this dissertation work to do debugging and execution monitoring for experiments and to provide a user interface for reserves. One is a reserve monitor with a graphical user interface, and the other is a usage monitor that operates in batch mode to gather usage statistics for experiments.

5.6.1 Reserve monitor

A reserve monitor, called `rmon`, provides the user at the console with a graphical user interface to monitor and control processor reserves. The two important aspects of this tool are its presentation of usage information and its support for control of resource reservation.

5.6.1.1 Usage information

The primary view of `rmon` displays basic information about all of the processor reserves in the system. This information consists of the name of the reserve, a graphical representation of the recent usage information, normalized to the reservation period, and the reservation period itself. Figure 5-7 shows a screen dump of this primary view.

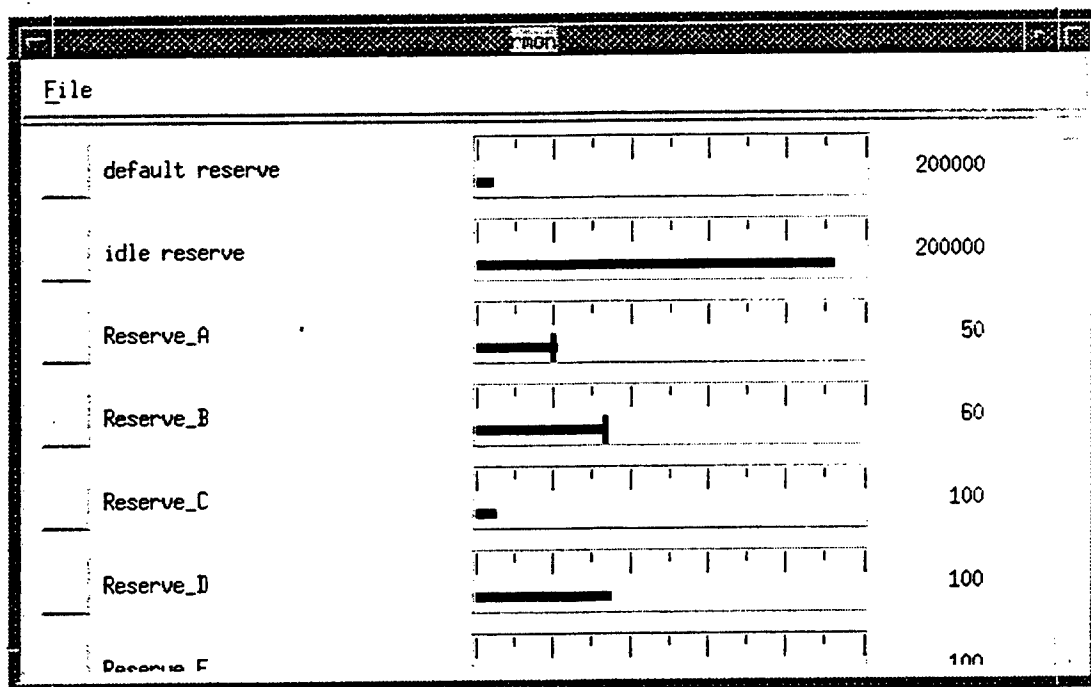


Figure 5-7: `rmon` Main View

As the figure shows, each reserve takes one row in the display. Each row contains the following elements:

- detail button - Pressing the detail button pops up another window which shows more detailed information associated with the reserve.
- reserve name - Each reserve may have a name associated with it for ease of identification.
- graphical usage display - This graphic displays a bar with length corresponding to the percent resource usage over the last several reservation periods. The usage is normalized to the reservation period, and the graphic includes markings to indicate the scale of the usage.
- reservation period - The reservation period indicates the averaging interval of the usage measurement.

As reserves are created and terminated in the system, corresponding rows are created and destroyed in the primary view. The two system reserves (called “default reserve” and “idle reserve”) always exist. So they always appear in the view. The default reserve is where all the usage is charged for applications that do not have their own private reserves. It has no actual resource capacity reservation associated with it; it just accumulates the usage of the unreserved programs. The idle reserve accumulates the usage of the idle thread; it also lacks an actual resource reservation.

For each reserve that has a resource capacity reservation associated with it, rmon displays a vertical bar in the usage graphic to indicate the level of the reservation, in terms of normalized capacity. In Figure 5-7, the reserves named “Reserve_A” and “Reserve_B” have resource capacity reservations associated with them whereas “Reserve_C” and “Reserve_D” do not. The vertical bars in the usage graphics of Reserve_A and Reserve_B indicate that they have reservations of 20% and 33%, respectively. The reservation periods are 50 ms and 60 ms, respectively.

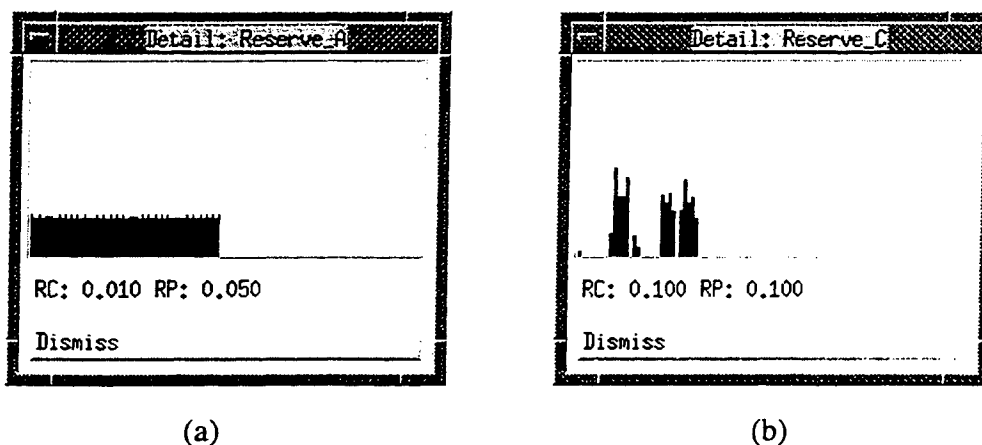


Figure 5-8: rmon Detail Views

Pressing the “detail” button at the beginning of a reserve’s row pops up another window of detailed information about the reserve. This includes a graphical display of the recent history of resource usage as well as the parameters of the reserve. Figure 5-8 shows two example detail windows. Part (a) of the figure shows the detail for an activity that has a reservation, and part (b) shows an activity without a reservation.

As shown in the figure, the recent history occupies the top portion of the detail window. It shows the normalized usage of the reserve over the last several reservation periods, and it advances in real-time in a manner similar to that of xload [74]. For the reserved activity in Figure 5-8(a), which is Reserve_A from the Main View, this usage is fairly constant over time. For the unreserved activity in Figure 5-8(b), Reserve_C, this usage is variable from period to period. Each window also displays the reserved computation time and the reservation period. The unreserved activity has zero reserved computation time but a non-zero reservation period (the implementation represents an unreserved activity using a reserved computation value equal to the reservation period, and this representation happens to be exposed in this view). This indicates that usage measurements for this activity will be taken based on the reservation period, but that there is no actual resource capacity reservation.

5.6.1.2 Allocation Control

The level of the reservation for a reserve, as indicated by the vertical bar in the normalized usage graphic, can be changed by clicking the mouse in that usage graphic at the level desired for the reservation. This action modifies the reserved computation time parameter of the reserve without changing the reservation period.

The upper screen view in Figure 5-9 shows several reserves at various reservation levels. Notice the position of the mouse pointer in the usage graphic of the reserve called Reserve_B, which is reserved at 20% of processor capacity. Clicking the mouse button with the pointer at this position changes the reservation level of Reserve_B to that shown in the lower screen dump in the figure. The reservation level is now about 40%, and the actual usage of the activity reflects the availability of that additional capacity.

5.6.2 Usage monitor

A usage monitor based on reserves was developed to aid in debugging and to support usage measurements for experiments. This monitor allocates a reserve and requests a reservation for its own execution. It periodically polls for the usage on specified reserves in the system, saving the usage numbers in a large buffer. Then it formats the usage information and writes it to a file for processing by a graphing tool.

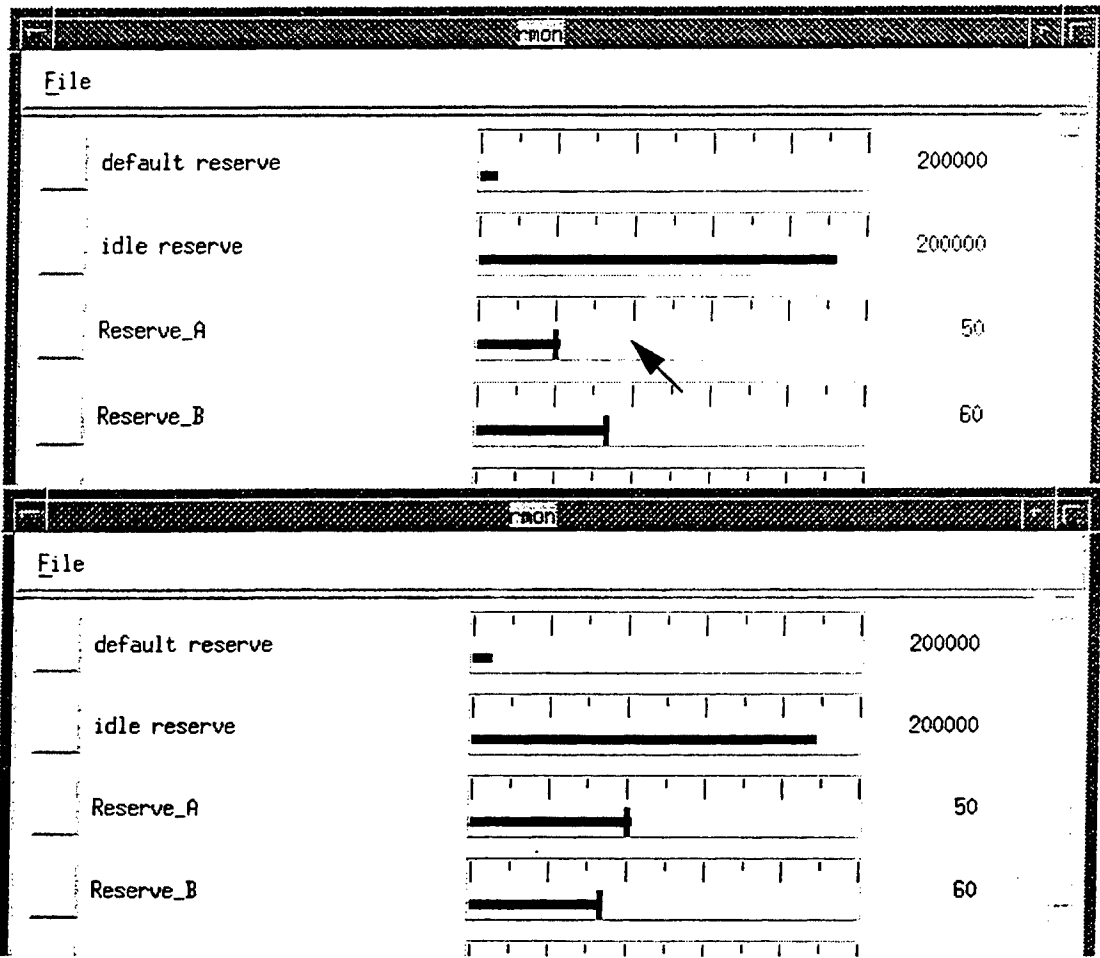


Figure 5-9: Modifying a Reservation

5.7 Chapter summary

This chapter describes the implementation several software components in the reservation system including:

- processor reserves in RT-Mach,
- reserved video players and a version of the X Server that uses reserves,
- a version of the Mach 3.0 socket library implementation modified to use reserves,
- a QOS manager,
- a tool for providing a user interface to the reservation system.

The description of the implementation of reserves presents operations that are supported for manipulating reserves as well as a description of how the scheduling and usage enforcement is handled. The implementation of two video players that use reservation is described along with that of a version of the X Server that was modified to use reserves. The section on the socket library implementation discusses issues in the organization of protocol processing software to support real-time packet processing. A description of the QOS manager indicates how it works and its relationship with applications and the reservation mechanism. Finally, a graphical user interface tool is described; it displays information about reserves and provides an interface for controlling reservation parameters.

Chapter 6

Experimental Evaluation

This chapter presents an experimental evaluation of the implementation of processor reserves in RT-Mach. The reservation mechanism was designed to support predictable behavior for real-time and multimedia applications, so the evaluation answers the questions: Can reserved programs achieve predictable behavior, and what is the price of predictability?

6.1 Overview

The experimental evaluation presented in this chapter answers two questions: Can reserved programs achieve predictable behavior, and what is the cost for predictability? These questions are addressed using synthetic benchmarks, real applications, and measurements of individual mechanisms. The chapter is divided into two main sections, one to address predictability and another to address scheduling costs.

The section on predictability shows that for a wide variety of task sets, real-time tasks exhibit predictable behavior and meet their timing constraints:

- Independent synthetic workload measurements show that for pure computations that have no interactions with other tasks, the reservation mechanism successfully guarantees timing constraints.
- Client/server experiments show that the reserve propagation mechanism helps guarantee the client's timing constraints, even when there are multiple clients with and without reservations.
- Results of experiments with the QTPlay QuickTime video player and the X Server show how this client/server pair is coordinated to meet the timing constraints of the video player, even when there are unreserved X clients competing for the attention of the X Server.
- Experiences with the mpeg_play decoder and the X Server demonstrate how an application can start with an inaccurate estimate of required com-

putation time and then adjust its reservation parameters to balance its usage requirements with the resource availability.

- Experience with a library-based network protocol software structure shows that protocol processing for real-time applications can be guaranteed using processor reserves.

The other main section of this chapter explores the scheduling costs of the reservation system. Two measurement techniques are used:

- A comparison of the system scheduling costs for periodic real-time programs that use reserves vs. periodic programs that do not use reserves shows that the cost varies, as expected, depending on the period of the program.
- Measurements of various internal operations such as reserve switch time, overrun timer handling, replenishment timer handling, and usage checkpoint operations provide a means of estimating scheduling cost for reserved task sets.

The measurements for most of these experiments were taken using RT-Mach version MK83j with UNIX server UX41. The mpeg_play and libsockets experiments used RT-Mach version MK83i, which does not differ significantly from MK83j in the features used in the experiments. The hardware platform for the first three sets of experiments was a 90MHz Pentium with 16 MB RAM and an Alpha Logic STAT! timer card. The timer card has a 48-bit free-running clock with 1 μ s resolution, and a 16-bit interrupting timer with 1 μ s resolution. For the remaining experiments, the hardware platform was the same except the processor was a 486 DX2 instead of a Pentium. For easy reference, the chart below summarizes which platforms were used for which experiments.

Experiment	RT-Mach Version	UX Version	Processor
Independent task sets	MK83j	UX41	Pentium
Client/server task set	MK83j	UX41	Pentium
QTPlay/X Server	MK83j	UX41	Pentium
mpeg_play/X Server	MK83i	UX41	486
libsockets	MK83i	UX42	486
Aggregate scheduling costs	MK83j	UX41	486
Micro measurements	MK83j	UX41	486

Table 6-1: Summary of Testbed Platforms

In general, the switch from the 486 to the Pentium speeds up the compute-intensive applications by about 30%. Since the micro measurements involve kernel instruction streams that access external devices such as the clock/timer card, these measurements are not expected to change significantly using a Pentium processor.

In summarizing the results of many of the experiments, percentiles are used to specify dispersion. While running these experiments on a desktop computer connected to the normal departmental network, occasional anomalies occurred. 5-percentiles and 95-percentiles are used to indicate the range of the strong majority of measurements while ignoring the occasional anomaly. As defined in Jain's book [50], the 5-percentile is obtained by sorting a set of n observations and taking the $[1 + n(.05)]$ th element in the sorted list (where $[.]$ is used to indicate rounding to the nearest integer). The 95-percentile is the $[1 + n(.95)]$ th element in the sorted list.

6.2 Predictability

What is meant by "predictability?" In the context of this work, a predictable application is one whose timing behavior can be determined from the application code, the resource reservations that it acquires, and its dependence on other programs. In particular, a predictable application that is not under-reserved will meet all of its timing constraints.

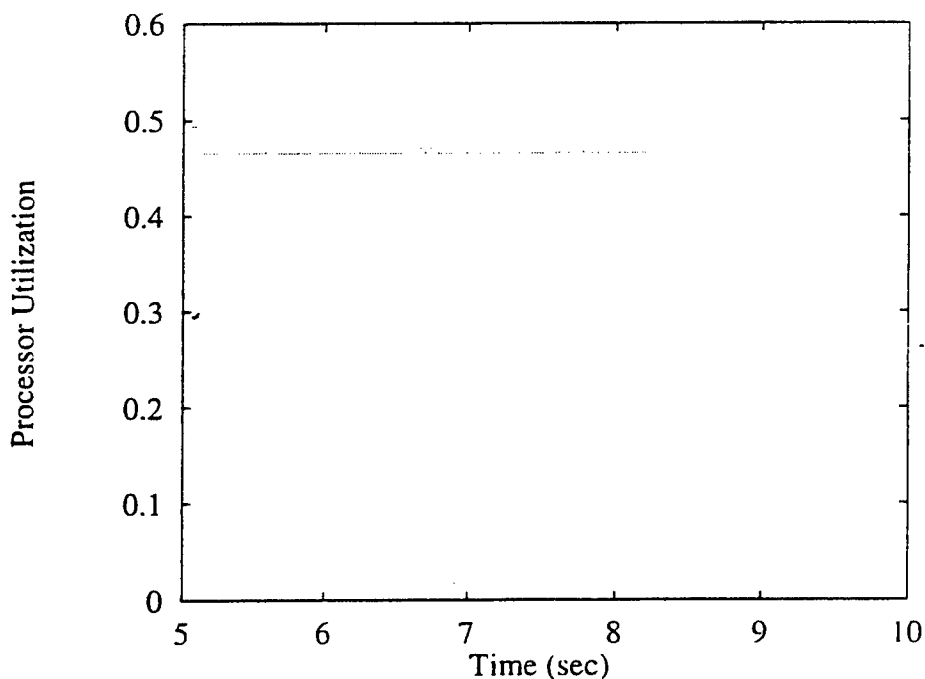


Figure 6-1: Compute-Bound Periodic Task with No Competition

As an example, consider a periodic application that computes for a fixed duration of time in each period and spends all of its computation time in a tight loop. Such an application should be able to allocate a reservation for its computation time, and it should exhibit the same behavior when it is executing concurrently with other activities as when it is executing in isolation. That is, it should be able to consume its reserved computation time in each period before the “deadline” at the end of the period.

