On Predictable Operating System Protocol Processing

Clifford W. Mercer, Jim Zelenka, and Ragunathan Rajkumar

May 1994

CMU-CS-94-165

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported in part by a National Science Foundation Graduate Fellowship and by the U.S. Naval Ocean Systems Center under contract number N00014-91-J-4061. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NSF, NOSC, or the U.S. Government.

Keywords: distributed multimedia systems, protocol processing, resource reservation, operating systems, predictability, reservation, admission control, scheduling, performance measurement

Abstract

Distributed continuous media applications that incorporate digital audio and video require predictable response from the operating system and the network. Much recent research in communication networks focuses on providing predictable service at the network level, but current operating systems do not typically provide end-to-end predictability. Our aim is to offer operating system support for predictability while still maintaining the dynamic, easy-to-use environment of a general purpose operating system. We explore the use of a resource reservation mechanism in the operating system with the idea that this mechanism can be used to support higher level quality of service management policies. We also employ a protocol software structure that uses priorities, preemptive packet servicing, and reservation enforcement to allow the reservation scheduling policy to make fine-grained scheduling decisions to control the protocol processing activities. By reserving resources such as buffers, computation time for application-level activities, and computation time for protocol processing, our system can support predictable behavior for distributed real-time application programs. We implemented our reservation mechanism using the Real-Time Mach operating system as a base, and we present performance results which demonstrate that our approach provides predictable network service to distributed multimedia applications.

1. Introduction

Applications that transfer and process digital audio and video data in real-time must be designed for predictable behavior, and these programs also require predictable behavior from the operating system and network services that they use. Networking researchers are actively exploring ways to provide predictable network service [1, 2, 3, 21]. This work has been very successful; many networks have been designed to deliver predictable service to the host interface. The problem arises, however, when the operating system gets involved in the communication path. General purpose operating systems are not designed to move time-constrained data from the network interface to the user-level programs in a predictable way. Consequently, multimedia packets from the network may be subject to unpredictable delays before being delivered to the user, and multimedia packets generated by user applications may be delayed in the operating system before being transmitted to the network. Non-time-constrained data packets may also interfere with real-time computations that do not involve the network.

To provide predictable end-to-end performance that will enable user applications to depend on timely delivery of continuous media data, the operating system must be designed to cooperate with the network in supporting time constraints. There are many ways to design an operating system to do this. For example, the system can be static with a fixed schedule in the form of a timeline [12], it can use a schedulability analysis based on detailed measurements of the computation times and resource requirements of all system components [7], or it can reserve resources in order to isolate real-time activities from unpredictable interference from other activities. In our current work, we have chosen to explore the resource reservation approach, with special attention to how processor time can be reserved for processing network packets. We have developed a novel *reserve* abstraction which provides the operating system mechanism for reserving resources [14].

In this paper, we describe how we can use resource reserves and scheduled protocol processing along with a predictable low-level network service to provide predictable end-to-end behavior in a distributed multimedia system. We demonstrate these ideas which we implemented in the context of Real-Time Mach [20].

1.1. Scheduling 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 [17]. 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 [19].

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 [8]. This gives protocol processing higher priority than *any* schedulable activity in the system, higher than any system or user processes. 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. At the other extreme is the method used in the ARTS real-time kernel where the protocol processing software was structured using preemptible threads [18]. Each thread handled a different packet priority class, and the priority of the thread matched the priority of the packets it handled. 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.

Several principles guide the design of predictable protocol processing software [15]:

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 recently implemented for the Mach operating system [13] is amenable to 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.

1.2. Predictability in the operating system

To support a predictable network service, the operating system must cooperate with the network in scheduling networking activities. Two common approaches to predictable systems are: static real-time scheduling for guaranteed service and statistical multiplexing techniques for (mostly) good service and high utilization. Static real-time scheduling requires a fixed schedule or a fixed priority scheduling rule based on careful measurement of the execution times of each component in the system. This approach is less appropriate for the dynamic, flexible, easy-to-use environment that we want to support. 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. We 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 pretty straightforward proposition, we have concentrated on how a reservation mechanism for the processor would work. The reservation mechanism has four parts:

1. an interface to specify reservation requests,
2. an admission control policy,
3. a scheduling algorithm, and
4. a mechanism to enforce reservations.

