Realizing Services for Guaranteed-QoS Communication on a Microkernel Operating System

Ashish Mehra[†], Anees Shaikh, Tarek Abdelzaher, Zhiqun Wang, and Kang G. Shin

Real-Time Computing Laboratory
University of Michigan
Ann Arbor, MI 48109–2122
{*ashaikh,zaher,zqwang,kgshin*}@*eecs.umich.edu*

[†]Server and Enterprise Networking
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598-0704
mehraa@watson.ibm.com

Abstract

Provision of end-to-end QoS guarantees on communication necessitates appropriate support in the end systems (i.e., hosts) and network routers that form the communication fabric. This paper focuses on the architectural and implementation challenges involved in realizing QoS-sensitive host communication subsystems on contemporary microkernel operating systems with limited real-time support. We motivate and describe the components constituting our integrated service architecture that together ensure QoS-sensitive handling of network traffic at both sending and receiving hosts and demonstrate a communication framework that can implement alternative QoS models by applying appropriate policies. An experimental evaluation in a controlled configuration demonstrates the efficacy with which QoS guarantees are maintained, despite limitations imposed by the underlying operating system.

1 Introduction

With the continued upsurge in the demand for networked multimedia and real-time applications, a key issue is to identify and resolve the challenges of realizing QoS-sensitive communication subsystems at end systems (i.e., network clients and servers). Traditional design of communication subsystems has centered around optimizing average performance without regard to the performance variability experienced by applications or end users. As such, simple and efficient schemes have been employed for traffic and resource management, as exemplified by the first-come-first-serve service policy. Provision of QoS guarantees, however, requires sophisticated traffic and resource management functions within the communication subsystem, and hence significantly impacts its structure and performance.

In this paper we explore QoS-sensitive communication subsystem design for contemporary operating systems. We describe the general architecture, implementation, and evaluation of a guaranteed QoS communication service for a microkernel operating system. Microkernel operating systems continue to play an important role in operating system design [20], and are being extended to support real-time and multimedia applications [31]. We describe how to map the architectural components of a QoS-sensitive communication subsystem onto the support furnished by the operating system in order to provide appropriate QoS guarantees. We discuss the difficulties in realizing real-time behavior on such platforms and our approach to providing predictability within platform limitations. While we have focused on a microkernel operating system, we believe that our design approach and issues highlighted are equally applicable, although with necessary modifications, to the in-kernel protocol stacks of monolithic Unix-like operating systems [5, 14].

When implementing the service architecture, lack of appropriate operating system mechanisms for scheduling and communication may negatively impact real-time communication performance. Accordingly, we have developed compensatory mechanisms in the communication subsystem to reduce the effects of platform unpredictability. For purposes of admission control, we parameterize the communication subsystem via detailed profiling of the send and receive data paths. Based on this parameterization, we identify the relevant overheads and constraints and propose run-time resource management mechanisms that, along with an admission control procedure, bound and account for these overheads. Execution profiling is, therefore, a key component of our architecture. An experimental evaluation in a controlled configuration demonstrates the efficacy with which QoS guarantees are maintained, within limitations of the inherent unpredictability imposed by the underlying operating system.

For application-level QoS guarantees, an end system must provide adequate computation as well as communication resources to simultaneousely executing applications. We focus on QoS-sensitive communication subsystem design while recognizing that real-time performance cannot be fully guaranteed without additional support from the operating system kernel. Such support could be in the form of processor capacity reserves for the service [28] or appropriate system partitioning [7], and are beyond the scope of this paper.. We envision a system structure with the *communication subsystem* distinct from the *computation subsystem*. The communication subsystem handles all activities and resources involved in transmission and reception of data to and from the network. The computation subsystem, on the other hand, comprises all application processes and threads that perform tasks other than communication processing.

We believe that the communication subsystem is a resource

[†]The work of this author was performed at the University of Michigan. The work reported in this paper was supported in part by the National Science Foundation under grant MIP–9203895 and the Defense Advanced Research Project Agency under grant DOD-C-F30602-95-1-0044. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or DARPA.

management domain distinct from the computation subsystem, since the QoS requirements and traffic characteristics of applications might not necessarily be tied to application importance. While we do not consider integration of QoS-sensitive communication and computation subsystems in this paper, we argue that the architectural support described in this paper is complementary to the underlying operating system support required for application-level QoS guarantees. We are currently investigating architectural approaches to integrate the two subsystems in a flexible manner.

Our primary contribution lies in realizing and demonstrating a QoS-sensitive communication subsystem that partially compensates for the unpredictability in a contemporary operating system, while exploiting the available support for provision of QoS guarantees on communication. This includes integration of the architectural components providing QoS guarantees with local communicaiton resources and management policies, support for dynamic scheduling of all communication processing, and detailed premeterization of the communication subsystem to incorporate underlying platform overheads for acccurate admission control. The insights gained from our work can benefit system designers and practitioners contemplating addition of elaborate QoS support in existing operating systems.

In the next section we note related work in the design of QoS-sensitive communication services. Section 3 presents the goals and architecture of the real-time (guaranteed-QoS) communication service. The components of the architecture are described in Section 4. Section 5 describes our prototype implementation and the issues faced in its realization on a platform with limited real-time support. Section 6 follows with results of an experimental evaluation of our implementation and Section 7 concludes the paper with a summary and directions for future work.

2 Related Work

A number of approaches are being explored to realize QoS-sensitive communication and computation in the context of distributed multimedia systems. An extensive survey of QoS architectures is provided in [9], which provides a comprehensive view of the state of the art in the provisioning of end-to-end QoS.

Network and protocol support for QoS: The Tenet real-time protocol suite [4] is an implementation of real-time communication on wide-area networks (WANs), but it did not address the problem of QoS-sensitive protocol processing inside hosts. Further, it does not incorporate implementation constraints and their associated overheads, or QoS-sensitive processing of traffic at the receiving host. While we focus on end-host design, support for QoS or preferential service in the network is being examined for provision of integrated and differentiated services on the Internet [6, 8, 12]. The signalling required to set up reservations for application flows can be provided by RSVP [34], which initiates reservation setup at the receiver, or ST-II [13], which initiates reservation setup at the sender.

