# Chapter 5

# Implementation

This chapter describes an implementation of processor reserves done using the Real-Time Mach operating system. It discusses applications that were modified to use processor reserves, network protocol processing software modified to use reserves, a QOS manager for negotiating resource allocation with applications, and tools for reserve monitoring.

## 5.1 Overview

This chapter describes the implementation of processor reserves in RT-Mach as well as several other components of the system. It also covers some applications that were modified or designed and implemented to use processor reserves. Figure 5-1 shows the various components and gives an indication of their relationship.

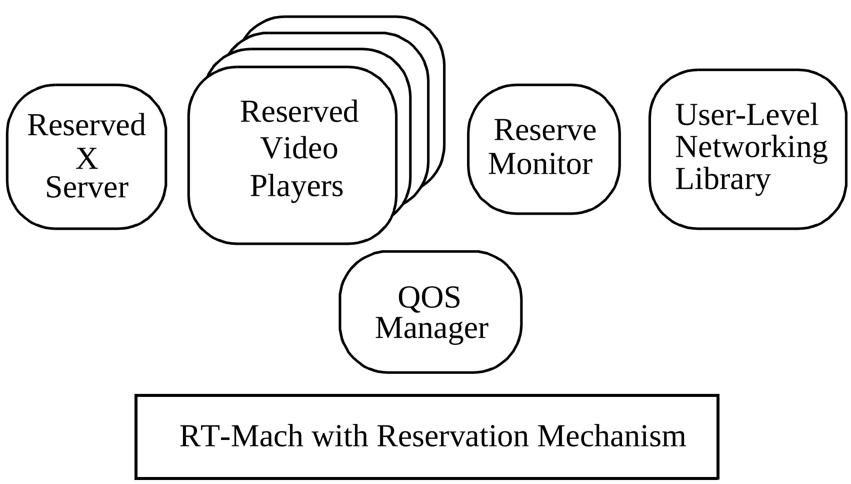



**Figure 5-1: System Components**

First there is the implementation of processor reserves in the RT-Mach kernel which is the basis for the rest of the implementation work. Reserves were implemented as a new kernel abstraction with operations for create/terminate, requesting reservation parameters, binding threads to reserves, and extracting usage information about reserves.

Several real applications were modified to use processor reserves including: a QuickTime video player developed at CMU called QTPlay, an MPEG decoder called mpeg_play [93], and a version of the X Server [34].

A version of the user-level socket library [70] was modified to use reserves as well. This socket implementation supports predictable performance for applications that send and receive network packets.

A QOS manager was implemented to allow for more sophisticated negotiation of reserve parameters than that provided by the kernel mechanism itself. The QOS manager interacts with applications to try to balance resource usage and negotiate with applications when conflicts arise in the resource reservation requests.

Tools were implemented to help manage reserves and to monitor resource allocations and measure usage. The rmon application is a reserve monitor that provides a graphical user interface for reserves. It displays the reservation levels and usage in recent history, and it also allows the user to change the reservation parameters from the graphical interface.

## 5.2 Reserves in RT-Mach

The implementation of processor reserves in RT-Mach involved adding the new reserve abstraction, implementing the operations on reserves, creating a new scheduler, and adding code for accurate usage measurement. In addition to the new scheduler, the reserve implementation also required modifications in the RT-Mach priority inheritance mechanisms to support reserve inheritance and reserve propagation.

The reserve abstraction in RT-Mach is managed much like the other abstractions like hosts, processor sets, tasks, threads, etc. that originated in Mach 3.0 [11]. These types of resources in Mach are referenced by ports, which are used as capabilities.

In RT-Mach, processor reserves are allocated from processor sets. In the uniprocessor version, there is only one processor set, so all reserves originate from this processor set.

### 5.2.1 Attributes and basic operations

Abstractions like tasks and threads offer basic operations such as create, destroy, get attributes, and set attributes. The basic operations on reserves are as follows:

**reserve_create(out reserve)** Creates a new processor reserve and returns it as an out parameter.

**reserve_terminate(reserve)** Terminates the given reserve, making its reserved capacity available for other requests.

**reserve_set_attribute(reserve, attr_name, attr_value, attr_value_size)** Set the value of an attribute of the reserve.

**reserve_get_attribute(reserve, attr_name, out attr_value, out attr_value_size)** Get the value of an attribute of the reserve.

**processor_set_reserves(processor_set, out reserve_list)** Returns the list of reserves associated with the given processor set.

The get attribute and set attribute operations give the programmer access to some of the attributes of reserves. The externally visible attributes that reserves have appear in the following list. The data types for the attributes are given in parentheses after the attribute names; "int" is an integer, "timespec_t" specifies a time value, "mach_reserve_name_t" is a fixed length string, and "boolean_t" is a boolean flag.

**name (mach_reserve_name_t)** A symbolic name for the reserve.

**ckpt_total (timespec_t)** The cumulative total usage measured at the at last period boundary.

**ckpt_time (timespec_t)** The absolute time of last period boundary (when the ckpt_total value was recorded).

**accum_total (timespec_t)** Cumulative total usage at the current time (usually updated when accessed).

**accum_time (timespec_t)** The time that the total usage was last updated.

**used (timespec_t)** Usage charged against the reserve so far in current period.

**next_period (timespec_t)** The absolute time of the next period boundary (end of the current period).

**period (timespec_t)** The duration of the reservation period.

**computation (timespec_t)** The reserved computation time.

**recent_checkpoint_position (int)** The position in the recent_checkpoints array for the next item to be written. The array is a circular buffer.

**ncheckpoints (int)** The number of how many checkpoint entries currently in the array.

**recent_checkpoints[MAX_CHECKPOINT_COUNT] (timespec_t)** The usage values for the recent checkpoints.

**recent_checkpoint_times[MAX_CHECKPOINT_COUNT] (timespec_t)** The times at which corresponding checkpoint usage values were recorded.

A "checkpoint" occurs at each period boundary for each reserve. At that time, the accumulated usage is recorded along with the absolute time of the period boundary. This information can be used later by applications or monitoring tools that need information about how much usage was charged against a reserve in a particular reservation period.