In Section 3, we will discuss these issues in more detail to provide some background for our work on predictable protocol processing. A more complete description of the design and implementation of this reservation system can be found elsewhere [14].

1.3. The rest of this paper

Section 2 discusses how different protocol processing software structures can impact the scheduling of packet processing. In Section 3, we give a more detailed description of the processor reservation mechanism that we have implemented, focusing on the features of this mechanism that enable better control over packet scheduling. Section 4 gives the results of some performance experiments which demonstrate the predictable behavior we can achieve. In Section 5, we present some concluding remarks.

2. Protocol software structure

A predictable network service depends on how the protocol processing for network packets is handled as well as how these activities are scheduled. 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.

2.1. Software interrupt vs. preemptive threads

Traditionally, protocol processing software is 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 [8]. 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 [6, 15].

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 [18]. This is similar to the method used in the *x*-kernel [5], 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.

2.2. Mach 3.0 networking

Hard real-time systems such as the ARTS Kernel are built for static task sets which can be analyzed offline. These systems do not provide the dynamic, easy-to-use operating system environment for which we argue in Section 3. Mach 3.0 [16] and Real-Time Mach [20] provide a general purpose computing environment with some basic real-time capabilities. Networking support in Mach 3.0 is mainly provided by the UX server which implements the 4.3 BSD operating system personality on top of Mach 3.0 [4].

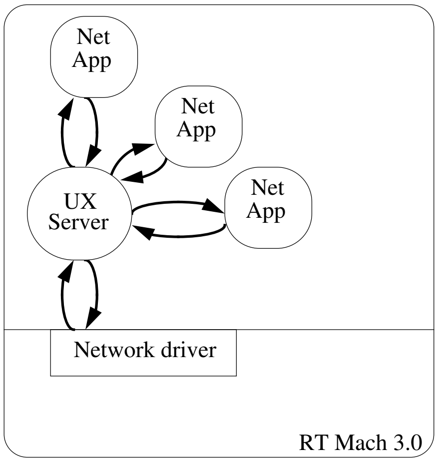


Figure 1: Networking with the UX Server

Networking in the context of the Mach 3.0 UX server 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 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 priority. In sum, this software does not satisfy our 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 [13]. Their library handles the protocol processing for sending and receiving packets and interacts with the network packet filter [22] 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 2 illustrates their networking software structure.

Maeda and Bershad report that their socket library yields much better performance in terms of throughput and delay than the UX server sockets implementation [13] . Coincidently, their 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 at the user's priority. It is also preemptible since it runs in user mode and shares nothing with other threads in other applications.

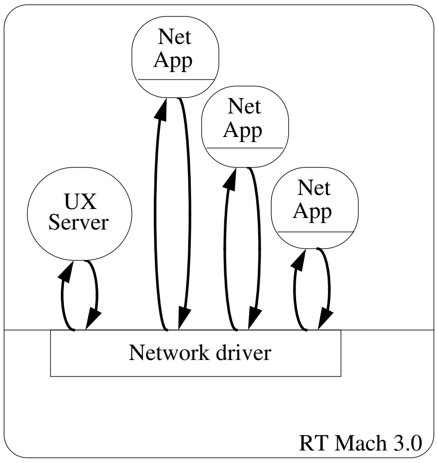


Figure 2: Networking with the Socket Library

2.3. 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, we can 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 tend 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, and 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 (to be described in Section 3) is free to make decisions about which applications should receive how much 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. We therefore expect very predictable networking behavior from applications that use the socket library with the reservation mechanism.

3. Reserves for operating systems

The goals of the reservation mechanism that we use to support predictable network service are to:

- schedule multiple real-time activities,
- handle non-real-time activities along with real-time activities,
- isolate real-time activities from interference from non-real-time activities,
- support dynamic program creation and termination,
- offer feedback to applications about timing performance, and
- allow dynamic adjustment of timing requirements.

There are many ways to attack this issue, including a direct approach where applications communicate to the system information about timing requirements, execution times of code segments, and user preferences for making load-shedding decisions. In this scenario, the system scheduler makes the appropriate scheduling decisions to satisfy the high level requirements and preferences. Making decisions based on a myriad of user preference rules is a very difficult problem, and the difficulty is exacerbated by the fact that the direct approach does not incorporate any structure to help organize the problem.