Figure 6-1 shows the processor usage over time of a periodic application with a local computation and no competition for resources. The x-axis is time measured in seconds; it shows several seconds of usage information for the application. The y-axis is the normalized processor utilization of the application. Time on the x-axis is divided into intervals that correspond to the period of the application, and the processor time used during each period is measured and the utilization computed for the period. The utilization is plotted at that constant level for each period. For the duration of the test shown, the periodic application has an average utilization of 0.467. The distribution of measurements is very closely packed around this average with a 5-percentile of 0.465 and a 95-percentile of 0.473.

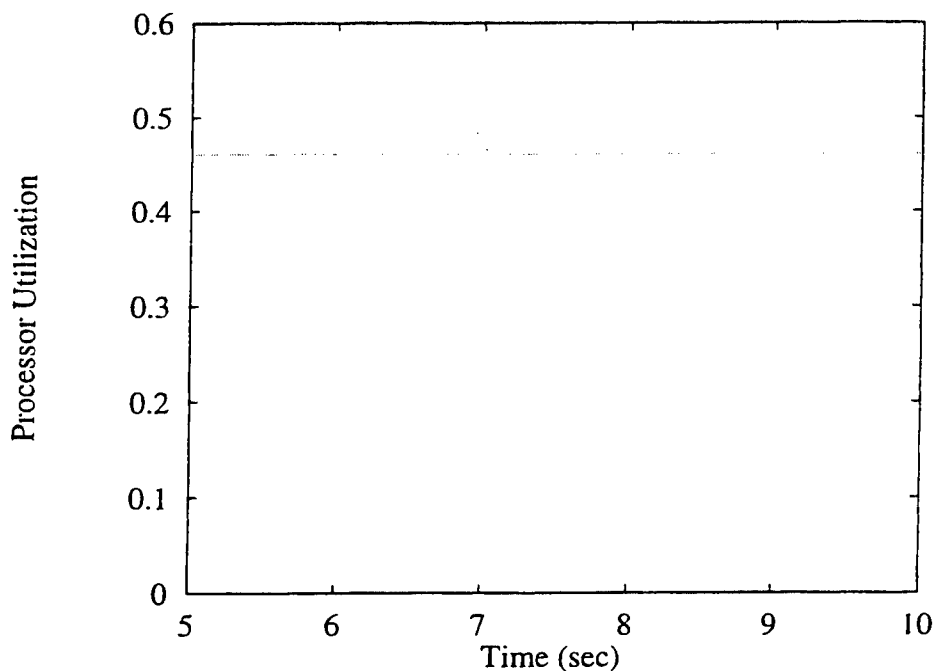


Figure 6-2: Compute-Bound Periodic Task with Competition

Figure 6-2 shows a similar graph of the processor usage of a periodic application that has a local computation but has competition for the processor from other programs (not shown). Even though there is competition, the reservation system ensures that the appropriate amount of processor time will be available to the application in each period. The application consumed an average of 0.462 of the processor in each period. As in the previous case,

most of the measurements were very close to the average; the 5-percentile is 0.460 and the 95-percentile is 0.470.

6.2.1 Independent synthetic workloads

Independent synthetic workloads are used to test whether the reservation system can successfully provide access to reserved processor resources. The example above demonstrates that the reservation system can ensure predictable behavior for a periodic application running with competition from other activities, and one of the experiments described below shows that multiple independent reserved applications can achieve predictable behavior, even with competing unreserved activities. Two additional experiments show that the reservation guarantee is independent of the number of competing activities, regardless of whether the competitors are reserved or unreserved.

6.2.1.1 Methodology

These experiments were run using two software tools developed for performance evaluation. A configuration manager parses the specification of a task set with timing parameters and reservation parameters and then creates programs with the appropriate parameters. Several different kinds of programs that exhibit different kinds of behavior can be specified in the task set. Each of these programs takes a start time, a duration to compute, a thread period, a computation time to reserve, and a reservation period. Two programs are used in these experiments:

- **arith** - Creates a periodic thread that executes in a tight loop for some duration of time in each period.
- **monitor** - Records the usage charged to reserves in the experiment. This program has a reservation of its own to enable it to run even when there are many reserved programs in the experiment.

A usage monitor is usually included in the task set to take usage measurements for all of the programs created by the configuration manager. The monitor buffers the measurements during the course of the experiment and then formats the data and writes them to disk after the experiment is completed. The data are then graphed.

Experiment 1 is designed to show that reserved activities are able to execute their periodic computations within their time constraints, even with competing unreserved activities. Even if the reservation parameters have different computation times and different periods the timing constraints of the reserved activities will be satisfied.

Table 6-2 shows the programs used in Experiment 1 along with the number of instances of each program, the timing parameters, and the reservation parameters. In this experiment there were 3 **arith** programs that were reserved with different timing and reservation parameters. One had a reservation of 5 ms of every 20 ms, the second had a reservation of 14 ms every 40 ms, and the third a reservation of 8 ms every 50 ms. The reservation is set slightly higher (1 or 2 ms) than the computation time that would be consumed by the program in isolation. This accommodates variation in the computation time due to cache effects

and context switches. In addition to those three, there were 5 `arith` programs running in infinite loops with no reservations; these provide compute-bound competition for the reserved activities. And finally, the experiment included a `monitor` program to collect usage numbers throughout the duration of the test. This monitor had 2 ms reserved of every 20 ms.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>arith</code>	1	4 ms	20 ms	5 ms	20 ms
<code>arith</code>	1	12 ms	40 ms	14 ms	40 ms
<code>arith</code>	1	6 ms	50 ms	8 ms	50 ms
<code>arith</code>	5	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-2: Experiment 1 Parameters

The other two experiments measure the sensitivity of reserved applications to competition. Both experiments consist of eight series of tests. Each test has a reserved `arith` program whose measurements are the focus of the test. The series differ in that the reserved `arith` program increases in reserved utilization in each series. Each series itself consists of a sequence of tests with an increasing number of competitors. In each series of Experiment 2, the one reserved `arith` program competes with an increasing number of unreserved `arith` programs. For each test, the 5-percentile and 95-percentile for the resource usage measured in each reservation period is reported. The parameters for an example task set in Series 1 of this experiment appear in Table 6-3.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>arith</code>	1	3 ms	40 ms	4 ms	40 ms
<code>arith</code>	2	infinite loop	N/A	0 ms	40 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-3: Example Parameters for Experiment 2

The task set that appears in Table 6-3 has one reserved `arith` program that computes 3 ms in every 40 ms and has a reservation of 4 ms for every 40 ms. It also has two unreserved `arith` programs and a `monitor` program. Other tests in the Series 1 have unreserved

competitors ranging in number from 0 to 9 competitors. This task set is designed to show that regardless of what the reserved utilization is and regardless of how many competitors there are (ranging from 0 to 9 compute-bound, unreserved competitors), a reserved activity will always be able to get its reserved allocation.

Experiment 3 is like Experiment 2 except that the competitors are reserved instead of unreserved. For convenience, the competitors all have identical reservations, so the number of competitors for a reserved activity is limited to the number of competitors that can be accepted by the admission control policy.

The tests of Experiment 3 are organized into 8 series with 10 tests, just as in Experiment 2. Again, the series differ in that the reserved `arith` program that is observed varies in its reserved utilization, and within each series, the number of competitors ranges from 0 to the highest number that can pass admission control along with the observed program. Variation in the usage of the measured reserved program is again characterized by the 5-percentile and 95-percentile. The parameters for an example test in this experiment appear below.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
<code>arith</code>	1	3 ms	40 ms	4 ms	40 ms
<code>arith</code>	2	2 ms	30 ms	3 ms	30 ms
<code>monitor</code>	1	N/A	20 ms	2 ms	20 ms

Table 6-4: Example Parameters for Experiment 3

The task set in Table 6-4 has one `arith` program with computation duration 3 ms and a period of 40 ms. The reservation given to this program is 4 ms every 40 ms. The table lists two other reserved `arith` programs with 2 ms computation time and 30 ms period, and these both have reservations of 3 ms every 30 ms. This is a test from Series 1 of Experiment 3, and other tests in this series have the different numbers of competing reserved programs. The number of competitors for Series 1 ranges from zero to seven, but the number of competitors for Series 8 is zero since no competitors could be admitted once the primary reserved program and the reserved `monitor` pass admission control. The purpose of this task set is to demonstrate that regardless of what the reserved utilization of the primary reserved program and regardless of the number of competitors, the primary reserved activity will get its reserved allocation virtually all of the time.

6.2.1.2 Results

The results from Experiment 1 demonstrate that multiple reserved programs meet their timing constraints, despite the competition between the reserved activities and competition from unreserved activities. Figure 6-3 shows a graph of the behavior of the three reserved programs in Experiment 1, leaving out the usage measurements of the competing unre-

served activities. These usage measurements are in the same format as the example case described earlier: the x-axis is time in seconds for the test, and the y-axis is processor utilization. The usage for each reservation period for each reserved program is computed and plotted on the graph, yielding three functions of utilization over the duration of the test.

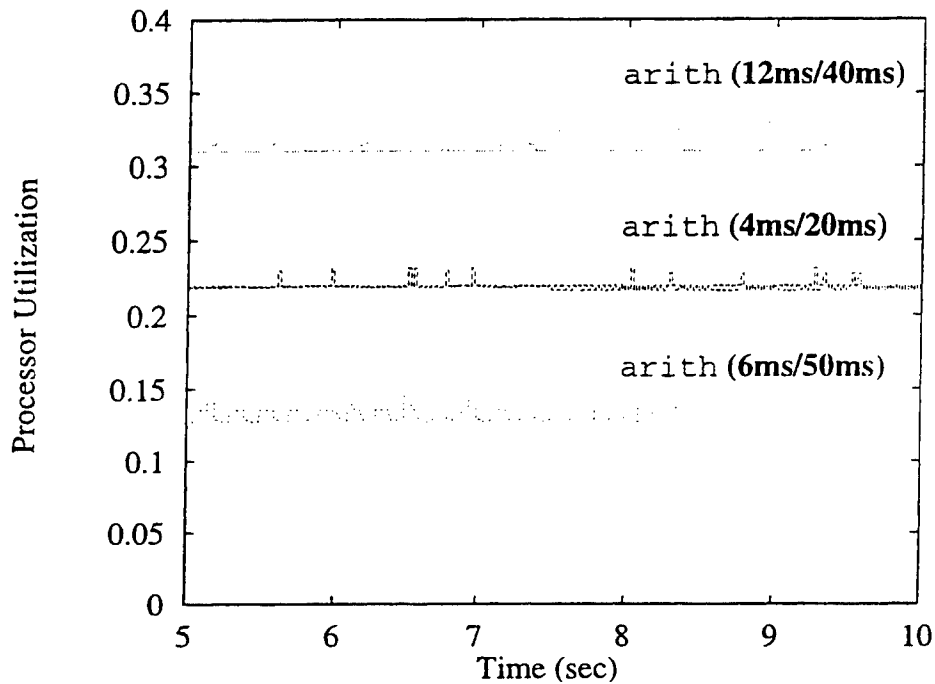


Figure 6-3: Experiment 1 Results

Each of the three reserved programs sustains a fairly constant utilization level throughout the entire test, in spite of the competition from reserved and unreserved activities. The reserved program with a computation time of 4 ms every 20 ms gets a fairly constant utilization with an average of 0.219 (context switching overheads and cache effects push the measured usage higher than what it would be in a quiescent system). The 5-percentile is 0.216 and the 95-percentile is 0.227, indicating that very few measurements fall far from the average. The reserved program computing 12 ms every 40 ms gets an average utilization of 0.312. It gets a 5-percentile of 0.310 and a 95-percentile of 0.320, so there is clearly very little variation in the utilization across periods. The program computing 6 ms every 50 ms gets an average utilization of 0.132 across the periods shown above. The 5-percentile is 0.126 and the 95-percentile is 0.141. Thus Experiment 1 shows that the reservation system can guarantee the timing constraints for multiple reserved programs even when there is competition from unreserved activities.

The results from Experiment 2, illustrated in Figure 6-4, show that the timing behavior of a reserved program is not affected by the number of unreserved competitors, regardless of the utilization of the reserved program. The graph in Figure 6-4 has the number of competitors on the x-axis and processor utilization on the y-axis. The data from the eight series are

plotted as functions on the graph. For example, the function for Series 1 starts with the average utilization for the test that has zero competitors. The function then continues to the average utilization for the test in that series that has 1 competitor and so on up to the average utilization for the test with nine competitors. At each point where the average utilization is plotted, there is also a range that gives the 5-percentile and 95-percentile to indicate the variation in the behavior of the reserved program on that test. The rest of the functions are similar. This graph shows that for each series, the average processor utilization is nearly constant, regardless of the number of competitors. Furthermore, the variation given for each measurement is quite small, indicating that for the vast majority of reservation periods, the reserved program is able to meet its timing constraints.

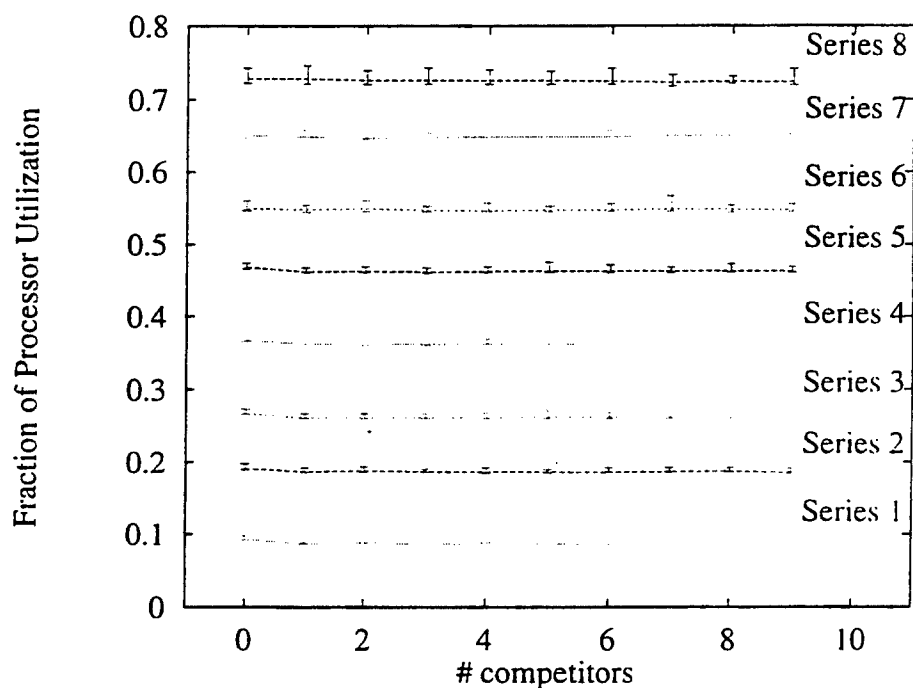


Figure 6-4: Experiment 2 Results

The results for Experiment 3 are presented in Figure 6-5. These results show that the behavior of a reserved program does not depend on the number of competing reserved activities, regardless of the utilization of the reserved program of interest. The graph for Experiment 3 is much like the graph for Experiment 2. The x-axis is the number of competing programs, and the y-axis is processor utilization. There are data from the same kinds of series, and the plot of average utilization as a function of number of competitors is the same. Each plotted point has a 5-percentile and 95-percentile range to indicate the variation. Since each of the competitors must pass the admission control policy, the number of competitors becomes more limited as the utilization of the measured reserved activity gets larger. So the maximum number of competitors for Series 1 is seven and for Series 8, no competitors can

pass admission control after the measured reserved activity and the monitor do. These results show that the average processor utilization for each series is nearly constant, i.e. it does not depend on number of competitors. The variation in processor utilization is very small, indicating that the reservations are available to allow the reserved activity to satisfy its timing constraints. This is true regardless of the processor utilization of the measured reserved program.

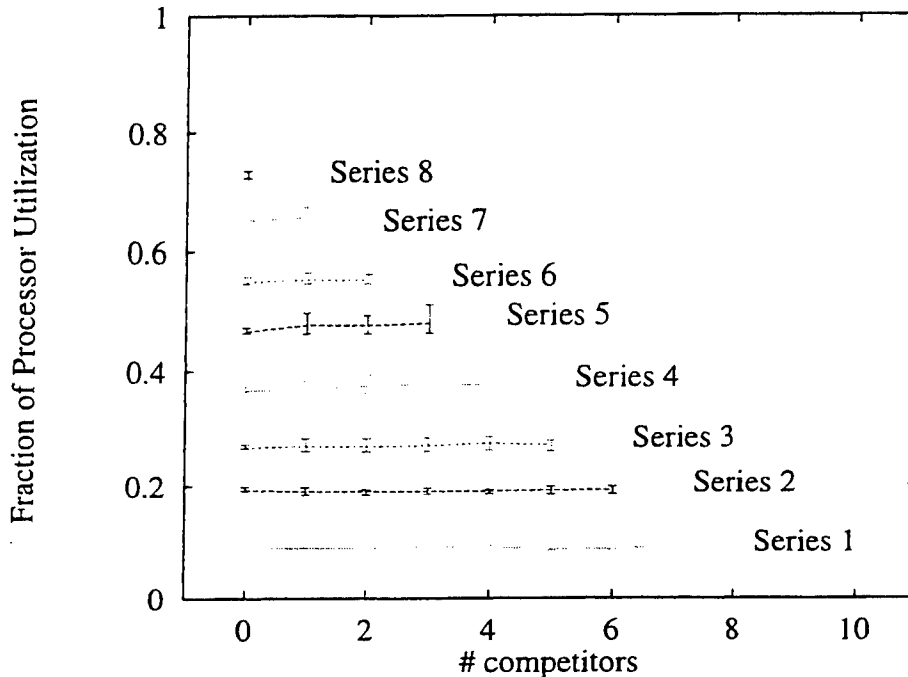


Figure 6-5: Experiment 3 Results

6.2.1.3 Analysis

These three experiments show that for cases where periodic threads allocate reserves for their computations, the reservation system is able to ensure that the reserved time is available as promised. The reserved time is available even when there are multiple reserved activities and unreserved competitors as in Experiment 1. Experiment 2 showed that a reserved activity is assured of being able to use its reserved time regardless of the number of unreserved competitors and regardless of whether the reservation is for a small computation time or a large computation time. Experiment 3 demonstrated that a reserved activity will get its reserved time regardless of the number of reserved competitors it has, and this is also true whether the reserved activity has a small amount of time reserved or a large amount of time.

Two issues are highlighted by these experiments. One is that the computation time for the reserved programs was always less than the reserved computation time by 1 to 2 milliseconds. The computations that the programs will execute are based on arithmetic computa-

tions that were timed on a quiescent system with only the timing program running. The synthetic workload was tuned in this environment. When these workloads are executed with other activities, there are additional overheads that are not included in the task set specifications. These overheads include:

- context switch times and cache effects, which are likely to increase as the task set size increases,
- periodic thread overhead associated with periodically releasing and resetting the computation,
- and interference from interrupts.

The reserved computation time is set to be 1 to 2 milliseconds larger than the pure computation amount to accommodate these overheads.