There are a few other reserve attributes that are only used internally. They form part of the scheduling state for a reserve and are not available through the get attribute operation. These internal attributes are:

**reserved (boolean_t)** The bit that indicates whether the reserve is in "reserved mode" or "unreserved mode".

**wait_replenish (boolean_t)** An internal flag indicating that the reserve has a reservation that has been depleted for the current period. The reserve is awaiting replenishment.

**start (timespec_t)** The absolute time at which the first period for the reserve started.

### 5.2.2 Reservation requests and admission control

When initially created, a reserve does not have an associated resource reservation. Getting a resource reservation for the reserve requires an additional call. The following operation allows a programmer to specify reservation parameters and request a reservation. This call is used when there is no reservation associated with the reserve, or if the programmer wishes to request a reservation with parameters that are different from the reservation associated with the reserve.

**reserve_request(reserve, reservation_parameters)** Requests a resource reservation to be allocated to the reserve. The caller provides the reservation parameters. Reservation parameters include desired reserved time per period, the period itself, and start time for reservation to take effect.

This operation is used to request a reservation with certain parameters. If there was previously no reservation associated with the reserve and the reservation request succeeds, then the operation returns "success" and the reservation is granted for that reserve. If the reservation request fails, the operation returns the error, and the reserve is left without a reservation.

If the reserve already had a reservation at the time the request call was made, the behavior is as follows. If the new reservation request is granted, the new reservation parameters will be associated with the reserve, and the old reservation will be freed in this process. If the new reservation request fails, the old reservation parameters remain in effect; that is, the old reservation will not be freed if the new reservation request cannot be granted.

The request operation invokes the admission control policy to determine whether the new reservation request can be accommodated given the collection of other reservations that have already been accepted for the resource. The RT-Mach implementation uses an admission test based on rate monotonic analysis, but the decision is somewhat optimistic in that it uses a utilization bound of 90% for testing for schedulability. This is based on the analysis of average schedulable bound [63], which says that for a randomly generated task set the schedulable bound is 88% on average.

### 5.2.3 Scheduling

The scheduler in RT-Mach was structured so as to make it easy to develop and incorporate scheduling policies. The scheduler uses a well-defined interface for scheduler operations, and a function pointer table in the scheduler contains the operations for the scheduling policy in effect. The scheduling policy can be changed dynamically by putting the ready threads in a policy-independent queue, changing the pointers in the function table to refer to the operations for the new scheduling policy, and then transferring the ready threads into a policy specific queue.

Several scheduling policies, including the "Reserves" scheduler, are supported in RT-Mach [85]. The scheduling policies in RT-Mach are associated with "processor sets," and in the case of a uniprocessor, there is only one processor set in the system. The operations to get and set attributes of a processor set are used to query or set a scheduling policy.

> **processor_set_get_attribute(processor_set, attr_name, out attr_value)** To get the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". The operation sets the "attr_value" to reflect the currently active scheduling policy.
>
> **processor_set_set_attribute(processor_set, attr_name, attr_value)** To set the scheduling policy for a processor set, PSET_SCHED_POLICY_ATTR is specified for "attr_name". A value such as SCHED_POLICY_RESERVES is given for the "attr_value".

Several scheduling policies are implemented in RT-Mach. The original Mach time-sharing policy is available, as are several varieties of fixed priority and rate monotonic policies. Earliest deadline scheduling is available, and round robin scheduling is supported for experimental purposes. Reserve-based scheduling is also a policy option. The policies that are available in the MK83j version of RT-Mach are:

1. Mach Time-sharing - Original time-sharing policy.
2. Fixed Priority/RR - The fixed priority/round robin policy services threads in order of a fixed priority associated with each thread. Within a priority class, threads are scheduled round robin with a quantum.
3. Fixed Priority/FIFO - The fixed priority/FIFO scheduler uses fixed priorities as well, but within a priority class, threads are scheduled using a FIFO discipline (with no quantum).
4. Rate Monotonic - Rate monotonic scheduling based on the periods given to periodic threads.
5. Deadline Monotonic - Deadline monotonic scheduling based on the deadlines given to periodic threads.
6. Earliest Deadline First - Schedules based on the deadline information of threads.
7. Round Robin - Simple round robin scheduling (with a time quantum).

8. Reserves - Scheduling policy for the reservation system.

The “Reserves” scheduling policy uses one queue for “reserved mode” threads, which are listed in order from smallest reservation period to largest. It uses an additional table of queues for implementing a multi-level feedback queue for time-sharing or “unreserved mode” threads. The reserved mode threads are scheduled first, and when there are no more reserved mode threads, the scheduler services the unreserved mode threads. In order to prevent starvation of unreserved mode threads, the reservation parameters are limited. In the implementation, a reservation cannot have a period larger than one second. This ensures that no reserved computation time can be greater than 0.9 second, so in the worst case, reserved activities can hold the processor continuously for no longer than 1.8 seconds before time-sharing programs get a chance to use the processor.

### 5.2.4 Usage measurement and enforcement

The scheduler for the reservation mechanism requires very accurate usage measurement so that the system can keep track of how the resource usage of each activity relates to its reservation (if any). In particular, reserved activities must be prevented from over-running their reservations and interfering with other reserved and unreserved activities.

To accumulate very accurate usage measurements, the system has code in the context switch from an old thread to a new thread that does the following:

1. takes a timestamp from a high-resolution free-running clock.
2. computes the duration of time the old thread was running and charges that usage against the reserve associated with the old thread.
3. stores the timestamp for doing the same computation later for the new thread.

Those actions mean that threads get charged for the amount of time they spent on the processor rather than getting charged an estimate of the time they spent. This timestamp method much more accurate than the method used for accumulating usage in many time-sharing systems where the process running at the time of a clock tick is charged for the duration of the clock tick (whether it was running the whole time or not).

The reserve abstraction has several operations that provide the programmer with access to usage information in the reserve. The usage-related operations are:

**reserve_get_checkpoint(reserve, out checkpoint_total, out checkpoint_time, out accum_total, out accum_time)** Get reserve’s checkpoint information, taken from the last reservation period boundary.

**reserve_get_attribute(reserve, attr_name, out attr_value, out attr_value_size)** The reserve_get_attribute operation can be used with “attr_name” set to RESERVE_RECENT_CHECKPOINTS to get an array of the recent checkpoint values for the last several period boundaries.