We believe that we can fruitfully apply the principle of separation of policy and mechanism [10] to the problem of managing operating system resources to ensure quality of service (QOS) requirements for distributed applications. We separate the mechanism for controlling resource consumption from the policy that determines how the resource capacity should be allocated, and we are left with the separate questions of how to deliver resource capacity to programs (the mechanism) and how to determine which programs should receive some measure of the capacity (the policy). To provide the mechanism to deliver resource capacity, we have introduced an operating system abstraction called a *reserve* which represents a reservation of capacity. For discrete resources like physical pages, reserves represent sets of the resources that have been allocated and dedicated to the reservation holder. For processors, reserves represent processing capacity.

Since processor capacity reserves present a more delicate problem than reserves for discrete resources, we focus on processor reserves. In our system, processor capacity requirements are specified by two numbers which indicate how much computation time is required per period of real time. For example, a program might be designed to execute for 3 ms every 10 ms, yielding a utilization of 30%. A capacity requirement of 30 ms every 100 ms results in the same utilization but very different timing requirements, so the additional information about the granularity of the requirement is important.

3.1. Admission control and scheduling

The processor capacity specification method described above is used to make reservation requests of the system. Programs may issue requests for reservations, and the system uses an admission control policy to decide whether an incoming request can be accepted or not. In our case, we base our admission control policy on the real-time scheduling analysis of Liu and Layland [11]. They showed that the behavior of periodic programs with bounded computation time and fixed period could be predicted by summing the utilization factors for all of the programs. In particular, they gave a schedulable bound for two different scheduling algorithms: rate monotonic scheduling and earliest deadline first. A schedulable bound is a utilization level below which the scheduling algorithm can guarantee that all programs will be able to complete their computations within the corresponding period, i.e. before the next period starts and a new computation begins. If the total utilization of a task set falls below the bound, the task set is guaranteed to be schedulable. If the utilization is greater than the bound, the task set may or may not be schedulable.

The schedulable bound for the rate monotonic scheduling algorithm is 69% although this is a pessimistic value for a pathological task set. Lechoczky *et al.* used an exact schedulability criterion to determine that for randomly generated task sets, the average schedulable bound is 88% [9]. This means that we can expect a task set with total utilization less than 88% to be schedulable. The schedulable bound for earliest deadline first is 100%, meaning that any task set that does not overload the processor can be scheduled.

The notion of schedulable bound is particularly useful for defining an admission control policy for the periodic scheduling framework we use in our reservation system. If we use rate monotonic scheduling, we can admit reservation requests that result in a total reserved utilization of less than 88%. For earliest deadline first scheduling, we would ideally be able to accept reservation requests summing to 100% of the processor, but in practice, it is probably a good idea to leave about 10% of capacity as a cushion against certain kinds system overhead that are difficult to reserve like cache miss penalties and clock interrupts. So a utilization bound of about 90% would be appropriate for use in the admission control policy when using either rate monotonic or earliest deadline first scheduling.

3.2. Enforcement

One of the assumptions in a periodic scheduling framework is that the computation time be bounded. A program must not exceed its stated computation time during any one period since this might interfere with other programs' reservations. A system may rely on programs to behave properly, or it may enforce this assumption by tracking the usage of programs and refusing to schedule a program that has consumed its computation time for a period. This issue of enforcement is a key point that separates hard real-time systems from general purpose operating systems. For hard real-time, system designers typically measure the execution characteristics of the programs and use those measurements to construct timelines or perform a scheduling analysis to determine whether the system is feasible. Since the task set is static and well-characterized at design time, no enforcement mechanism is required.

To make real-time scheduling policies accessible in a general purpose environment, we would like to avoid this tedious and difficult measurement and analysis process. We also want to avoid the rigidity of a statically defined task set. It must be possible to dynamically execute and terminate programs in a general purpose operating system, and we prefer an adaptive negotiation process between programs and the system to dynamically measure and adjust execution time and resource requirements.

In order to provide the additional usage measurement functionality and to isolate programs from interference by other programs, we use an enforcement mechanism. The mechanism measures the usage of each program and uses the program's reservation information to make sure that the usage does not exceed the reservation during any reservation period.

