QoS-Aware Resource Management for Distributed Multimedia Applications

Klara Nahrstedt, Hao-hua Chu, Srinivas Narayan*
Department of Computer Science
University of Illinois at Urbana Champaign
klara,h-chu3,srnaraya@cs.uiuc.edu

Abstract

The ability of operating system and network infrastructure to provide end-to-end quality of service (QoS) guarantees in multimedia is a major acceptance factor for various distributed multimedia applications due to the temporal audio-visual and sensory information in these applications. Our constraints on the end-to-end guarantees are (1) QoS should be achieved on a general-purpose platform with a real-time extension support, and (2) QoS should be application-controllable.

In order to achieve the users' acceptance requirements and to satisfy our constraints on the multimedia systems, we need a QoS-compliant resource management which supports *QoS negotiation*, *admission* and *reservation* mechanisms in an integrated and accessible way. In this paper we present a *new resource model* and a *time-variant QoS management*, which are the major components of the QoS-compliant resource management. The resource model incorporates, the *resource scheduler*, and a new component, the *resource broker*, which provides negotiation, admission and reservation capabilities for sharing resources such as CPU, network or memory corresponding to requested QoS. The resource brokers are intermediary resource managers; when combined with the resource schedulers, they provide a more predictable and finer granularity control of resources to the applications during the end-to-end multimedia communication than what is available in current general-purpose networked systems.

Furthermore, this paper presents the QoS-aware resource management model called QualMan, as a *loadable middleware*, its design, implementation, results, tradeoffs, and experiences. There are tradeoffs when comparing our QualMan QoS-aware resource management in middleware and other QoS-supporting resource management solutions in kernel space. The advantages of QualMan is that it is flexible and scalable on a general-purpose workstations or PC. The disadvantage is the lack of very fine QoS granularity, which is only possible if supports are built inside the kernel.

Our overall experience with QualMan design and experiments show that (1) the resource model in QualMan design is very scalable to different types of shared resources and platforms, and it allows a uniform view to embed the QoS inside distributed resource management, (2) the design and implementation of QualMan is easily portable, (3) the good results for QoS guarantees such as jitter, synchronization skew, and end-to-end delay, can be achieved for various distributed multimedia applications.

1 Introduction

With the temporal audio-visual and sensory information in various distributed multimedia applications, the provision of *end-to-end quality of service (QoS) guarantees* is a major acceptance factor for these applications. For example, multimedia applications such as video-conferencing require bounded *end-to-end delay* with a minimal jitter for meaningful audio and video communication. Video-on-Demand applications

*The work of all authors was supported by the NSF Career Award under the agreement number NSF CCR 96-23867 and the NSF CISE Infrastructure grant under the agreement number NSF CDA 96-24396.

require minimal *jitter* and *loss rate* to accomplish a good viewing quality of retrieved movie. Figure 1 shows a distributed multimedia system environment where we consider the end-to-end QoS issues.

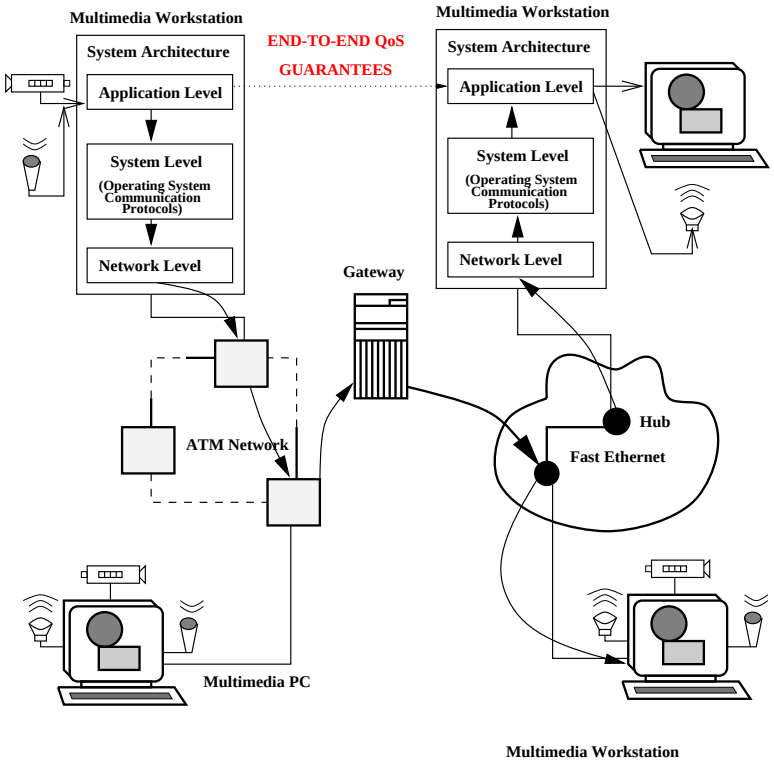


Figure 1: The End-to-End Scenario of Distributed Multimedia Applications

The environment consists of general-purpose workstations and PCs equipped with multimedia devices such as video cameras, microphones, and speakers. Our assumption about the general-purpose operating systems in these end-points is that they support *real-time extensions* with mechanisms such as priority scheduling and memory pinning, which are now available in most of the UNIX platforms and Windows NT platforms. The multimedia end-points are connected via local area networks such as ATM (Asynchronous Transfer Mode) and Fast Ethernet, which are currently widely available in academia and industry. One important issue about this general-purpose environment is that not all components along the end-to-end path (e.g., from video retrieval at the server workstation to video display at the client PC) have QoS support. For example, ATM network provides a QoS support (bandwidth reservation), but the end-points (workstations, PCs) do not have any specific support of QoS (the RT extensions are necessary, but not sufficient for QoS support). Our goal is to present a *solution* at the end-points of the end-to-end multimedia communication path which (1) contributes to end-to-end guarantees, and (2) allows the applications to access and control the end-to-end QoS parameters. We assume in this framework that the underlying network (e.g., ATM) has some capability of QoS provision such as bandwidth reservation and enforcement.

To achieve this goal, we utilize and build on our experience, knowledge and lessons learned during the design and experiments with the end-point OMEGA architecture and QoS Brokerage [NS96a, NS96b]. OMEGA architecture consisted of the QoS Broker, a centralized end-point entity for handling QoS at the edges of the network, and end-to-end communication protocols using resources negotiated by the broker. The QoS broker entity was integrating QoS translation, negotiation, admission control for every end-point

resource, and computation of a static scheduler, considering functional dependencies of the application. These functions were performed during the connection establishment phase. The enforcement of QoS relied only on usage of real-time priorities under the assumption that the application is well behaved, and the network is lightly loaded. Research around OMEGA architecture concentrated on QoS management and not on resource management. OMEGA did not provide any explicit reservation, enforcement, or adaptation mechanisms in case of QoS violation or degradation due to misbehaved applications or heavy load on networks.

The lessons learned from OMEGA showed us that QoS management is only a part of the end-to-end QoS solution and we need a powerful QoS-aware resource management when we want to provide end-to-end QoS guarantees. This leads us to new design, services, protocols and other significant changes in comparison to the our previous work within the QoS Broker and OMEGA architecture research:

- We split the functionality of the QoS Broker in the OMEGA architecture and distributed the individual QoS functions such as resource admission control and resource negotiation closer to the resource management. This distributed approach allows us provision of scalable solutions because different types of applications (local, remote) can be efficiently supported, hence not every resource is always involved.

- We left the central QoS broker at the end-point with translation functionality, and support for application QoS negotiation.

- We introduced a coordination protocol for reservation requests into the QoS Broker for reservation deadlock prevention during the resource reservation phase. At this point it is important to mention that this protocol evolved due to the step going from the centralized QoS brokerage approach to the distributed resource brokerage approach. In OMEGA architecture, the QoS Broker had all the information about the individual QoS and resource requests, hence could make immediate decisions about resource availability. In our new design, the QoS broker must communicate with the underlying resource management entities to obtain the resource availability and make the final reservation decision for the user.

- We designed and embedded reservation, monitoring, enforcement and partial adaptation mechanisms into our resource management entities so that QoS guarantees can be properly enforced in case of misbehaved applications or heavy loaded CPU/network.

- We designed the new QoS-aware resource management platform as a middleware in the user space which can be used independently by any application (local or remote) to receive QoS guarantees. The first design of OMEGA was not done with such an independence in mind.

- OMEGA provided only GUI (Graphical User Interface) API for QoS specification, where our new platform allows either GUI, command-line or system-based APIs for QoS specification and access to QoS services.

Our approach is to provide a distributed and QoS-aware resource management platform in form of a *loadable middleware* between the applications and the actual general-purpose operating system. Our new platform, called *QualMan*, consists of a set of resource servers using a new resource model and a robust time-variant QoS management, accessible to any application. The resource model incorporates, in addition to a resource scheduler, a new component, called the *resource broker*, which provides QoS, negotiation, admission, and reservation capabilities for sharing resources such as CPU, network, or memory according to QoS requirements. The resource brokers are intermediary resource managers which provide, together with resource schedulers, a more predictable and finer granularity control of resources to the applications during

the end-to-end multimedia communication than what is accessible in current general-purpose networked systems.

There are trade-offs when comparing our QualMan QoS-aware resource management in middleware and other QoS-supporting resource management solutions in kernel space:

- The **advantage** is that QualMan platform is *flexible*, and *scalable* at a general-purpose workstation or PC any time the end-point should be used for distributed multimedia applications. It is flexible because it allows the user to load and configure its general-purpose environment into a multimedia-supporting environment. The user starts the middleware and uses the API (Application Programming Interface) which allows the user to access and control the QoS offered by the middleware. It is scalable because it allows to provide QoS guarantees for local applications such as local MPEG players, or distributed applications such as Video-on-demand. The application requests from QualMan either CPU reservation only, or CPU and memory reservation only, or CPU, memory and network reservation all together, depending on the type of application.
- The **disadvantage** is the lack of very fine QoS granularity, which is particularly visible in the provision of timing constraints. The reason is that in order to achieve flexibility and load-ability for any platforms, there are no changes in the kernel. Hence the timing quality has lower resolution than if some of the algorithms were embedded in the kernel itself, where we would have access to much finer clock resolution. However, our achieved timing control is sufficient for multimedia applications and our results show that the middleware support provides much better temporal quality support than any application could achieve running on top of a general-purpose environment without our middleware.

In this paper we will presents the QoS and resource model as well as the placement of the QualMan architecture in the overall multimedia communication architecture in Section 2. This conceptual section will be followed by the description of individual elements of the QualMan architecture. Section 3 describes the CPU server, Section 4 presents the memory server, and Section 5 discusses the communication server. Section 6 presents the API to our QoS-aware resource management and other implementation details. Section 7 describes the results and experiences with the QualMan architecture. Section 8 discusses the related work. Section 9 concludes the paper.

2 QoS-Aware Resource Management Architecture

To achieve an end-to-end quality of service (QoS) along multimedia communication paths for distributed multimedia applications, we need to provide services and protocols in the end-points and networks which understand what quality of service is and how to map this quality into the required resource allocation. Furthermore, the underlying resource management must have services and protocols which know how to negotiate, admit, reserve, and enforce requested resource allocation according to requested QoS requirements.

In this section, we will present our QoS and resource model which will provide the basis for the QoS-aware resource management architecture (QualMan). Based on those models, we will give an overview of the QualMan architecture and its placement in the end-to-end multimedia communication architecture.

2.1 QoS Model

We consider parameterization of the QoS because it allows us to provide quality-controllable services. We will consider a *deterministic specification* of parameters, where the QoS parameters will be represented by a real number at a certain time t, i.e., $QoS : T \rightarrow R$ where T is a time domain representing the lifetime of a

service and R is the domain of real numbers. The overall quality of service will be specified either by a single value, by a pair of value such as QoS_{min} and QoS_{max}, or by a triple of value such as best value QoS_{max}, average value QoS_{ave} and worst value QoS_{min}. We will use the *single value* QoS_{ave}, or the *pair value* (QoS_{min}, QoS_{max}) specification in our service and protocol design. Particularly, the pair value specification will allow us to define range representation with *acceptable quality regions* $(QoS_{min} \leq QoS(t) \leq QoS_{max})$ and *unacceptable quality regions* $(QoS(t) < QoS_{min})$ as shown in Figure 2.

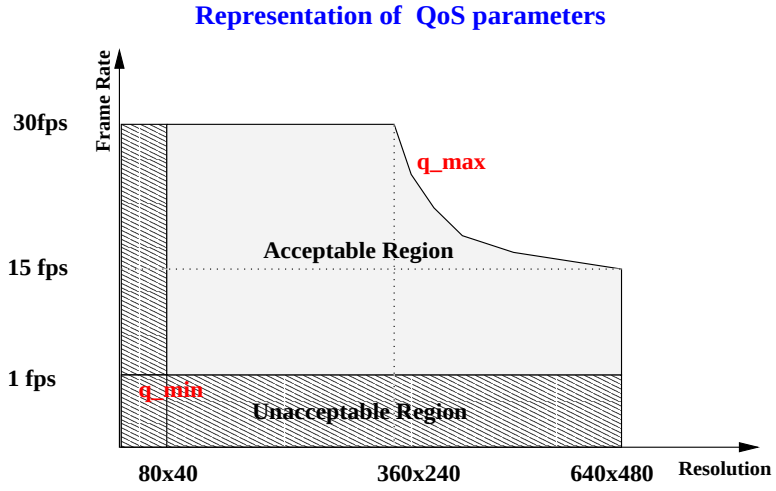


Figure 2: Range Representation of QoS Parameters. The figure shows two quality parameters, *resolution* of a video frame (X-axis) and *frame rate* of a video stream (Y-axis). The user/application specifies that receiving video frame rate of 1fps or below is *unacceptable* even if the resolution of the frame is very good. This specification determines the unacceptable region. Similarly, the user/application might specify that a video with a very small resolution below 80x40 pixels is not useful and we get another unacceptable region. The region above 1fps and 80x40 pixels defines the acceptable region. The upper right corner of the acceptable region is cut off which is determined by the maximal boundaries of the bare computer hardware/architecture. In our example, the hardware architecture cannot provide 30fps with the resolution 640x480 pixels.