In addition to the accurate usage measurement, the enforcement mechanism uses timers to keep threads from over-running their reserved computation time and to replenish the reserved time for a reserve appropriately. Two kinds of timers are used for doing these things: the overrun timer and replenishment timers.

The overrun timer is set at each context switch, and it is set for the maximum time the new thread could run before over-running its reserved computation time for its current reservation period. If the thread is still running when the timer expires, the system will update the reserve to show that the activity used all of its time for that period, and it will change the activity to unreserved mode. Then the scheduler will get an opportunity to reevaluate ready threads, and it may decide to switch to another thread.

If the current time is close to the end of the reservation period for the new thread and the reserved time is longer than the difference between the current time and the end of the reservation period, the overrun timer is set to expire at the end of the reservation period. If the timer goes off at that point, the reserve will be replenished and the activity will again be eligible to run.

The second kind of timer is the replenishment timer. Each reserve has a replenishment timer that is initially set at the reserve's start time. The timer is set to expire at the end of the reservation period (or the beginning of the new reservation period). When a replenishment timer expires, the system changes the state of the reserve to reflect that it has a new allocation of its reserved time for the next reservation period. The reserve is set to reserved mode, and the replenishment timer is set to expire again at the end of the new reservation period.

### 5.2.5 Reserve propagation

One of the key features of the reserve abstraction is that reserves can be bound to threads as appropriate for particular applications. This feature is useful in the situation where an application initiates a reserved activity that may invoke services of server processes locally or even on remote machines. When invoking a server, an application can make its reserves available for the server to use in its computations. The server can then take advantage of having the resources available, and the time it takes to perform the computation on behalf of the client can be charged to the client's reserves. In this way, reserves provide a method for consistently measuring resource usage of entire activities, even if threads in different protection domains cooperate on behalf of the broader activity.

The following operations are related to the binding of reserves to threads:

**thread_set_current_reserve(thread, reserve)** Each thread has a current reserve and a base reserve. The value of the current reserve may be the result of a reserve propagation, but it is not necessarily permanent. It may eventually revert to the base reserve. This primitive sets the current reserve of a thread.

**thread_restore_base_reserve(thread)** Makes the base reserve the current reserve for the given thread.

**thread_set_reserve(thread, reserve)** Set the reserve of a thread.

**thread_get_reserve(thread, out reserve)** Get a thread's reserve.

In addition to the binding operations, the priority inheritance mechanism of RT-Mach [59,86] aids in ensuring bounded delay for access to servers and mutexes. In the context of reserve scheduling, "priority inheritance" means "reserve inheritance" in the following sense. Interpreting "priority" in the broadest sense, one could think of a thread's reserve information and time-sharing priority information as combining to yield a total ordering for values of these fields. The scheduler schedules threads based on this total ordering from highest "priority" to lowest:

1. threads that have the "reserved" bit set are ordered with smaller reservation period having higher "priority" than larger reservation periods.

2. threads with the "reserved" bit cleared are ordered according to their time-sharing priority, which is a field of each thread (not a field of the reserve).

This concept of "priority" comes into play in the priority inheritance mechanism in RT-Mach. As an example, consider a single-threaded server with several clients. When a client makes a call to the server, the server takes on the "priority" of the client (done in the priority inheritance mechanism) and binds its own thread to the client's reserve to charge its time to it (using the bind operation).

If during this service time a second client with a higher "priority" makes a request, the IPC mechanism enqueues the new request for the server. It then calls on the priority inheritance mechanism to change the "priority" of the server to that of the newly enqueued client (thus limiting the duration of the "priority" inversion). The server continues to charge time to its first client's reserve, however, so that reserve will reflect the true resource usage required for the computation. After the service is finished, the server stops charging against the first client's reserve, picks up the request from the second client, and starts charging against the second client's reserve. The server continues to execute under the "priority" of the second client.

Priority inheritance for reserved activities presents an additional complication beyond what fixed priority inheritance mechanisms must face. In particular, with reserves (and with other dynamic priority disciplines), the "priority" the server takes may change during the service. For example, if the server executes for longer than the reserved time of its client's reserve, the reserve will be degraded into unreserved mode, and the "priority" thus changes. In the implementation, the priority inheritance mechanism is informed when this happens so that it can set the priority of the server to the appropriate value given the list of clients waiting for that server. For example, if a server uses all the reserved computation time for a particular client it would normally have its reserve downgraded and its "priority" decreased. However, if another reserved client is waiting for the server, the server will inherit the "priority" of that client so as to avoid a priority inversion.

## 5.3 Applications

A number of applications were modified to use reserves to show that real applications could actually achieve predictable behavior using the reservation system. A QuickTime video player and an MPEG decoder were modified to use reserves, and a version of the X Server [34] was modified to cooperate with reserved applications to provide predictable window system services.

### 5.3.1 QuickTime video player

A QuickTime video player, called QTPlay, was implemented at CMU. The player can display JPEG encoded video as well as raw, unencoded video. The player was modified to use reserves to achieve predictable performance.

QTPlay avoids interactions with system components that have not been modified to support predictable performance via reserves, such as the UX server. It loads a short clip of video into memory during initialization to avoid interaction with the UX server during playback. For experiments, the player loops over the clip for the duration of the test. Figure 5-2 summarizes the structure of the reserved QTPlay application.

```
load short video clip
allocate reserve with command-line parameters
create periodic threads
bind thread to reserve
...
while not done
   save start timestamp
   display a frame
   save end timestamp
...
dump timestamps to a file
```

**Figure 5-2: QTPlay Outline**

At initialization, QTPlay reserves time on the processor and binds the periodic thread responsible for frame processing to the reserve. The start time of the periodic thread and the start time of the reservation are synchronized so that when the thread becomes ready at the beginning of each period, the allocation of processor time will be available as well. The other reservation parameters, in particular the reserved computation time and the reservation period, are given as command-line arguments. They are typically determined by measurements made prior to the execution of the player. This particular application does not dynamically discover the appropriate reservation parameters nor does adjust the reservations after execution begins.

The player uses a version of the X Window System library, Xlib, that was modified to cooperate with a reserve-enabled X Server. This library passes a reference to the thread's reserve when the player opens the connection to the X server. The X server then uses the reserve for operations requested by the player (such as DisplayBitmap).