The second issue is that in some rare cases, a series of interrupts or a large critical region in the kernel may preempt a program and cause it to miss its reserved time and subsequently miss its deadline for the period. To mask these rare instances in Experiments 2 and 3, the 5-percentile and 95-percentile are given. This shows that for the vast majority of reservation periods for these threads, the usage observed is that which is expected based on the reservation.

6.2.2 Client/server synthetic workloads

Most interesting applications are not independent, so it is important to consider experiments that characterize the effect of interactions such as client/server relationships in reserved applications. The experiments described below show that reserved activities can achieve predictable behavior, even when the activities involve coordination between clients and servers.

6.2.2.1 Methodology

The following experiments use the same kind of software environment as described in Section 6.2.1. There is a configuration manager that reads a specification of a task set and then creates the programs for the task set. In addition to the `arith` and `monitor` programs described above, these experiments use the following programs:

- `tsclient` - Creates a periodic thread that invokes a server to compute for some duration of time specified in the invocation. The invocation is performed using the regular Mach IPC mechanism.
- `tsserver` - Services requests sent in from instantiations of the `tsclient` program.
- `rclient` - Creates a periodic thread that invokes a server to compute for some duration of time, but unlike the `tsclient`, the `rclient` uses RT-IPC instead of regular Mach IPC, and the `rclient` sends a reserve to the server so that it can charge the computation time to the client's reserve.

- **rserver** - Services requests sent from instantiations of the **rclient** program. Uses RT-IPC and charges the computation requested by a client to the client's reserve, which is passed as an argument with the invocation.

Experiment 4 is designed to show the processor usage of a client/server pair that has no competition from other programs; this is the base case, showing the desired behavior for the client and server. It uses a task set with an instance of the **tsclient** program using Mach IPC to periodically invoke an instance of **tsserver** to perform a computation. A monitor records the usage for later analysis. In this case, the client sends the computation time amount (to be consumed in a tight loop) to the server. And the server computes for that amount of time and returns a result. The parameters for the programs in this task set are given in Table 6-5. The client is periodic and has a reservation associated with it. As long as the computation time requested by the client is smaller than the period, the server with no competition should be able to finish the computation by the end of the period, yielding a fairly constant utilization over time.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
tsclient	1	10 ms	40 ms	10 ms	40 ms
tsserver	1	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-5: Experiment 4 Parameters

In Experiment 5, the task set includes competition from unreserved programs as well as the **tsclient**, **tsserver**, and **monitor**. This experiment is meant to show how competition for the processor from unreserved activity can interfere with the coordinated activity of a client and server using a typical IPC mechanism.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
rclient	1	8 ms	40 ms	10 ms	40 ms
rserver	1	N/A	N/A	0 ms	40 ms
arith	5	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-6: Experiment 6 Parameters

Experiment 6 is designed to determine whether a client/server pair, using an IPC mechanism integrated with the reservation system in terms of queueing and scheduling, can sustain a predictable, coordinated activity even with competition for the processor. Table 6-6 shows the task set specification for Experiment 6. The rclient has a processor reservation, and the rclient and rserver communicate using RT-IPC. The competition for the processor comes from five arith programs which are unreserved.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
rclient1	1	8 ms	40 ms	10 ms	40 ms
rclient2	1	8 ms	50 ms	10 ms	50 ms
rclient3	1	8 ms	60 ms	10 ms	60 ms
rserver	1	N/A	N/A	0 ms	40 ms
arith	5	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-7: Experiment 7 Parameters

Experiment 7 is intended to show whether several reserved clients can execute in a manner that satisfies their timing constraints when using the same server and competing with unreserved, compute-bound programs. The task set, shown in Table 6-7 shows three reserved instances of the rclient program with different reservation parameters. There is also an instance of the rserver program. The competition comes from five instances of the arith program which are unreserved, and there is a reserved monitor program.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
rclient	1	8 ms	40 ms	0 ms	40 ms
rclient	1	8 ms	50 ms	10 ms	50 ms
rclient	1	8 ms	60 ms	10 ms	60 ms
rserver	1	N/A	N/A	0 ms	20 ms
arith	5	infinite loop	N/A	0 ms	40 ms
monitor	1	N/A	20 ms	2 ms	20 ms

Table 6-8: Experiment 8 Parameters

Finally, Experiment 8 is designed to determine whether reserved clients can meet their timing constraints if there is an unreserved client that is using the same server. As shown in Table 6-8, the task set for Experiment 8 contains one `rcclient` program with no reservation and two `rcclient` programs with reservations. There is an `rserver` and five competing `arith` programs which are unreserved. A monitor is also included in the task set.

6.2.2.2 Results

The results from Experiment 4, shown in Figure 6-6, illustrate the processor usage pattern of the periodic client and its server. The x-axis is time over the duration of the test measured in seconds, and the y-axis is processor utilization. The usage measurements for both the client and the server are taken from the corresponding reserves. The client has a reserve that it charges for its own computation, and the server has a reserve that it charges against when performing an operation for a client. For each of these reserves, the monitor records the computation time used in each reservation period. Those computation times are then normalized with respect to the length of the corresponding reservation periods and plotted as constant for the duration of each reservation period.

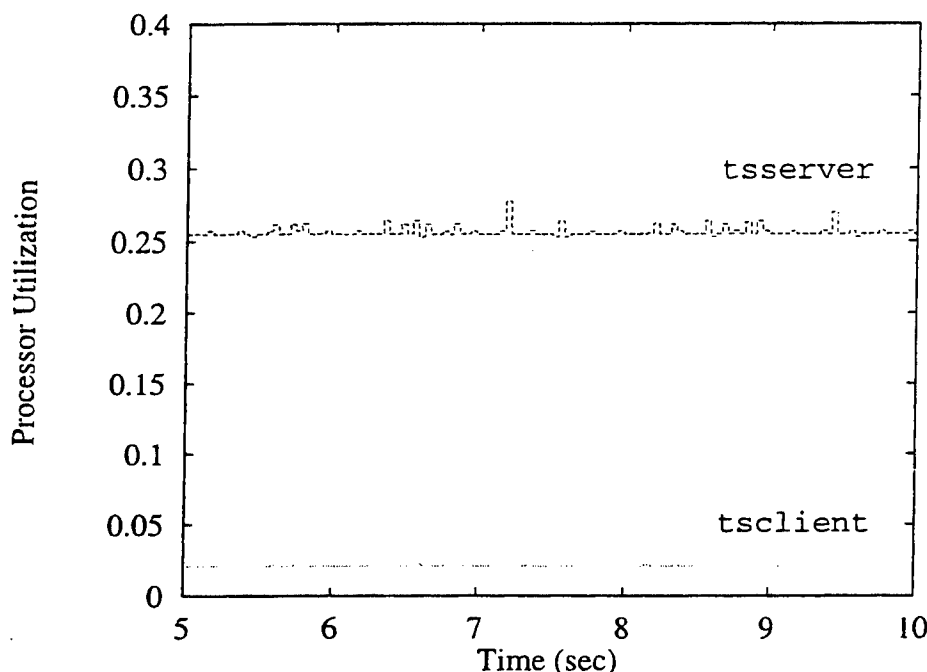


Figure 6-6: Experiment 4 Results

Figure 6-6 shows that the processor utilization charged to the server's reserve is fairly constant over the duration of the test. The average is 0.256 with a 5-percentile of 0.255 and a 95-percentile of 0.264 for the measurements graphed in the figure. The normalized charges to the client reserve average only 0.0210; the 5-percentile is 0.0207 and the 95-percentile is 0.0213. In this setup, the server is doing most of the work while the client does no

work other than sending off requests and receiving replies. Since there is no competition and the client makes the same request in every period, the utilization is fairly constant over the duration of the test.

Figure 6-7 shows the results of Experiment 5 where the same client/server pair has competition for the processor from unreserved activities. As before, the x-axis is time measured in seconds, and the y-axis is processor utilization. In this case, the client and server do not have constant utilization numbers over each reservation period. For the measurements shown in the graph, the server has an average utilization of 0.191, which is significantly lower than the desired level. The 5-percentile for the server is 0, and the 95-percentile is 0.262. The client has an average utilization of 0.0117 across the periods shown in the graph: its 5-percentile is 0 and its 95-percentile is 0.0158. With the client and the server recording 0 utilization in a significant number of periods in a row, it is clear that the client/server combination is not achieving the desired behavior.

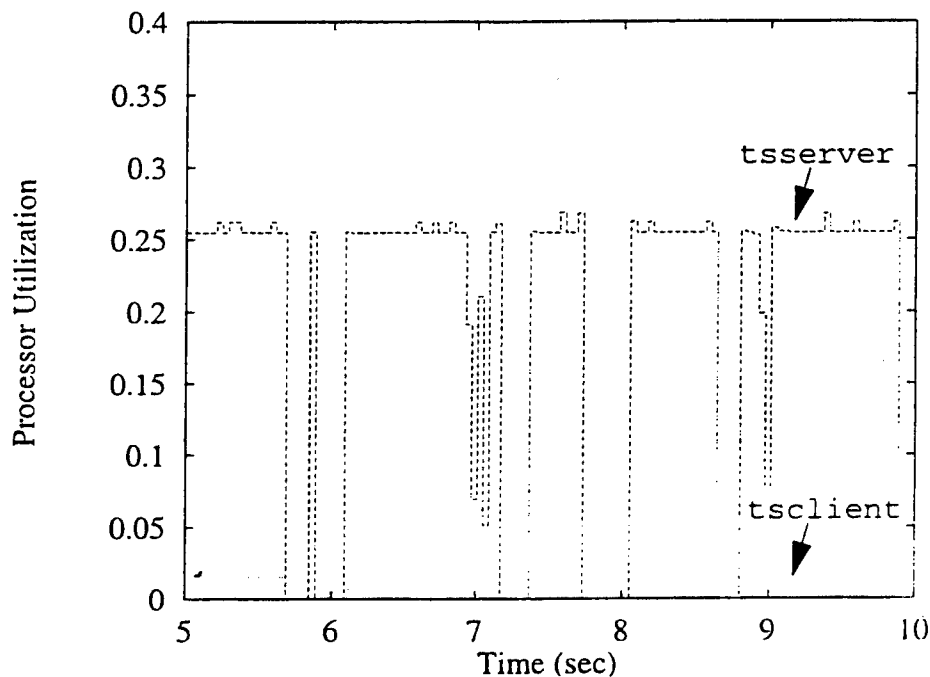


Figure 6-7: Experiment 5 Results

The competition from unreserved programs interferes with the execution of the server, and completely locks out the server for up to 100 to 200 ms at a time. During these periods, there is no usage recorded by either the server or the client, since the client cannot make progress without the server making progress. The usage for both the client and the server falls to zero for several reservation periods. This kind of behavior is clearly undesirable since many instances of the computation cannot take place, and the deadline is missed each time.

The results of Experiment 6 (Figure 6-8) show that when a client and server use an IPC mechanism that is integrated with the reservation scheduling policy, in this case a version of RT-IPC extended to work with the reservation scheduling policy, the combined client/server activity is quite predictable. The RT-IPC mechanism propagates the client's reserve to the server and supports a server that charges the computation time of each client to the client's reserve. So most of the computation in this experiment is being charged to the client (as it should be) instead of to the server (as in the previous case). The utilization charged to the client's reserve averages 0.223 with a 5-percentile of 0.222 and a 95-percentile of 0.228. The server utilization for the graphed intervals is 0.0114 on average; the 5-percentile is 0.0112 and the 95-percentile is 0.0117. These numbers indicate very predictable performance for these programs over time.

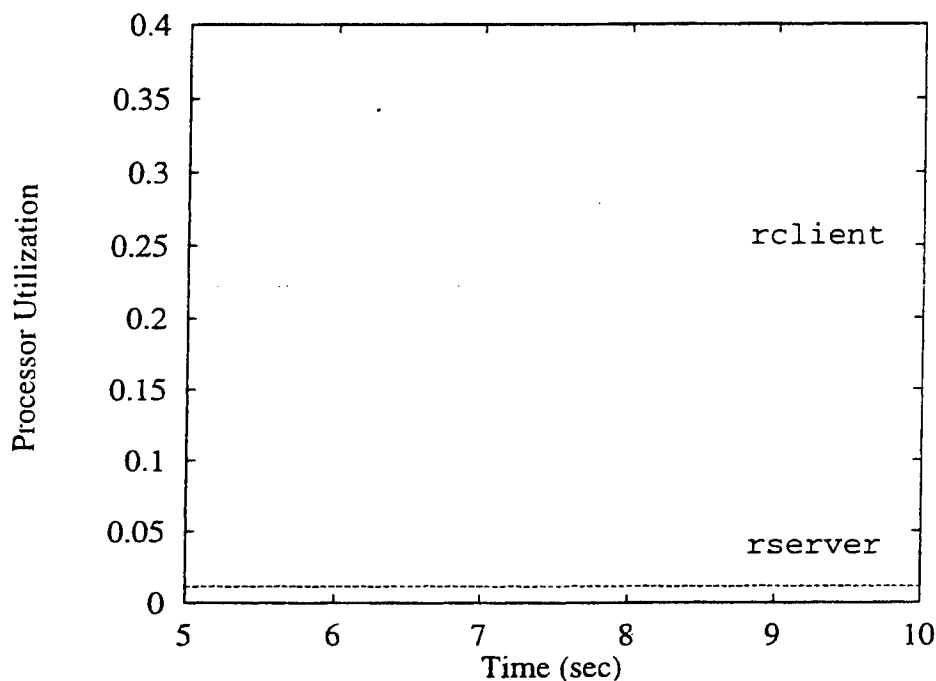


Figure 6-8: Experiment 6 Results

Figure 6-9 shows the results from Experiment 7. These results show that even when several reserved clients are using the same server, they can all meet their periodic timing constraints (subject to the admission control policy). This is true in spite of the presence of competition from unreserved `arith` programs.

Experiment 7 shows the usage charged to the reserves of the three reserved clients. The client with the 8ms/40ms synthetic computation has an average utilization of 0.223 with a 5-percentile of 0.221 and a 95-percentile of 0.232. The client with the 8ms/50ms computation has an average utilization of 0.178, a 5-percentile of 0.176 and a 95-percentile of 0.184. And the client with the 8ms/60ms computation gets an average utilization of 0.150; the 5-percentile is 0.147 and the 95-percentile is 0.157. These numbers indicate fairly tight distributions

around the averages for these applications, even though they are competing for the server and even though there are additional unreserved programs competing for the processor as well. The servers computation time seems erratic, and there are two reasons: it has a small reservation period (which just determines the usage measurement period), and its clients all have different periods and request different computations at different rates.

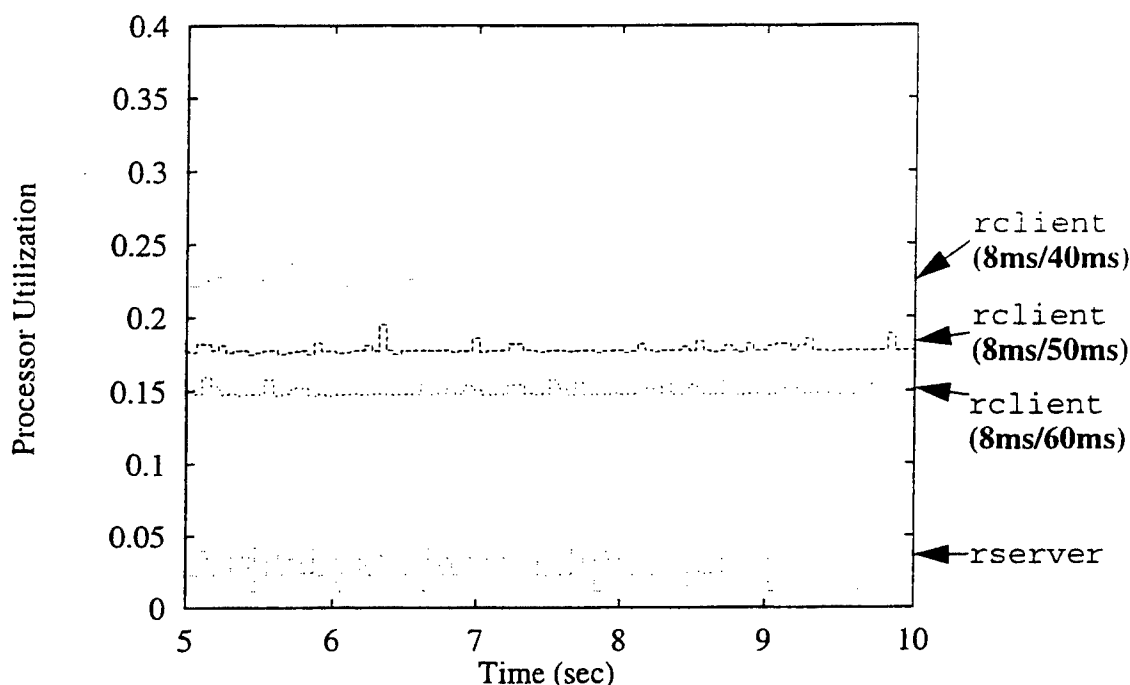


Figure 6-9: Experiment 7 Results

Finally, the results of Experiment 8 appear in Figure 6-10. These results show that in the case where reserved clients compete with an unreserved client for a single server, the reserved clients are still able to satisfy their timing constraints.

The reserved client with the 8ms/50ms computation has an average utilization of 0.181. It has a 5-percentile of 0.176 and a 95-percentile of 0.187. The reserved client with the 8ms/60ms computation has an average utilization of 0.151 with a 5-percentile of 0.148 and a 95-percentile of 0.158. Thus, these reserved programs are able to get the processor time they have reserved. As the graph shows, the unreserved client manages to complete its computation during some of its periods, but not in others. So the usage function goes back and forth between getting about 0.22 utilization in the periods where the computation is completed and getting 0 utilization in the periods where it does not get to complete the computation. The average utilization for this unreserved client is 0.131; the 5-percentile is 0.0059 and the 95-percentile is 0.222. This of course confirms that the dispersion of the utilization measurements for this unreserved client is large.