There are many possible QoS parameters such as visual tracking precision, image distortion, packet loss rate, jitter of arriving frames, synchronization skew, and others. They can be classified from different aspects. One aspect we are considering is according to the layered multimedia communication architecture which consists of four main layers: users, application, system, and network layers[NS95b]. If we assume this type of end-point layering, then we can separate QoS into *perceptual QoS* (e.g., TV quality of video), *application QoS* (e.g., 20 frames per second video), *system QoS* (e.g., 50 ms period cycle) and *network QoS* (e.g., 16 Mbps bandwidth) classes. This classification allows each layer to specify its own quality parameters. However, this classification also requires translations at the boundaries between individual layers[NS96b]. Some examples of application and system QoS parameters for MPEG-compressed video streams are shown in Table 1.

In this paper we consider the *system QoS parameters* such as the CPU QoS, memory QoS, and communication QoS parameters when discussing the QualMan, the QoS-aware resource management platform. Furthermore, our focus will be on controlling *time-variant QoS parameters* such as the *jitter* (J_A) of arriving frames within a continuous media stream, which implicitly influences *synchronization skew* $(Sync_A)$ between two or more continuous streams, and *end-to-end delay* (E_A) between two end-points because they have the most significant impact on the acceptance of distributed multimedia applications.

QoS type	Specification	QoS parameter	Symbol
Application QoS	Processing requirements	Sample size	M_A
		Sample size (I, P, B)	M_A^I, M_A^P, M_A^B
		Sample rate	R_A
		Number of frames per GOP	G
		Compression pattern	G_I, G_P, G_B
		Original size of GOP	M_G
		Processing size of GOP	M_G'
		Degradation factor	D
	Communication	End-to-end delay	E_A
		Synchronization Skew	$Sync_A$
		Jitter	J_A
System QoS	CPU	Computation time	C
		Cycle time	T
		CPU Utilization	U
	Memory	Memory request	Mem_{req}
	Communication	Packet size	M_N
		Requested packet rate	R_N
		Requested bandwidth	B_N
		End-to-end delay	E_N

Table 1: Application and System QoS Parameters (Examples)

2.2 Resource Model

To provide QoS, each of the shared resources at the end-points must be modeled autonomously enough to provide its own QoS control as well as being able to adapt to possible occurrences of non-deterministic system changes/overruns on general-purpose systems. We extend the shared resource model with the brokerage functionality as shown in Figure 3. This general model allows us to provide a *uniform view* at any shared resource in a distributed multimedia system with QoS requirements[1]. The uniform resource view then allows for development of feasible heuristics algorithms to solve the distributed resource allocation problem which is otherwise NP-complete problem [BCSW86]. We provide piecewise solutions at individual resource servers such as algorithms for resource reservation and enforcement, and reservation protocols and coordination within communication protocols integrate the distributed resource servers in an end-to-end computing and communication environment.

The access to a shared resource is based on the *client/server model*:

- The general model of the *client* consists of two main parts: the *client broker* and the *client process*. The client broker requests and negotiates with the resource broker during the establishment or adaptation phase of a multimedia communication connection. The client broker specifies desired QoS_{des} (QoS_{ave} or $< QoS_{min}, QoS_{max} >$) parameters. The client process utilizes negotiated resources during the processing/transmission phase.

- The general model of the *server* provides equivalent services for controlling the time-variant QoS parameters: jitter (J), synchronization skew ($Sync$), end-to-end delay (E) and their adaptation

[1] Note, that in our previous work within OMEGA architecture we did not have this uniform resource model.

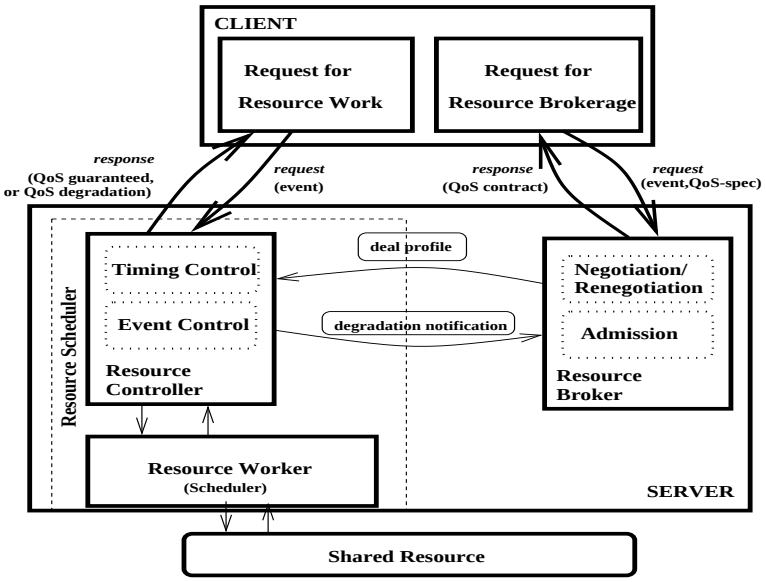


Figure 3: Resource Model with corresponding services. The client/server model for access to a resource is extended by the brokerage functionality which provides QoS negotiation, admission, and reservation capabilities.

to the clients requests. Upon the brokerage request, the client broker and resource broker negotiate/renegotiate a *QoS contract* between the client and server. The resource broker performs admission services to make decisions about resource availability. Note that in order for the resource broker to perform admission control, it must have the knowledge about the resource amount requested (e.g., processing time of a process/thread/task). If the client does not know the request amount, then it can acquire this information through the *probing service* [NHK96] done at the beginning of the application negotiation phase. This service determines statistical average of the requested resource amount and stores it in a *QoS profile*. The client relies and provides these values to the resource broker for admission control. The *resource scheduler* consists of two parts: the resource controller and the resource worker. The *resource controller* is invoked to control the *resource worker*. The controller gets the *QoS contract* which includes not only the parameters, but also a feasible scheduling policy satisfying timing and event flow control of resource usage. The resource broker communicates the information to the resource controller via a *contract profile*. Once the controller has the initial information, it takes over and issues appropriate schedulable units[2] to the resource worker according to the control policy. Furthermore, the resource controller is responsible for QoS monitoring and possible adaptation if short-term QoS variations occur. Larger QoS variations are communicated to the resource broker which decides further processing according to rules specified by the client.

Timing and event scheduling control within the resource controller provide control for the jitter and synchronization skew. They are derived from continuous media QoS requirements, and from client's program specifying timing and other events during the lifetime of a client(parsing of client's program during the pre-processing phase)[3]. The timing and event graphs are a general representation of resource access behavior and they allow the resource servers to make predictions of application behavior, hence they pro-

[2]Schedulable units are packets, scheduled in the network, processes, scheduled by the operating system, or disk blocks scheduled by the disk controller.

[3]Current design and implementation of QualMan derives the timing and event scheduling control from continuous media QoS requirements only.

vide customized scheduling, which leads to the capability of QoS provision. Figure 4 shows an example of an event and time flow control.

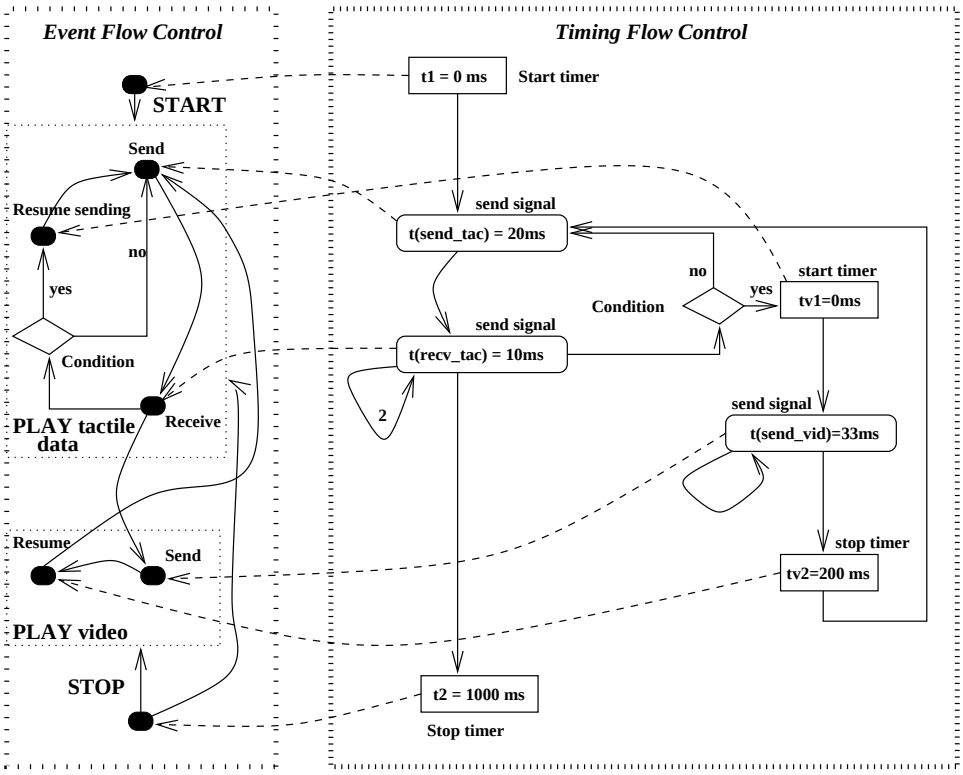


Figure 4: Local event and time flow control. The solid lines represent the transitions from one state to another within the individual flow control. The dashed lines represent the time signaling of a corresponding event.

The monitoring and adaptation between the resource controller and worker create a closed feedback loop which provides a basic functionality for the adaptation capability.

2.3 QualMan's Architecture and Placement in Multimedia Communication Architecture

The above described resource model has implications for the overall multimedia communication architecture. We can apply this model to each layer of the end-system (application, system, and network) where individual brokers and resource controllers communicate with each other and create an integrated end-to-end solution (see Figure 5). The network brokers and network protocols provide the lowest level of QoS provision. They are responsible for the low level end-to-end network quality guarantees. The resource brokers and resource schedulers in the system level provide control of local end-point resources such as CPU, memory, and disk as well as communication entry points to the networking environment. The application broker and scheduler can handle application-specific quality control and respond to the results of the lower level resource allocation.

In our further refinement of the end-point architecture, the system layer will be divided into the *QualMan middleware* (our QoS-aware resource management platform) and the core *OS kernel* shown in

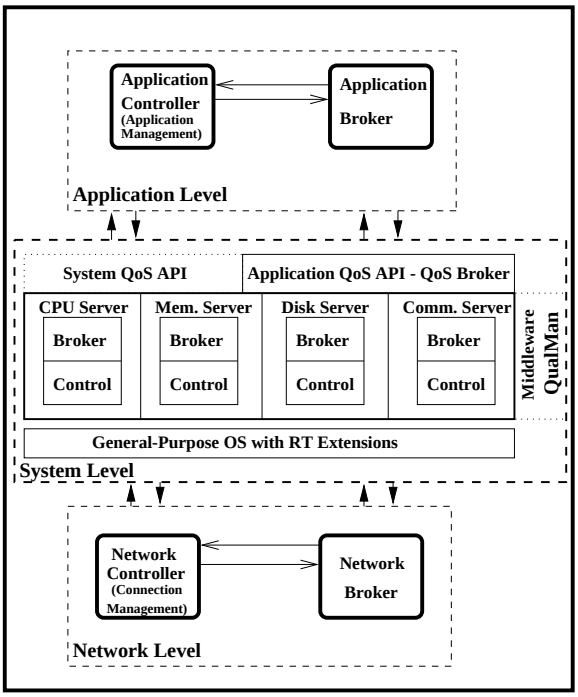


Figure 5: Multimedia Communication Architecture with a Detailed View on System Layer-Middleware

Figure 5. The middleware can interface with the application level through the *application-system interface* using either the *application QoS API*, or through the *system QoS API*. The middleware itself consists of *resource servers* (CPU, memory, communication, and disk). In this paper we will discuss the resource servers[4] design, implementation, and their results as well as on the system QoS API.

Before we go into the details of the individual resource servers of the QoS-aware resource management, it is important to point out the complexity of the *application QoS API* interface between the application and system layers. The interface is implemented by the *QoS broker* and it incorporates several functionalities such as the *translation* between the application QoS and system QoS parameters, *negotiation* protocols between the individual resource servers at the local site and the remote site, and *resource reservation coordination* to avoid/detect deadlocks. The translation service allows each domain (application or system) to express the QoS parameters in its own language. The negotiation protocol at this level needs to implement negotiation between the QoS broker and the resource brokers, as well as negotiation between the distributed QoS brokers to get the results of negotiation/reservation of resources at the local and remote sites. The resource reservation coordination needs to coordinate the reservation of resources so that deadlock can be avoided (apply Banker's algorithm[SG94] to request and reservation edges) or it can be detected and resolved. Figure 6 shows a possible deadlock scenario between processes P_3 and P_6, where P_3 has reserved disk resource and waits for CPU reservation, and P_6 has allocated CPU resource, but waits for disk resource which is contracted to P_3. The resource coordination needs to rely on robust policies to *satisfy* some reservations in case of resource contention and to *hold* resources for committed reservations.

[4]We will concentrate on CPU, memory and communication servers, because these are currently the most significant components in our system. The disk is local, hence the CPU and memory control of accessing the files on the disk are sufficient to achieve good access times to the disk. However, we are working on a more elaborate disk server in case the disk resides remotely.

Due to the limit on the length of this paper, we will omit a detailed description of this interface and refer the reader to our papers [NS95a, NS96b, KN97].