3.3. Quality of service using reserves

The reservation mechanism described above provides a foundation on which more sophisticated QOS policies can be implemented. Reserves give us an intermediate abstraction of resource capacity that handles the details of making low-level scheduling decisions and that isolates activities from interference. Thus QOS management need only be concerned with where to allocate resources without worrying about what scheduling decisions have to be made to allocate resources, whether activities will interfere with each other's timing, and how to enforce the QOS parameters that are in effect. We assume that such a QOS manager would coordinate resource reservation in the network as well as the resource reservation mechanism in the operating system.

To illustrate the workings of a QOS manager in relation to the reservation mechanism, consider a manager that coordinates QOS requests and negotiations among all the applications in a system (shown in Figure 3). Each application must be able to give information about the timing requirements of its computation and how long the computation takes. It may also provide minimum and maximum resource capacity requirements along with some indication of its importance to the user to help the reservation manager make decisions about how to allocate resources. A toolkit provided with the reservation manager could be used to measure the computation time required by the application. This information is then passed along to the reservation manager which will make a policy decision about whether the request should be granted.

4. Performance evaluation

We have implemented the processor capacity reservation mechanism described in Section 3 using Real-Time Mach version MK83 with the UX42 version of the UX server. For organizing the information

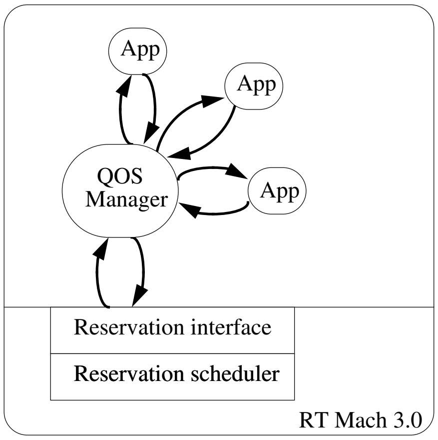


Figure 3: QOS Manager

associated with reservations, we introduced a new kernel abstraction call a *reserve*. In particular, our reserves are processor capacity reserves, and they contain information about processor capacity reservations that have been granted by the system. They also contain usage statistics which are used by the reservation system and which are available to the application. The usage numbers can be utilized by the application to learn about its execution behavior, computation time, etc. and to adapt its behavior if that is desirable.

Maeda and Bershad's socket library [13] is also available in the MK83/UX42 version of the system, and we use this library with our reserves to show how operating system networking can be predictable if the software is structured and scheduled properly.

Our tests use four different configurations of the RT Mach 3.0 system running on Gateway 2000 486 machines (66 MHz). We show the behavior of several task sets using UX server sockets with time-sharing scheduling and with reservation scheduling. We also show the behavior of the same task sets using the socket library with time-sharing scheduling and with reservation scheduling. These dimensions are illustrated in Figure 4.

Software Structure	Scheduling	
	Time-sharing	Reservation
UX Server		
Socket Library		

Figure 4: Test Case Dimensions

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), and 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. These task sets are summarized in Figure 5. The aim in structuring the synthetic benchmarks this way is that we can examine the interference caused by different types of competing activities, and we can also look at the interference that arises from an integrated task set with the various types of activities combined.

Task Set	Measured activity	Competitive activity
(a)	Predictable transmitter	No competition
(b)	Predictable transmitter	Arithmetic competition (compute-bound)
(c)	Predictable transmitter	Stdio competition
(d)	Predictable transmitter	Background networking competition
(e)	Predictable transmitter	Combined competition

Figure 5: Task Set Summary

In the following sections, we present measurements of the processor usage of our single predictable thread. We find that the different types of competition affect the behavior of this thread in different ways, depending on the competition and on the system configuration.

4.1. RT Mach 3.0/UX server with time-sharing scheduling

In this experiment, we use RT Mach 3.0 with the UX server providing the networking service to applications. The scheduling is Mach time-sharing. Figure 6 shows the configuration of the system and the structure of the task sets. The figure includes the combination of all types of competition, but different task sets include only portions of the competition. In the figure, the communication paths are represented by bold arrows.