QoS architectures: The OMEGA [30] end point architecture provides support for end-to-end QoS guarantees with a focus on an integrated framework for the specification and translation of application QoS requirements, and allocation of the necessary resources. OMEGA assumes appropriate support from the operating system for QoS-sensitive application execution, and the network subsystem for provision of transport-to-transport layer guarantees (the subject of this paper). QoS-A [10] is a communication subsystem architecture which provides features similar to our service, but its realization would necessitate architectural mechanisms and extensions like those presented in this paper. A novel RSVP-based QoS architecture supporting integrated services in TCP/IP protocol stacks is described in [5]. A native-mode ATM transport layer has been designed and implemented in [3]. These architectures provide support for traffic policing and shaping but not for scheduling protocol processing and incorporating implementation overheads and constraints.

Operating system support for QoS-sensitive communication: Real-time upcalls (RTUs) [17] are used to schedule protocol processing for networked multimedia applications via event-based upcalls [11]. In contrast to RTUs, our approach adopts a thread-based execution model for protocol processing, schedules threads via a modified earliest-deadline-first (EDF) policy [22], and accounts for a number of implementation overheads. Similar to our approach, rate-based flow control of multimedia streams via kernel-based communication threads is also proposed in [33]. However, in contrast to our notion of per-connection threads, a coarser notion of per-process kernel threads is adopted. Also, the architecture outlined in [33] does not provide signalling and resource management services within the communication subsystem.

Explicit operating system support for communication is a focus of the Scout operating system, which uses paths as a fundamental operating system structuring technique [29]. A path can be viewed as a logical channel through a multilayered system over which I/O data flows. As we demonstrate, the CORDS path abstraction [16], similar to Scout paths, provides a rich framework for development of real-time communication services. Our architecture generalizes and extends the path abstraction to provide dynamic allocation and management of communication resources according to application QoS requirements. Recently, processor capacity reserves in Real-Time Mach [28] have been combined with user-level protocol processing [23] for predictable protocol processing inside hosts [21]. However, no support is provided for traffic enforcement or the ability to control protocol processing priority separate from application priority.

3 Real-Time Communication Service Architecture

Our primary goal is to provide applications with a service to request and utilize guaranteed-QoS unicast connections between two hosts. The overall service is currently being utilized in the ARMADA project [1], which implements a set of communication and middleware services that support end-to-end guarantees and fault-tolerance in embedded real-time distributed applications.

Common to QoS-sensitive communication service models are the following three architectural requirements: (i) maintenance of per-connection QoS guarantees, (ii) overload protection via per-connection traffic enforcement, and (iii) fairness to best-effort traffic [25]. Earlier work in [25] presented and justified a high-level architectural design in the context of a specific communication service model. We generalize the architecture to apply to a number of service models, and focus on techniques and issues that arise in implementing the generic architectural components.

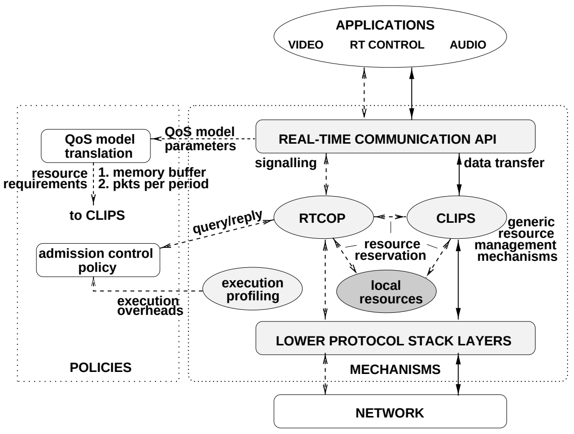


Figure 1. Real-time communication service architecture

Figure 1 illustrates the high-level software architecture of our guaranteed-QoS service at end-hosts. The core functionality of the communication service is realized via three components that interact to provide guaranteed-QoS communication. Applications use the service via the real-time communication application programming interface (`RTC API`); `RTCOP` coordinates end-to-end signalling for resource reservation and reclamation during connection set-up or tear-down; and `CLIPS` performs run-time management of resources for QoS-sensitive data transfer. Since platform-specific overheads must be characterized before QoS guarantees can be ensured, an execution profiling component is added to measure and parameterize the overheads incurred by the communication service on a particular platform, and make these parameters available for admission control decisions. The control path taken through the architecture during connection setup is shown in Figure 1 as dashed lines. Data is then transferred via `RTC API` and `CLIPS` as indicated by the solid lines.

Together, these components provide per-connection communication resource management, including signalling, admission control and resource reservation, traffic enforcement, buffer management, and CPU and link scheduling. We organize these functions into reusable core mechanisms that can implement alternative QoS communication paradigms given the appropriate policies.

We have approached the architectural component design with the goal of separating mechanisms that provide QoS-sensitive communication from the policies that dictate the nature of QoS guarantees. A relaxed admission control policy, for example, coupled with these component mechanisms could be used to implement a statistical guarantee model. Similarly, changing the policy for expression of application QoS requirements, along with a suitable admission control policy, facilitates QoS negotiation and adaptation, as is demonstrated in [2].

4 Architecture Component Design

Below, we discuss the salient features of each architectural component of the service along with its interaction with other components to provide QoS guarantees. We also describe how the components are used to realize a particular service model.

4.1 RTC Application Interface

The programming interface exported to applications comprises routines for connection establishment and teardown, message transmission and reception on established connections, and initialization and support routines. Table 1 lists some of the main routines currently available in `RTC API`. The API has two parts: a top half that interfaces to applications and is responsible for validating application requests and creating internal state, and a bottom half which interfaces to `RTCOP` for signalling (i.e., connection setup and teardown), and to `CLIPS` for QoS-sensitive data transfer.

The design of `RTC API` is based in large part on the well-known socket API in BSD Unix. Each connection endpoint is a pair `(IPaddr, port)` formed by the IP address of the host (`IPaddr`) and an unsigned 16-bit port (`port`) unique on the host, similar to an INET domain socket endpoint. In addition to unique endpoints for data transfer, an application may use several endpoints to receive signalling requests from other applications. Applications willing to be receivers of real-time traffic register their signalling ports with a name service or use well-known ports. Applications wishing to create connections must first locate the corresponding receiver endpoints before signalling can be initiated.