In each period, the thread displays a frame of the video and then saves the start time and completion time for the frame in a buffer in memory. Just before the player exits, it dumps the contents of this timestamp buffer to a file for subsequent analysis.

### 5.3.2 MPEG decoder

The Berkeley MPEG decoder [93] was modified to use processor reserves in RT-Mach. This version of mpeg_play reserves processor capacity during its initialization and periodically evaluates its performance and makes adjustments to its processor reservation and timing constraints as necessary.

The original Berkeley MPEG decoder works by repeatedly reading MPEG encoded macroblocks from an input stream, transforming them, and displaying the frames. The underlying mpeg library has some features for managing the timing of frames, but the simple player that is provided to demonstrate the use of the library displays frames as fast as possible without attempting to regulate their timing.

A number of changes and extensions to the MPEG player were required to enable predictable performance and to take advantage of the timing features of RT-Mach as well as the processor reservation mechanism. Figure 5-3 summarizes the code structure of the modified version of mpeg_play.

```
load short video clip
allocate reserve with command-line parameters
create periodic thread
bind thread to reserve
...
while not done
   save start timestamp
   display a frame
   save end timestamp
   if frame_number mod 30 == 0 then
       evaluate usage
       adjust reservation parameters and/or algorithm
```

**Figure 5-3: mpeg_play Outline**

As with the QTPlay, mpeg_play prefetches a short clip of video into memory to avoid interacting with the file system during runtime. The frames are decoded and displayed by a periodic thread that has the period desired for video playback, typically 33 ms.

During the initialization of the modified MPEG player, it requests a processor reservation based on an estimate of the computation time and the length of the period. Since the computation time may vary on different hardware platforms and different MPEG data streams, it is very difficult to get an accurate estimate before running the application, and this player tunes its reservation parameters as it executes.

For each frame, the player records the time the computation was started, the time it ended, and the amount of processor time it used during its execution (taken from the usage information in the processor reserve). It periodically computes statistics on these numbers for the recent periods to find out how much computation time was required for each frame and whether the delay for the computation is excessive. This information is used to decide what adjustments need to be made (if any). In its evaluation, mpeg_play distinguishes three cases:

1. reservation level okay, do nothing.

2. reservation level too low but some capacity is available to be reserved, increase reserved computation time while keeping the same period.

3. reservation level too low and no additional capacity is available to be reserved, increase the reserved computation time to the desired amount and increase the reservation period proportionally.

The modified version of mpeg_play incorporates some basic adaptive techniques, but it could be extended in a number of directions to improve its flexibility and performance. The decoding and display phases of the player should be decoupled to allow the variation in decoding time to be masked by buffering with the application. Incremental decoding techniques would yield several options for how much computation to do for each frame, and changing the dithering algorithm dynamically would increase flexibility as well.

### 5.3.3 X Server

Each real-time X client acquires a processor reserve and charges its own execution time against that reserve as well as providing the reserve to the X Server so that the Server can charge service time done on behalf of that client to the appropriate reserve. We have modified a version of the X Server to order service requests according to their timing constraints and to charge service time to the client for which the service is performed. Basically, the server should mimic as closely as possible the behavior that would be observed if each client could do its own graphical display within the context of its own address space and scheduling domain. Thus the modified X Server has the following properties:

1. The Server ensures that the activities of real-time clients are isolated from unwanted interference from non-real-time X clients by ordering all request from real-time clients down through non-real-time clients and servicing them in that order.

2. The Server itself is isolated from unwanted interference from non-real-time applications (even applications which are not X clients) by virtue of the processor reservation mechanism. The reservation system ensures that, while the X Server is running under a client’s reservation, the resource capacity associated with that reservation is available to the X

Server.

3. Other real-time applications that are not X clients are isolated from unwanted interference from the X Server and its clients if they use the reservation system. This would not be true if the X Server were just assigned a “high priority” or if it over-reserved resources.

The goal of this work is to achieve predictable performance for real-time applications that make use of the graphical display services provided by the X Window System. “Predictable performance” means that real-time applications will be scheduled based on their timing requirements, and their graphical display requests serviced by the window system will be scheduled to meet the timing requirements. Thus, the abstract activity for each real-time client, consisting of the computations within each client application and the associated computations within the window system, should suffer only bounded delays due to other real-time and non-real-time applications sharing the window system.

The processor capacity reserve mechanism provides this kind of timing isolation for independent programs which do not communication or synchronize with each other. However, when applications share a single software resource such as the X Server, the same kind of timing isolation provided by reserves must be extended into the Server’s computations. To provide this isolation and bounded delay, the following is required of the server:

1. Requests from different clients that queue up in the server should be serviced in the order that the clients would be serviced if they were doing the work themselves and being scheduled by the processor reservation mechanism. In other words, the server should handle requests in order of client “priority” (where client priority refers to an implied ordering among clients defined by the reservation system).

2. Computation performed in the server on behalf of a client should enjoy resources, such as processor capacity, that have been reserved for that client. The server should execute at the priority of the client whose request it is servicing. Likewise, the resource usage for such a computation should be charged to the client’s reservation, so that a client is prevented from getting more than its reserved time by sending some work to the server and then doing other work locally.

3. Priority inversion should be minimized (and unbounded priority inversion completely avoided) in servicing the clients’ requests. Thus if the X Server is occupied with a request from a client when another request comes in from a higher priority client, the server should inherit the priority of the newly arrived client.

These requirements have many implications for the coding of the server. The idea that the server should mimic the behavior of individual threads performing the same computations places some restrictions on how the server can be designed. Also, each of the three specific requirements listed above has some additional implications for the coding of the server.

First we address the desire to have the server behave as the individuals would. This is conceptually easy to achieve by thinking of spawning a new thread for each client's request as it arrives at the server, binding the thread with the reservation or priority of its client, and then allowing the threads to be scheduled by the processor reservation system based on that information. Unfortunately, spawning a potentially large number of new threads is expensive, and while there exists a version of the X Server that is multi-threaded [110], the one on which this work is based has only a single thread. With a single-threaded server, we try to mimic the desired behavior by satisfying the three requirements listed above as follows:

1. Client requests are enqueued in the server in priority order.

2. At the beginning of the computation for each service request, the server takes on the resource allocation persona of the client, enjoying the resource reservations of the client and charging usage against the client's reserve.