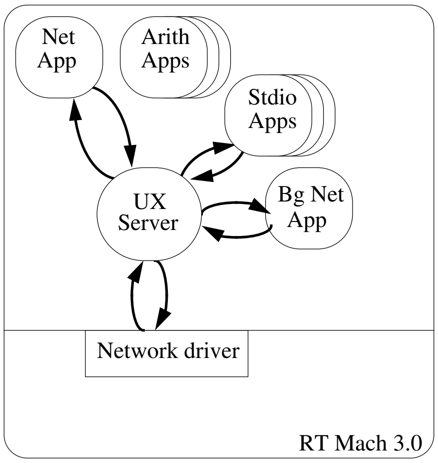


Figure 6: RT Mach 3.0/UX Test Case Structure

The task set definitions with respect to the components in the figure are as follows:

(a) Net App with no competition.

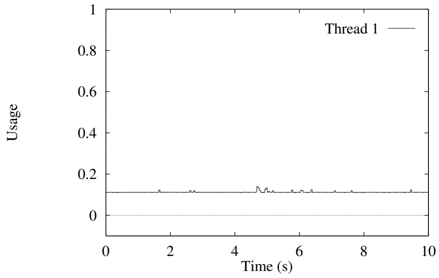
(b) Net App with Arith Apps in competition.

(c) Net App with Stdio Apps in competition.

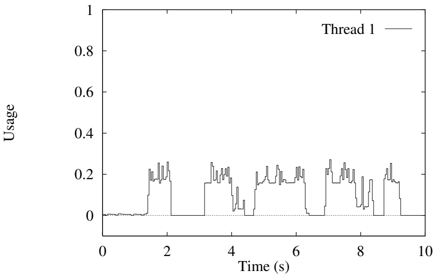
(d) Net App with Bg Net (background network) App in competition.

(e) Net App with all competitive components.

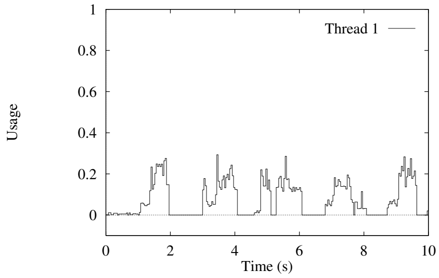
The same task set definitions will be used for the following three experiments as well.



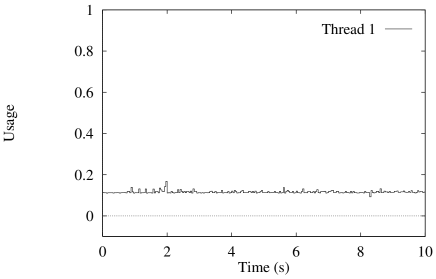
(a)



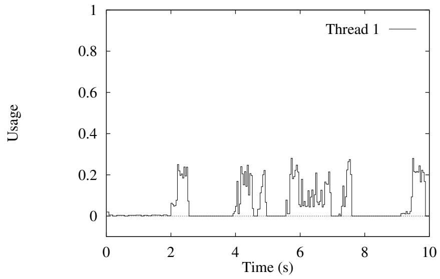
(b)



(c)



(d)



(e)

Figure 7: RT Mach 3.0/UX (Mach Time-sharing Scheduling) Measured Behavior

In Figure 7(a) we see the usage (percent of the processor) 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 Net App as well. This tends to delay service for the Net App, reducing the amount of time it has to do useful work.

4.2. RT Mach 3.0/UX server with reservation scheduling

The system structure for this experiment is the same as above (see Figure 6), the difference is the scheduling policy. We use reservation scheduling here instead of Mach time-sharing scheduling. The point is to demonstrate that simply using reservation scheduling does not solve the problem; the software structure is very important for predictable behavior.

The task definitions are the same as the previous experiment.