The key aspect which distinguishes `RTC API` from the socket API is that the receiving application *explicitly approves* connection establishment and teardown. When registering its intent to receive signalling requests, the application specifies an agent function that is invoked in response to connection requests. This function, implemented by the receiving application, determines whether sufficient application-level resources are available for the connection and, if so, reserves necessary resources (e.g., CPU capacity, buffers, etc.) for the new connection.

The QoS-parameters passed to `rtcCreateConnection` are translated, for generality, into abstract resource requirements. These are, (i) a specified message buffer size to be reserved for the connection, and (ii) a specified number of packets to be transmitted per specified period. These parameters are passed to `CLIPS` so that it can perform resource management. In addition, optional (QoS model-specific) parameters can be specified and interpreted by the admission policy. Typically, such parameters would constitute additional constraints, such as message deadline for example that affect admission control decisions.

4.2 Resource Reservation with RTCOP

Requests to create and destroy connections initiate the Real-Time Connection Ordination Protocol (`RTCOP`), a distributed end-to-end signalling protocol. `RTCOP` is composed of *request and reply handlers* that manage signalling state and interface to the admission control policy, and a *communication module* that handles the task of reliably forwarding signalling messages. This separation allows simpler replacement of admission control policies or connection state management algorithms without affecting communication functions. Note that signalling and connection establishment are non-real-time (but reliable) functions. QoS guarantees apply to the data sent on an established connection but signalling requests are sent as best-effort traffic.

When processing a new signalling request, the request handler at each hop invokes an admission control procedure to decide

Function	Parameters	Operation Performed
`rtcInit`	none	service initialization
`rtcGetParameter`	chan id, param type	query parameter on specified real-time connection
`rtcRegisterPort`	local port, agent function	register local port and agent for signalling
`rtcUnRegisterPort`	local port	unregister local signalling port
`rtcCreateConnection`	remote host/port, max rate, max burst size, max msg size, max delay	create connection with given parameters to remote endpoint ; return connection id
`rtcAcceptConnection`	local port, chan id, remote host/port	obtain the next connection already established at specified local port
`rtcDestroyConnection`	chan id	destroy specified real-time connection
`rtcSendMessage`	chan id, buf ptr	send message on specified real-time connection
`rtcRecvMessage`	chand id, buf ptr	receive message on specified real-time connection

Table 1. Routines comprising `RTC API`

whether or not sufficient resources are available for the new request. When a connection is admitted at all nodes on the route, the reply handler at the destination node generates a positive acknowledgment on the reverse path to the source to commit connection resources. These resources include packet and message buffers and a `CLIPS` connection handler thread.

The communication module handles the basic tasks of sending and receiving signalling messages, as well as forwarding data packets to and from the applications. Most of the protocol processing performed by the communication module is in the control path during processing of signalling messages. In the data path it functions as a simple transport protocol, forwarding data packets on behalf of applications, much like UDP.

`RTCOP` exports an interface to `RTC API` for specification of connection establishment and teardown requests and replies, and selection of logical ports for connection endpoints. The `RTC API` uses the latter to reserve a signalling port in response to a request from the application, for example. `RTCOP` also interfaces to an underlying routing engine to query an appropriate route before initiating signalling for a new connection. In general, the routing engine should find a route that can support the desired QoS requirements. However, for simplicity we use static (fixed) routes for connections since it suffices to demonstrate the capabilities of our architecture and implementation.

4.3 CLIPS-based Resource Scheduling

The Communication Library for Implementing Priority Semantics (`CLIPS`), implements the necessary resource-management mechanisms to realize QoS-sensitive real-time data transfer. It provides a simple interface that exports the abstraction of a guaranteed-rate communication endpoint, where the guarantee is in terms of the number of packets sent during a specified period. The endpoint also has an associated configurable buffer to acommodate bursty sources. We call this combination a *clip*. To control jitter, `CLIPS` also accepts a deadline parameter. Within each period packets will be transmitted (via the clip) by the specified deadline measured from the start of the period.

Internal to `CLIPS`, each clip is provided with a *message queue* to buffer messages generated or received on the corresponding endpoint, a *communication handler thread* to process these messages, and a *packet queue* to stage packets waiting to be transmitted or received. Once a pair of clips are created for a connection, messages can be transferred in a prioritized fashion using the `CLIPS` API. The `CLIPS` library implements the key functional components illustrated in Figure 2.

QoS-sensitive scheduling: The communication handler thread of a clip executes in a continuous loop either dequeuing outgoing messages from the clip's message queue and fragmenting them (at the source host), or dequeuing incoming packets from the clip's packet queue and reassembling messages (at the destination host). Each message must be sent within a given local delay bound (deadline) that is specified to the clip as a QoS parameter. To achieve the best schedulable utilization, communication handlers are scheduled based on an earliest-deadline-first (EDF) policy. Since most operating systems do not provide EDF scheduling, `CLIPS` implements it with a user-level scheduler layered on top of the operating system scheduler. The user-level scheduler runs at a static kernel priority and maintains a list of all kernel threads registered with it, sorted by increasing deadline. At any given time, the `CLIPS` scheduler blocks all of the registered threads using kernel semaphores except the one with the earliest deadline, which it considers in the running state. The running thread will be allowed to execute until it explicitly terminates or yields using a primitive exported by `CLIPS`. In the context of executing this primitive, the scheduler blocks the thread on a kernel semaphore and signals the thread with the next earliest deadline. This arrangement implements non-preemptive EDF scheduling within a single protection domain.

Policing and communication thread scheduling: Each communication handler is assigned a *budget* to prevent a communication client from monopolizing resources. The budget is expressed in terms of a maximum number of packets to be processed per period and is replenished at the start of a new period. Communication handlers call the CLIPS scheduler after processing each packet to decrement the budget. To police non-conformant sources, the handler is blocked when its budget expires. A handler is rescheduled for execution when the budget is replenished. We implement a "cooperative preemption" mechanism that prevents handlers with large periods and budgets from inflicting unacceptable jitter on the execution of handlers with smaller periods. Each handler participates in cooperative preemption by voluntarily yielding the CPU after processing a small number of packets. If no handler of higher priority is ready for execution at that time, `CLIPS` returns control to the yielding handler immediately. Otherwise, the higher priority handler is executed. Thus, a handler may be rescheduled by the communication thread scheduler when the it blocks due to expiration of its budget, or when it yields the CPU.