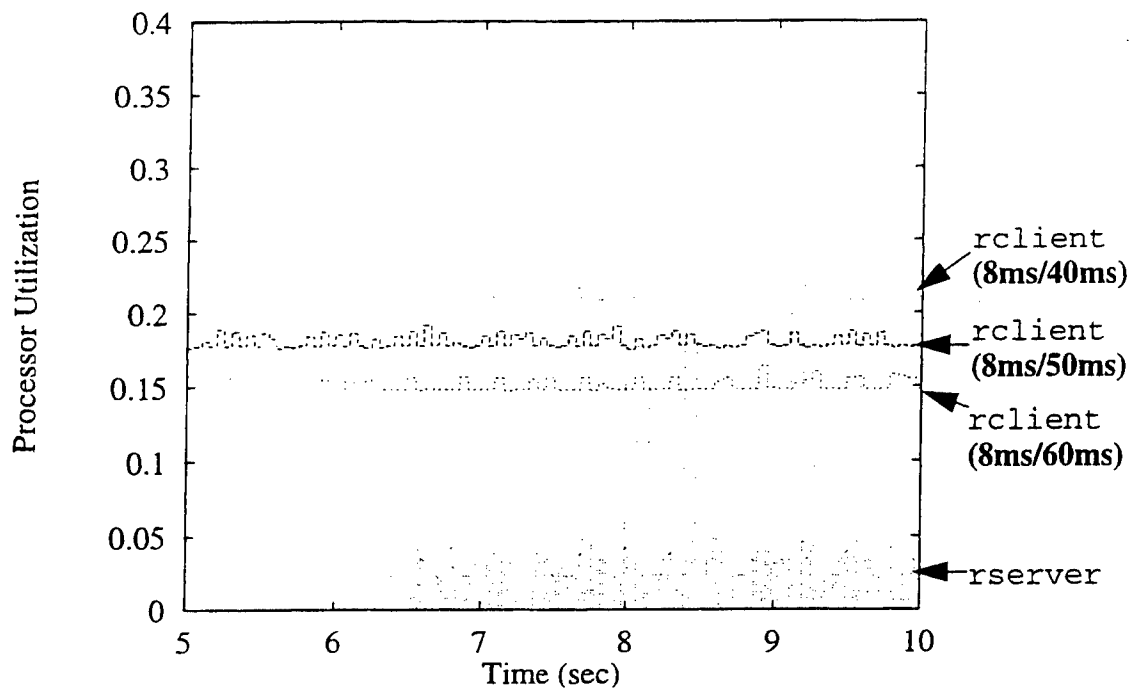


Figure 6-10: Experiment 8 Results

6.2.2.3 Analysis

The experiments with unreserved and reserved client/server pairs demonstrate the importance of doing reserve propagation properly between client and server when the server is designed to use the client's reserve. Experiment 4 shows the baseline behavior for the client/server pair with a periodic client driving the timing of the activity.

Experiment 5 demonstrates the problem that can occur when the reserve propagation is not handled properly. In this experiment, the client allocates a reserve and passes it to the server, which then uses it to charge the client's service time. However, this client/server pair does not use the "priority" inheritance mechanism to ensure that the server takes on the "priority" of the client as soon as the RPC is enqueued in the servers input queue. With competition from unreserved activities, this lack of "priority" inheritance results in many missed deadlines for the client/server activity.

The results of Experiment 6 show that the proper periodic behavior of the client/server pair can be restored by using the "priority" inheritance mechanism. Priority inheritance makes sure that the competing unreserved activities do not interfere with the server as it attempts to read the request from its input queue and switch to the client's reserve.

The last two experiments demonstrate that the reserve propagation mechanism, which includes "priority" inheritance and the server binding to the client's reserve, works properly even when there are multiple reserved clients or there are unreserved clients in addition to

reserved clients. Experiment 7 demonstrates that with three reserved clients (all with different computation times and timing constraints), the server can service them all in time to make their deadlines. This is true even though there is competition from unreserved activities, which can exploit any lapses in an incorrectly implemented reserve propagation mechanism and cause delays in the client/server activities.

Experiment 8 shows that when two reserved clients and one unreserved client share a server in an environment with competition from other unreserved activities, the reserved clients will always meet their deadlines. In this case the reserved clients meet their deadlines even though other unreserved competitors sometimes delay the unreserved client. This experiment further tests the integrity of the reserve propagation mechanism by making sure that the “priority” of the unreserved client is not propagated to the server at the wrong time, causing the server to appear unreserved and resulting in interference from the unreserved competitors.

6.2.3 QTPlay/X Server

Experiments using task sets with synthetic workloads provide evidence that the reservation system can support the predictable execution of real-time programs. However, experiments with real applications that use the reservation system to meet timing constraints provide stronger evidence of the usefulness of the reservation system in real-world situations. The experiments described in this section use a video player that has been modified to use processor reserves and a version of the X Server that has been modified to support reserves.

6.2.3.1 Methodology

These experiments use the QuickTime video player, called QTPlay, and the reserved X Server, both of which were described in the previous chapter. Since these programs are described in detail elsewhere, the description here is brief.

The QTPlay application prefetches a short video clip into main memory and repeatedly displays that clip to avoid interaction with the file system and disk (which are not reserved in this system) during the experiments. It allocates a reserve during initialization based on command-line arguments and then starts playing the video. For each frame, the player records in a buffer the start time and end time for the frame processing, and at the end of the experiment these data are written to a file on the disk for later analysis.

When QTPlay connects to the X Server, it passes a reference to its reserve for the X Server to use when performing frame display operations. The X Server was modified to order requests based on reservation information and to charge the computation time for each operation to the appropriate client’s reserve.

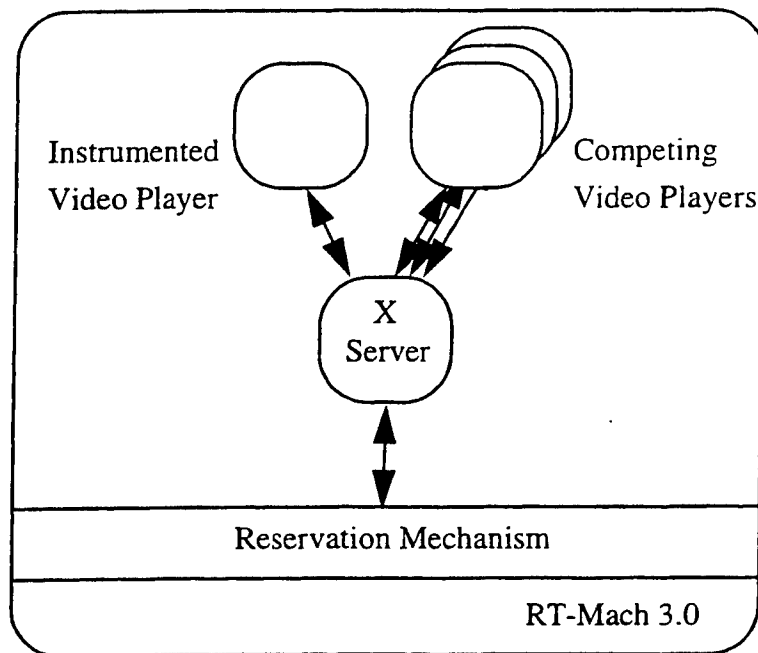


Figure 6-11: Software Configuration

Figure 6-11 shows the basic software structure that is used for all of the experiments in this section. There is an instrumented QTPlay application which may or may not have a reservation and which records timestamps for each frame at the beginning of the frame display computation and then again at the completion of frame display. Other instances of QTPlay may compete with this instrumented player. These are unreserved and continuously display frames as fast as possible (providing the maximum competition).

QTPlay can display frames at a particular period or in a continuous loop, and in all of the experiments below, the frame resolution is 160x120 pixels with 8 bits/pixel. The timing is specified by command-line arguments.

Experiment 9 is designed to illustrate the usage pattern for QTPlay with no competition for the processor. The parameters for QTPlay are given in Table 6-9. The period is 33 ms, which corresponds to a frame rate of 30 frames/second.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	0 ms	0 ms

Table 6-9: Experiment 9 Parameters

Experiment 10 is intended to show what can happen when QTPlay is executed under a time-sharing scheduler with a competing instance of the QTPlay program. The parameters for this experiment appear in Table 6-10. The QTPlay instance listed in the first row of the table is instrumented to provide timing information and the other just competes for the resources for displaying frames by continuously displaying frames as fast as possible.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	0 ms	0 ms
QTPlay	1	Continuous	N/A	0 ms	0 ms

Table 6-10: Experiment 10 Parameters

Experiment 11 is designed to show how well an instance of QTPlay with a reservation can coordinate with the reserved X Server to achieve a constant playback rate for frames. Table 6-11 gives the parameters for the experiment. The instrumented QTPlay application has a reservation and competition from one other unreserved QTPlay instance.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	14 ms	33 ms
QTPlay	1	Continuous	N/A	0 ms	0 ms

Table 6-11: Experiment 11 Parameters

Experiment 12 is similar to Experiment 10 in that it explores the behavior of an unreserved QTPlay with competition from 3 unreserved QTPlay instances rather than just one. For completeness, the parameters appear in Table 6-12.

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	0 ms	0 ms
QTPlay	3	Continuous	N/A	0 ms	0 ms

Table 6-12: Experiment 12 Parameters

Experiment 13 is similar to Experiment 11; it looks at the behavior of a reserved QTPlay instance with three competing QTPlay instances. The parameters are given in Table 6-13. These last two experiments look at the behavior of unreserved and reserved QTplay applications under adverse conditions (intense competition from multiple unreserved X clients).

Program	#	Mode	Period	Reserved Computation	Reservation Period
QTPlay	1	Periodic	33 ms	14 ms	33 ms
QTPlay	3	Continuous	N/A	0 ms	0 ms

Table 6-13: Experiment 13 Parameters

6.2.3.2 Results

The results for Experiment 9 illustrate the timing behavior of a QTPlay application with no competition. Figure 6-12 shows these results. For each frame, the player records the starting time and ending time. The x-axis is the frame number, counting frames starting at the 200th frame through to the 400th. For each frame value, the difference between the start time and end time is computed, and the y-axis is this frame delay measured in milliseconds.

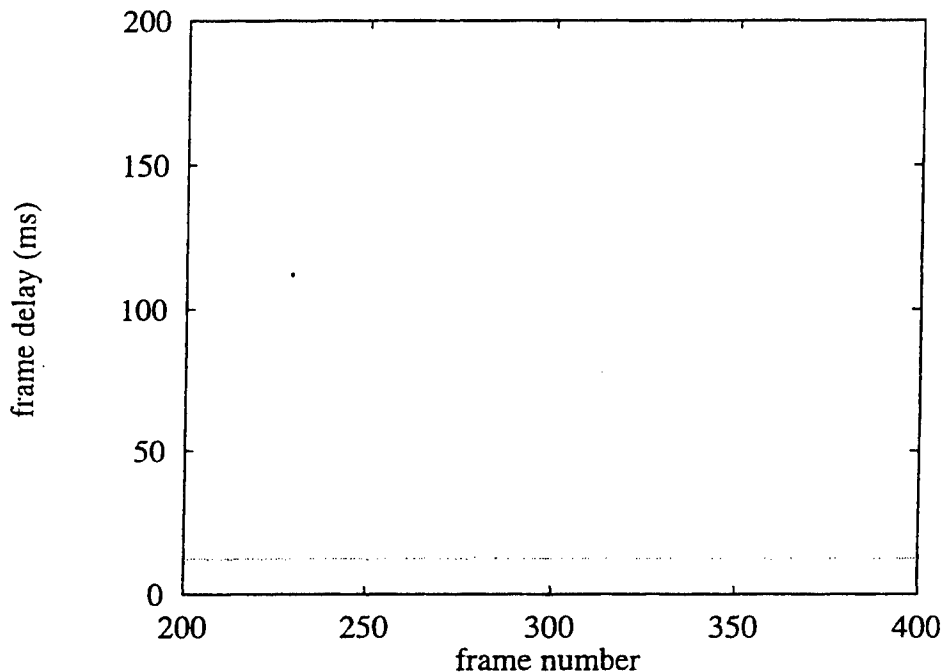


Figure 6-12: Experiment 9 Results

In the figure, the frame delay averages 12.1 ms with a 5-percentile of 11.8 and a 95-percentile of 12.6. This indicates that the software took an average of about 12 ms to perform all the computations necessary to display a frame when there was no competition for resources.

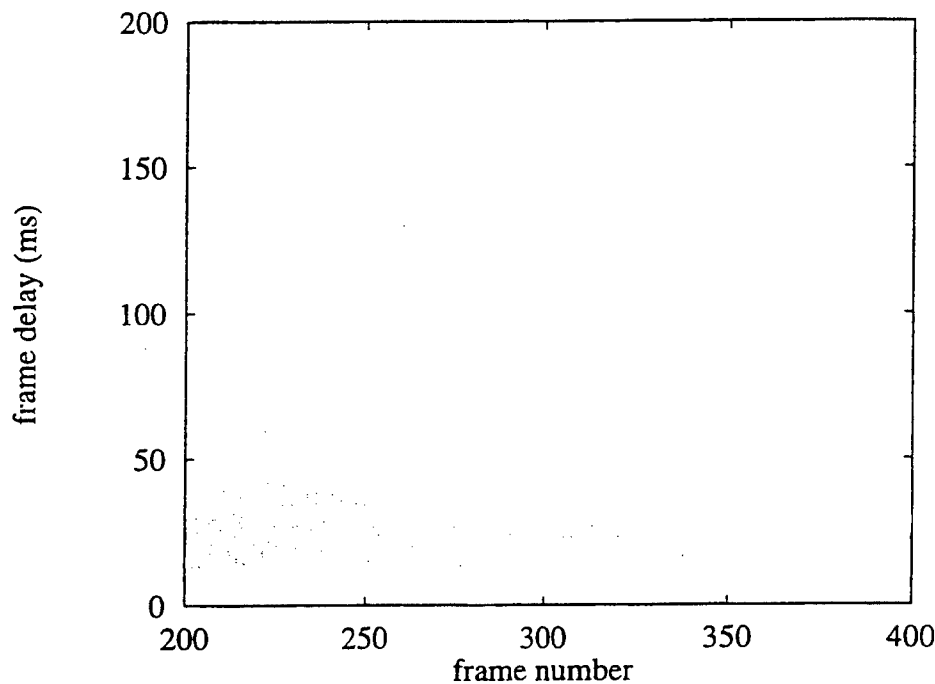


Figure 6-13: Experiment 10 Results

The results from Experiment 10 show a slightly different picture in Figure 6-13. Again, the x-axis is frame number, and the y-axis is frame delay measured in ms using the same method. With time-sharing scheduling and one competing QTPlay, the instrumented QTPlay sees quite a bit of interference in its frame delay time. The frame delay is much more variable. The average delay is 25.8 ms with a 5-percentile of 11.2 and a 95-percentile of 45.0.

In the results from Experiment 11, the instrumented QTPlay has a reservation, and its frame delay is much less variable even with competition from one unreserved QTPlay instance. Figure 6-14 shows the timing behavior. As before, the x-axis is frame number; the y-axis is frame delay in milliseconds.

The frame delay still has a bit of variation, but it is much less variable than the case where the QTPlay application is unreserved. The average delay is 19.4 ms with a 5-percentile of 14.7 and a 95-percentile of 24.3. So QTPlay is almost always able to display each frame within its 33 ms period.

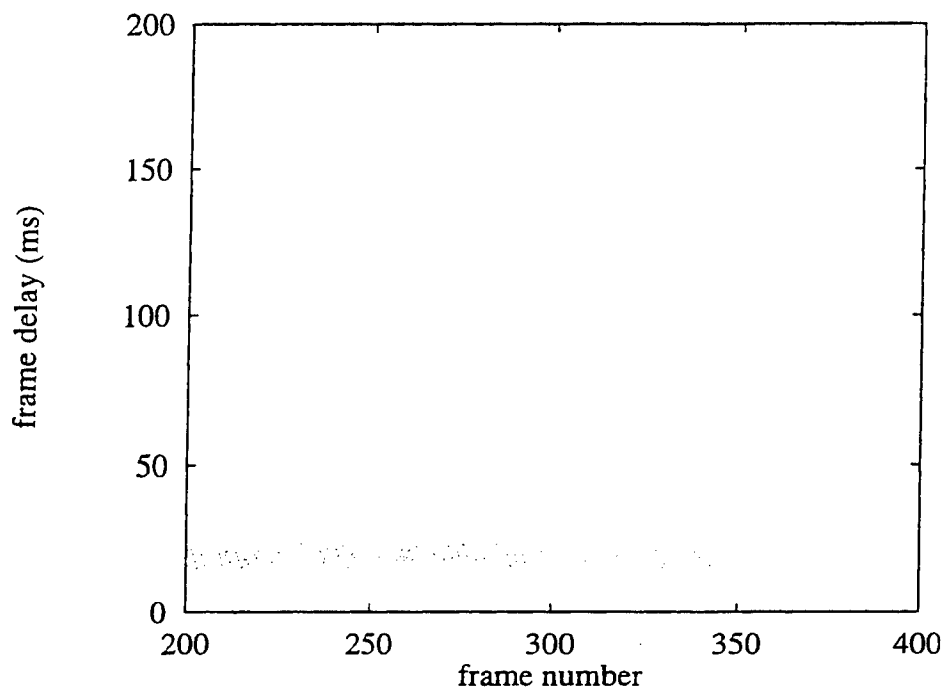


Figure 6-14: Experiment 11 Results

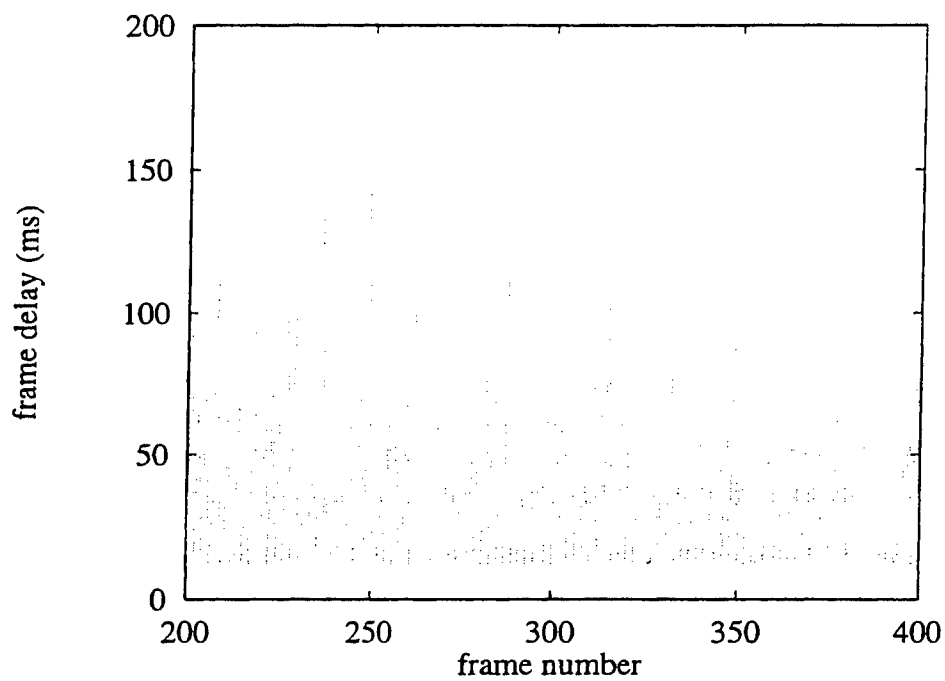


Figure 6-15: Experiment 12 Results

The results from Experiment 12 show how much the frame delay variation can be for an unreserved QTPlay instance that has three unreserved QTPlay applications competing to display frames. Figure 6-15 shows these results. As the figure shows, the variation in frame delay is quite large. The 5-percentile for the frame delay is 11.5 ms, and the 95-percentile is 107 ms with an average frame delay of 40.3 ms. A delay of 150 ms (which does not show up in the 95-percentile number but occurs a number of times during the test) or even 100 ms in a sequence of video frames is clearly noticeable to the human eye. Frame rates of 15 frame/second or more are required to sustain the illusion of smooth motion. This implies that with delays above 66 ms or so, the illusion of smooth motion may be destroyed.

