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.

[1]This 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 priorities[2]. 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 additoin, 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.

[2]This 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.

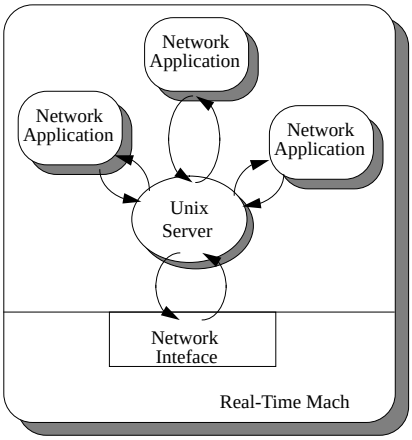


Figure 3-1: Networking with the Unix Server on RT-Mach

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.

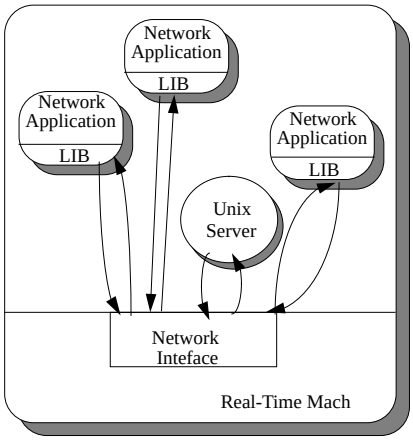


Figure 3-2: Networking with the Socket Library

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 scheduling) 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 libary 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 threads[4] 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).

---

[3]The 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-server`s 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.

[4]In 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 affectead 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.

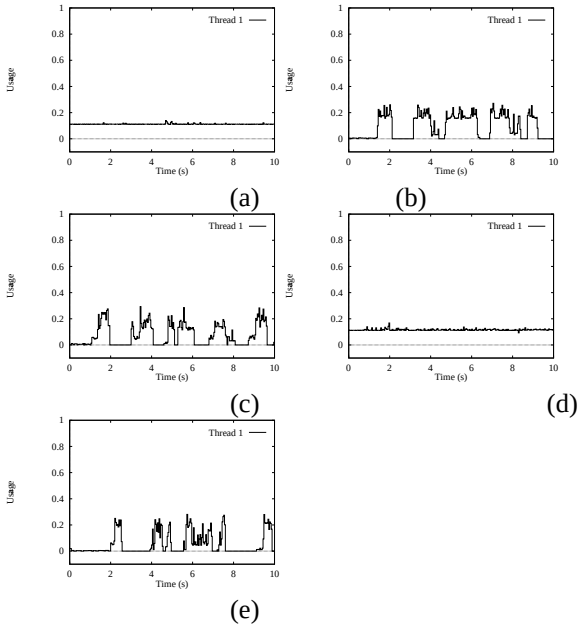


Figure 4-1: Measured Behavior under RT-Mach with Unix sockets, Mach Time-Sharing Policy

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-

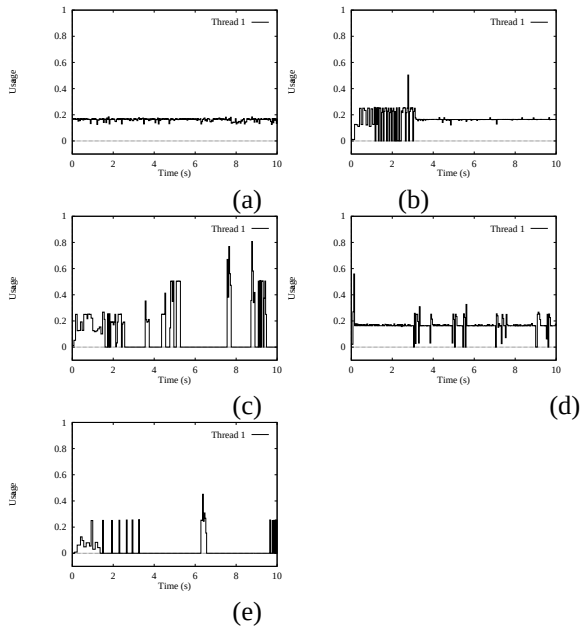


Figure 4-2: Measured Behavior under RT-Mach with `libsockets-rt` and Mach Time-Sharing

