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

Chen Lee, **Ragunathan Rajkumar** and **Cliff Mercer**
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
USA
{clee,raj+,cwm@cs.cmu.edu}

Abstract

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

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

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

1. Introduction

The need for real-time resource management is a key element which explicitly distinguishes multimedia applications from other traditional applications. The Real-Time Mach kernel [19] supports many primitives for constructing predictable and dynamic real-time applications. These primitives include a wide choice of real-time scheduling policies (including fixed priority scheduling, rate-monotonic scheduling and earliest deadline scheduling), periodic and aperiodic real-time threads with the notion of deadlines and handlers for deadline misses, real-time synchronization primitives [12] and real-time inter-process communication primitives which support priority inheritance [15], virtual memory wiring, a real-time shell and a network protocol server, a processor reserve abstraction, an integrated toolset for schedulability analysis and real-time monitoring of scheduling decisions [2], and real-time X11 windowing services [10]. In this paper, we build upon the processor reserve abstraction to provide dynamic quality of service support, and to predict and achieve end-to-end delays in distributed multimedia applications.

1.1. Processor Reserves in Real-Time Mach

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

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

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

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

1.2. Related Work

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

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

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

Nahrstedt and Smith [11] used AIX for their telerobotic application and showed that the AIX real-time priorities are not enough to control protocol task behavior when used for implementation of rate-monotonic or deadline-based scheduling, unless severe restrictions are made which includes only one user allowed, one multimedia application running on the RS/6000, application/transport protocols implemented in a single user process with real-time priority etc. "Only with these restrictions satisfied can we map rate-monotonic scheduling onto the real-time priority scheme of AIX to provide (approximate) predictability for guaranteed services" [11].

1.3. Organization of the Paper

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

2. Dynamic Quality of Service Control

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

2.1. Quality Management and its Implications to Systems Support

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

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

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

2.2. The QOS Server

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

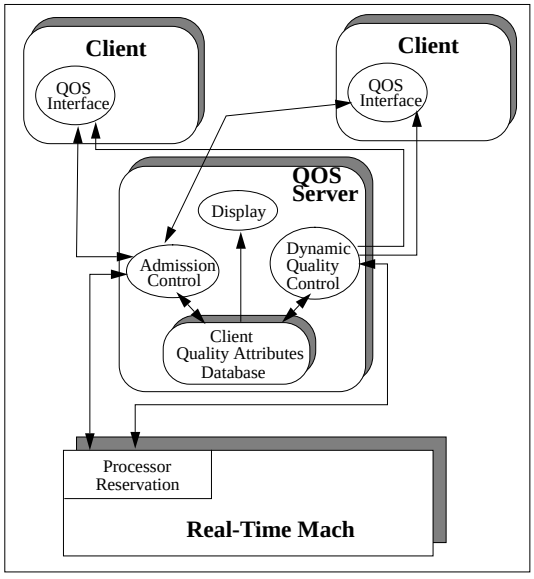

Figure 2-1: The Architecture of the Dynamic QOS Server.

The QOS server is based on the following complementary aspects:

- *QOS attributes* are used by application threads to specify their acceptable quality levels to the QOS server.
- An application can choose from one of many *quality adjustment policies* that the QOS server can use on it if the quality guarantee given to the application has to be upgraded or downgraded. In other words, this determines how an application wants to be treated when its current resource allocation has to be altered. Included in this policy choice is an application *quality adjustment priority* which determines the order by which the QOS server selects the beneficiary (victim) of a server decision to upgrade (downgrade) the quality of one or more applications.
- One of many *admission control adjustment policies* is used by the QOS server to admit as many new application requests as possible without violating the needs of currently registered applications.
- One of many *overrun control policies* is used by the QOS to satisfy the needs of registered applications whose current CPU demand exceeds their actual allocation. This is done by the QOS server exploiting the slack that may be available from applications not using their current resource allocation.