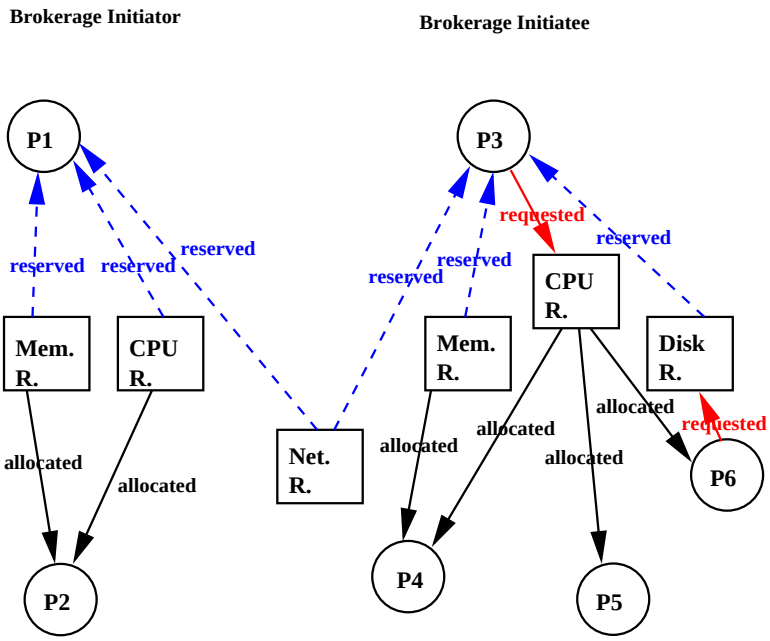


Figure 6: Resource Reservation and Allocation Graph comprising a Deadlock Situation at the Broker Initiatee Site

In summary, the QoS broker provides an integrated and automated translation when accessing QoS-aware resource management. Our final goal is to make the QoS broker together with the underlying QualMan *CORBA-compliant*. This functionality will allow users to achieve end-to-end QoS guarantees within the CORBA framework.

3 CPU Server

The CPU server[5] provides a QoS control to the application over the shared CPU resource. It differentiates during its processing among *waiting real-time(RT) processes* which wait to be scheduled, *active RT processes* which are currently scheduled, and *time sharing(TS) processes*. The passive and active RT processes are scheduled by the CPU server, and the TS processes are scheduled by the UNIX scheduler. The CPU server architecture is modeled closely according to the resource model described in Section 2, and it contains three major components–the *resource broker*, the *dispatch table*, and the *dispatcher*. The dispatcher is equivalent to the resource scheduler and its two parts (controller and worker), as shown in Figure 3. They are integrated into a single entity in the current CPU server. The reason is that the timing and event control in the current CPU server consists of periodic timer interrupts at the boundaries of constant size time slots. Each component is described in details in the following subsections. In addition, we describe *probing/profiling* which is used to provide a good estimate of task processing time used for the reservation.

[5]Early version of the CPU server was published in IDMS'97 proceeding [CN97]

Slot Number	Time	Process PID
0	$0 - 10ms$	721
1	$10 - 20ms$	773 774 775
2	$20 - 30ms$	721
3	$30 - 40ms$	free

Table 2: A Sample Dispatch Table.

3.1 Broker

The resource broker receives requests from client RT processes (client's broker). It performs the admission control test: $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, where C_i is the execution time, and T_i is the period (cycle time) of the $i-th$ client process. This determines whether a new client process can be scheduled. If it is schedulable, the broker will put the RT process into the waiting RT process pool by changing it to the waiting priority. The broker also computes a new schedule based on a desirable scheduling algorithm, the new schedule is written to the *dispatch table*.

The broker process is a root daemon process running at a normal dynamic priority. It can be started at the system boot time, like any other network and file system daemons. It will wake up when the new client request arrives. The broker needs to be a root process so that it can change processes into the fixed RT priority. The broker process does not perform the actual dispatching of the RT processes, instead it will fork a separate real-time *dispatcher* process. The reason is that the admission and schedulability test in the broker may have variable computation time, hence it may affect the timing of dispatching. The admission and schedulability test do not need to be done in real time; as a result, the broker runs at a dynamic priority. The separation of RT dispatching in the *dispatcher* process and the *schedulability test* in the TS *broker* process is an essential feature that allows both *dispatcher* and *broker* to do on-line computation without compromising the precision of RT processes dispatching.

The client RT processes must start their processing at the TS dynamic priority level. The broker and the dispatcher will change them into the fixed RT priority when they are accepted and dispatched. This is an improvement over the current UNIX environment, because our scheme allows any user to run processes at the fixed priority in a fair and secure manner.

3.2 Dispatch Table

The *dispatch table* is a shared memory object where the *broker* writes the computed schedule to and the *dispatcher* reads from in order to know how to dispatch RT processes. It is locked inside memory for efficient reading and writing. The dispatch table contains a repeatable *time frame* of slots, each slot corresponds to a time slice of CPU time. Each slot can be assigned to a RT process pid, a group of cooperating RT process pids, or be free which means yielding the control to the UNIX TS scheduler to schedule any TS processes. Let us consider the example in Table 2. The repeatable *time frame* for all accepted RT client processes is $40ms$ ($GCD(T_{721}, T_{773,774,775})$), and it contains 4 time slots of $10ms$ each. The sample dispatch table is a result of a rate-monotonic(RM) schedule with the process pid 721 at *period*=$20ms$, *execution time*=$10ms$, and process pid 773/774/775 at *period*=$40ms$, *execution time*=$10ms$. There is one free slot, which means $10ms$ out of every $40ms$ of CPU is allocated to the TS processes.

The minimum number of free slots is maintained by the broker to provide a fair share of CPU time to the TS processes. In the Table 2, 25% ($10ms$ out of $40ms$) of the CPU is guaranteed to the TS processes. The site administrator can adjust the TS percentage value to be what is considered fair. For example, if

	Priority	Process
RT class	highest	Dispatcher
	2nd highest	Running RT process
	..	Not used
TS class	any	Any TS processes
RT Class	lowest	Waiting RT processes

Table 3: Priority Scheduling Structure.

the computer is used heavily for RT applications, the TS percentage can be set to a small number, and vice versa.

3.3 Dispatcher

The *dispatcher* is a periodic server (process) running at the highest possible fixed priority. The dispatcher process is created by the broker and it is killed when there are no RT processes to be scheduled in the system. When there are only TS processes running, the system has no processing overhead associated with the RT server.

The dispatcher contains the shared memory dispatch table and a pointer to the next dispatch slot. At the beginning of the next dispatch slot, a periodic RT timer signals the dispatcher to schedule the next RT process. The length of time to switch from the end of one slot to the start of the next one is called the *dispatch latency*. The dispatch latency is the scheduling overhead which should be kept at a minimal value.

The dispatcher is based on the following priority scheduling [KYO96]. The dispatcher runs at the highest possible fixed-priority, the waiting RT process waits its scheduling turn at the lowest possible fixed-priority (called the waiting priority), and the active RT process runs at the 2nd highest fixed-priority (called running priority). The priority structure is shown in Table 3. The dispatcher wakes up periodically to dispatch the RT processes by moving them between the waiting and the running priority; during the other time, it just sleeps. When the dispatcher sleeps, the RT process at the running priority executes. When no RT processes exists, the TS processes with dynamic priorities execute using the fair time sharing scheduler of UNIX. This provides a simple mechanism to do RT scheduling in UNIX. It also has many desirable properties which other approaches such as the *processor capacity reserves* [MT94] do not provide:

- It requires no modification to the existing UNIX/POSIX.4 kernels. The scheduling process can be implemented as an user-level application.
- It has very low computation overhead.
- It provides the flexibility to implement any scheduling algorithms in the scheduler, e.g., rate monotonic, earliest deadline, or the hierarchical CPU algorithms.

We will demonstrate the scheduling policy of the dispatcher using the following example. Let us consider the dispatch table in Table 2 with time slot starting at $10ms$. The dispatcher is moving from slot 0 to slot 1, and the following steps are taken.

- The periodic RT timer wakes up the dispatcher process, and the process 721 is preempted (1 context switch).

- The dispatcher changes the process 721 to the waiting priority and processes 773/774/775 to the running RT priority (4 system calls to set priority).
- The dispatcher puts itself to sleep, and one of the processes 773/774/775 is scheduled (1 context switch).

The program code segment that corresponds to the above steps is executed repeatedly, and is locked into memory to avoid costly page faults. The dispatch latency can be bounded by the time to do 2 context switches and (the maximum number of processes in any 2 adjacent slots) set-priority system calls.

In our real time programming model, we require the RT process to mark the end of its execution within a given period using our yield() API call. The yield() call generates an event to the dispatcher. Like the signal from the periodic timer, the event wakes up the dispatcher to make a new scheduling decision. We define a process *underrun* within a period as a state in which the process finishes before using up all its reserved slots. It is detected when the dispatcher receives the yielding event from the RT process prior to the end of its reserved slots. When a process underrun occurs, the dispatcher will assign its remaining reserved slots to TS processes. At the start of its next period, the under-running process will be scheduled again by the dispatcher in its reserved slots.

We define a process *overrun* within a period as a state in which the process does not finish after using all its reserved slot. It is detected when the dispatcher does not receive the yielding event from the RT process at the end of its reserved slots. When a process overrun occurs, the dispatcher will not allow the over-running process to consume more time slots. Instead the RT process is demoted as a TS priority process until the start of its next period. Since the dispatcher will not allow any RT processes to use more than their reserved slots, the reserved processing time of the RT process is guaranteed and protected from potential overruns of other RT processes.

Our scheduler allows the application to query the amount of processing time that it has consumed in its current period and its previous period. The application will know if it is having underrun or overrun. If the application is experiencing constant overruns or underruns, it can re-negotiate to increase or decrease its reservation so that its reserved processing time matches its actual consumed processing time.

3.4 RT Clients and Probing/Profiling

Our client's system QoS request has a form of QoS specification: *period=T, CPU utilization in percentage=U*, where $U = \frac{C}{T} \times 100\%$. For example, the specification $(T = 100ms, U = 40\%)$ means that $40ms$ out of every $100ms$ is reserved for this RT process. The QoS specification can be generalized to be in a form of a time graph as shown in Figure 7.

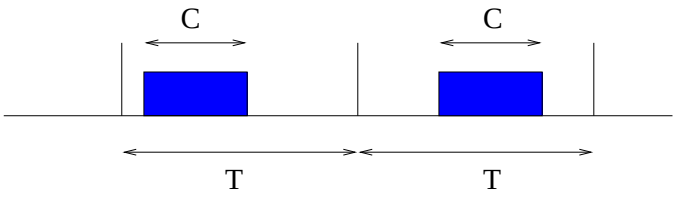


Figure 7: Time Graph

Given that our CPU server can provide a scheduling mechanism to guarantee processing time through a reservation, the application programmers still face the formidable task of figuring out exactly how much

processing time C to submit in a reservation. Since the application is usually written to be platform independent, it can be compiled and run on a variety of hardware platforms and operating systems. Hence, it is impossible to hard-code a fixed C value into the program. For example, the average processing time to decode one MPEG frame differs significantly between a SUN Sparc 10 machine and a much faster SUN Ultra Sparc machine.

Probing allows the client applications to get an accurate estimation of how much processing time to reserve, prior to making a reservation. During the probing phase, we run a few iterations of the application with no CPU reservation and we measure the actual CPU usage. At the end of the probing phase, we compute the average usage time from the measurements as our probed processing time. The processing time is then recorded in a *QoS profile* associated with the application running on that particular hardware platform. For example, we may have a profile called *mpeg_decoder.profile* with the following entries: (platform=Ultra-1, resolution= 352x240, C=40ms), and (platform=SPARCstation-10, resolution= 352x240, C=80ms). With the probed values in the profile, the client application can compute the CPU utilization $U = \frac{C}{T}$ to make the reservation.

The computation of the period T is as follows: $\frac{1}{R_A}$ (e.g., $R_A = 40$ frames per second video player has a period of $T = 25ms$). There is a restriction on the lower bound of the period size, which cannot be smaller than the resolution of the system periodic timer. Smaller period leads to smaller time slice, which may result in higher number of context switchings and inefficient CPU utilization.

4 Memory Server

The execution time of client's RT process also depends on the state of memory contention and the resulting number of page faults. We designed a *memory broker* where the RT process can reserve memory prior to their RT execution.

The memory server consists of the *broker* and the *memory scheduler* according to the resource model in Figure 3. The memory server is a root process that can be started at the system boot up time. It is initialized with a parameter called *global_reserve*, which is the maximum amount of pinned memory (in bytes) that the server can allocate to RT processes. The *global_reserve* should be chosen carefully so that it does not starve the TS processes and the kernel. The server waits for requests from RT processes.

The RT process begins with the reservation phase. It contacts the memory broker to try to establish a *memory reserve* with a specified amount of memory request in bytes. The reserve should be an estimated amount of the pinned memory that the process needs in order to satisfy its timing requirement. It should include all its text, data, and shared segments. Once the memory broker receives the request, it performs the following admission test: $Mem_{req} \leq Mem_{avail}$, i.e., $Mem_{req} + \sum_{j=1}^{k} Mem_{acc_j} \leq Mem_{glob-resv}$ to check that the incoming request for memory reserve Mem_{req}, added to the already accepted memory reserves $\sum_{j=1}^{k} Mem_{acc_j}$, does not exceed the `global_reserve` $Mem_{glob-resv}$. If the admission test succeeds, the memory broker returns a reserve id (`rsv_id`) to the process, and it creates an entry in its table (`rsv_id`, Mem_{acc}). The process should then lock its text segment using the reserve rsv_id.

During (or prior to) the execution phase, the process can send the memory controller a request `(rsv_id, size)` to acquire the pinned memory data allocation (e.g., `malloc()`). Once the request is received, the server checks whether there is enough reserve to satisfy this request. If so, it decreases `size` bytes of memory from the reserve `rsv_id`. The server then allocates the pinned memory in the form of *shared memory* to the process. The server creates a shared memory segment of `size` using `shmid = shmget(key, size)` and locks it using `shmctl(shmid, SHM_LOCK)`. The shared memory key is then passed to the process which attaches the shared memory segment into its address space.

When the process wants to free its pinned memory, it detaches the shared memory segment and sends a request containing the shared memory key to the memory server. Then the server destroys the share

memory segment and increases the corresponding memory reserve.

We choose not to apply the probing and adaptation in our memory server because the application programmer can usually determine the actual amount of memory the process needs throughout its runtime. However, we do allow the process to increase or decrease the amount of its memory reservation, but there is no system initiated monitor and adaptation as in the case of CPU reservation.

4.1 Relation between Processes and Memory Reserves

The relationship between the memory reserves and processes can be many to many. A process can establish multiple reserves to protect memory usage among various parts of the same program. For example, a distributed video playback application can assign separate reserves for its display, decoded, and network buffers. It will restrict the growth of some buffers that use pinned memory. Multiple processes can also share the same reserve. For example, a distributed video playback application may require services from the network, decoder, or display processes (or modules/drivers) which can charge their memory usage to the application's reserve.