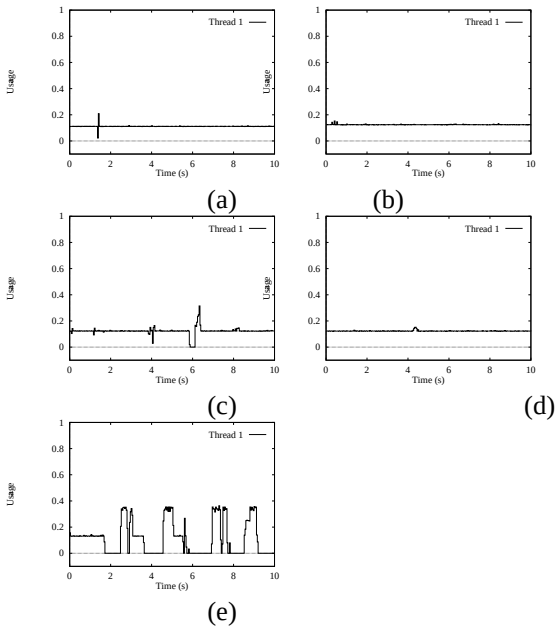


Figure 4-3: Measured Behavior under RT-Mach with UX sockets, RT-Mach Reservations

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.

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.

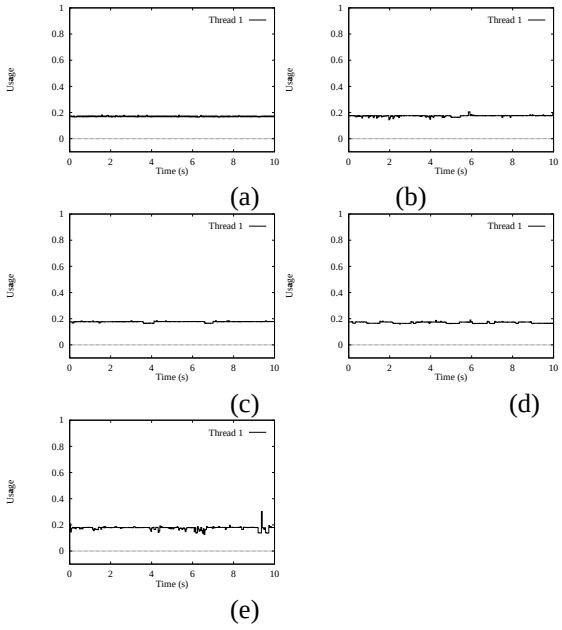


Figure 4-4: Measured Behavior under RT-Mach with `libsockets-rt`, RT-Mach Reservations

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 μsecond. 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.

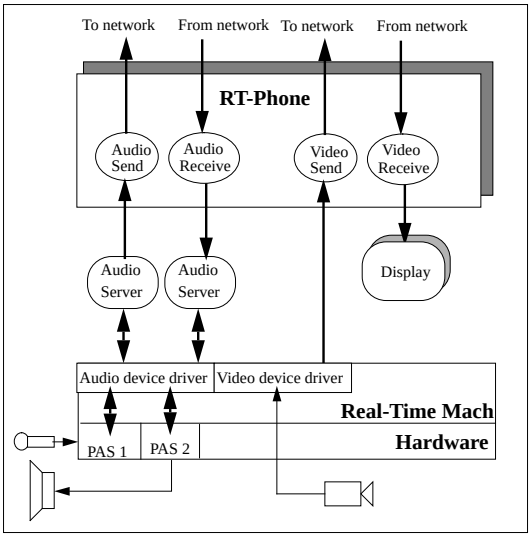


Figure 5-1: RT-Phone Video-Conferencing Support

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.

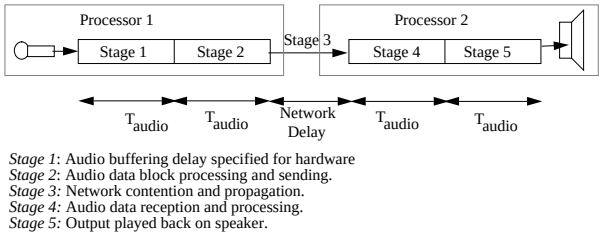
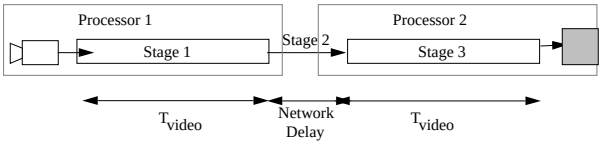


Figure 5-2: The Audio Processing Pipeline

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 $3T_{audio} + d_{network\_audio}$, 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.



Stage 1: Video frame grab, processing and sending.
Stage 2: Network contention and propagation.
Stage 3: Video frame reception and display.

Figure 5-3: The Video Processing Pipeline

Hence, the worst-case end-to-end delay for each video stream is given by $2T_{video} + d_{network\_video}$, where $d_{network\_video}$ 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 `libsockets-rt`		With `libsockets-rt`	
		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 $3T_{audio}$ 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 `libsockets-rt`		With `libsockets-rt`	
		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 latency[5] 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.

[5]The 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. .

Table of Contents

List of Figures

List of Tables