QoS-sensitive link bandwidth allocation: Modern operating sys-

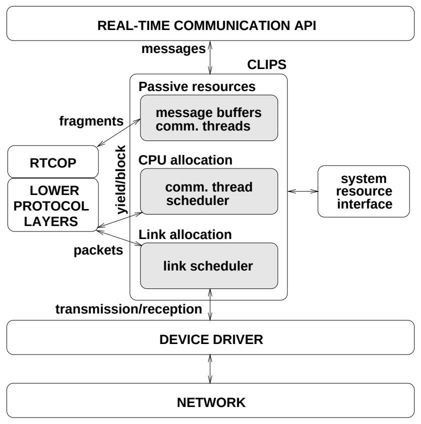


Figure 2. Functional structure of CLIPS.

tems typically implement FIFO packet transmission over the communication link. While we cannot avoid FIFO queuing in the kernel's network device, CLIPS implements a dynamic priority-based *link scheduler* at the bottom of the user-level protocol stack to schedule outgoing packets. The link scheduler implements the EDF scheduling policy using a priority heap for outgoing packets. To prevent a FIFO accumulation of outgoing packets in the kernel (e.g., while the link is busy), the CLIPS link scheduler does not release a new packet until it is notified of the completion of previous packet transmission. Best-effort packets are maintained in a separate packet heap within the user-level link scheduler and serviced at a lower priority than those on real-time clips.

4.4 Execution profiling

The execution profiling component is invoked when the system is deployed on a new platform, or upon system upgrades. It abstracts the communication overheads and costs of the host hardware and software platform and makes them available to admission control to account for protocol processing delay, packet transmission latency, message send delay, etc. Details of our profiling methodology, including measured parameters of our service implementation, are available in [26].

4.5 Service Model Instantiation

Our real-time communication architecture may be used to realize a family of service models that differ in the choice of QoS-parameters and admission control policy, as long as QoS parameters can be converted into a rate constraint (maximum number of packets sent per period), a storage constraint (maximum packet buffer size), and a deadline on each node. We have implemented a communication paradigm amenable to such an abstraction, namely the real-time channels model [15, 19]. A real-time channel is a unicast virtual connection between a source and destination host with associated performance guarantees on message delay and available bandwidth. In requesting a new channel, the application specifies its message generation process to allow the communication subsystem to compute resource requirements and decide whether it can guarantee the desired quality-of-service. The generation process is expressed in terms of the maximum message size (M_{max}), maximum message rate (R_{max}), and maximum message burst size (B_{max}). The burst parameter serves to bound the short-term variability in the message rate and partially determines the necessary buffer size (i.e. in time t, the number of messages generated must be no more than $B_{max} + t \cdot R_{max}$). The QoS requirement is expressed as an upper bound on end-to-end communication delay from the sending application to the receiving application. This deadline parameter influences admission control decisions at all nodes in the route during signalling.

The admission control policy for real-time channels implements the D_order algorithm to perform schedulability analysis for CPU and link bandwidth allocation. Details on D_order and subsequent extensions to account for CPU preemption costs and the relationship between CPU and link bandwidth are available in [19] and [24], respectively.

5 Service Implementation

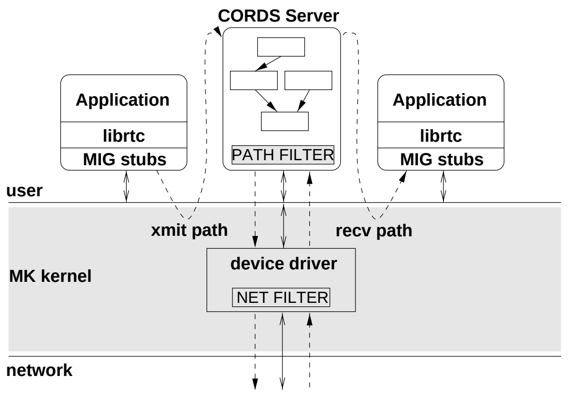
Our experimental testbed and implementation environment is based on the MK 7.2 microkernel operating system from the Open Group Research Institute. The hardware platform consists of several 133 MHz Pentium-based PCs connected by a Cisco 2900 Ethernet switch operating at 10MB/s.

While not a full-fledged real-time OS, MK 7.2 includes several features that facilitate provision of QoS guarantees. Specifically, though it provides only preemptive fixed-priority scheduling, the 7.2 release includes the CORDS (Communication Object for Real-time Dependable Systems) protocol environment [16] in which our implementation resides. CORDS is based on the x-kernel object-oriented networking framework originally developed at the University of Arizona [18], with some significant extensions for controlled allocation of system resources. CORDS is also available for Windows NT and, as such, serves as a justifiable vehicle for exploring the realization of communication services on modern microkernels with limited real-time support.

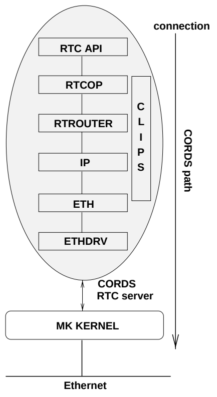
5.1 Service Configuration

Figure 3(a) shows the software configuration for the guaranteed-QoS communication service. While the CORDS framework can be used at user-level as well as in the kernel, we have developed the prototype implementation as a user-level CORDS server. There are several reasons for this choice as discussed in Section 5.2 below, the most obvious being the ease of development and debugging, resulting in a shorter development cycle. Applications link with the librtc library and communicate real-time connection requests and data via IPC with the user-level CORDS server.

The service protocol stack is configured within the server as shown in Figure 3(b). RTC API interfaces with applications via Mach Interface Generator (MIG) stubs, translating application requests to specific invocations of operations on RTCOP (for signalling) or CLIPS (for data transfer). RTCOP serves as a transport protocol residing above a two-part network layer composed of RTROUTER and IP. Though we currently use default IP routing, we provide RTROUTER as a go-between protocol to keep the routing interface independent of IP so that RTCOP may eventually work with more sophisticated routing protocols that support QoS- or policy-based routing. The IP, ETH, and ETHDRV protocols are standard implementations distributed with the CORDS framework. ETH is a generic hardware-independent protocol that provides an interface between higher level protocols and the actual Ethernet driver. ETHDRV is specific to the user-level implementation of the



(a) Server configuration



(b) CORDS protocol stack

Figure 3. Service implementation as CORDS server: In (a) the communication path used by applications is shown; (b) illustrates the configurable protocol stack.