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

2.3. QOS Attributes

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

2.4. Quality Adjustment

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

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

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

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

- A *Keep-Reservation-Constant* policy is obtained by specifying the minimum, maximum and desired values of the computation time to be the same, and similarly for the period.
- A *Step-wise Adjustment* policy is available for the computation time (period) in which the computation time (period) will be adjusted only in discrete steps, the size of which is specified by the application.
- A *Negotiation Policy* tells the QOS server to notify the client when an adjustment needs to be made (along with information about the maximum available reservation at the time) and the client can negotiate the computation time and reservation parameters of the new adjustment. This policy leaves the actual choice of resource allocation parameters to the application and can therefore be used by those applications which cannot be satisfied by any of the other policies. This policy is particularly useful for real-time applications such as feedback control, where controllers may only be defined in the application for some specific values of the period[2].

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

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

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

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

2.5. Admission Control Policies

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

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

2.6. Dynamic Quality Control Policies

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

2.7. The QOS Server Threads

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

2.8. The Client Interface to the QOS Server

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

3. The *RT-Phone* Teleconferencing Application

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

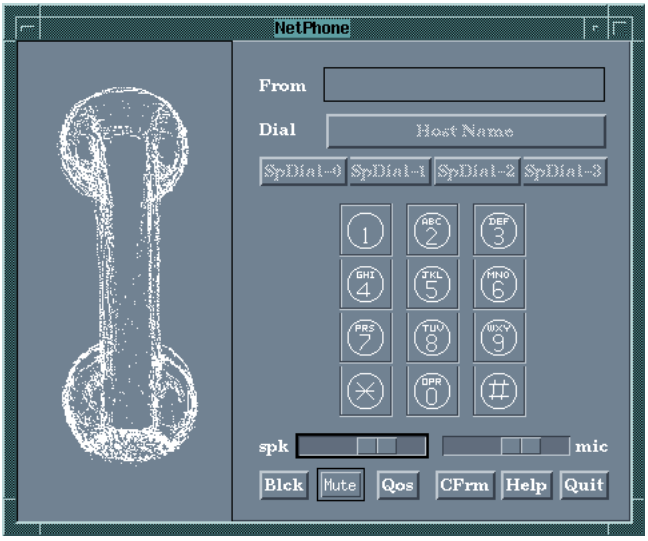

Figure 3-1: The *RT-Phone* user interface on starting up.

3.1. RT-Phone

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

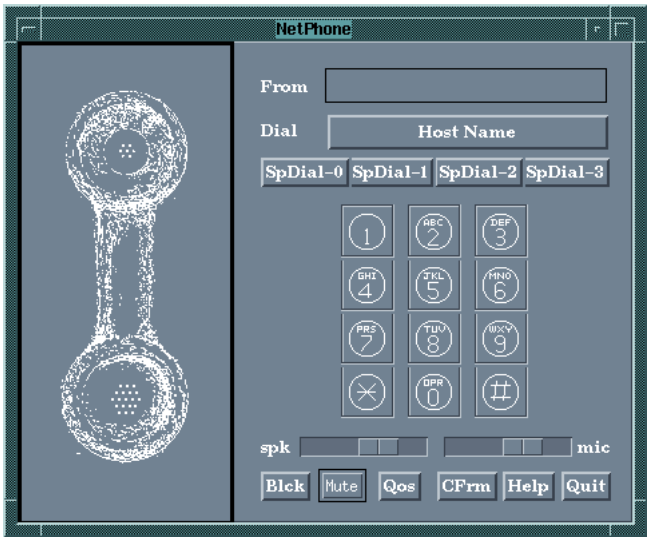

Figure 3-2: The *RT-Phone* user interface in the "off-the-hook/ready-to-dial" state.