The underlying shared memory implementation also helps to eliminate the copying overhead when various processes need to pass data around. Consider a network module that assembles packets into frames and passes the frames to the decoder process. The network module and the decoder process can establish a joint memory reservation and create a common shared memory region. The network module charges the reserve for every new frame it uses, the decoder process gets the frames through the shared memory region without copying.

4.2 Small Data Allocation

For small data `MEM.alloc()`, e.g., 20 bytes, it is too expensive to create a separate shared memory segment. There is an adjustable parameter called `MIN_ALLOC_SEGMENT`, which is the minimum size of the shared memory segment size. The default `MIN_ALLOC_SEGMENT` is the page size, which is 8K on the SUN workstation. All the MEM.alloc()s which are less than 8K are grouped together into a shared memory segment of size 8K. For example, the first 20 bytes of a MEM.alloc() request will create (be charged) a 8K shared memory segment with the 20 bytes allocated to this request. The subsequent small requests will be allocated within the same shared memory segment. To reduce fragmentation within the shared memory segment, we use a resource map memory allocator with the First Fit algorithm.

Parameters	Default Value	Suggested Value
shmsys:shminfo_shmmax	1MB	5MB
shmsys:shminfo_shmmni	100	1000
shmsys:shminfo_shmseg	6	100

Table 4: Shared memory parameters in the system configuration file.

In the Solaris Operating System, there are several system tunable parameters that place limitations on the various aspects of the shared memory segments as shown in Table 4: `shmsys:shminfo_shmmax` is the maximum size of a shm segment, `shmsys:shminfo_shmmni` is the maximum number of shm identifiers, and `shmsys:shminfo_shmseg` is the maximum number of shm segments per process. Their default value is too small. The system administrator can increase these parameters to the suggested value in the table by modifying the `/etc/system` configuration file.

4.3 Limitations

There are several limitations in our shared memory implementation of the the memory reserve. The first one is that the memory reserve covers only the text and data segments, but not the stack segment. We have found that it is difficult to monitor and manage the stack segment without modifications inside the kernel. In a typical program, its stack segment is usually much smaller than its text or data segments. Therefore, it is unlikely that the stack segment will get swapped out.

The second limitation is with the data allocation in the linked/shared library. Users can not modify the data allocations in the linked libraries (e.g., X library) to call our memory reserve routines. These data segments in these libraries are not pinned nor accounted for in the reservation.

We have chosen the shared memory implementation because it can be done at the "user-level" and without modifications in the kernel. These limitations can be overcome with another choice of implementation which involves modifications to the virtual memory system. However, this would mean a defeat of the desired loadable capability which our current middleware has.

5 Communication Server

Similarly to the CPU and memory servers, the *communication server* consists of two components according to the resource model in Figure 3.: the *communication broker*, which admits and negotiates the network QoS and the *multimedia-efficient transport protocol (METP)*, which enforces the communication QoS at the end-points and propagates the ATM QoS parameters/guarantees to the higher communication layers.

5.1 Communication Broker

The communication broker is a management daemon which in conjunction with the transport protocol provides QoS guarantees required by the distributed multimedia application. The broker performs service registration, admission control, negotiation, connection setup, monitoring and adaptation as follows:

5.1.1 Service Registration

The multimedia application (RT client) is required to register with the communication broker and to specify a name identification, the type of data being transmitted, and the quality parameters requested from the connections.

The parameters which the communication broker needs from the RT client for further decision making are the *peak, mean, and burst bandwidth* ($B_{peak}, B_{mean}, B_{burst}$), *size* of the application protocol data unit (APDU) M_A, *end-to-end delay* E_A, *specification of data flow* either simplex or duplex, *reliability* enforcement either total or partial, and *timeout* duration t_{out} which specifies how long to wait for a PDU or for an acknowledgment in our reliability mechanism. The broker tabulates these information and sets up a message channel for future communications with the RT client (application). This channel is used to inform the RT client of incoming connections, as well as to send messages about upgrading or degrading the requested communication QoS.

5.1.2 Admission Control and Negotiation

Once the application specifies its communication QoS parameters at the time of connection setup, the broker performs checks to verify that the parameters can be guaranteed. The admission control mechanism, using an admission condition, decides if the requested QoS can be met or suggests a lower achievable value. The communication broker performs admission on bandwidth availability and end-to-end delays.

For communication *bandwidth availability*, the admission condition is $\sum_{i=1}^{k} B_{acc_i} + B_{req} \leq B_{HI}$, where B_{acc_i} is the accepted bandwidth for the i-th connection and B_{req} is the requested bandwidth of the new connection[6].

The *end-to-end delay* depends on a number of factors such as the application PDU size, load on the network, loads on the end hosts, and the bandwidth reserved for the connection. Admission control for end-to-end delay is performed using a *profiling scheme*. A QoS profile of the end-to-end delays for various APDU sizes is created (measured off-line) and used as the seed[7].

APDU size (KB)	EED (ms)
20	8
50	17
80	21
110	28
140	35
170	39
200	48
230	56

Table 5: EEDs for different APDU sizes

The *APDU end-to-end delay (E_A)* is defined as the time required for one APDU to completely reach the destination from the source. Due to missing of information about the packet service time in intermediate components, it is difficult to get a precise E_A. Hence we estimate the E_A as follows. Let t_1 be the time at which the source starts sending the Transport PDU (TPDU)s belonging to the considered APDU. Let t_2 be the time at which an acknowledgment for the APDU is received by the source. The time interval $(t_2 - t_1)$ is the time required for the entire APDU to be sent across (APDU EED) and the time E_N required for an acknowledgment PDU to be sent from the destination to the source (TPDU EED)[8]. The E_N can be estimated using the standard round trip time measurement algorithm which involves sending one TPDU across, receiving an acknowledgment for it, and dividing the result value by two. Hence, the APDU EED is given by $E_A = (t_2 - t_1 - E_N)$, where E_N represents the end-to-end delay of the group acknowledgment.

When the user supplies the APDU size and an end-to-end delay requirement, the APDU size is matched with the closest larger size in the table, and the end-to-end delay value specified is checked against the value in the profile. If the user specified value is greater than the value in the profile, the network admission control is passed.

For the *CPU bandwidth and memory availability* in METP, the communication broker contacts the CPU and memory servers. The communication broker needs to have an information about the processing time C and size M_A corresponding to the processing of APDUs in the transport tasks (e.g., segmentation of APDUs to TPDUs, header creation, movement of PDUs) in METP. The period T of the transport tasks is derived from the frame rate R_A. The broker gets the size M_A from the user who knows the size of the

[6]The bandwidth actually represents the bandwidth specification calculated from the application stream characteristics plus the header overheads coming from the transport protocol and from the AAL/ATM layers. The reason is that the B_{HI} bound is the bandwidth achieved at the ATM layer. The achieved bandwidth in the user space is possible to determine, but it depends on the actual CPU load and CPU bandwidth availability for communication activities in the end-point. Hence it is not a reliable upper-bound.

[7]This profile is strongly platform dependent. Table 5 shows measurement using our ATM/SPARC 10 platform and we use the table as an example to show the profiling concept.

[8]As we describe later, our acknowledgment scheme uses group acknowledgment which is sent after all TPDUs, belonging to the considered APDU, are received.

APDU to be sent out. The processing time C of APDUs within transport tasks is acquired by the probing service as discussed in Section 3. During the CPU probing time, the CPU broker monitors the processing times of the transport tasks and stores them in a corresponding QoS profile. The processing time includes the time of METP tasks after receiving APDU by METP to send the segmented TPDUs in a burst every $T_A = \frac{1}{R_A}$. The communication broker reads the QoS profile of the processing time and uses the information to get reservation from the CPU broker for the transport tasks.

5.1.3 Connection Setup

Connection setup includes negotiation between the communication brokers of remote entities. When the negotiation is done, the connection is established using the ATM API for setup of its VC and QoS parameters.

The connection setup request to the communication server is initiated from the RT client (application). The connection setup protocol is shown in the Figure 8. First, the initiator RT client sends the *register request* along with its QoS parameters. Second, the initiator broker performs admission control on the specified QoS. If the admission test is successful, the initiator broker will send an *open connection* request to the initiatee side (broker), which also performs an admission test. If the admission test is also successful at the initiatee side, the initiatee RT client will accept the connection setup, and the protocol will respond with an *accepted allocation* message. Upon receiving the accept message the initiator side completes the connection setup.

The communication broker holds a table with connections and reserved/accepted QoS parameters. The number of supported connections at the end system is bounded by the available CPU and network bandwidth. Once the connections are admitted, the CPU server takes over the connection scheduling. The CPU and bandwidth allocation are guaranteed, and the CPU server allows for timely switching among individual connections. Note that the connections are not multiplexed at the METP level because the QoS of individual connections would be lost from the multiplexing [Fel90]. Hence each connection has its own CPU reservation. The multiplexing of different connections occurs at the ATM level in the device which is out of the CPU server responsibility. Hence, the proper CPU reservation for transport tasks processing individual connections will enforce timely traffic shaping into the ATM device as well as reception of data out of the ATM device.

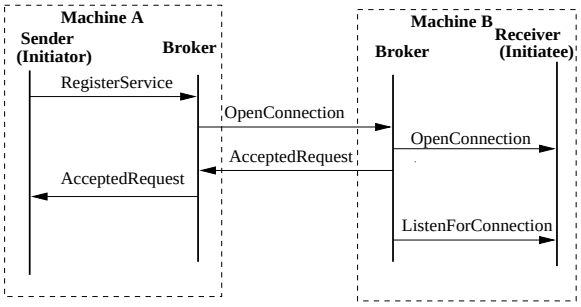


Figure 8: Connection Setup Protocol

We also provide a possibility of *partial acceptance* when the initiatee does not have available requested resources for end-to-end delay provision, and it sends back a message with *partially fulfilled* content (only bandwidth guarantees are given). The initiator of the QoS connection decides if this is sufficient. If this is

the case, *accepted allocation* message is sent back to the initiatee, and a connection opens at the initiatee side with degraded quality.

The third possibility is to send out a *reject request* message when bandwidth and EED tests are both violated. When that happens, the initiator must wait until the requested resources become available again.

5.1.4 Monitoring and Adaptation

Monitoring and adaptation are needed in order to allow for upgrading and degrading in the quality of connections. A monitoring thread examines the amount of available resources whenever a connection is closed. It checks if the freed resources can be used to satisfy any partially fulfilled connections. When such a connection is identified, the monitoring thread sends a message to its application over the register channel and informs it about the possible upgrade.

5.2 Multimedia-Efficient Transport Protocol

The communication server includes a thin layer of transport service support. For support of jitter and other temporal QoS requirements, this multimedia-efficient transport extension requests an appropriate amount of CPU bandwidth and memory from the CPU and memory servers so that its transport tasks can move and process TPDUs in a predictable fashion. Furthermore, this protocol expands the native ATM mode (AAL API) to provide efficient reliability capability which is not provided by the AAL layer and enforces optimal movement of data through the transport extension.

The architecture of the transport layer is depicted in Figure 9. The protocol is described in two sections for the sending side and the receiving side.

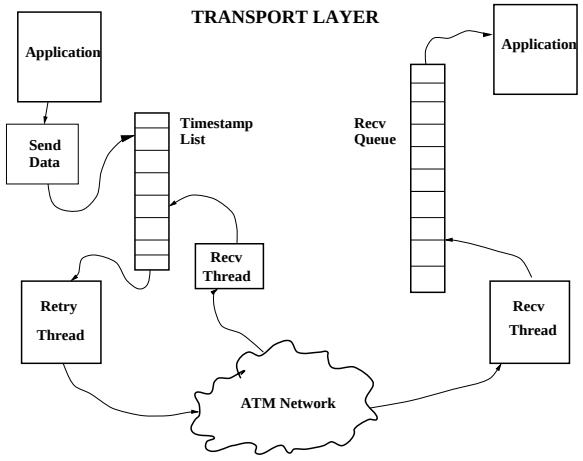


Figure 9: Components of the Transport Layer

5.2.1 Send Protocol

The application data is segmented into TPDUs. The size of the TPDU is configurable. Each TPDU has a header section and a data section. In traditional transport layers, memory for the TPDUs is allocated afresh in kernel space and the application data is copied into the newly created TPDUs, which contain additional space for headers. In our transport layer, a simple but efficient scheme is used to achieve a

zero copy send (above the device driver level). Since memory for the data has already been allocated by the application, the same memory can be used to store the headers too. The basic idea is to locate the beginning of each TPDU in the application chunk and to overwrite the preceding bytes with the header of the TPDU. Those few bytes are backed up beforehand and can be accessed if the previous TPDU needs to be retransmitted. This scheme avoids a copy of the entire application chunk. The size of the header is usually small compared to the size of the data in the TPDU[9]. To give an example, the maximum amount of data that can be sent in one TPDU is 64 kilobytes and the size of the header is a fixed 24 bytes.

The sending side functions as follows :

1. The sending function locates the beginning of each TPDU and overwrites the preceding bytes with the header of the TPDU. The TPDU thus formed is transmitted. Information about each transmitted TPDU is stored in a list. This list is used to retrieve information if any TPDU needs to be retransmitted. The information stored includes:

   - the location of the TPDU within the APDU
   - the time-stamp corresponding to the sending time of the TPDU
   - the size of the TPDU
   - statistical information such as the number of retransmissions

2. After all PDUs in the APDU have been transmitted once, the sending side waits for a response from the receiver. The response could be one of the following:

   (a) A *group positive acknowledgment (GPACK)*: A GPACK is received (recv function) if all the TPDUs sent have reached the receiver successfully. When a GPACK is received, all TPDUs in the range of sequence numbers specified in the GPACK are removed from the list of unacknowledged PDUs.

   (b) A *group negative acknowledgment (GNACK)*: A GNACK is received when one or more of a sequence of TPDUs does not reach the receiver. When a GNACK is received, all the TPDUs in the range of sequence numbers specified in the GNACK are retransmitted. The time of retransmission and the number of retries are updated.

3. When a timeout (t_{out}) occurs, the sending side checks to see if all the TPDUs in APDU have been acknowledged. If there are unacknowledged TPDUs, there are two possible scenarios:

   - The pessimistic scenario is that all unacknowledged TPDUs were lost during the transmission, and they all need to be retransmitted.
   - The optimistic scenario is that some or all of the TPDUs reached the receiver, but the acknowledgment sent by the receiver was lost.