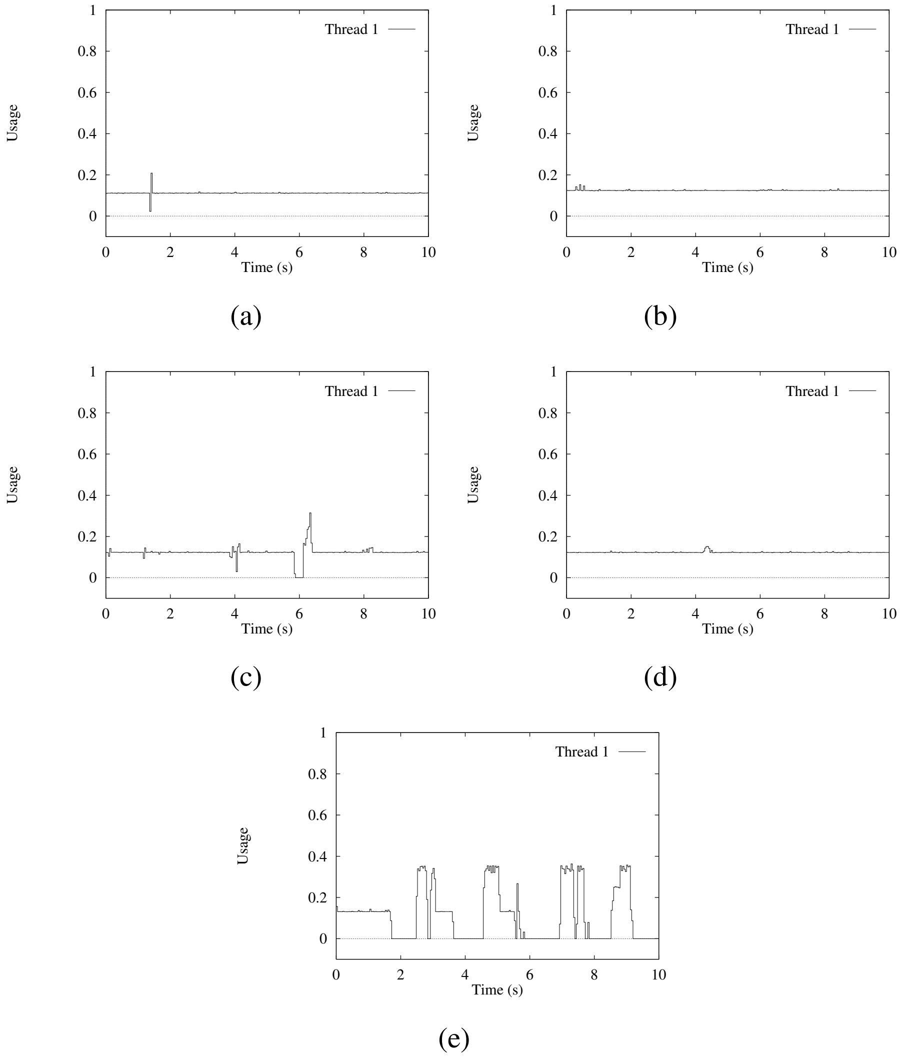


Figure 8: RT Mach 3.0/UX (Reservation Scheduling) Measured Behavior

Figure 8(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 8(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.

4.3. RT Mach 3.0/socket library with time-sharing scheduling

The tasks in this experiment use the socket library for networking. The task set is the same as in the previous two experiments except for the changes in communication pattern implied by the socket library. The new software structure appears in Figure 9 which shows that, when sending packets, network applications now communicate with the network device driver directly instead of through the UX server. The scheduling for this experiment is Mach time-sharing.

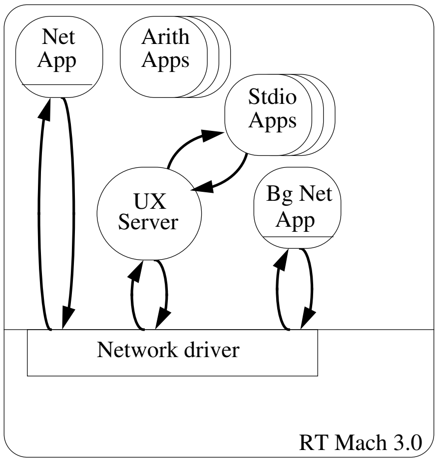


Figure 9: RT Mach 3.0/Socket Library Test Case Structure

Figure 10(a) shows the Net App in isolation. In parts (b) and (c), we can see that the Net App is sensitive to interference 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.

4.4. RT Mach 3.0/socket library with reservation scheduling

The final synthetic benchmark experiment uses the same task set as the others, it uses the socket library, and the scheduling policy is reservation scheduling. This is the system configuration which has all of the desirable features we discussed in Sections 2 and 3.