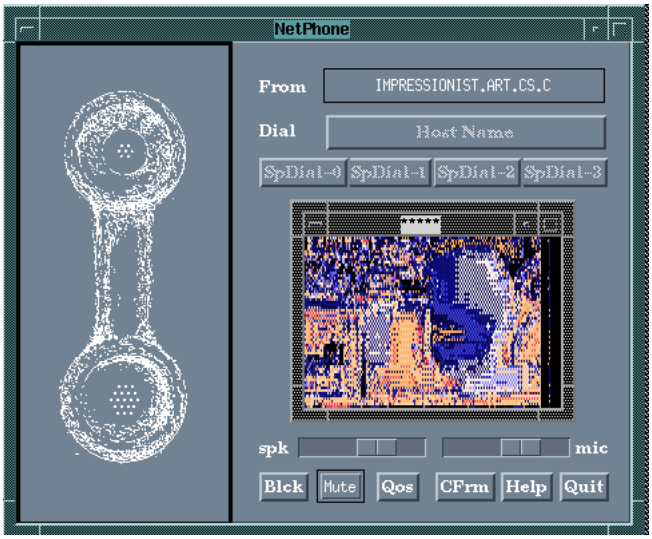

Figure 3-3: The *RT-Phone* user interface after a network phone connection is established.

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

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

3.2. The Architecture of the RT-Phone Application

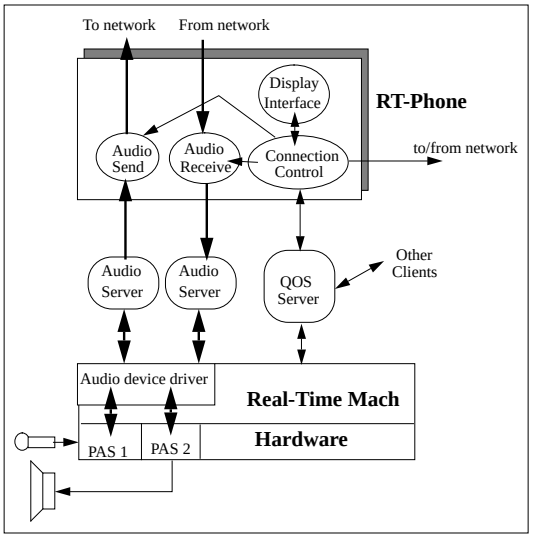

Figure 3-4: The Architecture of *RT-Phone* at each end-point.

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

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

Although the PAS card supports bi-directional audio (capture and output), it cannot do both simultaneously (in duplex mode). Hence, the user-level audio-servers can only do either audio input or output at any given time. We therefore use two PAS cards and two audio servers, one for continuous audio sampling and another for continuous audio output respectively. A periodic interrupt based on the sampling rate and buffer size triggers the input audio server to copy the buffer into its requesting client, an *audio-send* thread within *RT-Phone*. A shared memory buffer is used between the audio server and the *RT-Phone* thread. The *audio-send* thread then transmits the data to the remote callee across the network. An *audio-receive* thread receives the audio data and outputs it to its speaker through the use of the output audio-server[4]. A mirror image of the same streams transmits audio from the callee to the caller.

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

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

4.1. The Quality Parameters of *RT-Phone*

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

4.2. The Guaranteed End-To-End Delay

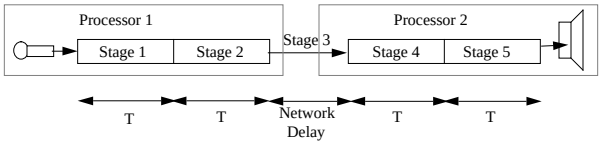

Figure 4-1: The Audio Stream Pipeline Stages in the End-To-End Delay

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

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

[3]The AudioFile utility supports a similar server as well.

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

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

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

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

$$\textit{worst–case end–to–end delay} = 3T + \textit{network propagation delay}$$