CORDS server. It is an out-of-kernel device driver that interacts with the network device driver in the Mach kernel through system calls to a Mach device control port. Note that CLIPS spans the protocol stack, providing scheduling and resource management services at both the message and packet levels.

When an application sends a message to the CORDS server for transmission on an established real-time connection, an API thread waiting on the corresponding Mach port first deposits it into a connection-specific message queue. CLIPS then schedules the connection's handler thread to perform protocol processing and fragment the message into packets. These packets are labeled with their local deadline and staged in the CLIPS packet heap. From this point the CLIPS link scheduler thread retrieves packets and transmits them according to their dealines. A packet arriving at the receiving host is demultiplexed into its connection-specific packet buffer when it enters the CORDS server from the kernel. The connection handler thread, scheduled by CLIPS, retrieves the packet and shepherds it up the protocol stack performing protocol processing and message reassembly. Once reassembled, the message is deposited in the connection message queue and the corresponding API thread is notified of the message arrival (if it is waiting). Finally, the API thread constructs a MIG message containing the data and delivers it to the application task.

5.2 Implementation Issues and Platform Support

Below we highlight several issues and challenges in implementing the communication service. We discuss limitations in the underlying platform that lead to unpredictable behavior and the compensatory mechanisms that we used to circumvent them. We also describe platform features that are useful in realizing a real-time communication service and their use in our implementation.

Server-based implementation: While a server-based implementation is natural for a microkernel operating system, it may perform poorly compared to user-level protocol libraries due to excessive data copying and context switching [23, 32]. Implementing the service as a protocol library, however, distributes the functions of admission control and run-time resource management among several address spaces. Since applications may each compete for communication resources, controlling system-wide resources is more effectively done when these functions are localized in a single domain. Moreover, in the worst case, compared to user-level protocol libraries a server configuration only suffers from additional context switches. While this has significant implications for small messages, the relative degradation in performance is not as significant for large data transfers performed by the guaranteed-QoS communication service, although it may affect connection admissibility. Once developed and debugged, the server can be moved into the kernel to improve performance and predictability. Our design approach and lessons learned are applicable to communication subsystems realized as user-level libraries, co-located kernel servers, or integrated kernel implementations.

Network device interface: A server-based implementation presents a number of problems for data input and output in our architecture. The bottom layer of the protocol stack interfaces with the kernel device driver via the kernel's IPC mechanism. Device output is initiated by the CLIPS link scheduler, as close as possible to the device driver without being in the kernel. The kernel device driver cannot directly invoke the user-level link scheduler in response to transmission completion interrupts unless the OS supports mapped device drivers or user-level upcalls. In the absence of such support, user-level link scheduling cannot be done in interrupt context. Instead, we utilize user-level threads to perform synchronous device transfers and link scheduling is realized in the context of a high priority thread.

Resource reservation with Paths: Resource reservation must be coordinated in an end-to-end fashion along the route of each connection during connection establishment. CORDS provides two abstractions, *paths* and *allocators*, for reservation and allocation of system resources within the CORDS framework. Resources associated with paths include dynamically allocated memory, input packet buffers, and input threads that shepherd messages up the protocol stack [16]. Paths, coupled with allocators, provide a capability for reserving and allocating resources at any protocol stack layer on behalf of a particular connection, or class of messages. With packet demultiplexing at the lowest level at the receiver (i.e., performed in the device driver), it is possible to isolate packets on different paths from each other early in the protocol stack. Incoming packets are stored in buffers explicitly tied to the appropriate path and serviced by threads previously allocated to that path. Moreover, threads reserved for a path may be assigned one of several scheduling policies and priority levels. We use paths to facilitate per-connection resource reservation during connection setup.

Packet classification: Proper handling of prioritized real-time data at the receiving host requires that packet priority be identified as early as possible in the protocol stack, and that packets be served accordingly. CORDS associates outgoing packets with paths and demultiplexes incoming traffic into per-path buffers, essentially acting as a specialized packet filter. The data link device driver examines outgoing packets and adds an appropriate path identifier to allow early path-based demultiplexing at the receiver. This allows packet handling to be done in path-dependent order and facilitates imposing relative priorities among paths (e.g., packets of one path can be served before those of another). While this technique is natural for networks supporting a notion of virtual circuit identifiers (VCI) such at ATM, it is not so for traditional data link technologies such as Ethernet. In the case of Ethernet, the CORDS driver adds a new *path identifier* to the data link header. This creates a non-standard Ethernet header that would not be understood by hosts not running the CORDS framework.

Packet queuing: While packet classification, as discussed above, occurs in the QoS-sensitive communication server, we cannot assume that the kernel supports prioritized packet processing. The in-kernel network device driver simply relays received packets to the communication server in FIFO order via the available IPC mechanism, in our case Mach ports. This FIFO ordering does not respect connection QoS requirements, however, since urgent packets can suffer unbounded priority inversion when preceded by an arbitrary number of less urgent packets in the queue. Also, since the same queue is used for real-time and non-real-time traffic, depending on packet arrival-time patterns, real-time data maybe dropped when the queue is filled by non-real-time packets. These two problems cannot be solved without modifying the kernel device driver. To ameliorate this unpredictability, a high priority thread waits on the communication server's input port and dequeues incoming packets as soon as they arrive, depositing them in their appropriate path-specific queues. This prevents FIFO packet accumulation in the kernel and allows the server to service packets in priority order according to the connection path.

Application-server IPC: A problem similar to the above arises when applications use kernel-level IPC mechanisms to send messages via the QoS-sensitive communication server. Unless synchronous communication (e.g., RPC) is used to send messages to the server, successive application messages will accumulate in the kernel buffers for delivery to the server. In a QoS-sensitive system the length of such a queue should be derived from the application traffic specification, for example based on message rate, size, and burst. If the queue is too small, application messages may be dropped or require retransmission from the application. If the queue is overly long, application messages may reside in it longer than anticipated and result in deadline violations. We do not assume that we can control allocation of the kernel-level IPC queues. Our strategy is to drain them as fast as possible, transferring messages to connection-specific queues in the communication server, which are sized in accordance with the connection's traffic specification. We dedicate an API thread per connection within the server whose function is to consume messages from the corresponding Mach port queue. Once an application sends a message to the server, the corresponding API thread reads it from the Mach port and queues it for the corresponding communication handler. The thread, whose execution time is charged to the handler's budget, runs at handler priority, and is allowed to continue running at background priority when the handler's budget expires. Like the handler, the API thread adheres to the cooperative preemption model by yielding to waiting, higher priority messages after processing a fixed amount of message data.