In Figure 11(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 competition alone, or background network competition alone. And in Figure 11(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

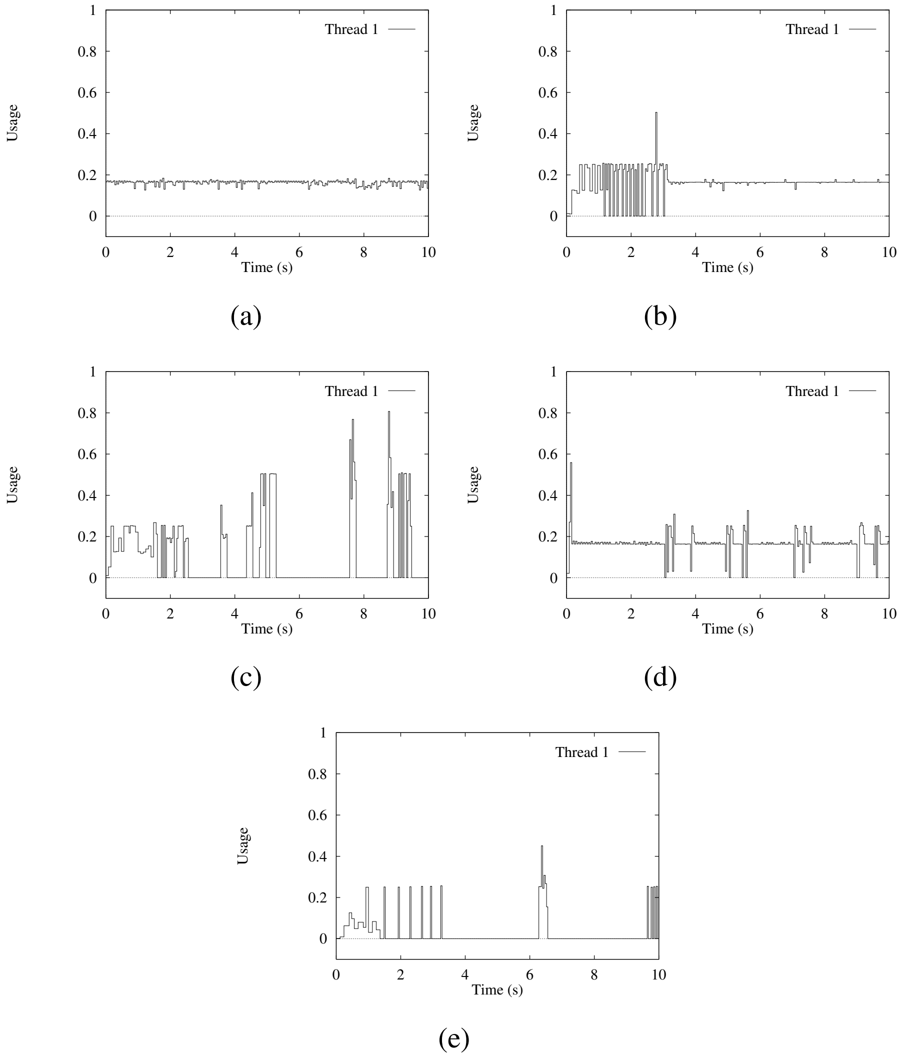


Figure 10: RT Mach 3.0/socket library (Mach Time-sharing Scheduling) Measured Behavior

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. Conclusion

We have described an operating system resource reservation model and an implementation of the model in Real-Time Mach 3.0. This reservation system addresses the need for the operating system coordination with predictable networks which is essential for end-to-end predictability in real-time multimedia applications. One of the two major components of the predictable end-to-end system is a protocol processing software structure that exhibits the features necessary for good real-time performance: priority scheduling and preemptibility. The other is a processor reservation and scheduling mechanism that incorporates timing constraints, usage measurement, and reservation enforcement. We measured the performance of the system for several different task sets, and these numbers demonstrate that our approach provides predictable behavior in a general-purpose distributed operating system.