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

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

4.3. Bundling of Threads to a Reserve

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

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

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

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

4.4. Performance Measurements

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

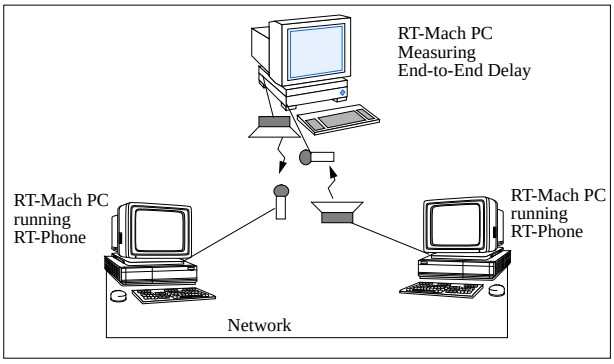

Figure 4-2: Measurement of Audio End-to-End Delay

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

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

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

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

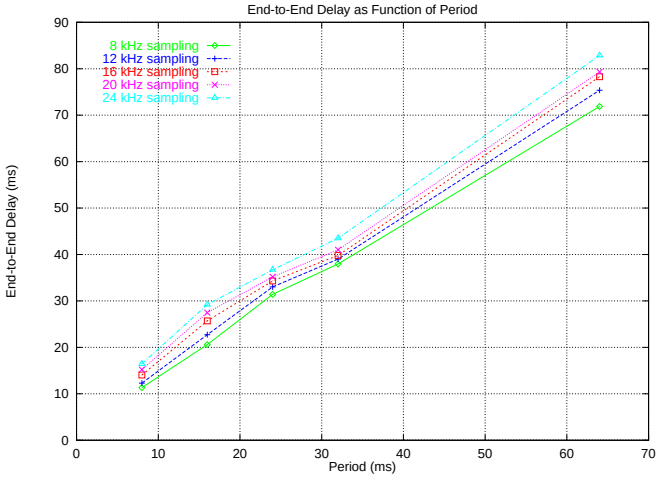

Figure 4-3: The End-To-End Delay w/ varying reservation periods and audio sampling rates.

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

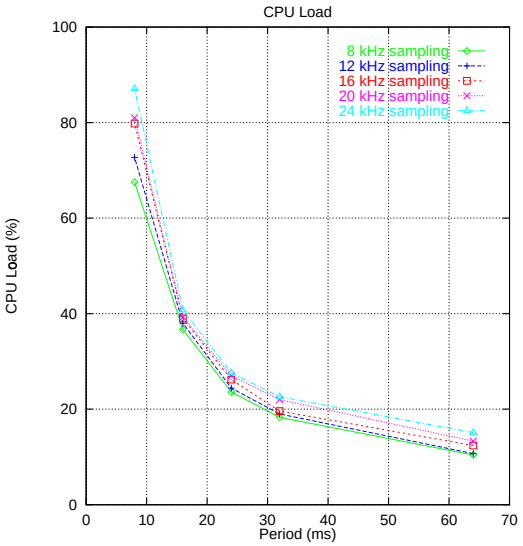

Figure 4-4: The CPU Load with different reservation periods and audio sampling rates.

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

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

4.5. QOS Interface for Changing Quality Parameters Dynamically

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

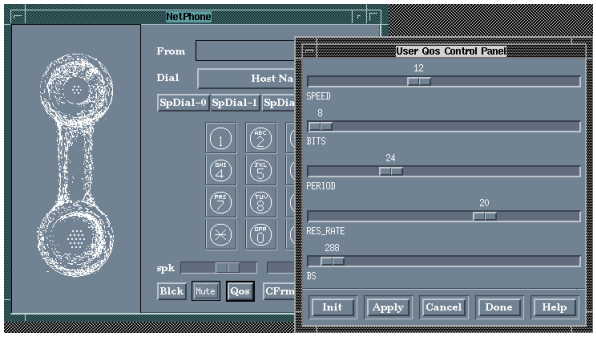

Figure 4-5: The interface to set system quality parameters.

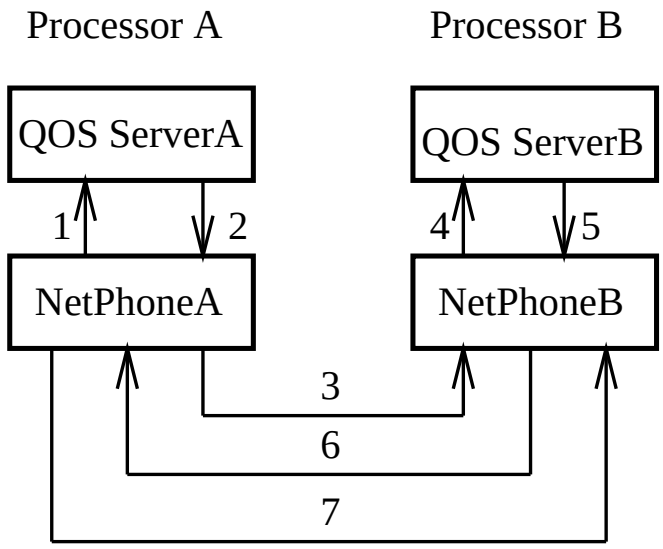

Figure 4-6: The Multi-Phase Protocol to *dynamically* adjust QOS during teleconferencing