In contrast, the results of Experiment 13, shown in Figure 6-16, demonstrate how well the reservation system can control the variability in frame delay for the reserved QTPlay application, even with much competition from unreserved instances of the QTPlay program. The frame delay in the figure is somewhat variable, but the variation is much less than in the case of the unreserved QTPlay with three competitors. The 5-percentile is 13.4 ms and the 95-percentile is 34.2 ms with an average of 20.7 ms. This is well below the target of the 60 ms period necessary for smooth-looking motion.

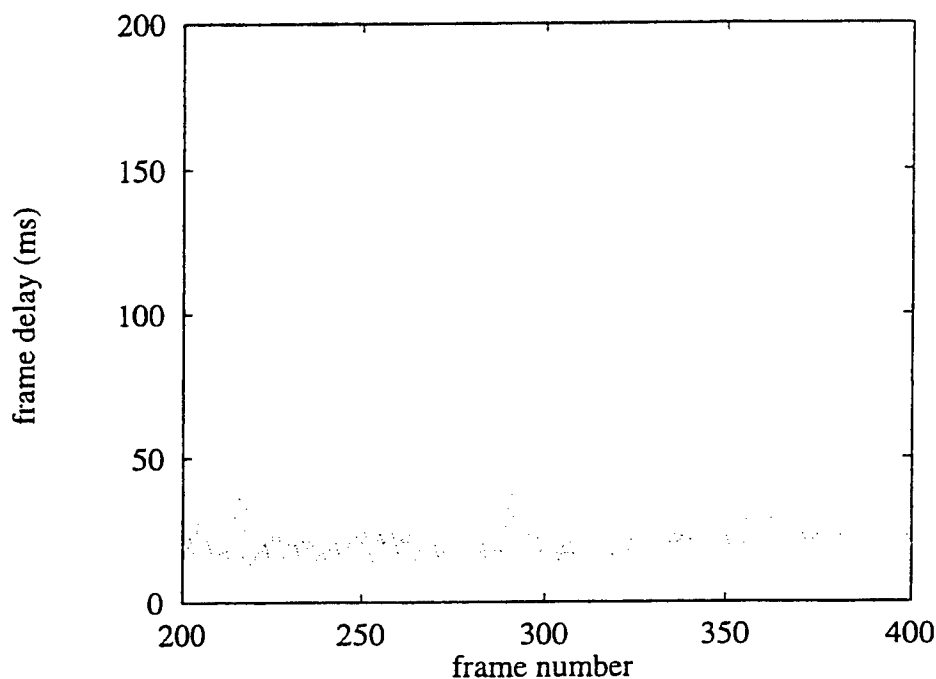


Figure 6-16: Experiment 13 Results

6.2.3.3 Analysis

The QTPlay/X Server experiments show that even with real applications like a Quick-Time video player and the X server, the reservation system can provide predictable perfor-

mance for real-time programs that must meet timing constraints to achieve satisfactory performance.

Experiment 9 shows the baseline behavior that is desired for the video player. This experiment has no competing activity, so there is no contention for resources and the behavior is very regular. Experiment 10 has a competing video player in addition to the instrumented video player, and with time-sharing scheduling, this competitor causes some scheduling delay in the instrumented player. In Experiment 12, there are three competitors, and the interference to the instrumented player is very bad. The player frequently has frame delays of 60 to 100 ms and even as high as 150 ms. These kinds of delays are clearly noticeable to a human observer of the video stream.

In Experiment 11, the instrumented video player is reserved with one competitor, and although there is a little fluctuation in the frame delay, it is limited to 24.3 for the 95-percentile. In Experiment 13, the reserved video player has competition from three unreserved video players. The frame delays show a little more variability, but are still limited to a 95-percentile of 34.2 ms compared to the 100 and even 150 ms delays experienced with time-sharing scheduling.

The question of why the reserved QTPlay/X Server combination suffered any delay arises. The reason is that the X Server was not implemented from scratch to use reserves. The extensions to allow it to use reserves did not completely restructure the request input queue, in particular. So the server reads requests from its input queue, orders them internally, and then performs the operations. If server's client interface code were completely rewritten to support reserves, the behavior would be comparable to that of the client/server synthetic benchmarks where the servers were designed from scratch.

6.2.4 mpeg_play/X Server

In addition to the QuickTime video player, a version of the `mpeg_play` decoder was modified to use reserves and coordinate with the reserved X Server. This decoder uses some simple usage measurement and adaptation techniques to tune the reservation parameters and timing parameters based on changing system conditions.

6.2.4.1 Methodology

The `mpeg_play` modifications were described in detail in the previous chapter, so the description here is brief. The player prefetches a video clip into memory to avoid interference in the file system. It requests a processor reservation and passes the reference to its reserve to the X Server. While it is executing, the decoder keeps track of its resource usage and timing characteristics, and it makes adjustments to the reservation parameters and/or period of the program based on usage.

Experiment 14 is designed to determine whether the `mpeg_play` decoder can successfully modify its reservation parameters and/or behavior based on existing conditions. The decoder starts executing with a reservation that is too small for its computation time. Competition is then introduced in the form of reserved and unreserved activities.

6.2.4.2 Results

The behavior of the `mpeg_play` application under the conditions of Experiment 14 is illustrated in Figure 6-17. The decoder is able to tune its reservation parameters based on run-time information and stabilize its own behavior.

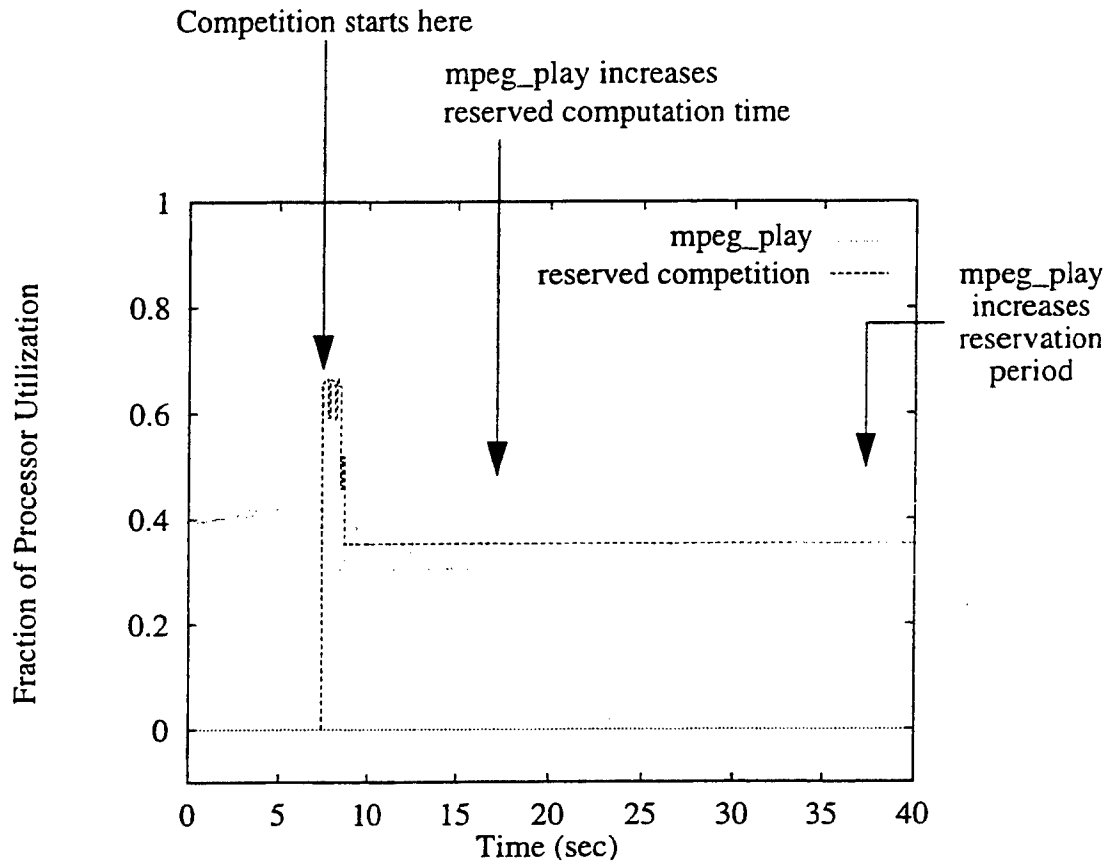


Figure 6-17: Experiment 14 Results

Figure 6-17 shows the processor utilization of the `mpeg_play` decoder over a period of 40 seconds. The x-axis is time in seconds, and the y-axis is processor utilization. During the first 7 seconds, `mpeg_play` averages about 40% of the processor, even though its reservation is only 30 ms every 100 ms. Since there is no competition, it consumes 40%. At about 7 seconds into the experiment, a reserved program (shown in the graph) and several unreserved programs (not shown) are introduced. The usage of `mpeg_play` immediately drops to its reserved level of 30% of the processor. This is not enough to sustain its previous frame rate, so some frames are dropped.

The reserved activity, which had a usage spike at the time it started, settles down to a constant 37%. The spike occurs since the time-sharing algorithm initially allows the new program to get more cycles than its reservation. After consuming a large percentage of the

processor, however, the reserved activity no longer gets additional cycles from the time-sharing algorithm, and the usage flattens.

After several seconds, `mpeg_play` realizes its frame rate has fallen and attempts to increase its reservation. At about 17 seconds into the experiment, the decoder increases its reservation to 41 ms reserved every 100 ms, and its frame rate increases accordingly. Again at about 37 seconds into the experiment, the decoder changes its reserved computation time to 47 ms and its reservation period to 111 ms to fine-tune the reservation even further.

6.2.4.3 Analysis

This experiment demonstrates that an application can adjust its reservation parameters and adapt its behavior based on the usage information from the reservation mechanism. In this case, the initial reservation of the `mpeg_play` application did not have to be accurate since the application automatically adjusted the reservation levels based on usage measurements.

6.2.5 Protocol processing workloads

The experiments in this section explore the real-time behavior of the socket library (called `libsockets`) protocol processing. The behavior obtained by an application using the socket library is compared to the behavior using the UX Server's socket service under both time-sharing scheduling and reserves.

6.2.5.1 Methodology

The `libsockets` experiments use the same software environment as the experiments with independent tasks and client/server workloads. There is a configuration manager that reads the task set specification and creates the specified programs. In addition to the workload programs introduced so far, these experiments use the following programs:

- `stdio` - Creates a periodic thread that calls a sequence of file operations.
- `udps` - Creates a periodic thread that sends some number of packets (specified in the program computation field) in each period. This program uses the UX Server to send packets.
- `udpls` - Creates a periodic thread that sends some number of packets in each period. This program uses `libsockets` to send the packets rather than interacting with the UX Server.
- `udpr` - Creates a periodic thread that receives some number of packets in each period using the UX Server to receive the packets.
- `udplr` - Creates a periodic thread that receives some number of packets in each period using `libsockets`.

Experiment 15 is designed to show how a packet-sending activity can be disturbed by competition from a combination of compute-intensive and I/O-intensive activities, especially when all of those activities use system services which interact with each other as they do in the UX Server. The task set for this experiment is given in Table 6-14. It shows two udp senders (udps) with slightly different workloads. The competition for this experiment (as for the next three experiments) consists of five compute-intensive arith programs and five I/O-intensive stdio programs. In this experiment, the packets sent by the udps programs are received on a remote machine by yet another program that records a timestamp when the each packet arrives. This program buffers the timestamps and dumps them out at the end. The timestamp data can be used to judge whether the packet senders were able to send their packets out as desired or not.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udps (A)	1	4 pkt	40 ms	0 ms	40 ms
udps (B)	1	2 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-14: Experiment 15 Parameters

Experiment 16 is designed to determine whether the packet-sending applications can send their packets on time when using libsockets for their network protocol processing. The parameters are given in Table 6-15. This experiment differs from Experiment 15 in that the UDP packet senders use libsockets instead of the UX Server and they have reservations instead of being scheduled by the time-sharing scheduler. The competition is the same: five compute-intensive programs and five I/O-intensive programs.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udpls (A)	1	4 pkt	40 ms	10 ms	40 ms
udpls (B)	1	2 pkt	40 ms	6 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-15: Experiment 16 Parameters

The purpose Experiment 17 is to determine whether a UDP packet-receiving program that attempts to receive a number of packets periodically can meet that objective. The `udpr` program attempts to receive some number of packets in each period. This program receives packets through the UX Server and runs without a reservation. The competing activities are identical to the previous two experiments. Table 6-16 presents the parameters for this experiment.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udpr (A)	1	4 pkt	40 ms	0 ms	40 ms
udpr (B)	1	3 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-16: Experiment 17 Parameters

Experiment 18 is designed to determine whether a reserved packet receiver that uses `libsockets` can predictably execute periodically to receive a number of packets. The `udplr` program is a periodic packet receiver that uses `libsockets` instead of the UX server. This experiment includes the same competition from compute-intensive and I/O intensive tasks as the other experiments in this section. The parameters are given in Table 6-17.

Program	#	Program Computation	Program Period	Reserved Computation	Reservation Period
udplr (A)	1	4 pkt	40 ms	0 ms	40 ms
udplr (B)	1	3 pkt	40 ms	0 ms	40 ms
arith	5	various	various	0 ms	40 ms
stdio	5	various	various	0 ms	40 ms
monitor	1	N/A	20 ms	1 ms	20 ms

Table 6-17: Experiment 18 Parameters

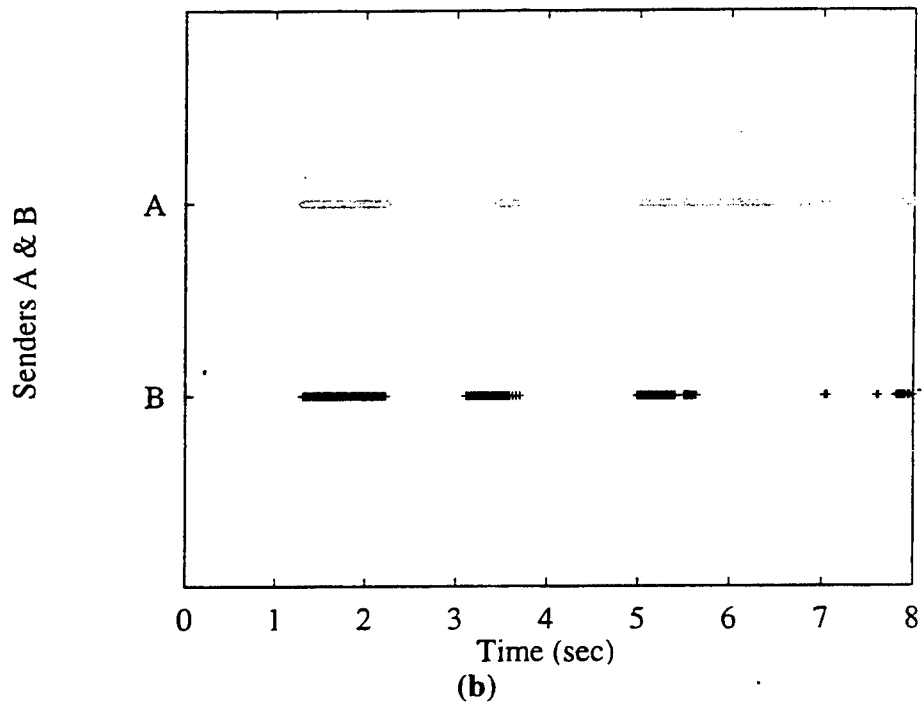
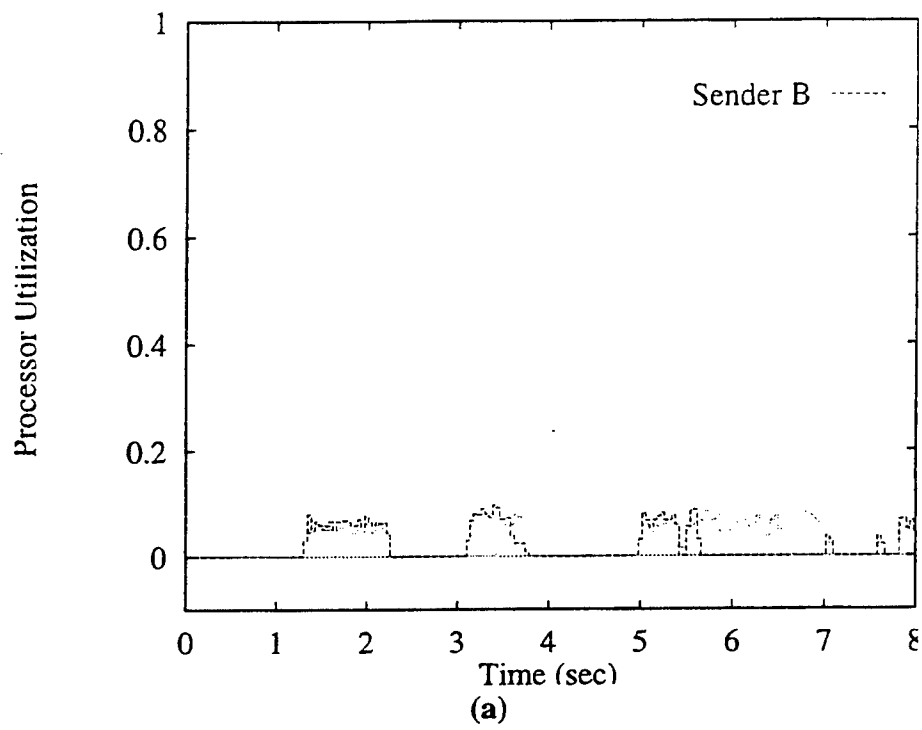


Figure 6-18: Experiment 15 Results

6.2.5.2 Results

The results of Experiment 15 are shown in Figure 6-18. Part (a) of the figure shows the processor utilization over time for the two packet senders, which use the UX Server implementation of sockets and run without reservations. The x-axis is time in seconds, and the y-axis is processor utilization. The two programs, denoted "Sender A" and "Sender B", show a very erratic usage pattern. Frequently, the usage drops to zero for over 1 second. The average utilization for Sender A is 0.023 with a 5-percentile of 0 and a 95-percentile of 0.0741. The average utilization for Sender B is 0.0193 with a 5-percentile of 0 and a 95-percentile of 0.0796. The dispersion is clearly substantial for these applications, and that dispersion in utilization achieved translates directly into missed deadlines.