3. The RT-IPC [59] mechanism handles priority inheritance to minimize the effects of priority inversion.

These are the modifications made to the X Server to provide predictable performance. However, there are some problems with the X Server that interfere with real-time applications and which are very difficult if not impossible to fix. Several of these problems are addressed in the development of a window system intended for real-time performance [105]. Briefly, the problems are:

1. The X Protocol supports a "grab server" operation which blocks out all other operations for an unbounded period of time.

2. The X library batches requests for higher throughput. This can increase the delay of single operations as multiple operations are combined into one.

Despite these hindrances to 100% guaranteed real-time performance, the modified X server can provide good real-time behavior for typical multimedia applications such as video players.

## 5.4 Reserved network protocol processing

A predictable network service depends on how the protocol processing for network packets is handled as well as how these activities are scheduled. This section examines several different approaches to protocol processing software design and discusses the advantages and disadvantages of these approaches.

### 5.4.1 Software interrupt vs. preemptive threads

Traditionally, protocol processing software has been designed to take packets from the network interface and immediately begin processing them at high priority. For example, 4.3

BSD protocol processing is done at a "software interrupt level" which executes at a higher priority than any schedulable activities in the system (like processes) but at a lower priority than hardware interrupts [62]. Unfortunately, network packets associated with a low priority activity may flood the protocol processing software and execute while higher priority processes are delayed. This is an example of priority inversion [48,75].

To prevent this kind of priority inversion, it is necessary to associate priorities with packets so that they can be queued and serviced in priority order. It may also be helpful to be able to preempt the processing of one low priority packet in favor of a higher priority packet, especially if the computation time required for protocol processing is significantly more than that required for a context switch. One approach, used in the ARTS real-time kernel, has preemptible threads to shepherd packets through the protocol software [124]. This is similar to the method used in the *x*-kernel [45], but unlike the *x*-kernel threads, ARTS protocol processing threads were preemptive. This approach provides fast response to high priority packets and prevents low priority network activities from interfering with high priority work on the processor.

## 5.4.2 Mach 3.0 networking

Networking in the context of the Mach 3.0 UX server [36] is accomplished by calling the 4.3 BSD networking primitives, which are handled by the UX server. The UX server interacts directly with the network device drivers to send and receive packets.

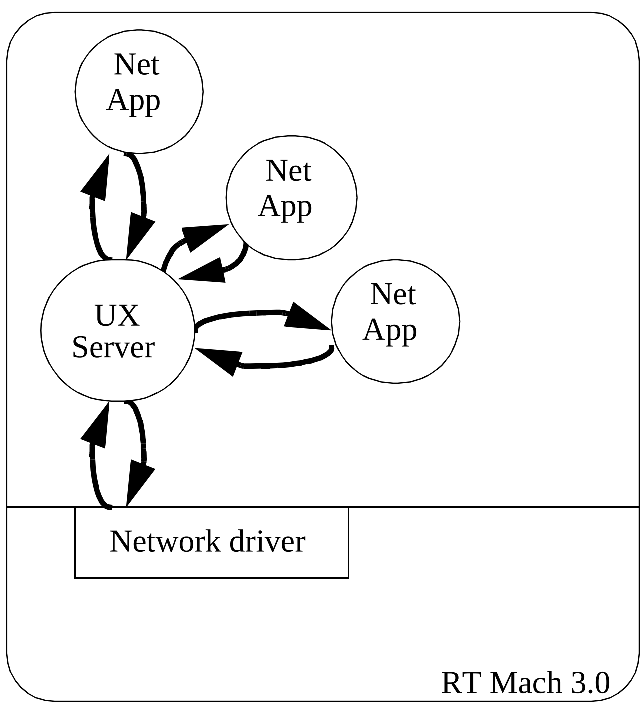



**Figure 5-4: Networking with the UX Server**

As shown in Figure 5-4, this makes the UX server a single point of contention for all activities that are using the network. Unfortunately, the networking code inside the UX server does not support priority. So this software does not satisfy the requirements for priority and preemptibility in predictable protocol processing software.

Another problem with networking under the UX server of Mach 3.0 is that the interprocess communication (IPC) required between the application and the UX server and between the UX server and the network device drivers adds overhead to network communication. This decreases throughput and increases latency. To alleviate these problems, Maeda and Bershad created a library implementation of TCP/IP and UDP/IP sockets [70]. Their library handles the protocol processing for sending and receiving packets and interacts with the network packet filter [139] and network device drivers directly. The library can be linked in with applications that use the networking calls, so each application can do its own protocol processing in its own scheduling domain (i.e. within its own threads). The library only interacts with the UX server to create and destroy connections and for a few other control operations. The fast path for sending and receiving packets is confined to the library itself (and the device drivers). Figure 5-5 illustrates their networking software structure.

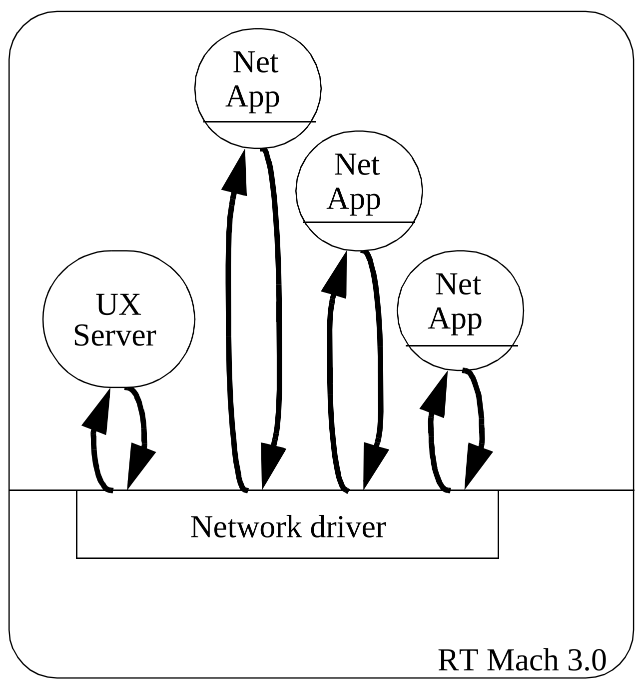



**Figure 5-5: Networking with the Socket Library**

Maeda and Bershad report that their socket library yields much better performance in terms of throughput and delay than the UX server sockets implementation [70]. Coincidentally, their implementation also satisfies the requirements for effective real-time scheduling of protocol processing. By including the code in a user library, the computation is done by the user thread at the user's priority. It is also preemptible since it runs in user mode and shares nothing with other threads in other applications.