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

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

5. Lessons Learned and Future Work

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

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

Acknowledgements

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

References

1. S. R. Ahuja, J. R. Ensor and D. N. Horn. The Rapport Multimedia Conferencing System. Proceedings of the Conference on Office Information Systems, March, 1988, pp. 1-8.

2. Rajkumar, R. "The Advanced Real-time Monitor: User Manual". *Real-Time Mach Project, Carnegie Mellon University* (November 1994).

3. A. Bricker, M. Gien, M. Guilllemont, J. Lipskis, D. Orr and M. Rozier. "Architectural Issues in Microkernel-base Operating Systems: the CHORUS Experience". *Computer Communication 14*, 6 (July 1991), 347-357.

4. D. Ferrari and D. Verma. "A Scheme for Real-Time Channel Establishment in Wide Area Networks". *8*, 3 (April 1990).

5. H. Gajewska, J. Kistler, M. S. Manasse and D. D. Redell. Argo: A System for Distributed Collaboration. Proceedings of the Second ACM International Conference on Multimedia, Oct., 1994, pp. 433-440.

6. K. Jeffay, D. L. Stone and F. D. Smith. "Kernel Support for Live Digital Audio and Video". (Nov. 1991), 10-21.

7. J. Kurose. "Open Issues and challenges in Providing Quality of Service Guarantees in High-Speed Networks". *ACM Computer Comm. Review 23*, 1 (January), 6-15.

8. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

9. Mercer, C. W., Savage, S. and Tokuda, H. "Processor Capacity Reserves: OS Support for Multimedia Applications". *IEEE International Conference on Multimedia Computing Systems* (May 1994).

10. C. W. Mercer and R. Rajkumar. Design and Evaluation of a Real-Time X Server. Tech. Rept. , ART Project, Carnegie Mellon University, Oct., 1995.

11. K. Nahrstedt and J. Smith. "The QOS Broker". *IEEE Multimedia Spring* (1995), 53-67.

12. Nakajima, T., Kitayama, T., Arakawa, H. and Tokuda, H. "Integrated Management of Priority Inversion in RT-Mach". *to appear in IEEE Real-Time Systems Symposium* (December 1993).

13. Zelenka, J. "The Audio Server: User Manual". *Real-Time Mach Project, Carnegie Mellon University* (November 1994).

14. *Adobe Premiere for Macintosh.* 1993.

15. Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991. ISBN 0-7923-9211-6.

16. P. Robin, G. Coulson, A. Campbell, G. Blair and M. Papathomas. Implementing a OoS Controlled ATM Based Communications System in Chorus. Technical Report MPG-94-05, Dept. of Computing, Lancaster University, March, 1994.

17. Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K. "Mode Changes in a Prioritized Preemptive Scheduling Environment". *The Real-Time Systems Journal* (December 1989).

18. D. B. Terry and D. C. Swinehart. "Managing Stored Voice in the Etherphone System". *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 3-27. Also available in Xerox PARC report CSL-89-2, May, 1989.

19. Tokuda, H. The Real-Time Mach Operating System. Carnegie Mellon University, Pittsburgh, PA, September, 1991.

20. H. M. Vin and P. T. Zellweger and D. C. Swinehart and P. V. Rangan. "Multimedia Conferencing in the Etherphone Environment". *IEEE Computer 24*, 10 (Oct. 1991), 69-79.

21. A. Vogel, B. Kerherve, G. Bochmann and J. Gecsei. "Distributed Multimedia and QOS: A Survey". *IEEE Multimedia* (1995).

22. L. Zhang, S. Deering, D. Estrin, S. Shenker and D, Zappala. "RSVP: A New Resource Reservation Protocol". *IEEE Network* (September 1993).

Table of Contents

List of Figures