Figure 6-18(b) shows the record of timestamps that were received by a remote receiver for both senders. The x-axis in the graph is time in seconds, and there are two horizontal lines, one for Sender A and the other for Sender B. A mark on the line corresponding to Sender A at a particular time indicates that a packet arrived at that time and likewise for Sender B. This graph shows that the packets were received on the remote host sporadically. The pattern of packet receptions corresponds closely with the pattern of usage seen in part (a) of the figure. The largest gaps between received packets were 1.28 seconds and 1.22 seconds for Sender A and 1.93 seconds and 1.40 seconds for Sender B. These senders are attempting to send packets every 40 ms, so clearly they are not able to schedule the message sending activity as desired.

The results of Experiment 16 appear in Figure 6-19. In this case, the senders have reservations and use the `libsockets` library to avoid depending on the UX Server for networking. The graph shown in part (a) of the figure shows time in seconds on the x-axis and processor utilization on the y-axis. The processor utilization for both of the senders is very regular, indicating that in each reservation period, the programs were able to send the packets they were supposed to send. There are no long intervals of zero usage as in the previous case. The average utilization of Sender A is 0.103 with a 5-percentile of 0.101 and a 95-percentile of 0.110 indicating a very tight distribution around the average. Likewise for Sender B, the average utilization is 0.0591 and the 5-percentile is 0.0582 with a 95-percentile of 0.0624. This is also a tight distribution.

The graph in Figure 6-19(b) supports the conclusion that the senders in this experiment are able to send their packets very regularly in each period. As before, the x-axis is time in seconds and two horizontal lines indicate the timestamps for packets received from the two senders. A point on the line corresponding to Sender A represents a packet that was received at the associated time. In this experiment, packets are received very regularly from each sender. There are no significant gaps in the reception pattern. The maximum gaps for packets received from Sender A are 0.0519 seconds and 0.0423 seconds, and the maximum gaps for packets received from Sender B are 0.0438 seconds and 0.0435 seconds. So the conclusion is that the combination of `libsockets` with reserved resources yields predictable behavior for packet senders.

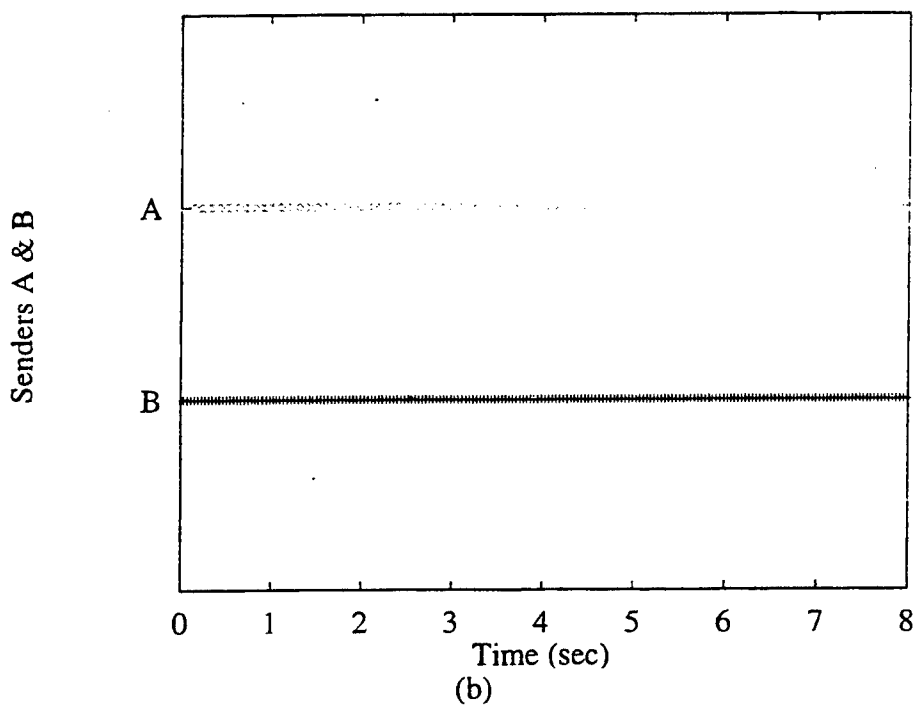
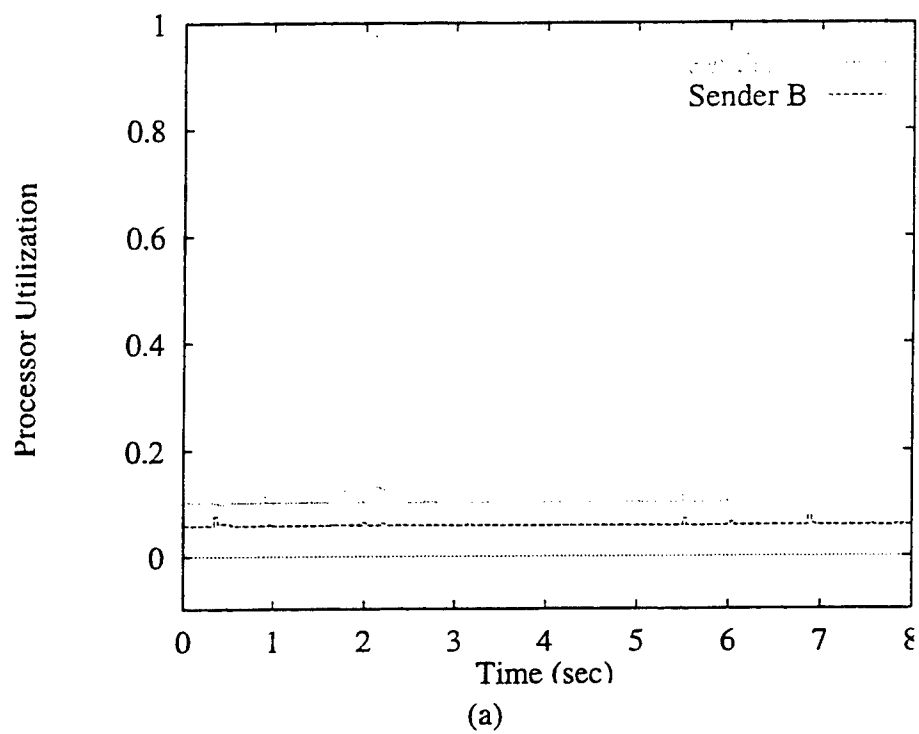


Figure 6-19: Experiment 16 Results

Figure 6-20 shows the results of Experiment 17. In this case, a remote sender periodically sends packets to the two receivers described in the task set. In this experiment, the two receivers are unreserved and use the UX Server's implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The behavior is very erratic. The remote host sends packets periodically, but the receiver is not always able to run long enough to receive the packets. Receiver A has an average utilization of 0.0134 with a 5-percentile of 0 and a 95-percentile of 0.0617. This is not the kind of timely behavior desired in the packet receiver. Receiver B has an average utilization of 0.0220 with a 5-percentile of 0 and a 95-percentile of 0.0726. Again, the dispersion is significant. The usage for both receivers frequently drops to zero, indicating that the packets are being dropped for significant periods of time.

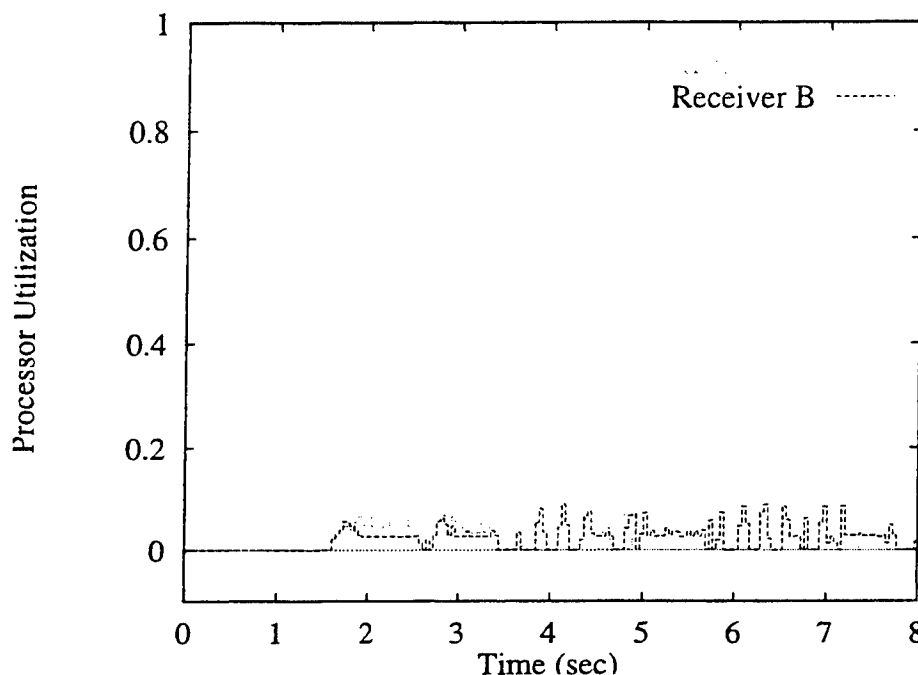


Figure 6-20: Experiment 17 Results

The results of Experiment 18 are shown in Figure 6-21. Again, a remote sender periodically sends packets to two receivers which have reservations and which use the `libsockets` implementation of sockets. The graph has time in seconds on the x-axis and processor utilization on the y-axis. The usage functions of the two receivers are plotted over the duration of the experiment. The average utilization for Receiver A in this case is 0.149 with a 5-percentile of 0.139 and a 95-percentile of 0.173. This indicates a fairly tight distribution. Receiver B has an average utilization of 0.130 with a 5-percentile of 0.115 and a 95-percentile of 0.140. Again the utilization achieved is quite consistent across periods for the duration of the test. It is clear that the receivers do not drop to very low utilizations for significant intervals of time. Their relatively constant usage indicates that they are able to process the incoming packets in a predictable manner.

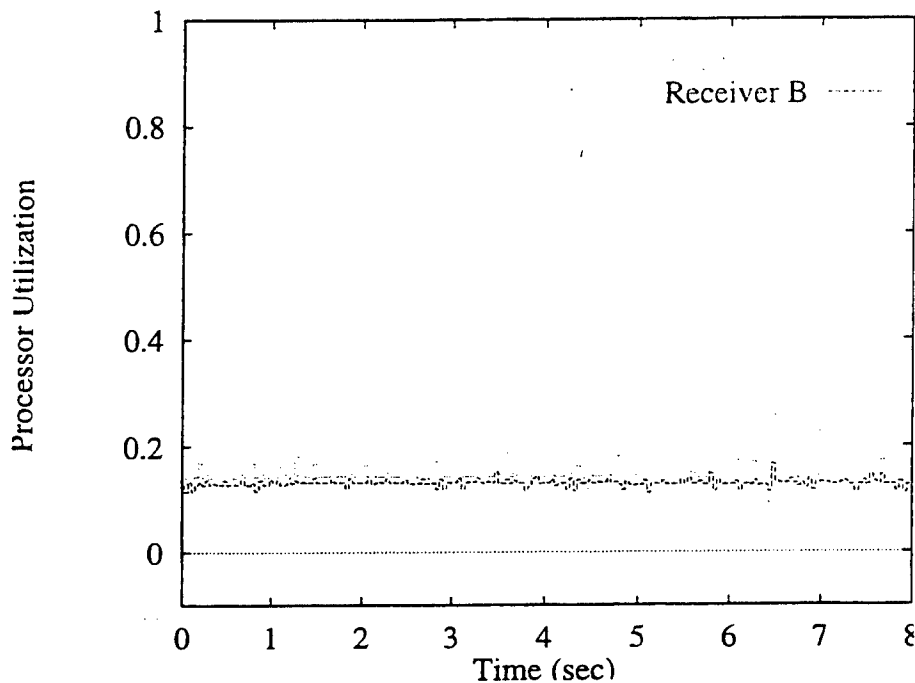


Figure 6-21: Experiment 18 Results

6.2.5.3 Analysis

The results from the `libsockets` experiments showed that when packet senders and receivers ran in time-sharing mode using the socket implementation provided by the UX Server, their behavior was erratic when other programs were competing for the processor and access to other services provided by the UX Server. Reserved packet senders and receivers that used `libsockets` for handling network packets had much better behavior. They were able to execute periodically and perform each sending or receiving computation by the end of the corresponding period. Other experiments (not presented here) showed that the behavior of reserved programs that used UX sockets was just as bad as that of unreserved programs with UX sockets. Also, unreserved programs that used `libsockets` exhibited unpredictable behavior when executing with competition as well. These experiments indicate that using a reservation mechanism or a `libsockets` mechanism alone does not ensure predictability in programs; both mechanisms are needed.

6.3 Scheduling cost

This section addresses the scheduling costs of predictable programs that can meet their timing constraints under the reservation system. It looks at measured aggregate costs for the system as well as measurements of scheduling operations that contribute to the costs. Such measurements enable cost projections to be made for specific task sets.

6.3.1 Measured aggregate scheduling cost

The reservation system ensures that reserved resources will be available to enable real-time programs to meet their timing constraints, but this predictability has costs associated with it. In particular, the accurate measurements and the timers necessary for enforcement take some time. In measuring the aggregate scheduling cost, the intention is to determine how much time is consumed by scheduling costs in the case of a reserved periodic thread compared to that of an unreserved thread.

6.3.1.1 Methodology

Experiment 19 is designed to measure the scheduling cost for a periodic thread as the period varies. The task set includes the periodic thread and an "idle" thread that runs in the background to consume all processor time not consumed by the periodic thread and the system's housekeeping activities. The system scheduling cost is taken to be the total time of the test minus the time consumed by the periodic thread and the idle thread. This scheduling cost includes context switch times, associated cache effects, as well as the cost of timers and clock operations for the reservation mechanism.

One series of tests measures the scheduling cost for a reserved periodic thread whose reservation period ranges from 20 ms to 200 ms. Most of the cost for the reservation system is a fixed cost in each period, so a longer period implies a relatively smaller cost. The other series of tests measures scheduling cost for an unreserved periodic thread with a period that varies from 20 ms to 200 ms.

6.3.1.2 Results

The results of the scheduling cost measurements are presented in Figure 6-22. The graph consists of two functions: the scheduling cost associated with a reserved periodic thread as a function of period and the scheduling cost associated with an unreserved periodic thread as a function of period. The x-axis is the period in milliseconds, and the y-axis is the percentage of the processor that is lost to scheduling costs.

The scheduling cost for the reserved thread starts out about 3% for a reservation period of 20 ms and drops off as the reservation period is increased. For a 100 ms reservation period, the scheduling cost is about 0.5% and for a 200 ms reservation period, it is about 0.2%. For the unreserved thread, the scheduling cost is smaller, starting at about 2.2% for a 20 ms period. The cost drops off to about 0.5% for a 100 ms period and then to about 0.1% for a 200 ms period.

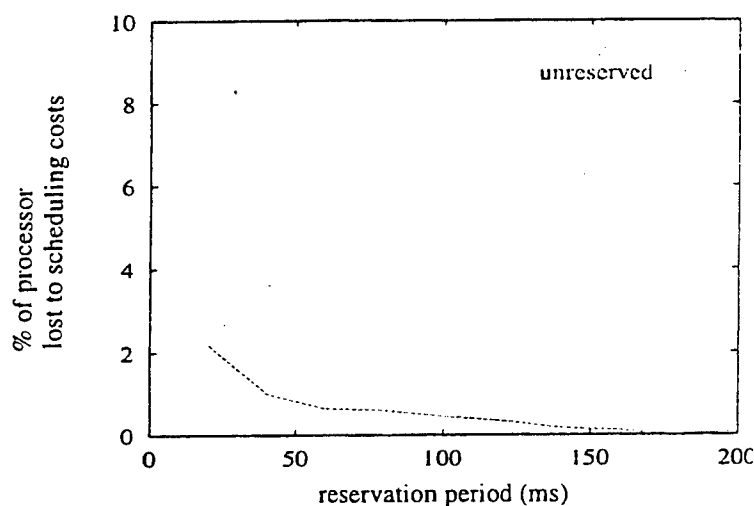


Figure 6-22: Scheduling Cost

6.3.1.3 Analysis

The scheduling cost measurements indicate that the cost of threads with very small reservation periods (smaller than about 30 ms) grows somewhat as the reservation period decreases. For reservation periods in the range of 40 to 100 ms, which would be an appropriate range for many audio and video applications for example, the scheduling cost is acceptable.

The scheduling cost is relatively high for the reservation system because the clock/timer card used in the experiments is very sensitive to the timing of loads and stores in its control and data registers. The card is used quite often in the reservation system to read a free-running clock, set an interrupting timer, or cancel a timer. Since each of these operations requires multiple reads and stores to device registers and since the device driver for the card contains many delay loops required to synchronize properly with the card, much time is wasted. With clock and timer support from a better card, the scheduling cost should be significantly lower.

6.3.2 Individual operations

This section presents measurements of the individual internal operations used by the reservation mechanism. These measurements can be used to project scheduling costs for task sets.

6.3.2.1 Reserve Switch

During a context switch, the system must switch the reserve to which it is charging computation time as it switches the thread that is running on the processor. This involves updat-

ing usage accumulators in the old reserve and possibly setting the overrun timer for the next reserve. The four measured cases for the reserve switch are:

- Neither the old or new activity was reserved - Just update the usage accumulators.
- The old activity was reserved - Cancel the overrun timer for the old activity and update usage.
- The new activity is reserved - Set up and arm the overrun timer for the new activity and update usage
- Both activities are reserved - Cancel the old overrun timer, set up the new overrun timer, and update usage.

The following table gives the measurements for these cases.

Action	Duration
Neither old nor new activity reserved	23 μ s
Old activity reserved	100 μ s
New activity reserved	100 μ s
Both activities reserved	180 μ s

Table 6-18: Reserve Switch

6.3.2.2 Overrun and Replenishment Timers

The measured cost for handling an overrun timer includes identifying the timer, setting some state in the reserve, and initiating a context switch. It does not include the cost of the context switch itself.

The cost for handling a replenishment timer includes identifying the timer, setting some state in the reserve, resetting the timer for the next reservation period boundary, and possibly resetting the overrun timer. Four different cases for replenishment timer handling were measured:

- A reserve with a reservation whose computation allocation had been depleted during the period (the overrun timer had expired for this activity), and it was waiting for a new allocation.
- A reserve with a reservation whose computation allocation had not been depleted.
- A reserve with no reservation whose replenishment timer expired while it was running.
- A reserve with no reservation which was not running at the time the replenishment timer expired.

The following table shows the measurements for the overrun timer and each of the replenishment timer cases.