   In order to save time and bandwidth, the transport layer first assumes the optimistic scenario and retransmits only the first unacknowledged TPDU. If the TPDU has reached the receiver along with some or all of the other TPDUs, the receiver sends out a GPACK. A GPACK contains a pair of sequence numbers defining a range of TPDUs which have reached the receiver. On receiving a GPACK,

[9] There are tradeoffs using this scheme. The advantage is that if APDU size is large, then large chunks of APDU payload are not copied. The overhead is the additional list of APDU parts which were overwritten by the transport headers for retransmission purposes. This overhead is small and this method is efficient if the APDU/TPDU size is large in comparison to the TPDU header. In case that the APDU/TPDU sizes are small in comparison to the header, the overhead of copying parts is equal or larger when comparing to copying of the APDU payload to transport layer space.

all TPDUs in the range specified by the GPACK are removed from the list of unacknowledged TPDUs. However, if there is no response from the receiver to the first retransmission, the pessimistic scenario is assumed and all unacknowledged timed-out PDUs are retransmitted.

This technique of optimized retransmission improves the performance of the transport layer. The idea is similar to the SMART technique [KM97] mentioned previously. The difference is that in our scheme there is no concept of a cumulative acknowledgment as in SMART. Also, in SMART retransmission, the selective retransmission is based only on the NACKs sent by the receiver and there is no scheme to perform optimized retransmissions when timeouts occur. Our scheme is more elaborate in the way it performs timeout-triggered retransmissions.

5.2.2 Receive Protocol

The receiving side takes care of receiving the TPDUs and reassembles them into the chunks required by the RT client. The data is visualized as a stream of TPDUs. So, the chunks sent out by the sending side can differ in size from the chunks read by the receiving side. Support for such a feature requires information about application chunks to be included in each TPDU. The receiving side functions as follows:

1. A receiving function receives TPDUs and inserts them into the correct position in a receiving queue. The receiving queue is ordered in ascending order of sequence numbers. Every TPDU also contains information about the application chunk it belongs to. This information is extracted and stored in a separate list.

2. If any data PDUs are missed in the sequence, the receiving function sends a GNACK to the sender. The GNACK carries two sequence numbers specifying the range of sequence numbers in which PDUs are missing.

3. If any duplicate PDUs are received, a GPACK is sent to the sender. The GPACK contains the lowest and highest acknowledged TPDU sequence numbers in the APDU with no unacknowledged TPDUs between them. This serves as an acknowledgment for either part or the whole of the APDU depending on the situation.

4. The receiving function determines the TPDUs to be retrieved using the application chunk information. The selected TPDUs are removed from the receiving queue and copied to their correct positions in the application memory.

5. If the transport layer is in the real-time mode and all TPDUs corresponding to one application chunk have not been received before it is time to receive the next chunk, the receiving function returns with whatever data has been received so far. If any TPDU belonging to the current application chunk arrives later, it is discarded.

5.2.3 Configurability

The transport layer has some important dynamically configurable features:

- *Reliability*: The transport layer can operate in two modes - a *totally reliable* mode and a *partially reliable* mode. In the totally reliable mode, there are no timing guarantees. The user is allowed to specify an expected QoS. An effort is made to fulfill the timing requirements, but priority is given to the reliability of delivery - so timing is compromised to ensure that every byte of data sent by the sender reaches the receiver. In the partially reliable mode (also termed the real-time

mode), the timing guarantees take precedence over reliability of delivery. The maximum time for *send* and *receive* operations can be specified. The transport layer sends as much data as possible within the time specified. This mode is particularly useful in frame-based video transmission. If the user requires a frame rate to be sustained, but is willing to compromise on the quality of the video, then the real-time mode may be used. The downside is that an intelligent video compression scheme, with the ability to deal with missing data, is needed. Thus, a real life scenario using the dynamic configurability can be the following - a video application can be started with the transport layer in the reliable mode. If the user is not satisfied with the speed of data delivery, the transport layer can be configured to execute in the real-time mode without interrupting the application.

- *Detachable descriptors*: Current transport layers are tightly coupled to the system file descriptors. For instance, once a TCP socket is created, it is not possible to change the data-link layer characteristics associated with the socket without closing it. Specifically in the case of TCP over ATM, there is no mechanism to change the quality of the ATM connection associated with the socket without closing and reopening the socket. Our transport layer provides configuration methods to dynamically attach connection descriptors. Thus, new ATM connections can be transparently created and attached to the transport layer while the applications continue to send and receive data without interruptions. Furthermore, the descriptor used for sending data can be different from the one used for receiving data. This allows connections with different QoS at the data-link level to be coupled with a single transport layer.

- *TPDU size*: The size of the TPDU can be configured dynamically. This feature is also provided by TCP. The limitation in the current implementation is that the sender and receiver should both issue the configuration command to change the size of the TPDU to the same value. This limitation can be easily overcome by adding more control capability to the transport layer. The size of the TPDU cannot exceed the size of the ATM MTU. In order to have larger TPDU sizes, the size of the ATM MTU must first be increased. However, this is observed to degrade the performance of the ATM card.

5.2.4 Real-time Features

The transport protocol possesses some real-time features designed with multimedia transmission in mind. These features can be activated by configuring the transport layer to run in its real-time mode. The features include:

- *Sender-side Timed-out-data Discard*: If a *send* operation takes longer than its allotted time, the sending side discards future data till it catches up with the timer. This is done in anticipation of a discard on the receive side. Since data arriving late is anyway discarded by the receiving side, the sending side saves bandwidth by avoiding transmission of the late data and instead transmits future data before its time in an attempt to perform a time-saving operation.

- *Dynamic timer adjustment*: Both the *send* and the *receive* operations use timers in order to provide real-time guarantees to the application. These timers are used for retransmission in the case of the *send* operation, and acceptance or rejection of data in the *receive* operation. As mentioned previously, the actual delay value depends on the load on the network. Hence it is necessary to dynamically tune the timeout value in order to achieve better throughput. The transport layer updates its timeout value using a simple averaging scheme. The timeout is set to the average of the current timeout value and the current application TPDU round-trip time. It is found that this scheme of timer adjustment reduces retransmissions and increases throughput without significant degradation of the QoS parameters.

6 Implementation and Application Program Interface(API)

6.1 Specific Issues about CPU Server

We have implemented our server architecture on a single processor Sun Sparc 10 running Solaris 2.5 Operating System. The Solaris Operating System has a default global priority range (0-159), 0 the least importance. There are 3 priority classes:RT class, System class, and TS class. The RT class contains fixed priority range (0-59), which maps to the global priority range (100-159). The dispatcher's priority is 59, the running priority is 58, and the waiting priority is 0. The waiting priority 0 needs to be mapped to the lowest global priority 0, and it must be lower than any TS priorities. This can be done by compiling a new RT priority table `RT_DPTBL` inside the kernel.

The changing priority is done by using the `priocntl()` system call. Its average cost is measured as $175\mu s$. The average dispatch latency (2 context switch + 2 `priocntl()` is measured as $1ms$. The interval timer is implemented using `setitimer()`. We set the time slot to be $10ms$. The overhead comes up to be 10%, which is acceptable. The CPU broker implements a rate monotonic(RM) scheduling algorithm to generate the dispatch table.

6.2 Specific Issues about Memory Server

In modern computer architecture, the memory hierarchy consists of 3 levels in decreasing order of access time – Cache (1st level and 2nd level), Physical Memory, and Disk. The penalty for a cache miss (2nd level) is in the range of 30-200 clock cycles (100s ns) [PH96]. As long as the cache miss ratio falls into a consistent range throughout a process execution, it has little impact on the on-time performance of the soft RT processes. Therefore, we do not provide any cache management or guarantee. However, the penalty for a virtual memory (physical memory) miss is in the range of 700,000-6,000,000 clock cycles (10s of ms) [PH96]. For a software video decoder/encoder running at 30 frames per second (or 33 ms per frame), a few virtual memory misses might lead to the loss of several frames.

In UNIX, each process has its own virtual address space. Within its virtual address space, a process memory is divided into several segments: text, stack, data, shared libraries, shared memory, or memory map. The text segment contains the program binaries. The stack segment contains the execution stack. The data segment contains the process data (e.g., malloc()).

Note that in C++, memory allocation for a new class object is done implicit through the constructor call (e.g., new CLASSNAME). In such cases, the memory allocation does not go through our `Mem::alloc()` API call and hence it is not pinned.

6.3 Specific Issues about Communication Server

We have implemented our communication server in an integrated fashion with the underlying ATM network, CPU, and memory servers. The communication server runs on SPARC 10 machines. The SPARC 10 machines have been installed FORE SBA-200E ATM adaptor cards, which are connected to a FORE ASX-200 switch. The switch is configured with 16 ports and with 155 Mbps capacity per port.

The bandwidth overhead of our METP is measured to be around 20%, which includes the ATM cell header overhead (8/53 Bytes), AAL MTU header overhead, and our Transport Layer PDU header. This means that if the application requests a connection with, e.g., 10 Mbps user-level bandwidth[10], our communication broker will reserve a connection with 10 Mbps * 120% = 12 Mbps of mean bandwidth B_{mean} allocation. Furthermore, our implementation integrates the peak and burst bandwidth into one parameter, the peak bandwidth, because our ATM adaptor card does not support the B_{burst} parameter

[10]This is an average bandwidth which considers an average size of the APDUs among the various frame sizes in the stream.

whereas the ATM standard has specification for it. This means that, if considering the above example, the peak bandwidth B_{peak} is set to be an additional 5 Mbps on top of the mean bandwidth (12 Mbps + 5 Mbps = 17 Mbps)[11]. The acknowledgment connection (reverse connection) is also established for sending acknowledgment information from the receiver back to the sender, its bandwidth is set to be one fifth of the forward connection's bandwidth.

We have measured and plotted two throughput performances, one using our transport protocol and the other one using the Fore AAL3/4 socket, for the achieved bandwidth vs. the reserved bandwidth as shown in Figure 10. The reserved bandwidth is the user-level mean bandwidth that the application specifies to the communication broker. Using the formula given above, the communication broker adds various overhead to derive the ATM-level mean and peak bandwidth for reservation. The maximum achievable user-level bandwidth for the METP protocol is measured to be around 30 Mbps, which is far below the ATM standard of 155 Mbps. However, the low performance of the METP is caused by the poor performance of the underlying FORE AAL3/4 layer which has a maximum performance of only 40 to 45 Mbps. As shown in the graph, when the reserved bandwidth is less than 30 Mbps, the METP can provide good guarantees with the achieved bandwidth meeting the reserved bandwidth.

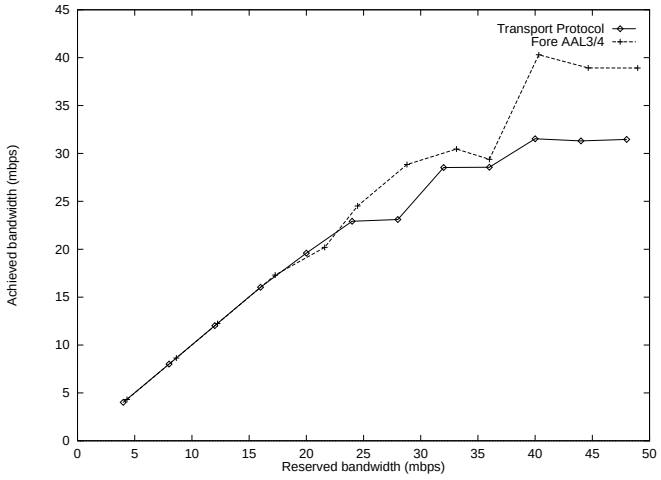


Figure 10: Reserved bandwidth (Excluding Overhead) vs. Achieved bandwidth

6.4 API

The CPU/Memory service provides a C++ programming interface to the user processes. The methods are encapsulated into C++ classes called Cpu and Mem. The interprocess communication (IPC) between the resource broker/scheduler and the user processes is hidden inside the API. This API includes methods for (1) reservation and allocation of CPU and memory resources through the CPU and Memory QoS parameters, (2) management of CPU and memory resources, and (3) modification and deallocation of the allocated resources.

[11] The 5 Mbps corresponds to the overhead under the following assumptions: (1) all video frames transmitted over the connections are I frames ($B^I = M_A^I * R_A$), and (2) our METP protocol segments APDU into a set of TPDUs which may create a burst bandwidth over a short period of time, and this burst bandwidth may be larger than the mean bandwidth B_{mean}, or B^I.

The communication service also provides a C++ programming interface to the user processes. Programs developed using the framework will need to communicate with the communication broker for *service registration, connection establishment, and connection tear-down.* The server API is encapsulated into the class ServerConnection, and the client API is encapsulated into the class ClientConnection.