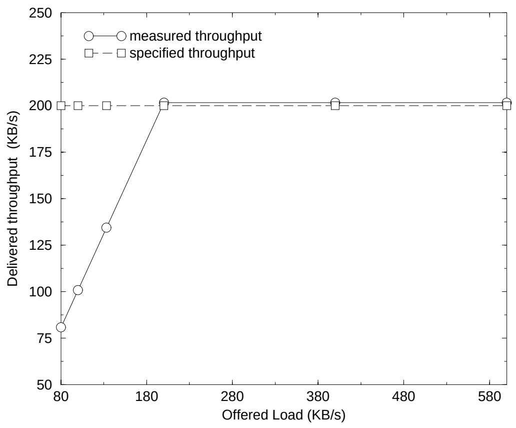


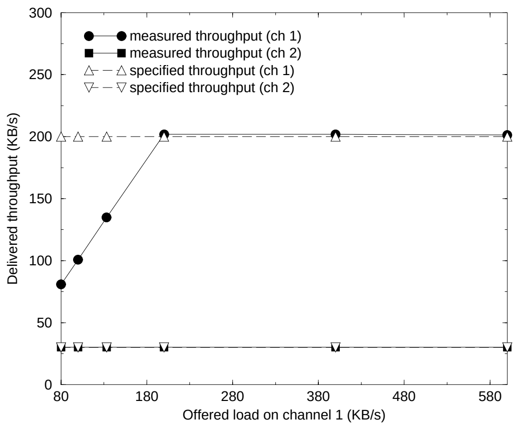
Figure 4. Traffic enforcement on a single real-time channel: $R_{max} = 5$ **msg/sec,** $M_{max} = 40$**KB,** $B_{max} = 10$**, and** $D = 200$**ms.**

Dynamic path creation and deletion: Real-time connections may be created and deleted repeatedly over an application's lifetime requiring that paths be dynamic entities with appropriate teardown and resource reclamation mechanisms. The CORDS framework envisions a relatively static use of paths, with a single path for best-effort traffic and a few paths for different classes of traffic. That is, there are never more than perhaps ten active paths, all of these long-lived and preconfigured. Accordingly, the CORDS path library does not support path teardown or resource reclamation operations. To facilitate a one-to-one association between real-time connections and paths, we have extended CORDS to support path destruction and reclamation of resources associated with a path. These mechanisms are invoked by RTCOP during signalling of teardown messages from connection source to destination.

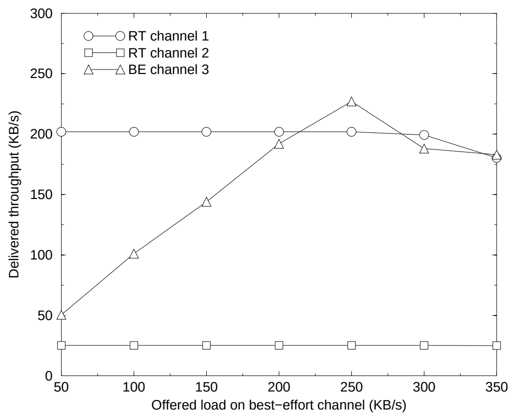
6 Experimental Evaluation

We conducted several experiments to evaluate the efficacy of our prototype implementation. The experiments demonstrate two key aspects of the QoS support provided: traffic enforcement (i.e., policing and shaping) on a single connection, and traffic isolation between multiple real-time and best-effort connections. We show that reasonably good QoS-guarantees can be achieved despite the lack of real-time scheduling and communication support in the kernel.

The experimental setup consists of two hosts communicating on a private segment through the Ethernet switch. To avoid interference from the Unix server, we suppress extraneous network traffic (e.g., ARP requests and replies) and configure the CORDS server to receive all incoming network traffic. This allows us to



(a) Isolation between real-time channels



(b) Isolation between best-effort and real-time traffic

Figure 5. In (a) channel 1 has the same specification as in Figure 4 and channel 2 has $R_{max} = 2$ msg/sec, $M_{max} = 15$ KB, $B_{max} = 10$, and $D = 100$ ms. In (b) channel 1 has a specification of $R_{max} = 10$ msg/sec, $M_{max} = 20$ KB, $B_{max} = 10$, and $D = 200$ ms and channel 2 has $R_{max} = 5$ msg/sec, $M_{max} = 5$ KB, $B_{max} = 10$, and $D = 100$ ms.

limit the background CPU load on each host and accurately control network traffic between them. For each experiment reported below, connections are created between MK client tasks running at the two hosts. Connection parameters are specified according to the real-time channel model, namely as maximum message size (M_{max}), maximum message burst (B_{max}), message rate (R_{max}), and message deadline (D). Message traffic is generated by threads running within the MK client task at the source and consumed by threads running within the destination client. Our evaluation metric is the per-connection application-level throughput delivered to the receiving task at the destination host.

6.1 Traffic Enforcement

For this experiment, we establish a real-time channel with a specified rate of 200 KB/s and a 200 ms deadline. The actual offered load on the channel is varied, however, by changing the interval between generation of successive messages, ranging from 500 ms to 0 ms (i.e., continuous traffic generation).

As shown in Figure 4, the delivered throughput increases linearly with the offered load until the offered load equals the specified channel rate. For example, at an offered load of 100 KB/s (corresponding to a message generation interval of 400 ms), the delivered throughput is 100 KB/s. Similarly, at an offered load of 200 KB/s, the delivered throughput is 200 KB/s. For offered loads beyond the specified channel rate, however, the delivered throughput equals the specified channel rate. This continues to be the case even under continuous message generation (message generation interval of 0 ms). These measurements show that the traffic enforcement mechanisms effectively prevent a real-time connection from violating its specified rate.