### 5.4.3 Reserved protocol processing

Since the socket library enables the protocol processing computation to be scheduled under the priority of the application and since it is also preemptible, the processor reservation system can be applied to programs which do socket-based communication [77]. Compared with a UX server socket implementation, the library partitions the data structures and control paths of all of the networking activities and places them in independent address spaces where they do not to interfere with each other. In the UX server, these different activities are forced to share the same queues without the benefit of a priority ordering scheme. Other activities such as file I/O, asynchronous signals, etc. may interfere with the protocol processing, thus delaying packets as a result of other operating system activity that is not even related to networking.

In the socket library, these components do not interfere with each other, so the reservation mechanism is free to make decisions about which applications should receive computation time and when. The control exercised by the reservation scheduler is not impeded by additional constraints brought on by the sharing of data structures and threads of control. Applications that use the socket library with the reservation mechanism should therefore achieve very predictable networking behavior.

## 5.5 QOS manager

A QOS manager was implemented to provide a central point for resource allocation decisions. It exports an interface that allows the application programmer to create and terminate reserves, to request a reservation at a specific desired level, and to set preferences for the minimum reservation level. If the reserved load becomes high and the server has difficulty granting minimum reservation levels for new requests, the server begins to downgrade some of the previously granted reservations to their minimum levels in order to admit the new reservation request at its minimum level.

In general, information about resource allocation requirements may come from a variety of sources and may change over time. Resource allocation information can come from applications themselves which may request resource and negotiate if the request cannot be satisfied immediately. It can come from static user preferences about which applications should be more resources under what circumstances. And it can come from various user interface elements designed to bring resource management decisions to the console user.

### 5.5.1 Information sources

The QOS manager uses the information it gathers to make policy decisions about how to allocate resources to various activities. The information may come from user preferences files, applications themselves, and graphical resource management tools. Figure 5-6 summarizes the information flow associated with the QOS manager.

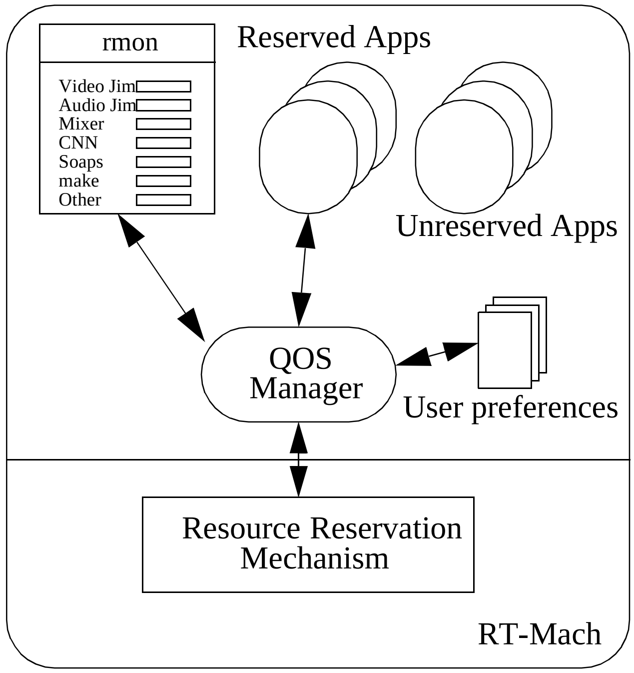



**Figure 5-6: Resource Management Schematic**

The static user preferences used by a QOS manager might come from a configuration file located in the user's home directory or in a system default directory. Such a file could contain arbitrarily sophisticated rules for the QOS manager to use in making allocation policy decisions. For example, the file might contain rules to indicate how the user's focus should affect resource allocation. It might have rules to determine which applications are more important (e.g. specifying that audio/video applications are more important that file transfer). There might be rules about how temporal properties indicate which applications are more important (e.g. giving recently created applications preference over older applications). And there might be rules about how past usage should affect future reservation.

The dynamic user preferences might come from the applications themselves, from a separate tool, or from some mechanism associated with a window manager. In any case, cues given by the user, which can be picked up in the user interface, are very important to the policy decisions that must be made about where to allocate resource capacity. These cues can be explicit, where the user makes certain gestures to change the resource capacity allocations of various activities. Or the cues can be implicit, as in the case of a window manager which notices which window has the user's focus (based on the position of the mouse pointer) and passes this information along to the QOS manager.

Information about the recent resource usage of various activities might be used to determine what the future resource reservation levels should be for those activities. For example, an audio player receiving transmissions over the network might become quiet due to long-lasting lull at the sender. When this happens, it may be appropriate to notice the lack of resource usage in the associated reserve and temporarily scale down the reservation level in order to free up more reservable capacity for other activities. A QOS manager with this feature would undoubtedly also provide a mechanism for such dormant applications to come back to life at their original reservation level once they become active again.

### 5.5.2 Admission control

The admission control policy of the QOS manager must be coordinated with the admission control of the system. The reservation system has an admission control policy that allows it to enforce reservations and keep itself internally consistent with respect to resource allocation and enforcement. The QOS manager must have a version of the same admission control policy so that it can evaluation reservation requests that it gets and look at more sophisticated issues such as how different requests it gets can be combined or changed to fit together better.

This design was chosen because it keeps the admission control test of the kernel simple and fast while allowing arbitrarily sophisticated admission control decisions and negotiations to be carried out in user-level QOS managers. It would be possible to combine the two policies but there are drawbacks to that approach. If the sophisticated policy with negotiation were implemented in the kernel, the system would become more complicated, slower, and less flexible. If the kernel depended on user-level admission control for its own consistency, it would be vulnerable to errors in the user-level QOS managers.