Action	Duration
Handle overrun timer	130 μ s
Reserved and waiting for new allocation	170 μ s
Reserved but not waiting	140 μ s
Unreserved and running when timer expired	140 μ s
Unreserved and not running when timer expired	140 μ s

Table 6-19: Replenishment Timer

6.3.2.3 Usage checkpoints

This section gives measurements of the system primitives that extract usage information from the kernel. The reserve usage data from the reservation system implemented in RT-Mach come in two forms: the current accumulated usage of a reserve, and the accumulated usage as of the last reservation period boundary. The record of the accumulated usage of a reserve taken at a reservation period boundary is called a checkpoint. The reservation system offers two system primitives for retrieving usage data: the first gives the data for a single checkpoint along with the current accumulated usage, and the second give the last 20 checkpoints from the last 20 reservation period boundaries. The following table gives the measured costs for both of these system primitives.

Action	Duration
Retrieve single checkpoint	130 μ s
Retrieve 20 recent checkpoints	220 μ s

Table 6-20: Checkpoint Cost

6.3.2.4 Analysis

The measurements described in this section can be used to estimate the cost associated with running specific task sets on the reservation system. The replenishment timer costs occur in every period of every reserved activity, and these numbers can help project the impact of having many tasks with small reservation period. The overrun timer costs detail the penalty associated with a program whose computation does not closely match its reservation. The checkpoint costs can be used in estimating the overhead for a monitor and in choosing timing properties of a monitor to balance accuracy of information with overhead cost.

6.4 Chapter summary

This chapter addressed the questions of whether the reservation system supports predictable application behavior and how much the predictability costs in terms of scheduling overhead. To show that the reservation system supports predictable behavior for applications, several experiments were done using task sets consisting of independent compute-bound synthetic workloads as well as client/server task sets with synthetic workloads. In an experiment with three reserved programs and five unreserved competitors, the reserved programs achieved measured processor utilizations that had 5-percentiles and 95-percentiles within 3-7% of their average utilizations, indicating that they were able to get their processor reservations with very little variance in their computation times. In the client/server experiments, even a case where an unreserved client was competing with two reserved clients for the same server (along with other independent unreserved programs competing for the processor), the reserved clients were able to achieve their timing constraints. The 5-percentiles and 95-percentiles for the two reserved clients in this case were within 5% of their average utilizations. These experiments showed that the reservation system guarantees very tight distributions of processor utilization even with different types of computation and with different combinations of client/server interactions.

Additional experiments used a reserved QuickTime video player and a modified version of the X Server to show that reserved applications can coordinate with shared servers to satisfy timing constraints on computations such as displaying video frames. In these experiments, the reserved players (even with interference from competing non-real-time X clients) had processor utilization measurements with 5-percentiles and 95-percentiles within 65% of their average utilizations. The utilization distributions were not as tight as the synthetic client/servers because the internal structure of the X Server is not ideally suited to reserve propagation and charging to clients' reserves. However, the performance of the reserved players was still much improved over that of unreserved players which had measurements with 5-percentiles and 95-percentiles that were as much as 166% of their average utilizations.

Experiments with libsockets showed that with an appropriate protocol processing structure, packet senders and receivers could achieve predictable behavior. The reserved senders in the experiments had processor utilization measurements with 5-percentiles and 95-percentiles that were within 7% of the average utilizations, and the reserved receivers had 5- and 95-percentiles that were within 16% of the average utilizations. This is compared with unreserved senders and receivers that had 5- and 95-percentiles of up to 360% of their average utilizations.

The scheduling costs of the reservation system were measured by running a periodic thread and measuring the idle time left over to find the scheduling cost. The results from several experiments with reserved and unreserved periodic threads with different periods show that the scheduling cost of reserved threads is typically about twice that of unreserved threads.

Chapter 7

Related Work

The work presented in this dissertation draws on research in several different areas. The reserve model depends on theoretical results from real-time scheduling, and the design and implementation were influenced by the work in real-time system design as well as the requirements for multimedia applications. This chapter presents an overview of related work: most of the related work focuses on systems issues although a section on applications discusses work on tools and application adaptation techniques.

7.1 System Implementation

The increasing integration of computer and telecommunications technologies has focused attention on the real-time issues that arise in processing digital audio and video. Handling multimedia data streams requires an understanding of the timing requirements, encoding techniques, and data formats of the new media types [13,69,94,135,136] as well as programming interfaces [60,101] and new application design techniques [35].

7.1.1 Multimedia support

A great deal of recent work has focused on how software systems (including operating systems) can be designed to support multimedia applications. Many researchers and practitioners consider resource reservation desirable if not absolutely necessary for real-time multimedia operating systems [3,46,53,60,87,102,127]. Herrtwich [42] gives an argument for resource reservation and careful scheduling in these systems. Others prefer a best effort approach to OS design for multimedia applications [19,21,90]. Recent survey papers [116,132] and books [12,115] discuss work in this area.

One of the simplest ways to do resource allocation for multimedia applications is to dedicate an entire machine to a single multimedia application. Multimedia applications for single-user personal computers typically assume that only a single multimedia activity will exist at any one time. Or if several multimedia activities exist, the assumption is that they

will be associated with a single application that can do cooperative scheduling of these activities. Adobe Premiere [95] is an example of such an application. If this assumption is violated, neither the applications nor the system will be in a position to assert anything about the behavior of multimedia applications. There is no admission control or any kind of overload protection in such systems.

Other systems such as OS/2 [60] put limits on the number of high-level activities, such as video streams being processed by the system, as a primitive form of admission control. This technique does not address the problem of resource allocation or admission control for arbitrary computations and media processing applications.

Jeffay *et al.* implemented an operating system designed for guaranteed real-time scheduling [53]. A video capture and playback application demonstrates that the analytical techniques can be successfully applied to real applications. In subsequent work, Jeffay has focused attention on transport mechanisms to detect and deal with variability in network behavior [54]. The reservation system described in this dissertation uses real-time scheduling analysis as does Jeffay. It focuses on enforcement whereas Jeffay's recent work focuses more on flexibility. The reservation system would benefit greatly from the increased flexibility of having adaptive mechanisms such as Jeffay's at higher levels.

Anderson *et al.* [3] argue for introducing more sophisticated timing and scheduling features into operating systems, and their DASH system design supports a reservation model based on linear bounded arrival processes (LBAP) [22,23]. They implemented their system design and were able to report some preliminary experiences with the system. They use earliest deadline scheduling for real-time traffic because it is optimal in the sense that if any algorithm can schedule a particular collection of tasks, the earliest deadline algorithm can do it. Admission control for this system is based on a time-line where new jobs are admitted only if they fit onto the time-line when the job request arrives [129]. The reservation system described in this dissertation uses an admission control algorithm based on a periodic scheduling framework with scheduling analysis rather than a timeline approach. One way for reserves to accommodate one-shot events would be to use a timeline based admission control policy and scheduling algorithm. The reservation system focuses more on enforcement of specified computation times.

Hyden [46] considered the problem of supporting QOS in operating systems in his thesis. He implemented a system that offers a virtual processor interface to applications. A video decoder application demonstrates how such an interface can be used by an application. In contrast, the reserve system provides flexibility in reservation binding for a more integrated view of resource usage reservation, measurement, and enforcement.

Coulson *et al.* [21] present a system design based on Chorus [103]. This system uses earliest deadline scheduling, but they do not provide any admission control and usage enforcement. QOS commitments can be revoked, and overload is permitted; commitments are degraded as a response to overload. The work focuses primarily on fast context switching and reducing protection domain crossings. The reserve system provides guaranteed resource reservation using an enforcement mechanism with QOS policy modules layered over the reservation abstraction.

Jones [56] describes some ideas on system design for multimedia applications which depend on value functions as a means of scheduling processes based on timing constraints, semantic importance, user preferences, and other information about application-level requirements. The reserve system divides the QOS provision problem into a reservation mechanism and QOS policy layers. Jones' work focuses more on the QOS policy and could benefit from mechanisms for guaranteed resource reservation.

Many multimedia cards and co-processor boards and boxes are actually embedded systems with their own processors and operating systems. These systems must manage resources for multiple real-time activities that must be multiplexed.

Hopper [44] described Pandora's Box which is a transputer system designed to be a multimedia peripheral for a traditional workstation. The Pandora system employs several transputers, each of which handles a particular function in the system such as compression, decompression, network traffic, and audio. A similar approach is being pursued in the context of the Desk Area Network [40] and related operating system efforts [9,81]. The reserve system addresses the sharing of devices and software resources by appropriate admission control policies, scheduling algorithms, and enforcement mechanisms. Multiplexing of resources results in more efficient resource utilization.

The Mwave system [47] consists of a digital signal processor (DSP) card intended for use with a PC. The Mwave card handles various audio processing and telephony tasks, depending on the host processor for control functions. A programmer can develop applications that are divided between the host system and the Mwave processor. There is a programming environment that supports application development, and the operating system running on the Mwave processor provides some real-time support for scheduling and enforcement. In particular, the Mwave/OS performs enforcement functions, resetting the card when any of the real-time activities misses a deadline. The reserve abstraction provides more functionality for flexible binding of reserves, and the reserve system also has a more flexible policy for handling timing failures. In particular, real-time activities are not affected by the timing failure of an independent real-time or non-real-time activity.

7.1.2 QOS architectures

The reserve abstraction is designed to support higher-level architectures for managing quality of service specification and negotiation. Several QOS architectures have been proposed assuming that the mechanisms for operating system resource management, such as the reserve system, would be developed.

Nicolaou [89] described a QOS architecture suitable for programming distributed multimedia applications. His work describes an architecture that one might use to design and implement multimedia applications. While some systems issues like resource management and scheduling are identified as being important, the cited work does not address those problems. A prototype implementation based on the architecture demonstrates the feasibility of this approach, but since the prototype is implemented on UNIX, its performance suffers from a lack of real-time scheduling techniques in the operating system. The reserve

model is designed to provide the kind of operating system support necessary for Nicolaou's QOS architecture.

Wolf *et al.* [137] defined a QOS architecture for a communication transport system, and they have an initial implementation of their system, called the Heidelberg Transport System. The reserve system focuses more on admission control, dealing with interaction between processes (such as client/server interactions) and usage enforcement.

The QOS Broker [84] provides an architecture for handling QOS negotiation among resource "buyers" and "sellers." Protocols are provided for carrying out the negotiation, and a version of the QOS Broker was implemented and used in the context of a telerobotics application. The Broker contains hooks for reserving resources in operating systems and networks when reservation mechanisms are available. The reserve system described in this dissertation provides the mechanism necessary for negotiating guaranteed service.

7.1.3 Networking

The idea of bandwidth reservation for network quality of service models has been explored by a number of researchers. The thesis work complements the work on networking by providing resource reservation in the end hosts as well as in the network. Thus the operating system resource reservation work enables predictable end-to-end performance for real-time programs.

Ferrari and Verma [31] describe a model for guaranteeing bandwidth in a wide-area network. Their analytical model provided a basis for subsequent work on network bandwidth reservation. In contrast, Clark *et al.* [18] describe another service model based on the idea of predictive service. In related work, L. Zhang *et al.* [140] gave a basic description of RSVP, a protocol for reserving resources across nodes in an internetwork. This thesis provides operating system support to implement these types of schemes. Reserves support guaranteed service for hard real-time applications as well as supporting predictive service for soft real-time applications that dynamically change their requirements on various system resources.

H. Zhang [141] describes a bandwidth reservation model and an implementation in an internetwork protocol. He was able to allocate and control network bandwidth in gateways, but did not address the allocation and control of resources like the processor in the more general operating system environment. The reserve system would make it possible to address resource reservation issues in general end systems, extending H. Zhang's work on resource reservation within the network and its routers.

Anderson *et al.* [4] describe a Session Reservation Protocol (SRP) which reserves resources along the route of a connection to ensure a particular bandwidth and delay for the connection. This protocol provides resource reservation at routers for predictable network performance. The reserve system would make it possible to reserve resources at end hosts as well as in the network routers.

7.2 Scheduling theory

Real-time scheduling theory has been an active area of research for years. Many important theoretical results date back to the 1950's and 1960's [20,49], and work on system design dates back to the same time period [71,73,82]. More recently, real-time systems research has focused on scheduling algorithms and techniques for building complex, distributed real-time systems [1,130,131].

The reserve system was designed using several results from real-time scheduling theory in its admission control policy, scheduling algorithm, and synchronization and communication primitives. The reserve model is based on the original rate monotonic scheduling analysis [67] as well as recent extensions [63]. The simple two-parameter reservation model is suitable for a wide range of applications, but applications that require different reservation semantics might need a model that takes advantage of other scheduling algorithms and analyses. For example, an application might require generalized rate monotonic analysis [39,64,107,109], aperiodic servers with different replenishment policies [111,112,120], earliest deadline first scheduling [27,49,67], sporadic task scheduling [51], or deadline monotonic scheduling [7].

The reservation system also depends of priority inheritance protocols for fixed priority scheduling [96,108], and similar inheritance protocols for earliest deadline first [8.16] would be required for a reservation model based on that policy.

The two-parameter periodic scheduling framework of the original rate monotonic scheduling analysis divides the capacity of the processor among multiple tasks. Other scheduling techniques such as processor sharing and fair share scheduling aim to provide a similar kind of proportional sharing of the processor. The primary difference is that the two-parameter periodic scheduling framework specifies a granularity of sharing by specifying a period. Processor sharing and fair share scheduling seek to support sharing at a very fine granularity.

A processor sharing technique [50] intended to be accurate enough to accommodate the timing constraints of arbitrary multimedia applications would require a very small quantum. This would imply a high scheduling overhead. Such a system would also require some method for controlling the effects of synchronization and communication between processes.

Fair share schedulers [41,58,138] provide for resource allocation like processor sharing, although at a coarser granularity. Such schedulers ensure that users who pay more for compute time get better service than others who pay less do. They record usage measurements to try to match usage with target allocation levels over the long term.

More recently, work on fair share scheduling and proportional share scheduling has addressed the integration of network scheduling and end-system scheduling [118,133,134]. Instead of focusing on ensuring that a certain amount of computation time will be available by a deadline, this work focuses on ensuring that a certain proportion of the processor is available to a compute-bound task in any interval.

7.3 Applications

Two classes of applications turn out to be very important to the resource reservation work. Adaptive applications are important because dynamic real-time applications must be very sensitive to the relationship between their (changing) resource requirements and their levels of resource reservations. In addition, design tools, performance monitors, and dynamic resource allocation tools are necessary for the design and on-line monitoring and tuning of resource reservations for dynamic real-time applications.

7.3.1 Adaptive applications

Recently there has been a focus on how systems and applications can adapt their computations to the resources they find available to them. For example, video compression algorithms are designed to allow for tradeoffs in resource requirements in various ways. Software techniques are also being developed, primarily in the area of mobile computing, for application awareness of resource requirements and adaptation based on system resources available.

The MPEG compression algorithms support several ways to trade off between bandwidth, computational resources, and image quality [17,61]. The MPEG-1 q-factor trades image quality for less bandwidth and computational resources, and the MPEG-2 hierarchical encoding scheme supports incremental improvements in image quality for additional bandwidth and computation time [10,17]. Other methods such as subband encoding [121], Hyden's method [46], and other hierarchical encoding schemes [2,17] support this tradeoff as well.

Recent work in mobile computing explores how applications can be sensitive to the resources that are available to them in terms of network bandwidth, processor power, and screen resolution among others [32,91,106]. These applications discover the resources that are available to them and then use this information to guide their own computations. For example, a video player residing on a high-end workstation might request from a video server a full color (24 bits per pixel), full motion (30 frames per second), relatively high resolution (640x480 pixels) video stream. The same video player on a low-end personal computer might request only 256 bits of color, 15 frames per second, and a resolution of 160x120 pixels. Supporting a scenario like this requires that all of the involved system components are aware of the resources they have available, the resources they need to do their work, and the level at which all of the other components in the end-to-end activity can perform.

7.3.2 Tools

The reserve system provides a mechanism for tools that monitor and control resource usage. This section discusses current tools for monitoring functions and for controlling resource usage.

Different tools intended for monitoring various aspects of system performance work at different levels. Some are intended for program design while others are intended for system-level debugging and on-line resource monitoring.

Performance analysis tools such as gprof [29,37] and PCA [28] use PC sampling techniques to characterize program runtime behavior and object code to determine the static structure. This method gives a great deal of insight into the behavior of individual programs, but there is no notion of tuning task sets as a whole. Also, the method of exercising control of the programs being analyzed is through the programs themselves, either changing their structure or modifying their parameters. The tool does not exercise this control directly.

System monitoring tools typically separate the functions of capturing performance data, analyzing the data, and having an effect on the system that was measured. The Advanced Real-Time Monitor (ARM) [123], which was originally designed for the ARTS Kernel [124] and more recently updated for RT-Mach [125], takes this approach. ARM uses a kernel mechanism to capture scheduling events and then sends those events over the network to an ARM application running on a different machine. ARM then displays a scheduling history based on the events, and this history can be viewed, analyzed, and saved for future use.

Tools like xload [74] provide a very simple view of the cumulative processor load on a workstation. The xload application does this on-line, but it leaves out some interesting information like a breakdown of which processes are consuming what percent of the load. It also lacks a control element to help the user have an effect on the load through the tool.

Tools like the Memory Sizer on the Macintosh [6] offer control over a system resource, but the resource information is very simple, and one cannot set the memory size of a program while it is running. The reserve system allows for much more sophisticated control of system resource allocation. With the help of a QOS manager, a tool such as rmon can graphically display resource usage information and interact with the console user to control resource allocation.

Chapter 8

Conclusion

This dissertation has presented a comprehensive model describing resource reserves, the operations they support, the scheduling algorithm and enforcement mechanism required, and how reservations for various resource types can be encapsulated in a single framework. An implementation and experimental evaluation demonstrate that real applications with non-trivial client/server interactions can achieve predictable real-time performance using resource reserves. The reserves ensure this predictability even when there are multiple real-time and non-real-time applications competing for the same resources.

This work shows that system mechanisms that address entire activities are important for real-time resource management. Furthermore, it shows that enforcement is essential, otherwise a reservation abstraction has no meaning. Programming techniques such as software pipeline architectures with synchronized periods and deadlines are useful for achieving predictable behavior. The following sections detail the contributions of this work and directions that this work opens for future research.

8.1 Contributions

The contributions of this work include the abstraction for operating system resource reservation, its implementation, real applications which use it, and an experimental evaluation of those applications. The following sections discuss these in more detail.

8.1.1 Resource reservation abstraction

The resource reserve abstraction provides a model for how real-time scheduling algorithms and analyses can be incorporated in an operating system design in an integrated manner. This is in contrast to the specification of scheduling algorithms and analyses in the context of a simplified task model where many practical systems issues and programming issues are ignored. Two key features of the abstraction are flexible binding of reserves to threads and enforcement of reservation parameters.