6.2 Traffic Isolation

In addition to proper traffic enforcement, recall that one of the architectural goals of the guaranteed-QoS communication service is to ensure isolation between different QoS and best-effort connections. We first consider traffic isolation between multiple real-time channels subject to traffic violation by a real-time channel. Two real-time channels are established between the hosts, one representing a high-rate channel (channel 1) and the other representing a low-rate channel (channel 2). The high-rate channel has the same traffic and deadline specification as before, i.e. a specified rate of 200 KB/s. The low-rate channel has a specified channel rate of 30 KB/s. Message generation on channel 2 is continuous, so that it sends at a persistent 30 KB/s. Channel 1 is controlled in order to vary the offered load.

Figure 5(a) shows the delivered throughput on channels 1 and 2 as a function of the offered load on channel 1. Once again, the delivered throughput on channel 1 increases linearly with the offered load until the offered load equals the specified channel rate (200 KB/s). Subsequent increase in offered load has no effect on the delivered throughput which stays constant at the specified channel rate. The delivered throughput on channel 2, on the other hand, remains constant at approximately 30 KB/s (its specified channel rate) regardless of the offered load on channel 2. That is, traffic violations on one connection (even continuous message generation) do not affect the delivered QoS for another connection.

We also consider traffic isolation between real-time and best-effort traffic under increasing best-effort load. For this experiment we create an additional best-effort channel in addition to two real-time channels. As before, one real-time channel (channel 1) represents a high-rate channel with a specified rate of 200 KB/s. The other real-time channel (channel 2) is a low-rate channel with rate 25 KB/s. Message generation on channels 1 and 2 is continuous, i.e., with a message generation interval of 0 ms. The offered load on the best-effort channel (channel 3) is varied from 50 KB/s to 350 KB/s by controlling the message generation interval.

Figure 5(b) plots the delivered throughput on each channel as a function of the offered best-effort load. A number of observations can be made from these measurements. First, the delivered throughput on channels 1 and 2 are roughly independent of the offered best-effort load. That is, real-time traffic is effectively isolated from best-effort traffic, except under very high best-effort loads as explained below. Second, best-effort traffic utilizes any excess capacity not consumed by real-time traffic, as evidenced by the roughly linear increase in delivered throughput on channel 3 as a function of the offered load. Once the system reaches saturation (beyond a best-effort offered load of approximately 250 KB/s), however, best-effort throughput declines sharply due to buffer overflows and the resulting packet loss at the receiver.

Under very high best-effort loads, the delivered throughput on channel 1 declines slightly. We believe that this is due to the overheads of receiving and discarding best-effort packets, which have not been accounted for in the admission control procedure. These overheads impact the delivered throughput on high-rate connections more than low-rate connections, as evidenced by the constant throughput delivered to channel 2 even under very high best-effort load.

6.3 Fairness to Best-Effort Traffic

While the load offered by real-time connections in the previous experiments was persistent (always greater than the reserved capacity), this experiment focuses on utilization of any reserved capacity not utilized by a real-time connection. It is desirable that this unused capacity be utilized by best-effort traffic, as per our goal of fairness. Other real-time connections should not be allowed to consume this excess capacity at the expense of best-effort traffic. We create two real-time channels and a best-effort channel as before. While the offered load on channel 2 is continuous, channel 1 only offers a load of 100 KB/s even though it is allocated a capacity of 200 KB/s. We consider two cases of message generation on channel 1, as explained below. In case 1, channel 1 carries 20 KB messages at 5 messages/second (half the specified rate). In case 2, it generates 10 KB messages at 10 messages/second.

Figure 6 plots the delivered throughput on all the channels as a function of the offered load on the best-effort channel (channel 3). Channel 1 receives a constant 100 KB/s throughput independent of the offered best-effort load. Similarly, channel 2 receives its allocated capacity of 25 KB/s. In case 1, Channel 3's delivered throughput increases linearly with the offered load until an offered load of 250 KB/s. Beyond this load, the delivered best-effort throughput falls as before, but continues to be higher than that obtained in Figure 5(b) when real-time channels were using their full reserved capacity.

We found, though, that best-effort traffic is unable to fully utilize the unused capacity left by channel 1. We suspect that this effect is primarily due to packet losses caused by buffer overflow at the receiver, either in the adapter or in the kernel device port queue used by the CORDS server to receive incoming packets. To validate this, we ran additional experiments with case 2, in which channel 1 generates smaller messages (10 KB) at a higher rate to offer the same average load of 100 KB/s. As can be seen in Figure 6, the delivered best-effort throughput in this case continues to increase linearly beyond 250 KB/s and shows no decline even for a best-effort load of 350 KB/s. These results suggest that best-effort traffic is able to fully utilize unused capacity when real-time traffic is less bursty (i.e., has fewer packets in each message).

6.4 Further Observations

With the user-level CORDS server configuration, the receiving task is able to receive packets at an aggregate rate of 450-500 KB/s (depending on the number of packets in a message), even though the sender can send at a maximum rate of approximately 750 KB/s.

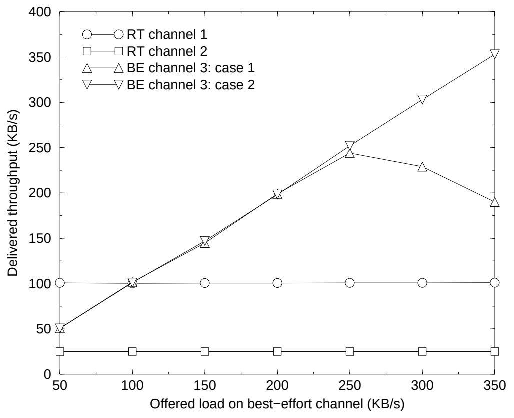


Figure 6. Channels 1 and 2 have the same parameters as in Figure 5(b) but channel 1 underutilizes its reservation. In case 1, channel 1 generates 20 KB messages at 5 msg/s; in case 2, it generates 10 KB messages at 10 msg/s.

This discrepancy is most likely due to CPU contention between the receiving application task and the CORDS server and the resulting context switching overheads, and the high cost of IPC across the client and server. Another reason could be the unnecessary copy performed by the lowest layer (ETHDRV) of the CORDS protocol stack whenever packets from multiple paths arrive in an interleaved fashion. Since this occurs frequently with multiple channels and under high traffic load, it is likely that this extra copy is slowing down the receiver significantly; this extra copy can only be eliminated by redesigning path buffer management in the CORDS framework. More importantly, none of these effects are accounted for in the admission control procedure, and must be addressed when the communication subsystem is integrated more closely within the host operating system. We expect to see significant improvements in the base performance for an in-kernel realization of our prototype implementation.