### 5.5.3 Extensions

The QOS manager reacts to new reservation requests that strain the available resource capacity by trying to free up resource capacity from among previously reserved activities, subject to the limitations that those activities allow as expressed by their minimum reservation levels. This policy could be extended to accommodate information about which activities should be downgraded first, whether new minimums could be negotiated with activities to free up even more capacity, or whether the activities requesting reservations should be denied to keep the previously reserved activities at their current reservation levels [79]. Another extension might upgrade reservations to the old desired values once reservable resource capacity became plentiful.

## 5.6 Tools

Two tools were developed in the course of this dissertation work to do debugging and execution monitoring for experiments and to provide a user interface for reserves. One is a reserve monitor with a graphical user interface, and the other is a usage monitor that operates in batch mode to gather usage statistics for experiments.

### 5.6.1 Reserve monitor

A reserve monitor, called rmon, provides the user at the console with a graphical user interface to monitor and control processor reserves. The two important aspects of this tool are its presentation of usage information and its support for control of resource reservation.

#### 5.6.1.1 Usage information

The primary view of rmon displays basic information about all of the processor reserves in the system. This information consists of the name of the reserve, a graphical representation of the recent usage information, normalized to the reservation period, and the reservation period itself. Figure 5-7 shows a screen dump of this primary view.

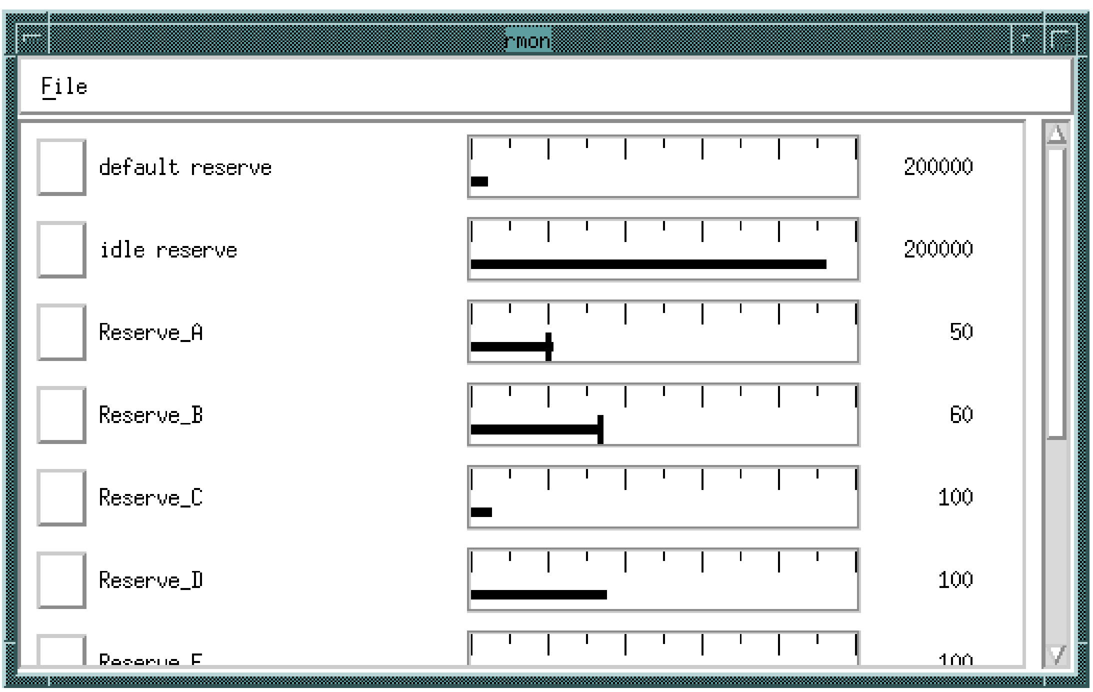


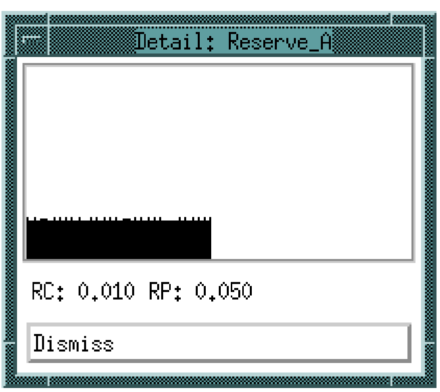

**Figure 5-7: rmon Main View**

As the figure shows, each reserve takes one row in the display. Each row contains the following elements:

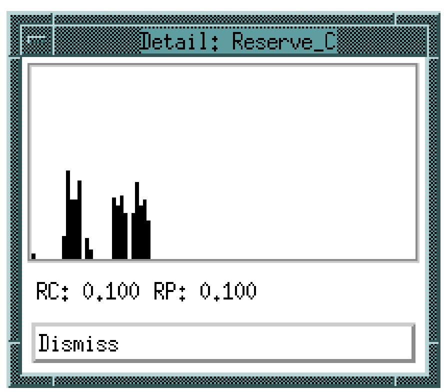
- detail button - Pressing the detail button pops up another window which shows more detailed information associated with the reserve.
- reserve name - Each reserve may have a name associated with it for ease of identification.
- graphical usage display - This graphic displays a bar with length corresponding to the percent resource usage over the last several reservation periods. The usage is normalized to the reservation period, and the graphic includes markings to indicate the scale of the usage.
- reservation period - The reservation period indicates the averaging interval of the usage measurement.

As reserves are created and terminated in the system, corresponding rows are created and destroyed in the primary view. The two system reserves (called “default reserve” and “idle reserve”) always exist. So they always appear in the view. The default reserve is where all the usage is charged for applications that do not have their own private reserves. It has no actual resource capacity reservation associated with it; it just accumulates the usage of the unreserved programs. The idle reserve accumulates the usage of the idle thread; it also lacks an actual resource reservation.

For each reserve that has a resource capacity reservation associated with it, rmon displays a vertical bar in the usage graphic to indicate the level of the reservation, in terms of normalized capacity. In Figure 5-7, the reserves named “Reserve_A” and “Reserve_B” have resource capacity reservations associated with them whereas “Reserve_C” and “Reserve_D” do not. The vertical bars in the usage graphics of Reserve_A and Reserve_B indicate that the have reservations of 20% and 33%, respectively. The reservation periods are 50 ms and 60 ms, respectively.