To illustrate the API described above in an integrated fashion, we provide a simple sample program below that shows how to use our CPU, memory, and communication servers. This sample program presents a piece of Video on Demand client code. For the purpose of illustration, we remove all parts dealing with error checking and resource rejection due to insufficient available resources.

```
/** Initiator side (client which displays video frames) **/
 ...
 Mem mem;
 int rsvId = mem.reserve(500000);   // Memory reservation in number of bytes.
 mem.lockTxt(rsvId);
 char* displayBuf = mem.alloc(rsvId, 76800);  // buffer allocation for display
 char* decoderBuf = mem.alloc(rsvId, 20000);  // buffer allocation for decoder

 CliConnection comm;
 Quality qos(APDU size=20000, endToEndDelay=300ms, sample rate=10 APDUs/sec);
 comm.open("server.cs.uiuc.edu", svcId, qos); // Communication reservation for connection

 Cpu cpu;
 int period = 1000/framerate;
 float util;

 cpu.probeOn(period);  // start of CPU Probing
 for (int i=0; i < 5; i++) {
   comm.recvData(decoderBuf, vsize);  // receive data
   decode(decoderBuf, displayBuf, vsize);  // decode data
   display(displayBuf); // display data
   cpu.yield(); // mark the end of one iteration
 }
 util = cpu.probeOff(); // end of CPU Probing

 cpu.reserve(getpid(), util, period); // CPU reservation.for a process
 cpu.start();
 for (;;) {
   comm.recvData(decoderBuf, vsize);
   decode(decoderBuf, displayBuf, vsize);
   display(displayBuf);
   cpu.yield();
 }

 cpu.freeReserve();  // free CPU reservation
 mem.freeReserve(rsvId); // free Memory reservation
 comm.close();  // free communication reservation
 ...
```

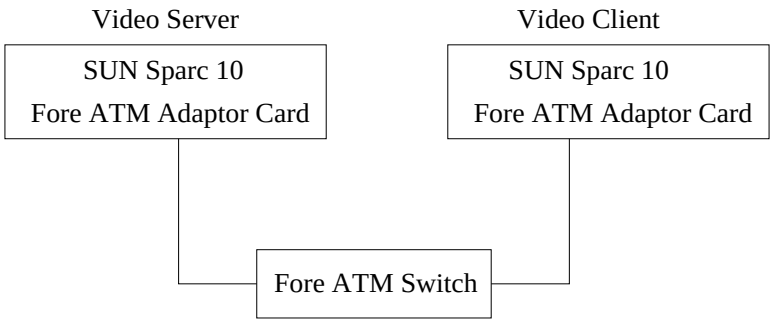


Figure 11: Experimental setup

7 Experiments and Results

The testbed where our implementation and experiments are running consists of two Sparc 10 workstations under Solaris 2.5.1 which they are connected via ATM fore networks as shown in Figure 11. The experiments are designed to show that with QualMan framework, end-to-end QoS requirements for bounded jitter, synchronization skew, and end-to-end delay for distributed multiple applications can be provided under additional load sharing the resources such as CPU, memory, and network bandwidth.

7.1 CPU Server Results

We have performed a number of experiments with the CPU server on a single processor Sparc 10 workstation running Solaris 2.5.1 OS with 32 MB of physical memory. The first experiment (CPU-Experiment-1) consists of the mixture of the following four frequently used applications running concurrently. The first application is a RT mpeg_play program, the later three applications are TS background programs.

- The Berkeley mpeg_play program (version 2.3) plays the TV cartoon Simpsons mpeg file at 10 frames per second (fps).
- The gcc compiler compiles the Berkeley mpeg_play code.
- A compute program calculates the *sin* and *cos* table using the infinite series formula.
- A memory intensive program that copies mpeg frames in a ring of buffers.

Figure 12 shows the measurement of intra-frame time on the mpeg_play program under the above specified load. Figure 12(a) shows the result under the normal TS UNIX scheduler without our server. Figure 12(b) shows the result of the 10fps mpeg_play program with 70% CPU reserved every $100ms$. Using the UNIX TS scheduling, noticeable jitter [12] over $200ms$ (equivalent to 2 frames time) occurs frequently–91 times out of the 650 frames (65 seconds). The largest jitter is about $450ms$ (over 4 frames time), which is clearly unacceptable. Using our server, noticeable jitter over $200ms$ does not occur at all.

The second experiment (CPU-Experiment-2) consists of two mpeg_play programs that plays the same TV cartoon Simpsons at 8fps and 4fps. The set of background TS jobs are the same as in CPU-Experiment-1. Figures 12(c) and 12(d) show the measurements of intra-frame time on the two mpeg_play programs. Figure 12(c) shows the result under normal TS UNIX scheduler without our CPU server. Figure 12(d) shows the result for the 8fps mpeg_play program with 60% CPU reserved every $125ms$, and for the 4fps mpeg_play program with CPU 30% CPU reserved every $250ms$. Using the UNIX TS scheduling, noticeable

[12] Jitter is computed as | intra-frame time - period($100ms$) |.

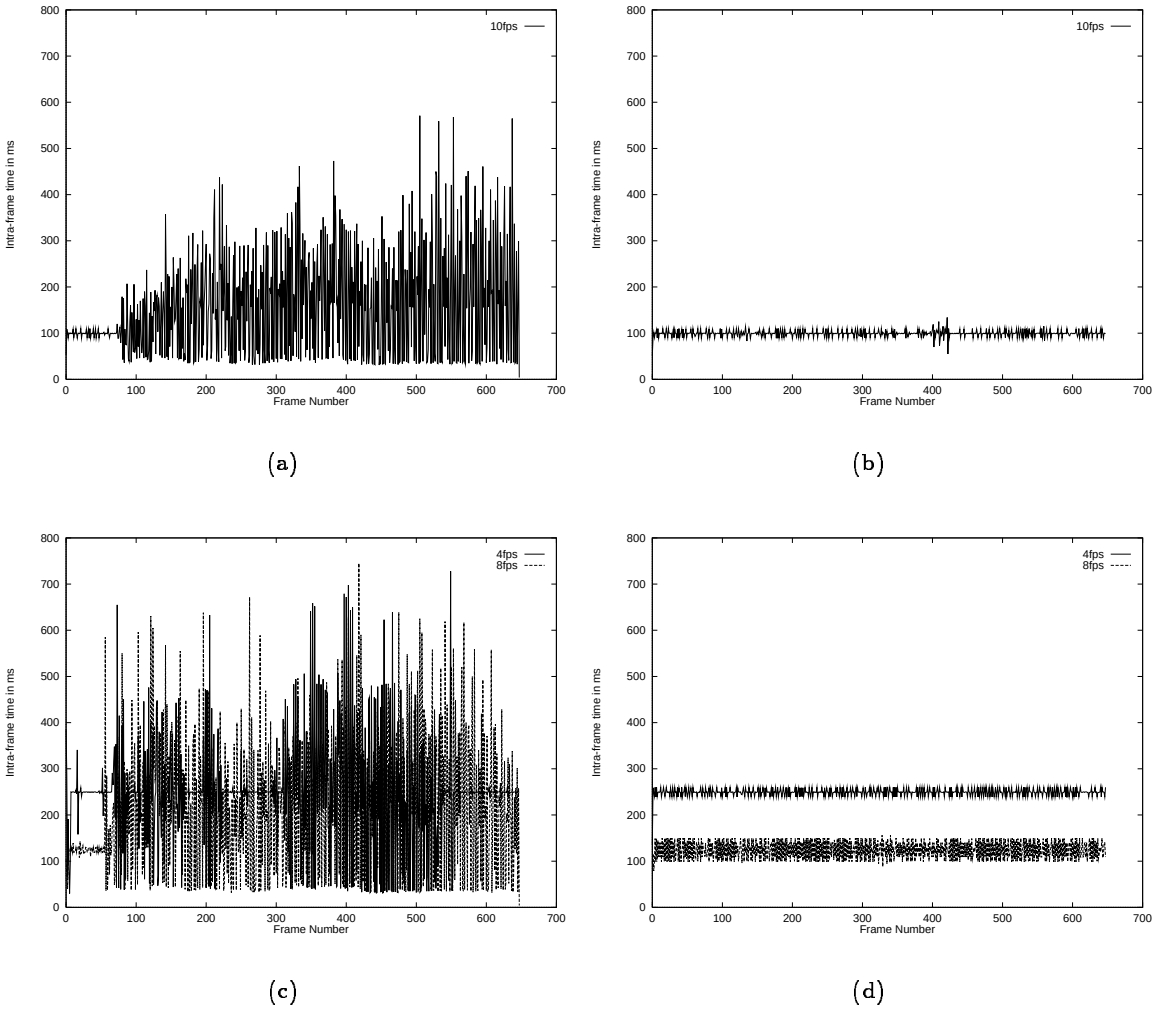


Figure 12: Intra-frame time measurement for the mpeg_play program with and without the CPU server.

jitter over $250ms$ (equivalent to 2 frames time) for the 8fps mpeg_play program occurs frequently at 106 times out of 650 frames (65 seconds), and the largest jitter is around $650ms$ (4 frames time) which is unacceptable. The 4fps mpeg_play program exhibits noticeable jitter over $250ms$ (1 frame time) 16 times. Using our server, noticeable jitter over $250ms$ do not occur for both 8fps and 4fps mpeg_play programs. We have tested other video clips (e.g, a lecture video clip and an animation clip), and we have found similar behavior.

7.2 Memory Server Results

We have tested our memory server together with the CPU server under the same system setup as in the CPU-only experiments in the previous subsection. The memory server is configured with a 10 MB of global_reserve, out of 32 MB of physical memory, serving potentially multiple mpeg_play programs.

The first experiment (CPU-MEM-Experiment-1) consists of the same mixture of applications as defined in the CPU-Experiment-1. It measures the intra-frame time on the 10fps mpeg_play program under the

specified load, and with the presence of both the memory and CPU servers. The mpeg_play program makes a CPU reservation of (70%, 100ms) and establishes a memory reservation of 3MB. All the malloc() calls are changed to MEM::alloc(). The result is similar to the CPU server only graph in Figure 12(b), with a further improvement in average jitter from 4.46ms to 3.94ms.

The second experiment (CPU-MEM-Experiment-2) consists of the same mixture applications as in the CPU-Experiment-1. It measures the intra-frame time on two mpeg_play programs (at 8fps and 4fps), under the specified load, and with the presence of both the memory and CPU servers. The 8fps mpeg_play program makes a CPU reservation of (60%, 125ms) and a memory reservation of 3MB, and the 4fps mpeg_play program makes a CPU reservation of (30%, 250ms) and also a memory reservation of 3MB. We observe more significant improvement in the average jitter from 17.82ms in CPU-Experiment-2 to 8.42ms in CPU-MEM-Experiment-2 for the 8fps mpeg_play program, and 5.49ms in CPU-Experiment-2 to 2.40ms in CPU-MEM-Experiment-2 for the 4fps mpeg_play program. Again, we have also tested other video clips, and we have found similar behavior.

7.3 Communication Results

We have tested our communication server together with the CPU and the memory servers. The network experiment uses two machines, one acting as a sender and the other one as a receiver. The ATM network configuration is described in the previous section. Except for the additional network support, the machines are of the same configuration as in the previous experiments.

The communication server experiment runs a video server program on one machine and potentially several client video programs running on other machines. The video server program forks a child server process to service each client, and the server's child process retrieves a requested MPEG stream and sends the compressed video frames via METP protocol. The video client mpeg_play program is built on top of the Berkeley mpeg_play program, with modifications to read data from our RT transport protocol instead of a file. The client program mpeg_play performs the same decoding and displaying as in the original Berkeley mpeg_play program.

In the first experiment (CPU-MEM-COMM-Experiment 1), the mpeg_play server and client mpeg_play programs are running concurrently with the same mixture of background TS programs on both the server and the client machines at 10 fps. Figure 13(a) has the client and server programs without any resource reservation. Figure 13(b) has the client program with reservation (CPU=80%,100ms; memory=3MB; net=1Mbps) and the server program with reservation (CPU=40%,100ms; memory=3MB; net=1Mbps). Without any resource reservation, noticeable jitter over 200ms occurs frequently at 49 times. The largest jitter is about 450ms. With resource reservation, noticeable jitter over 200ms does not occur.

The second experiment (CPU-MEM-COMM-Experiment-2) consists of two concurrent mpeg_play clients and servers at 6fps and 3fps. Figure 13(c) has the client and server programs without any resource reservation. Figure 13(c) has the 6fps client with reservation (CPU=60%,166ms; memory=3MB, net=0.6Mbps) and server with reservation (CPU=24%,166ms; memory=3MB, net=0.6Mbps), and the 3fps client with reservation (CPU=30%,333ms; memory=3MB, net=0.3Mbps) and server reservation at (CPU=12%,333ms; memory=3MB; net=0.3Mbps). Without any resource reservation, noticeable jitter over 333ms for the 6fps client mpeg_play program occurs frequently at 30 times; however jitter for the 3fps client mpeg_play program occurs less frequently because it consumes little resources at this low rate. With resource reservation, jitter stays within 20ms range for the 6fps client mpeg_play program and within 30ms range for the 3fps client mpeg_play program.

We now summarize the performance results on the mpeg_play (client mpeg_play) program under various degree of resource reservation in Table 6. The comparison metric is average jitter in ms.

Since the MPEG stream is compressed into low bandwidth, which is not a good stress test on our transport subsystem, we have performed additional set of experiments with the video server sending un-

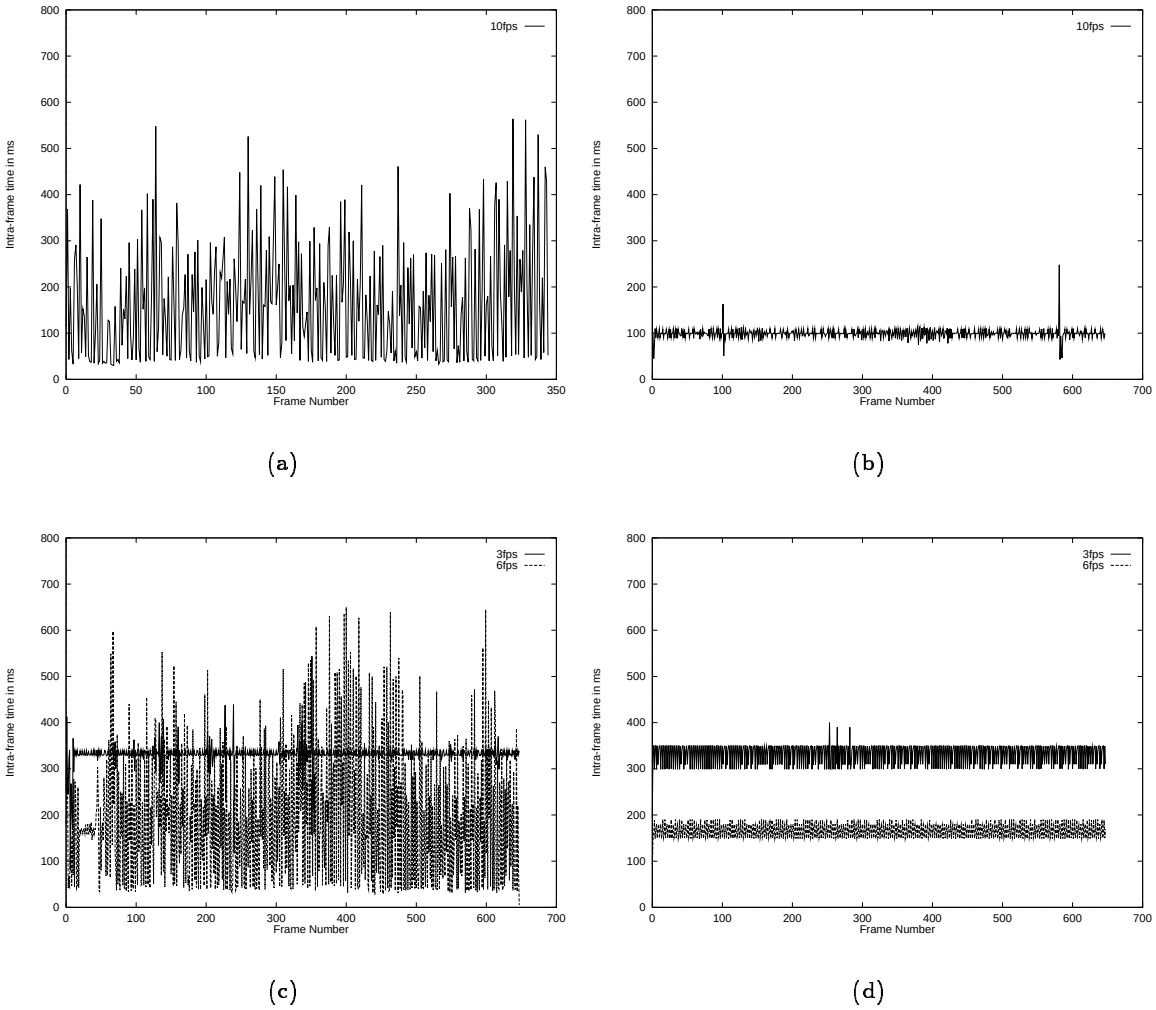


Figure 13: Intra-frame time measurement for the client and server mpeg_play programs with and without CPU, memory, and network servers.