7 Summary and Future Work

In this paper we have described our experiences with the design, implementation, and evaluation of a guaranteed-QoS communication service implemented on a contemporary microkernel operating system with limited real-time supprt. We designed three primary components that provide general mechanisms for real-time communication, including support for signalling and resource reservation, traffic enforcement, buffer management, and CPU and link scheduling. A fourth execution profiling component is responsible for the essential task of characterizing platform-specific communication overheads. When combined with with specific policies for admission control and interpretation of QoS parameters, these components can be used to implement several QoS-sensitive communication models.

We have tested our prototype with transmission of stored compressed video and playout using mpeg_play. We plan to conduct further experiments with a number of stored video traces. To allow for QoS-adaptation, we have implemented an end-host architecture for adaptive-QoS communication services [2]. In [26] we describe the complex process of parameterizing the overheads of the communication subsystem and target platform. These efforts illustrate the need for an automated approach to profiling and system parameterization. We have, therefore, also begun to explore self-parameterizing protocol stacks for QoS-sensitive communication subsystems [27].

References

[1] T. Abdelzaher, S. Dawson, W. Feng, S. Ghosh, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, J. Norton, A. Shaikh, K. Shin, V. Vaidyan, Z. Wang, and H. Zou. ARMADA middleware suite. In *Proc. of IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, pages 11–18, San Francisco, CA, December 1997.

[2] T. Abdelzaher and K. Shin. End-host architecture for QoS-adaptive communication. In *to appear in Proc. Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[3] R. Ahuja, S. Keshav, and H. Saran. Design, implementation, and performance of a native mode ATM transport layer. In *Proc. IEEE INFOCOM*, pages 206–214, March 1996.

[4] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang. The Tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Trans. Networking*, 4(1):1–11, February 1996.

[5] T. Barzilai, D. Kandlur, A. Mehra, D. Saha, and S. Wise. Design and implementation of an RSVP-based quality of service architecture for integrated services Internet. In *Proc. Int'l Conf. on Distributed Computing Systems*, May 1997.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. Internet Draft (draft-ietf-diffserv-arch-01.txt), August 1998.

[7] G. Bollella and K. Jeffay. Supporting co-resident operating systems. In *Proc. Real-Time Technology and Applications Symposium*, pages 4–14, June 1995.

[8] R. Braden, D. Clark, and S. Shenker. Integrated services in the Internet architecture: An overview. *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[9] A. T. Campbell, C. Aurrecoechea, and L. Hauw. A review of QoS architectures. *Multimedia Systems Journal*, 1996.

[10] A. T. Campbell, G. Coulson, and D. Hutchison. A quality of service architecture. *Computer Communication Review*, April 1994.

[11] D. D. Clark. The structuring of systems using upcalls. In *Proc. ACM Symp. on Operating Systems Principles*, pages 171–180, 1985.

[12] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proc. of ACM SIGCOMM*, pages 14–26, August 1992.

[13] L. Delgrossi and L. Berger. Internet stream protocol version 2 (ST-2) protocol specification - version ST2+. *Request for Comments RFC 1819*, August 1995. ST2 Working Group.

[14] R. Engel, D. Kandlur, A. Mehra, and D. Saha. Exploring the performance impact of QoS support in TCP/IP protocol stacks. In *Proc. IEEE INFOCOM*, San Francisco, CA, March 1998.

[15] D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[16] F.Travostino, E.Menze, and F.Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proc. IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, February 1996.

[17] R. Gopalakrishnan and G. M. Parulkar. A real-time upcall facility for protocol processing with QoS guarantees. In *Proc. ACM Symp. on Operating Systems Principles*, page 231, December 1995.

[18] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):1–13, January 1991.

[19] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-time communication in multi-hop networks. *IEEE Trans. on Parallel and Distributed Systems*, 5(10):1044–1056, October 1994.

[20] D. G. Korn. Porting UNIX to windows NT. In *Proc. USENIX Winter Conference*, January 1997.

[21] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in Real-Time Mach. In *Proc. of 2nd Real-Time Technology and Applications Symposium*, June 1996.

[22] C. Liu and J. Layland. Scheduling algorithms for multi-programming in hard real-time environment. *Journal of the ACM*, 1(20):46–61, January 1973.

[23] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *Proc. ACM Symp. on Operating Systems Principles*, pages 244–255, December 1993.

[24] A. Mehra, A. Indiresan, and K. Shin. Resource management for real-time communication: Making theory meet practice. In *Proc. 2nd Real-Time Technology and Applications Symposium*, pages 130–138, June 1996.

[25] A. Mehra, A. Indiresan, and K. Shin. Structuring communication software for quality of service guarantees. In *Proc. 17th Real-Time Systems Symposium*, pages 144–154, December 1996.

[26] A. Mehra, A. Shaikh, T. Abdelzaher, Z. Wang, and K. Shin. Realizing services for guaranteed-QoS communication on a microkernel operating system. Technical Report CSE-TR-375-98, University of Michigan, Dept. of Electrical Engineering and Computer Science, Ann Arbor, MI, September 1998.

[27] A. Mehra, Z. Wang, and K. Shin. Self-parameterizing protocol stacks for guaranteed quality of service. available at ftp://rtcl.eecs.umich.edu/outgoing/ashish/selfparam.ps, June 1998.

[28] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[29] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pages 153–168, October 1996.

[30] K. Nahrstedt and J. M. Smith. Design, implementation and experiences of the OMEGA end-point architecture. *IEEE Journal on Selected Areas in Communications*, 14(7):1263–1279, September 1996.

[31] S. Sommer and J. Potter. Operating system extensions for dynamic real-time applications. In *Proc. 17th Real-Time Systems Symposium*, pages 45–50, December 1996.

[32] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE/ACM Trans. Networking*, 1(5):554–565, October 1993.

[33] D. K. Y. Yau and S. S. Lam. An architecture towards efficient OS support for distributed multimedia. In *Proc. Multimedia Computing and Networking (MMCN '96)*, January 1996.

[34] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource ReSerVation Protocol. *IEEE Network*, pages 8–18, September 1993.