(a)




(b)

**Figure 5-8: rmon Detail Views**

Pressing the “detail” button at the beginning of a reserve’s row pops up another window of detailed information about the reserve. This includes a graphical display of the recent history of resource usage as well as the parameters of the reserve. Figure 5-8 shows two example detail windows. Part (a) of the figure shows the detail for an activity that has a reservation, and part (b) shows an activity without a reservation.

As shown in the figure, the recent history occupies the top portion of the detail window. It shows the normalized usage of the reserve over the last several reservation periods, and it advances in real-time in a manner similar to that of xload [74]. For the reserved activity in Figure 5-8(a), which is Reserve_A from the Main View, this usage is fairly constant over time. For the unreserved activity in Figure 5-8(b), Reserve_C, this usage is variable from period to period. Each window also displays the reserved computation time and the reservation period. The unreserved activity has zero reserved computation time but a non-zero reservation period (the implementation represents an unreserved activity using a reserved computation value equal to the reservation period, and this representation happens to be exposed in this view). This indicates that usage measurements for this activity will be taken based on the reservation period, but that there is no actual resource capacity reservation.

#### 5.6.1.2 Allocation Control

The level of the reservation for a reserve, as indicated by the vertical bar in the normalized usage graphic, can be changed by clicking the mouse in that usage graphic at the level desired for the reservation. This action modifies the reserved computation time parameter of the reserve without changing the reservation period.

The upper screen view in Figure 5-9 shows several reserves at various reservation levels. Notice the position of the mouse pointer in the usage graphic of the reserve called Reserve_B, which is reserved at 20% of processor capacity. Clicking the mouse button with the pointer at this position changes the reservation level of Reserve_B to that shown in the lower screen dump in the figure. The reservation level is now about 40%, and the actual usage of the activity reflects the availability of that additional capacity.

### 5.6.2 Usage monitor

A usage monitor based on reserves was developed to aid in debugging and to support usage measurements for experiments. This monitor allocates a reserve and requests a reservation for its own execution. It periodically polls for the usage on specified reserves in the system, saving the usage numbers in a large buffer. Then it formats the usage information and writes it to a file for processing by a graphing tool.

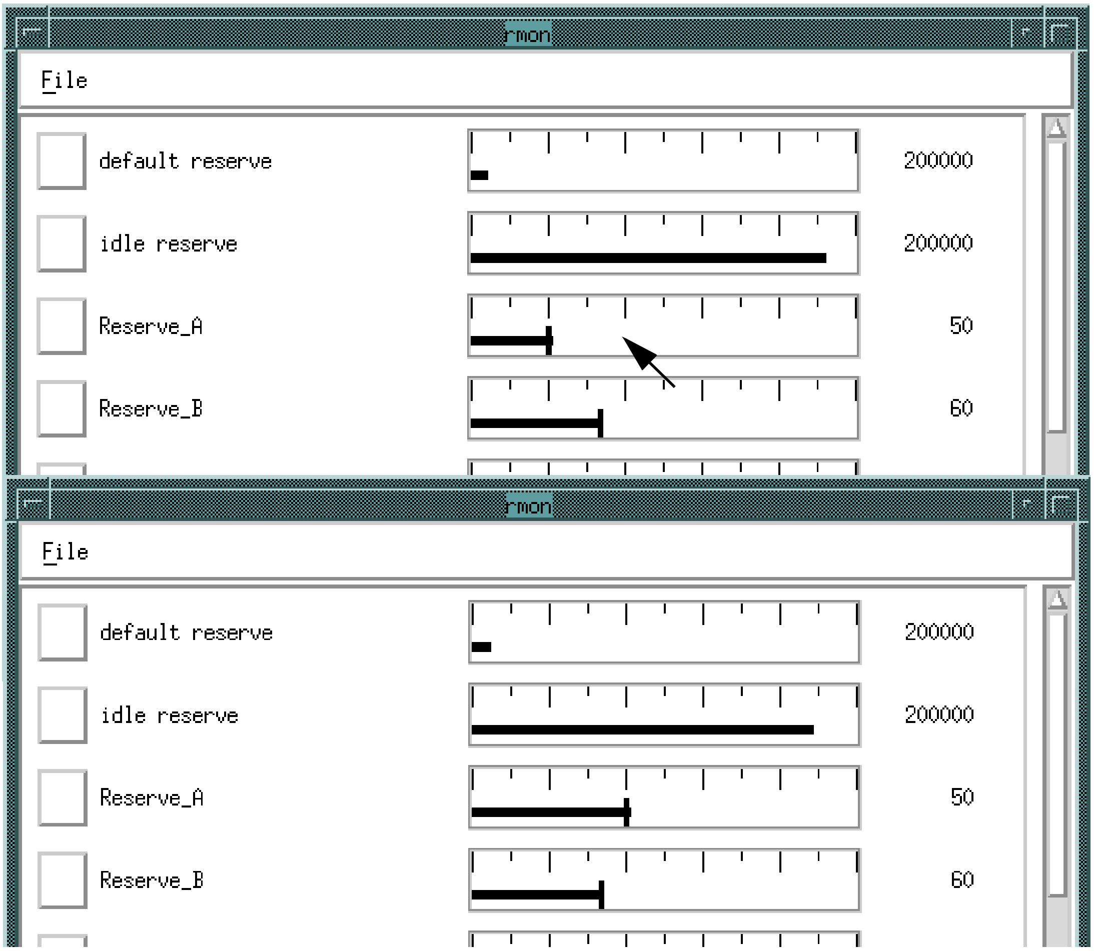



**Figure 5-9: Modifying a Reservation**

## 5.7 Chapter summary

This chapter describes the implementation several software components in the reservation system including:

- processor reserves in RT-Mach,
- reserved video players and a version of the X Server that uses reserves,
- a version of the Mach 3.0 socket library implementation modified to use reserves,
- a QOS manager,
- a tool for providing a user interface to the reservation system.

The description of the implementation of reserves presents operations that are supported for manipulating reserves as well as a description of how the scheduling and usage enforcement is handled. The implementation of two video players that use reservation is described along with that of a version of the X Server that was modified to use reserves. The section on the socket library implementation discusses issues in the organization of protocol processing software to support real-time packet processing. A description of the QOS manager indicates how it works and its relationship with applications and the reservation mechanism. Finally, a graphical user interface tool is described; it displays information about reserves and provides an interface for controlling reservation parameters.