compressed video frames to potentially multiple clients at a much higher bandwidth using METP. Each uncompressed video frame is of fixed size 200KB. The first experiment (CPU-COMM-Experiment-1) involves a single client program requesting video frames at 10fps (16Mbps) from a server program. The same mixture of TS background programs as described in Section 7.1 run concurrently with the video server and client programs on both the server and client machines. We measure the intra-frame time of the uncompressed video frame at the client side. Figure 14(a) has the client and server programs without any resource reservation. Figure 14(b) has the client program with reservation (CPU=40%,100ms; net=16Mbps) and the server program with reservation (CPU=30%,100ms; net=16Mbps). Jitter over $100ms$ (one frame time) under no resource reservation occurs frequently 64 times; whereas it does not occur under the resource reservation.

The second experiment (CPU-COMM-Experiment-2) consists of two concurrent clients that request video frames at 10fps (16Mbps) and 5fps (8Mbps). Again the same mixture of TS background programs run concurrently with the video server and client programs on both the server and client machines. Figure

Resource Reserve	One stream(10fps)	Two streams(8/4fps)	Two streams(6/3fps)
None	93.85ms	136.41ms/72.32ms	*
CPU	4.46ms	19.30ms/5.49ms	*
CPU/memory	3.94ms	8.42ms/2.40ms	*
CPU/memory/network	6.06ms	*	13.57ms/20.01ms

Table 6: Summary of performance results on the mpeg_play program.

14(c) has the client and server programs without any resource reservation. Figure 14(d) has the 10fps client program with reservation (CPU=40%,100ms; net=16Mbps), the 10fps server program with reservation (CPU=30%,100ms; net=16Mbps), and the 5fps client program with reservation (CPU=20%,200ms; net=8Mbps) and the 5fps server program with reservation (CPU=15%,200ms; net=8Mbps). Noticeable jitter over $200ms$ (two frames time) for the 10fps client occurs frequently 35 times under no resource reservation; whereas it does not occur under resource reservation.

Due the limit of the processing power (CPU bandwidth) on the Sparc 10 machine, we cannot run as many concurrent MPEG streams as we would like. The bottleneck is in the software MPEG decoding which takes a significant amount of processing time. However, our solution is perfectly scalable to support multiple streams when we have a faster processor or with a hardware MPEG decoder.

7.4 Synchronization Results

We have also tested *lip synchronization* using our communication servers together with the CPU and the memory server on two SUN Ultra-1 workstations. The video and audio streams are decoded and transported using separate processes and network channels.

The video clip we used in our testbed is MPEG video with a resolution of 352x240 pixels and a recording rate of 7 fps. The audio clip is also MPEG compressed with a recording rate of 20 samples per second. The first experiment runs without any background traffic. The CPU server reserves 20% every $50ms$ to the audio /video servers and clients. The memory server starts with $5MB$ serving the audio/video client processes. Figure 15(a) illustrates skew measurements at the client site. The result shows that the skew is not only in the desirable range of lip synchronization $(-80, 80)$ ms [SN95], but most (99.3%) of the skew results are in the more limited range $(-10, 10)$ ms with an average skew of $3.96ms$ and standard deviation of $0.003ms$. The positive skew value represents the case when audio is ahead of video and the negative skew value for the case when video is ahead of audio.

The second experiment adds a second video stream from server to client with no CPU and memory reservation on both server and client sides as a background load. This additional video stream is also MPEG with a resolution of 352x240 pixels and a recording rate of 20 frames per second. It imposes not only network load as a background traffic, but also processor load on both server and client sides. The result from the second experiment, shown in Figure 15(b), presents the average skew of $4.15ms$ and standard deviation of $0.003ms$. 99.1% of the skew values are within the range $(-10, 10)$ ms. The result shows that our QoS-aware resource management delivers QoS guarantees to a VOD application with the presence of network and OS loads. Actually this is exactly what we expect from a system with resource reservations and performance guarantees.

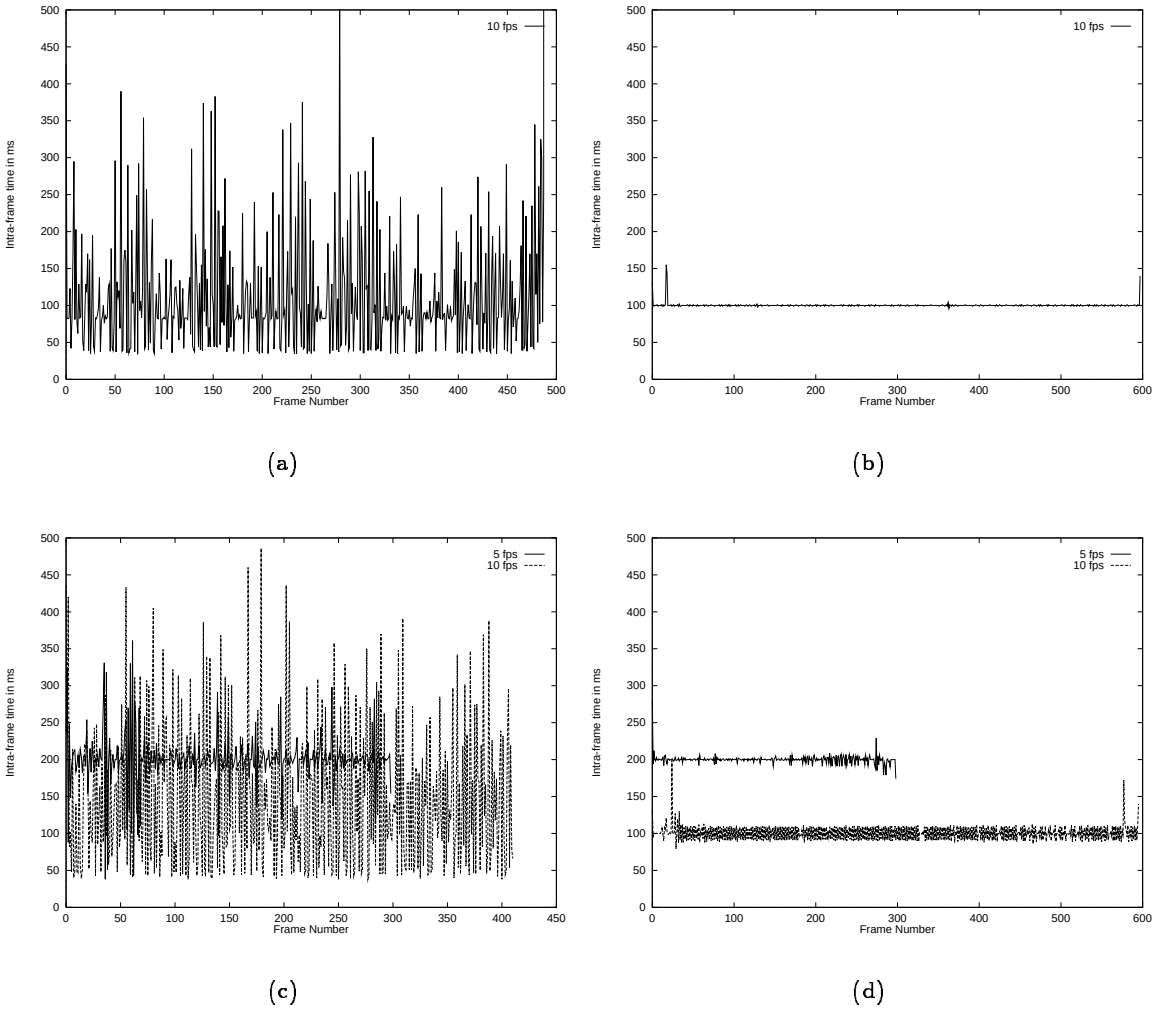


Figure 14: Intra-frame time measurement for the client and server uncompressed video programs with and without resource reservation.

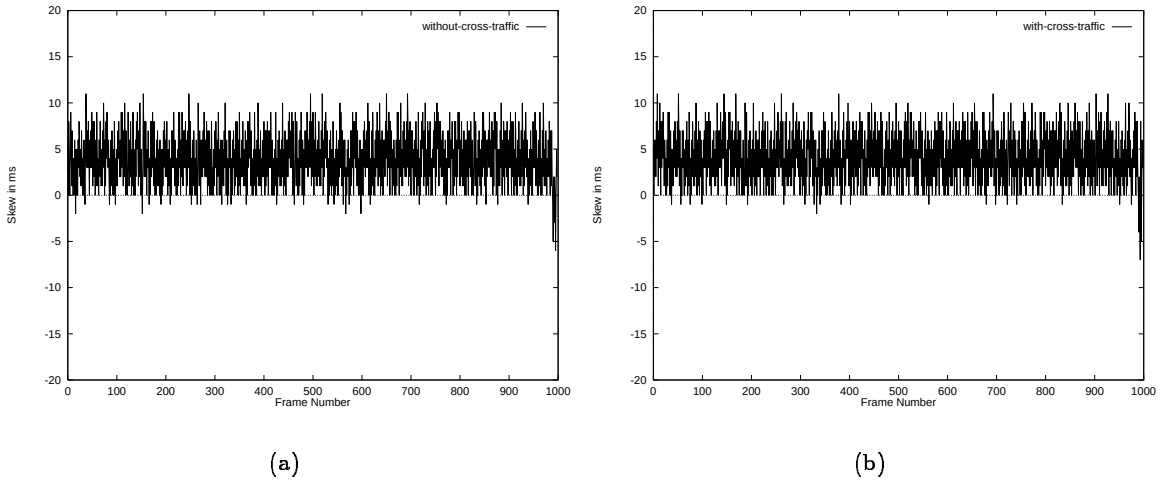
8 Related Work

8.1 QoS Framework

The current existing QoS systems either allow to access and control (1) network QoS such as the Lancaster QoS system [CCH93], or OMEGA end-point system [NS96b], or (2) CPU QoS parameters such as Nemesis [LMB+96], Real-Time Mach 'reserve' [LRM96].

8.2 CPU Scheduling

The area of accommodating scheduling of soft RT applications on the current UNIX platforms was addressed by several groups. Goyal, Guo, and Vin [GGV96] implemented the *Hierarchical CPU Scheduler* in the SUN Solaris 2.4. The CPU resource is partitioned into hierarchical classes, such as Real-time and Best-Effort classes, in a tree-like structure. A class can further partition its resource into subclasses. Each



(a) (b)

Figure 15: Reservation-based Synchronization Skew Results. Figure 15(a) shows the synchronization skew without cross traffic. Figure 15(b) shows the synchronization skew with cross traffic.

class can designate a suitable scheduler to meet the objective of its processes(leaf nodes). Protection between classes is achieved by the *Start-time Fair Queuing* (SFQ) algorithm which is a modification of the Weighted Fair Queuing Algorithm. SFQ is a fair scheduler that schedules all the intermediate classes and subclasses according to their partitioned resources. The major disadvantage of this approach is that their implementation requires modifications to the Solaris kernel scheduler. Fair sharing also does not translate directly into applications QoS guarantees that require a specific amount of CPU allocation and a constant periodicity. Furthermore, the scheduling overhead can be expected to rise proportionally with increasing depth and breath of the hierarchical trees.

Mercer, Savage, and Tokuda [MT94] implemented the *Processor Capacity Reserves* abstraction for the RT-threads in the RT Mach Operating System. A recent version [LRM96] supports dynamic Quality adjustment policy. A new thread must first request its CPU QoS in the form of *period*, and *requested CPU usage in percentage* during the reservation phase. Once it is accepted, a *reserve* of CPU processing time is setup and it is bound to this new thread. Any computation time done on behalf of this process, including all service time from various system threads, is charged to this reserve. The reserve is replenished periodically by its requested CPU usage time. This concept is similar to the *Token Bucket*. The accurate accounting of system service time is a superior but costly feature. It requires non-trivial modifications and computation overhead inside the UNIX kernel to support this abstraction, such as keeping track of the reserves database, and passing the client process's reserve to and between system threads.

Yau and Lam [YL96] implemented the Adaptive Rate-Controlled Scheduling, which is a modification of the *Virtual Clock* Algorithm. Each process specifies a reserve rate and a period for its admission control phase. During its execution phase, the reserve rate is adjusted upward or downward to match its actual usage rate in a gradual fashion. It is called *rate adaptation*.

SMART [NL97] is a real time scheduler that extends the TS UNIX scheduler. It is implemented inside the Solaris kernel. The SMART scheduler allows the RT processes to specify their timing constraints. The RT processes will receive notifications via an upcall from the scheduler when their timing constraints are violated. However, the SMART scheduler is still based on TS concept of proportional sharing. It does not have any mechanisms for admission control and reservation. Hence it does not provide any guarantees on CPU allocation to RT processes.