While the numbers for these experiments are quite promising, we continue to evaluate the perfor-

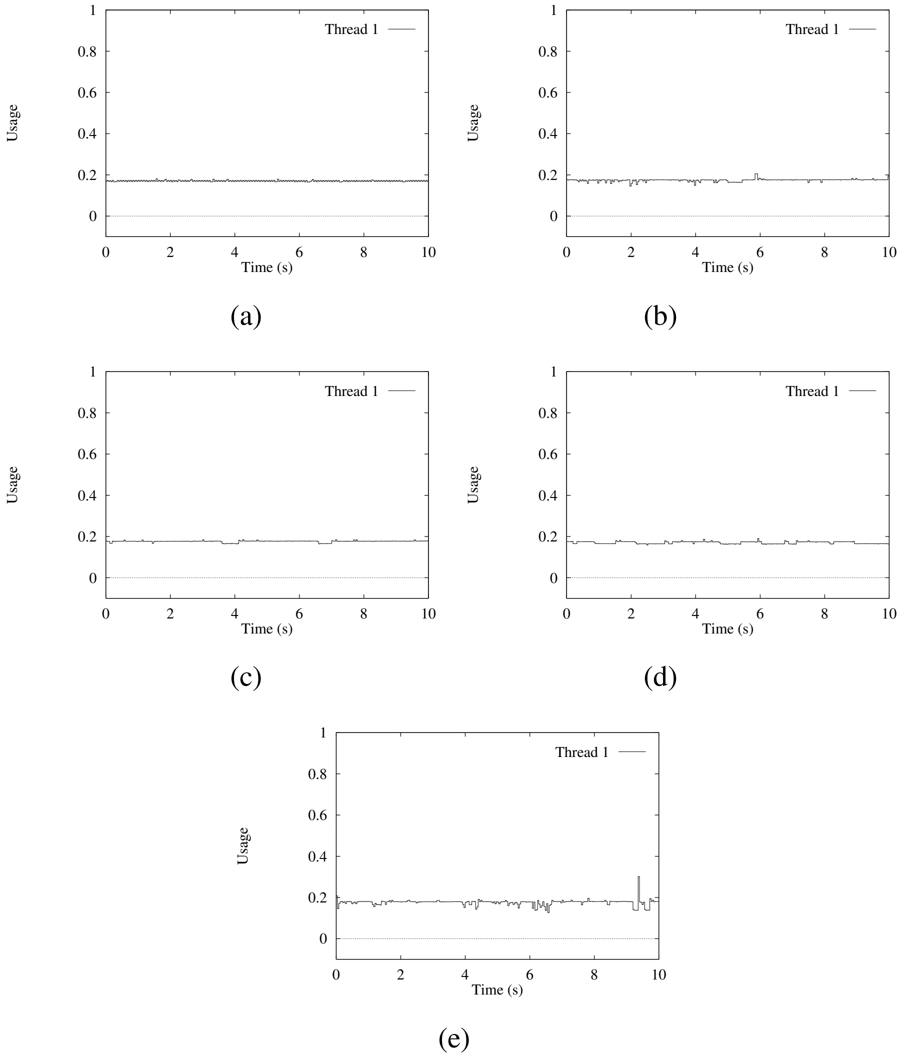


Figure 11: RT Mach 3.0/socket library (Reservation Scheduling) Measured Behavior

mance of our resource reservation approach in different kinds of network load scenarios, with other protocol implementations, and in other areas of the operating system including the paging system and the window system.

Acknowledgements

The authors would like to thank E. N. Elnozahy, M. Satyanarayanan, and Peter Steenkiste for their comments and suggestions, and Chris Maeda for help with using his socket library. Thanks also to Stefan Savage for his effort on the initial Real-Time Mach implementation of processor capacity reserves.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism Separation in HYDRA. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[11] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *JACM*, 20(1):46–61, 1973.

[12] 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.

[13] 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.

[14] 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. To appear.

[15] 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.

[16] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: A Foundation for Open Systems. In *Proceedings ofthe Second Workshop on Workstation Operating Systems (WWOS-II)*, pages 109–113, September 1989.

[17] 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.

[18] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.

[19] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok. Priority Inversions in Real-Time Communication. In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December 1989.

[20] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, October 1990.

[21] C. Topolcic. Experimental Internet Stream Protocol, Version 2 (ST-II). Internet Request for Comments, RFC 1190, October 1990.

[22] 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.