Since reserves are first class objects in the system rather than being tightly and permanently bound to threads (or processes), the management of resources is much easier. For example, reservation parameters can be allocated for a reserve by a thread and then a reference to the reserve can be passed to system service providers invoked by the thread. By allowing the binding of reserves to threads to be flexible, reserves can be passed around in this way, and the resource usage for the abstract activity is tracked and guaranteed throughout all of the server calls.

The enforcement mechanism eases program development and debugging for programmers of hard and soft real-time applications. Programmers of hard real-time applications can exploit the usage accumulation mechanism to measure the requirements of their code during development. During runtime, the enforcement mechanism and usage measurements can be used to isolate timing bugs. Programmers of soft real-time applications can use the enforcement mechanism to ensure isolation between applications and to provide information on resource usage requirements for adaptive applications. This relieves the programmer of doing the exhaustive measurements and analysis usually required to achieve real-time predictability.

8.1.2 Implementation

The implementation of processor reserves in Real-Time Mach and the implementation of several real applications that use reserves demonstrate the feasibility of the approach described in this document. The implementation shows how to design an enforcement mechanism and integrate it with the scheduling policy. It shows how a reserve propagation mechanism can be built to ensure consistent resource reservation and account for abstract activities that span multiple threads (or processes). It also demonstrates how QOS managers can be incorporated into the system to negotiate reservation parameters between reserved applications and the operating system, and how resource usage monitor and control can be used to promote awareness of resource requirements and dynamic adjustment of resource allocation.

8.1.3 Experimental evaluation

The experimental evaluation demonstrates that the applications using the reservation system can achieve predictable behavior with acceptable overhead costs. Experiments with synthetic benchmark applications were able to achieve very consistent real-time performance even in the face of competition from other real-time and non-real-time applications. In the experiments, periodic reserved applications ran over a relatively long duration of time, and measurements of the processor utilization during each period were recorded. The 5-percentile and 95-percentile numbers for these measurements were typically within 5-7% of the average utilization across all the periods, indicating a very tight distribution of processor utilization measurements across periods and a quite consistent pattern of real-time behavior. In other experiments with real applications such as a video player and X Server, processor utilization measurements yielded 5- and 95-percentiles which differed from the average utilization by up to 65%. This is not nearly as tight as the synthetic client/server benchmark applications due to the input/output organization of the X Server which is not

ideally suited to reserve propagation techniques. In any case, the behavior of reserved X clients was much better than that of unreserved X clients, which had 5- and 95-percentiles that were as much as 166% of their average utilizations. Experiments with network transmit/receive applications showed processor utilization measurements with 5- and 95-percentiles of 7-16% of the average utilizations. When unreserved, these applications had measurements with 5- and 95-percentiles of up to 360% of their average utilizations. In each case, the reserved application showed much more consistent real-time behavior than its unreserved counterpart.

8.2 Future directions

The work described in this document opens up many avenues for future research. By providing a general framework and testbed platform for operating system resource reservation, this work provides a concrete context for many research topics such as QOS provision and negotiation, resource allocation algorithms, and adaptive application programming techniques.

This work on resource reservation also provides a design point for comparison with other system design approaches [65] and for other scheduling paradigms [119]. The idea of enforced resource reservation can also be used as a building block for exploring higher-level concerns such as the role of user in resource management [57].

Bibliography

- [1] A. K. Agrawala, K. D. Gordon, and P. Hwang, editors. *Mission Critical Operating Systems*. IOS Press, Amsterdam, 1992.
- [2] D. Anastassiou. Digital Television. *Proceedings of the IEEE*, 82(4):510–519, April 1994.
- [3] D. P. Anderson, S. Tzou, R. Wahbe, R. Govindan, and M. Andrews. Support for Continuous Media in the DASH System. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 54–61, May 1990.
- [4] D. P. Anderson, R. G. Herrtwich, and C. Schaefer. SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet. Technical Report TR-90-006, International Computer Science Institute. February 1990.
- [5] D. P. Anderson and R. Kuivila. A System for Computer Music Performance. *ACM Transactions on Computer Systems*, 8(1), February 1990.
- [6] Apple Computer, Inc. *Macintosh User's Guide*, 1991.
- [7] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, May 1991.
- [8] T. P. Baker. Stack-based Scheduling of Real Time Processes. *The Journal of Real-Time Systems*, 3(1):67–99, March 1991.
- [9] P. Barham, M. Hayter, D. McAuley, and I. Pratt. Devices on the Desk Area Network. *IEEE Journal on Selected Areas in Communications*, 13(4):722–732, May 1995.
- [10] S. Baron and W. Robin Wilson. MPEG Overview. *SMPTE Journal*, 6(103):391–394, June 1994.
- [11] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming under Mach*. Addison-Wesley, 1993.
- [12] J. F. K. Buford, editor. *Multimedia Systems*. ACM Press and Addison-Wesley, 1994.

- [13] J. Burger. *The Desktop Multimedia Bible*. Addison-Wesley, 1993.
- [14] A. Campbell, G. Coulson, and D. Hutchison. A Quality of Service Architecture. *Computer Communication Review*, 24(2):6–27, April 1994.
- [15] K. Chen. A Study on the Timeliness Property in Real-Time Systems. *The Journal of Real-Time Systems*, 3(3):247–273, September 1991.
- [16] M.-I. Chen and K.-J. Lin. A Priority Ceiling Protocol for Multiple-Instance Resources. In *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, pages 140–149, December 1991.
- [17] T. Chiang and D. Anastassiou. Hierarchical Coding of Digital Television. *IEEE Communications Magazine*, 5(32):38–45, May 1994.
- [18] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 14–26, October 1992.
- [19] C. L. Compton and D. L. Tennenhouse. Collaborative Load Shedding for Media-Based Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 496–501, May 1994.
- [20] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, 1967.
- [21] G. Coulson, G. S. Blair, and P. Robin. Micro-kernel Support for Continuous Media in Distributed Systems. *Computer Networks and ISDN Systems*, 26(10):1323–1341, July 1994.
- [22] R. L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.
- [23] R. L. Cruz. A Calculus for Network Delay, Part II: Network Analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.
- [24] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.
- [25] R. B. Dannenberg. A Real Time Scheduler/Dispatcher. In *Proceedings of the International Computer Music Conference*, pages 239–242, 1988.
- [26] P. Dasgupta et al. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, 3(1):11–45, Winter 1990.
- [27] M. L. Dertouzos. Control Robotics: The Procedural Control of Physical Processes. In *Proceedings of the IFIP Congress*, pages 807–813, August 1974.
- [28] Digital Equipment Corporation. *VAX Performance and Coverage Analyzer User's Reference Manual*.
- [29] J. Fenlason and R. Stallman. *gprof: The GNU Profiler*. Free Software Foundation, Inc., 1988.

- [30] D. Ferrari. Client Requirements for Real-Time Communication Services. *IEEE Communications Magazine*, 28(11):65–72, November 1990.
- [31] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communication*, 8(3):368–379, April 1990.
- [32] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.
- [33] G. Gallassi, G. Rigolio, and L. Verri. Resource Management and Dimensioning in ATM Networks. *IEEE Network Magazine*, 4(3), May 1990.
- [34] J. Gettys, P. L. Karlton, and S. McGregor. The X Window System, Version 11. *Software – Practice and Experience*, 20(S2):S2/35–S2/67, October 1990.
- [35] S. J. Gibbs and D. C. Tsichritzis. *Multimedia Programming: Objects, Environments and Frameworks*. ACM Press and Addison-Wesley, 1995.
- [36] D. Golub, R. W. Dean, A. Forin, and R. F. Rashid. Unix as an Application Program. In *Proceedings of Summer 1990 USENIX Conference*, June 1990.
- [37] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, June 1982.
- [38] Rhan Ha and J. W. S. Liu. Validating Timing Constraints in Multiprocessor and Distributed Real-Time Systems. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, June 1994.
- [39] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing Analysis for Fixed-Priority Scheduling of Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, January 1994.
- [40] M. Hayter and D. McAuley. The Desk Area Network. *ACM Operating Systems Review*, 25(4):14–21, October 1991.
- [41] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1858, October 1984.
- [42] R. G. Herrtwich. The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems. In A. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science. pages 279–284. Springer-Verlag, 1991.
- [43] P. Hood and V. Grover. Designing Real Time Systems in Ada. Technical Report 1123-1, SofTech, Inc., January 1986.
- [44] A. Hopper. Pandora - an experimental system for multimedia applications. *ACM Operating Systems Review*, 24(2):19–34, April 1990.

- [45] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [46] E. A. Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge, February 1994.
- [47] International Business Machines. *Mwave/OS User's Guide*, 1993.
- [48] Y. Ishikawa, H. Tokuda, and C. W. Mercer. Priority Inversion in Network Protocol Module. *Proceedings of 1989 National Conference of the Japan Society for Software Science and Technology*, October 1989.
- [49] J. R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. Technical Report Research Report 43, Management Science Research Project, UCLA, 1955.
- [50] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.
- [51] K. Jeffay. Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.
- [52] K. Jeffay, D. F. Stanat, and C. U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1991.
- [53] K. Jeffay, D. L. Stone, and F. D. Smith. Kernel Support for Live Digital Audio and Video. *Computer Communications (UK)*, 15(6):388–395, July-August 1992.
- [54] K. Jeffay, D. L. Stone, and F. D. Smith. Transport and Display Mechanisms for Multimedia Conferencing Across Packet-Switched Networks. *Computer Networks and ISDN Systems*, 26(10):1281–1304, July 1994.
- [55] E. D. Jensen, C. D. Locke, and H. Tokuda. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.
- [56] M. B. Jones. Adaptive Real-Time Resource Management Supporting Composition of Independently Authored Time-Critical Services. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 135–139, October 1993.
- [57] M. B. Jones et al. Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1995.
- [58] J. Kay and P. Lauder. A Fair Share Scheduler. *CACM*, 31(1):44–55, January 1988.

- [59] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC extension for Real-Time Mach. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 91–104, September 1993.
- [60] W. Lawton, B. Noe, and M. Lopez. *Developing Multimedia Applications Under OS/2*. John Wiley & Sons, 1995.
- [61] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *CACM*, 34(4):46–58, April 1991.
- [62] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [63] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.
- [64] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, December 1990.
- [65] I. Leslie et al. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, September 1996.
- [66] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.
- [67] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.
- [68] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4(1):37–53. March 1992.
- [69] A. Luther. *Digital Video in the PC Environment*. Intertext Publications and McGraw-Hill, 2nd edition, 1991.
- [70] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.
- [71] G. K. Manacher. Production and Stabilization of Real-Time Task Schedules. *JACM*, 14(3):439–465, July 1967.
- [72] C. Martel. Preemptive Scheduling with Release Times, Deadlines, and Due Times. *JACM*, 29(3), 1982.
- [73] J. Martin. *Programming Real-Time Computer Systems*. Prentice-Hall, 1965.
- [74] Massachusetts Institute of Technology. xload man page, 1988. X11 Window System, Release 4, Massachusetts Institute of Technology.

- [75] C. W. Mercer and H. Tokuda. An Evaluation of Priority Consistency in Protocol Architectures. In *Proceedings of the IEEE 16th Conference on Local Computer Networks*, pages 386–398, October 1991.
- [76] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Technical Report CMU-CS-93-157, School of Computer Science, Carnegie Mellon University, May 1993.
- [77] C. W. Mercer, J. Zelenka, and R. Rajkumar. On Predictable Operating System Protocol Processing. Technical Report CMU-CS-94-165, School of Computer Science, Carnegie Mellon University, May 1994.
- [78] C. W. 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 (ICMCS)*, pages 90–99, May 1994.
- [79] C. W. Mercer and R. Rajkumar. An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, May 1995.
- [80] D. Merusi. *Software Implementation Techniques: VMS, UNIX, OS/2, and MS-DOS*. Digital Press, 1992.
- [81] S. J. Mullender, I. M. Leslie, and D. McAuley. Operating-System Support for Distributed Multimedia. In *Proceedings of the Summer 1994 USENIX Conference*, pages 209–219, San Francisco, CA, June 1994.
- [82] R. R. Muntz and E. G. Coffman, Jr. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *JACM*, 17(2):324–338, 1970.
- [83] K. Nahrstedt and R. Steinmetz. Resource Management in Networked Multimedia Systems. *IEEE Computer*, 28(5):52–63, May 1995.
- [84] K. Nahrstedt and J. M. Smith. The QOS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.
- [85] T. Nakajima and H. Tokuda. Implementation of Scheduling Policies in Real-Time Mach. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 165–169, September 1992.
- [86] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 120–130, December 1993.
- [87] T. Nakajima and H. Tezuka. A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach. In *Proceedings of the Second ACM International Conference on Multimedia*, pages 289–297, October 1994.
- [88] C. Nicolaou. An Architecture for Real-Time Multimedia Communication Systems. *IEEE Journal on Selected Areas in Communications*, 8(3), April 1990.

- [89] C. A. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems*. PhD thesis, University of Cambridge, 1991. Available as University of Cambridge Computer Laboratory Technical Report No. 220.
- [90] J. Nieh and M. S. Lam. SMART: A Processor Scheduler for Multimedia Applications. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, page 233, December 1995.
- [91] B. D. Noble, M. Price, and M. Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. *Computing Systems*, 8(4), 1995.
- [92] S. Oikawa and H. Tokuda. Efficient Timing Management for User-Level Real-Time Threads. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 27–32, May 1995.
- [93] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a Software MPEG Video Decoder. In *Proceedings of the First ACM International Conference on Multimedia*, pages 75–82, August 1993.
- [94] K. C. Pohlmann. *Principles of Digital Audio*. McGraw-Hill, 3rd edition, 1995.
- [95] *Adobe Premiere for Macintosh*, 1993.
- [96] R. Rajkumar. *Task Synchronization in Real-Time Systems*. PhD thesis, Carnegie Mellon University, August 1989.
- [97] R. Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
- [98] E. P. Rathgeb. Modeling and Performance Comparison of Policing Mechanisms for ATM Networks. *IEEE Journal on Selected Areas in Communications*, 9(3):325–334, April 1991.
- [99] J. F. Ready. VRTX: A Real-Time Operating System for Embedded Microprocessor Applications. *IEEE Micro*, 6(4):8–17, August 1986.
- [100] R. R. Riesz and E. T. Klemmer. Subjective Evaluation of Delay and Echo Suppressors in Telephone Communications. *The Bell System Technical Journal*, 42, 1963.
- [101] S. Rimmer. *Multimedia Programming for Windows*. Windcrest/McGraw-Hill, 1994.
- [102] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, April 1995.
- [103] M. Rozier et al. Overview of the CHORUS Distributed Operating System. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69, April 1992.
- [104] T. G. Saponas and R. B. Demuth. The Distributed iRMX Operating System: A Real-Time Distributed System. In A. K. Agrawala, K. D. Gordon, and P. Hwang, editors. *Mission Critical Operating Systems*, chapter 16, pages 208–231. IOS Press, Amsterdam, 1992.

- [105] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: A Platform for Evaluating Real-Time Window System Designs. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, pages 342–352, December 1995.
- [106] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware Adaptation for Mobile Computing. *Operating Systems Review*, 29(1), January 1995.
- [107] L. Sha and J. B. Goodenough. Real-Time Scheduling Theory and Ada. *IEEE Computer*, 23(4):53–62, April 1990.
- [108] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
- [109] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.
- [110] J. A. Smith. The Multi-Threaded X Server. *The X Resource*, 1:73–89, 1992.
- [111] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *The Journal of Real-Time Systems*, 1(1):27–60, June 1989.
- [112] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1990.
- [113] J. A. Stankovic and K. Ramamritham. The Design of the Spring Kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1987.
- [114] J. A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10), October 1988.
- [115] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications, and Applications*. Prentice-Hall, 1995.
- [116] R. Steinmetz. Analyzing the Multimedia Operating System. *IEEE Multimedia*, 2(1):63–84, Spring 1995.
- [117] R. Steinmetz. Human Perception of Jitter and Media Synchronization. *IEEE Journal on Selected Areas in Communications*, 14(1), January 1996.
- [118] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation for Real-Time, Time-Shared Systems. Technical Report Technical Report 96-18. Old Dominion University, May 1996.
- [119] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the Duality between Resource Reservation and Proportional Share Resource Allocation. Technical Report Technical Report 96-19, Old Dominion University, May 1996.
- [120] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

- [121] D. S. Taubman. *Directionality and Scalability in Image and Video Compression*. PhD thesis, University of California at Berkeley, 1994.
- [122] K. W. Tindell, A. Burns, and A. J. Wellings. Mode Changes in Priority Pre-emptively Scheduled Systems. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 100–109, December 1992.
- [123] H. Tokuda, M. Kotera, and C. W. Mercer. A Real-Time Monitor for a Distributed Real-Time Operating System. In *Proceedings of ACM SIGOPS and SIGPLAN workshop on parallel and distributed debugging*, May 1988.
- [124] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.
- [125] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, October 1990.
- [126] H. Tokuda, Y. Tobe, S. T.-C. Chou, and J. M. F. Moura. Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network. In *Proceedings of the SIGCOMM '92 Symposium on Communications Architectures and Protocols*, pages 88–98. ACM, October 1992.
- [127] H. Tokuda and T. Kitayama. Dynamic QOS Control based on Real-Time Threads. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 113–122, November 1993.
- [128] D. Towsley. Providing Quality of Service in Packet Switched Networks. In *Performance Evaluation of Computer and Communication Systems: Joint Tutorial Papers of Performance '93 and Sigmetrics '93*, pages 560–586, 1993.
- [129] S. Tzou, 1993. Personal Communication.
- [130] A. M. van Tilborg and G. M. Koob, editors. *Foundations of Real-Time Computing: Formal Specifications and Methods*. Kluwer Academic Publishers, 1991.
- [131] A. M. van Tilborg and G. M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.
- [132] A. Vogel, B. Kerherve, G. von Bochmann, and J. Gecsei. Distributed Multimedia and QOS: A Survey. *IEEE Multimedia*, 2(2):10–19, Summer 1995.
- [133] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 1994.
- [134] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Report Technical Memorandum MIT/LCS/TM-528, Massachusetts Institute of Technology Laboratory for Computer Science, June 1995.

- [135] J. Watkinson. *The Art of Digital Audio*. Focal Press, 1988.
- [136] J. Watkinson. *The Art of Digital Video*. Boston: Focal Press, 2nd edition, 1994.
- [137] L. C. Wolf and R. G. Herrtwich. The System Architecture of the Heidelberg Transport System. *ACM Operating Systems Review*, 28(2):51–64, April 1994.
- [138] C. M. Woodside. Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers. *IEEE Transaction on Software Engineering*, SE-12(10):1041–1048, October 1986.
- [139] M. Yuhara, B. N. Bershad, C Maeda, and J. E. B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994.
- [140] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.
- [141] H. Zhang. *Service Disciplines for Packet-Switching Integrated-Services Networks*. PhD thesis, University of California at Berkeley, 1993.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science
and Technology to meet Air Force unique requirements for
Information Dominance and its transition to aerospace systems to
meet Air Force needs.*