Gopalakkrishnan [Gop96] implemented the *Real Time Upcall (RTU)* on the NetBSD UNIX to support

periodic tasks. Each RTU is similar to a process, it contains an event handler that registers to the kernel its execution time and period. The kernel dispatcher is modified to schedule the RTUs using the Rate Monotonic algorithm. In order to increase the predictability and efficiency, the kernel dispatcher disallows preemptions during the middle of the RTU execution, called *delay preemption* and no asynchronous preemption.

Kamada, Yuhara, and Ono [KYO96] implemented the *User-level RT Scheduler* (URsched) in the SUN Solaris 2.4. The URsched approach is based on the POSIX.4 fixed priority extension and its priority scheduling rule. We use their technique of priority dispatching. However, our solution carries it much further by providing admission control, rate monotonic scheduling, probing/profiling, re-negotiation of reservation, enforcement of process overrun, and the resource broker architecture.

8.3 Memory

The SUN Solaris Operating System provides a set of system calls that allow a process to lock certain regions of its address space in physical memory [Mic94]. The `mlock(addr, len)`, `munlock(addr, len)` system calls lock or unlock for the address space region `[addr` $\sim$ `addr+len]`. The `mlockall()`, `munlockall()` locks or unlocks all the segments in the address space in physical memory. The `plock(op)` system call locks or unlocks the text or data segments in memory. These memory locking system calls have problems.

- Unlimited and unconstrained use of these system calls are dangerous to the system, given that memory is a shared and scarce system resource. A faulty process can potentially lock out a great portions of the physical memory and it can degrade the performance of other real-time or non-real-time processes.
- The memory locking system calls require processes with superuser privilege, but running video applications should not require superuser privileges.

Lynx Operating System [Sys97] supports the *priority threshold* in its *Demand-Paged Virtual Memory* management. TS processes running at priority lower than the *priority threshold* will get swapped out, while RT processes running at higher priority will not. This priority-based policy has problems. For example, a faulty high priority RT process can potentially block out great portions of the memory in the system and degrade other lower priority RT processes, or the non RT processes. It lacks a protection mechanism.

8.4 Multimedia Communication Protocols

Over the last couple of years, there was a number of fast and real-time transport protocols for multimedia transmission, considering network QoS management. Examples are ST-II [Top90], Tenet Protocol Suite [BFM+96, BM91], Lancaster Transport Subsystem [CCH93, Cam96], Heidelberg Transport Subsystem [DHH+93, DHVW93, DHH94, VHN92], Native ATM Protocol Stack [KS95], User Space TCP implementation [GP96], OMEGA architecture [Nah95], and QoS architecture for Internet Integrated Services [BKMS98]. Because of our ATM consideration for the communication server, to provide a multimedia-efficient transport protocol which will bring out the QoS guarantees provided by the ATM network to the application, we will compare from the above list of the protocols related work only transport subsystems which rely on ATM networks or influenced our METP design.

The *Native ATM protocol stack* [KS95] is a novel protocol stack which (1) is optimized specifically to work well over an ATM network running on PC platform; (2) attempts to provide QoS independent of the operating system environment, which is possible due to the PC's OS specifics; (3) exploits services of an underlying AAL5 layer; (4) uses a new retransmission scheme SMART (Simple Method to Aid ReTransmissions) [KM97], which performs significantly better; and (5) provides reliable and unreliable data delivery with a choice of feedback and leaky-bucket flow control. This framework is implemented and

optimized for a PC environment where part of the transport protocol resides in the kernel. Therefore, this protocol differs from our goal to design a loadable communication server as part of the middleware, which means to have the framework operate in the user-space. However, we applied several lessons learned from this protocol stack, and we expanded its functionality of reliability protocols as mentioned in section 5.

The User Space TCP implementation [GP96] project is a novel attempt to provide support for multimedia processing using existing protocols instead of designing new protocols. It uses an operating system feature called *Real-Time Upcalls* to provide QoS guarantees to networked applications. It (1) provides zero-copy operation based on shared User-Kernel memory, and using off-the-shelf adaptors; (2) eliminates all concurrency control operations in the critical protocol processing path; (3) avoids virtual memory operations during network I/O; and (4) uses the least amount of system calls and context switches. The changes to support upcalls were done in kernel which again differs from our objective for loadable communication server. Similarly to native ATM protocol stack, we applied their lessons learned to our protocol functions to optimize our performance.

The OMEGA architecture [NS96b] is an end-point architecture which extends network QoS services towards the applications. OMEGA consists of the *QoS Broker*, end-point QoS management entity for handling QoS at the edges of the network, and *end-to-end real-time communication protocols* using resources according to the deal negotiated by the broker [NS95a]. Via the QoS broker, it integrates the application requirements with the protocol stack and OS constraints, and supports translation of uncompressed strong periodic streams into the CPU and bandwidth system requirements during the connection setup. Besides the translation, the design of the centralized QoS broker functionality includes capabilities of the *CPU schedulability and network bandwidth admission services*, and *negotiation protocols* at the application and transport subsystems of the end-point system. The admission service checks the QoS requirements against the CPU and network bandwidth availability at each level of the system. The OS and transport QoS enforcement support in OMEGA are much simpler than in the QualMan work because none of the individual resources has the capability of brokerage and explicit QoS enforcement. For example, in OMEGA we rely on fixed priorities scheduling without preemption of real-time processes violating their negotiated CPU bandwidth contract. On the other hand, in QualMan the CPU server has the capability to monitor and preempt real-time processes violating their reservations. Overall, in QualMan system we applied various lessons learned from this end-point OMEGA architecture, made the QualMan end-point model and architecture more scalable and advanced, and provided QoS enforcement functions such as proper scheduling, monitoring, and adaptation within a range of QoS values performed by the resource servers.

The *Real Time Channel* [MIS96] is another novel approach in providing a communication subsystem with QoS guarantees. It implements an UDP-like transport protocol using the x kernel on the Motorola 68040 chip. Each RT channel is served by a periodic RT thread (called channel handler) which runs its protocol stack. The channel handler threads are scheduled by an EDF scheduler. The RT channel has a QoS reserve specification in the form of maximum message size, maximum message rate, and maximum burst size. From these parameters, the required memory and CPU time for the channel handler is computed and allocated. The EDF scheduler provides overload protection, which is similar to the concept of overrun protection for the CPU. The real time channel cannot cause other well-behaved real channels to violate their deadline by sending more bandwidth than it has reserved.

9 Conclusion

In this paper we presented a resource management which allows the applications to specify QoS parameters in terms of CPU, memory and communication QoS parameters, and therefore to control the resource allocation according to the quality desired by the application. We pointed out that in order to give an application such control, the resource management needs to be extended with brokerage and reservation

capabilities. Our new resource model for shared resources includes the resource broker, which provides negotiation, admission, and reservation capabilities over the shared resource. It is an important assistant to the resource scheduler to achieve predictable performance and to improve the quality guarantees to the application.

This model is especially beneficial to multimedia distributed applications which have timing constraints during the processing and communication of continuous media. We showed through numerous experiments and results that the integrated system layer architecture, QualMan, consisting of CPU, memory, and communication servers, is feasible. These servers are implemented as loadable middleware on a general purpose platform which supports real-time extensions. Our results have shown that QualMan provides acceptable and desirable end-to-end QoS guarantees for various multimedia applications such as the MPEG player and video-on-demand application. Perceptually, it makes a huge difference in user acceptance if one watches the display of jitter-full video streams vs. smoothed streams.

Overall, our experiments with QualMan showed that it is easily scalable and portable to different platform. We are currently running this framework on the SGI and Windows NT platforms, in addition to the SUN platform. Applications such as tele-microscopy, tele-robotics, and video one demand are using QualMan for their QoS provision.

References

[BCSW86] J. Blazewicz, W. Cellary, R. Slowinski, and J. Weglarz. *Scheduling under Resource Constraints – Deterministic Models*, volume 7. Baltzer Science Publishers, 1986.

[BFM+96] A. Banerjea, D. Ferrari, B.A. Mah, M. Moran, D.C. Verma, and H. Zhang. The Tenet Real-Time Protocol Suite: Design, Implementation and Experiences. *ACM Transaction on Networking*, 4(1):1–10, February 1996.

[BKMS98] T. Barzilai, D. Kandlur, A. Mehra, and D. Saha. Design and Implementation of an RSVP-based QoS architecture for Integrated Services Internet. *IEEE JSAC (to appear)*, 1998.

[BM91] A. Banerjea and B. Mah. The Real-Time Channel Administration Protocol. In *2nd International Workshop on Network and Operating System for Digital Audio and Video*, Heidelberg, Germany, November 1991.

[Cam96] A. Campbell. *A Quality of Service Architecture*. PhD thesis, Lancaster University, Lancaster, England, 1996.

[CCH93] A. Campbell, G. Coulson, and D. Hutchison. A Multimedia Enhanced Transport Service in a Quality of Service Architecture. In *Workshop on Network and Operating System Support for Digital Audio and Video '93*, Lancaster, England, November 1993.

[CN97] H. Chu and K. Nahrstedt. A soft real time server in unix operating system. In *IDMS'97(European Workshop on Interactive Distributed Multimedia Systems)*, September 1997.

[DHH+93] L. Delgrossi, Ch. Halstrick, D. Hehmann, R. G. Herrtwich, O. Krone, J. Sandvoss, and C. Vogt. Media Scaling for Audiovisual Communication with the Heidelberg Transport System. Technical Report 43.9305, IB¡ ENC Heidelberg, Heidelberg, Germany, 1993.

[DHH94] L. Delgrossi, R. G. Herrtwich, and F. O. Hoffmann. An Implementation of ST-II for the Heidelberg Transport System. *Internetworking Research and Experience*, 5, 1994.

[DHVW93] L. Delgrossi, R. G. Herrtwich, C. Vogt, and L. C. Wolf. Reservation Protocols for Internetworks: A Comparison of ST-II and RSVP. Technical Report 43.9315, IBM European Networking Center, Heidelberg Germany, 1993.

[Fel90] D. Feldmeier. Multiplexing Issues in Communication System. *SIGCOMM*, September 1990.

[GGV96] P. Goyal, X. Guo, and H. Vin. A Hierarchical CPU Scheduler for Multimedia Operating System. In *Second USENIX Symposiun on Operating System Design and Implementation*, 1996.

[Gop96] R. Gopalakrishnan. *Efficient Quality of Service Support Within Endsystems for High-Speed Multimedia Networking*. PhD thesis, Department of Computer Science, Washington University, St. Louis, MI, 1996.

[GP96] R. Gopalakrishnan and G.M. Parulkar. Efficient User Space Protocol Implementation with QoS Guarantees using Real-Time Upcalls. Technical report, Department of Computer Science, Washington University, St. Louis, 1996.

[KM97] S. Keshav and S.P. Morgan. SMART Retransmission: Performance with Random Losses and Overload. In *INFOCOM'97*, 1997.

[KN97] K. Kim and K. Nahrstedt. QoS Translation and Admission Control for MPEG Video. In *5th IFIP International Workshop on Quality of Service*, May 1997.

[KS95] S. Keshav and H. Saran. Semantics and Implementation of a Native-Mode ATM Protocol Stack. Internal technical memo, AT&T Bell Laboratories, Murray Hill, NJ, January 1995.

[KYO96] J. Kamada, M. Yuhara, and E. Ono. User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler. Technical report, Toshiba, Inc., Tokyo, Japan, June 1996.

[LMB+96] I.M. Leslie, D. McAuley, R. Balc, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[LRM96] Ch. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *IEEE Multimedia Systems'96*, Hiroshima, Japan, June 1996.

[Mic94] Sun Microsystems. Solaris 2.5 manual AnswerBook. Sun Microsystems, Inc., 1994.

[MIS96] A. Mehra, A. Indiresan, and K. Shin. Structuring Communication Software for Quality-of-Service Guarantees. In *Proc. of 17th Real-Time Systems Symposium*, December 1996.

[MT94] C. W. Mercer and S. Savage H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Nah95] K. Nahrstedt. *An Architecture for End-to-End Quality of Service Provision and its Experimental Validation*. PhD thesis, University of Pennsylvania, August 1995.

[NHK96] K. Nahrstedt, A. Hossain, and S. Kang. A Probe-based Algorithm for QoS Specification and Adaptation. In *Proceedings of 4th IFIP Workshop on Quality of Service*, pages 89–100, Paris, France, March 1996.

[NL97] J. Nieh and M.S. Lam. SMART UNIX SVR4 Support for Multimedia Applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Ottawa, Canada, June 1997.

[NS95a] K. Nahrstedt and J. M. Smith. The QoS Broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[NS95b] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE COMPUTER*, pages 52–63, May 1995.

[NS96a] K. Nahrstedt and J. Smith. End-Point Resource Admission Control for Remote Control Multimedia Applications. In *IEEE Multimedia Systems*, Hiroshima, Japan, June 1996.

[NS96b] K. Nahrstedt and J. M. Smith. Design, Implementation and Experiences of the OMEGA End-Point Architecture. *IEEE JSAC, Special Issue on Distributed Multimedia Systems and Technology*, 14(7):1263–1279, September 1996.

[PH96] D. Patterson and H. Hennessy. *Computer Architecture: a Qualitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[SG94] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison-Wesley, 1994.

[SN95] R. Steinmetz and K. Nahrstedt. *Multimedia:Computing, Communications, and Applications*. Prentice Hall, Inc., 1995.

[Sys97] Lynx Real-Time Systems. LynxOS, Hard Real-Time OS Features and Capabilities. http://www.Lynx.com/products/ds_lynxos.html, 1997.

[Top90] C. Topolocic. Experimental Internet Stream Protocol, Version 2 (ST II). Internet Network Working Group, RFC 1190, October 1990.

[VHN92] C. Vogt, R. G. Herrtwich, and R. Nagarajan. HeiRAT: The Heidelberg Resource Administration Technique, Design Philosophy and Goals. In *Proceedings of Conference on Communication in Distributed Systems*, München,Germany, 1992. Also published in Informatik aktuel, Springer.

[YL96] D. Yau and S.S. Lam. Adaptive Rate-controlled Scheduling for Multimedia Applications. In *ACM Multimedia Conference'96*, Boston, MA, November 1996.