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
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*ashish,ashaikh,zaher,zqwang,kgshin*}@*eecs.umich.edu*

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. Typically, the support is in the form of suitable extensions to the communication subsystem and the underlying operating system for specification and maintenance of QoS guarantees. 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. We separate the policies from mechanisms in each component, demonstrating a communication framework that can implement alternative QoS models by applying appropriate policies. We also report the results of a detailed execution profile of the system to characterize communication costs for the purposes of admission control. 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 involved in 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

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.

role in operating system design [1, 2], and are being extended to support real-time and multimedia applications [3]. 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. The issues and techniques explored in this paper are also relevant to contemporary monolithic operating systems such as UNIX and its variants.

To meet the timeliness and predictability requirements of applications, our service architecture includes three primary components: (i) `RTC API`, the programming interface exported to applications that wish to use the service; (ii) `RTCOP`, a protocol that coordinates end-to-end signaling, admission control, and QoS-sensitive resource allocation and reclamation; and (iii) `CLIPS`, a resource management library that handles real-time data transfer and implements QoS-sensitive CPU scheduling for protocol processing and link scheduling of packet transmissions. These components together ensure QoS-sensitive handling of network traffic at sending *and* receiving hosts.

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 can be regarded as the fourth 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.

We realize this service as a user-level server, even though server-based protocol stacks perform poorly compared to user-level protocol libraries or in-kernel implementations [4, 5], for a number of reasons. A server configuration considerably eases software development and debugging, particularly the location and correction of timing-related bugs. Also, since several applications can establish multiple QoS connections, admission control and run-time resource management of these connections must be localized within one resource management domain. A protocol library in which resource management is distributed would preclude this localization. For most contemporary operating systems, this corresponds to a single protection domain and address space, in our case the server. Once developed and debugged, the server can be placed within the kernel to improve performance and predictability.

For application-level QoS guarantees, an end system must provide adequate computation as well as communication resources to one or more applications executing simultaneously. 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 [6] or appropriate system partitioning [7]. We envision a system structure with the *communication subsystem* distinct from the *computation subsystem*. The communication subsystem comprises all activities and resources that participate in transmitting data and processing received data from the network. The computation subsystem, on the other hand, comprises all application processes and threads that perform activities other than communication processing. The two subsystems exchange network data through appropriate buffers and queues in memory. Available CPU resources are then shared between the two subsystems via appropriate CPU partitioning and multiplexing, details of which are beyond the scope of this paper.

We believe that the communication subsystem presents a resource management domain distinct from that presented by 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 and extensible 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 different architectural components providing QoS guarantees with local communicaiton resources and management policies, support for dynamic scheduling of all communication processing, and detailed prameterization 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.

Our design approach and lessons learned are applicable to communication subsystems realized as user-level libraries, co-located kernel servers, or integrated kernel implementations. Specifically, 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 [8, 9].

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 comprising the architecture are described in Section 4, with an emphasis on their internal design, interaction, and support for real-time behavior. Section 5 describes our prototype implementation and the issues faced in its realization on a platform with limited real-time support. System profiling and parameterization of the platform and our implementation are presented in Section 6, followed in Section 7 by results of an experimental evaluation to determine the efficacy of our implementation in providing QoS guarantees. Finally, Section 8 concludes the paper with a summary and directions for future work.

2 Related Work

In this section we compare and contrast related work in QoS-sensitive communication and computation with the contributions made by this paper. 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 [10], which provides a comprehensive view of the state of the art in the provisioning of end-to-end QoS. In the discussion below, we highlight a subset of these approaches, focusing on enhancements and the associated implications for end hosts. We first highlight communication architectures for QoS, followed by related work in multimedia and real-time operating systems, and then discuss approaches to QoS negotiation and adaptation.

Network and protocol support for QoS: The Tenet real-time protocol suite [11] is an implementation of real-time communication on wide-area networks (WANs), but it did not consider incorporation of protocol processing overheads into the network-level resource management policies. In particular, the above efforts do not address the problem of QoS-sensitive protocol processing inside hosts. Further, they do not consider the incorporation of 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 [12–15]. Several classes of service are being considered, including guaranteed service (similar to our work) which provides guaranteed delay bounds, and controlled load service which has more relaxed QoS requirements. The expected QoS requirements of applications and issues involved in sharing link bandwidth across multiple classes of traffic are explored in [16]. The signalling required to set up reservations for application flows can be provided by RSVP [17], which initiates reservation setup at the receiver, or ST-II [18], which initiates reservation setup at the sender (similar to `RTCOP`). We note that the architectural approach, mechanisms, and extensions developed in this paper are applicable to unicast as well as multicast sessions, for both sender-initiated and receiver-initiated signalling.

QoS architectures: The OMEGA [19] end point architecture provides support for end-to-end QoS guarantees. The primary focus of OMEGA is development of an integrated framework for the specification and translation of application QoS requirements and allocation of the necessary resources. Application QoS requirements are translated to network QoS requirements by the QoS Broker [20], which negotiates for the necessary host and network resources. The OMEGA approach 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 [21] is a layered architecture focusing on QoS provisioning within the communication subsystem and the network. It provides features such as end-to-end admission control, resource reservation, QoS translation between layers, and QoS monitoring and maintenance. QoS-A specifies a functionally rich and general architecture supporting networked multimedia applications. Practical realization of QoS-A, however, would necessitate architectural mechanisms and extensions similar in flavor to the ones presented in this paper. A novel RSVP-based QoS architecture supporting integrated services in TCP/IP protocol stacks, running on legacy (e.g., Token Ring and Ethernet) and high-speed ATM LAN networks is described in [8]. A native-mode ATM transport layer has been designed and implemented in [22]. These architectures also provide support for traffic policing and shaping; however, no support is provided for scheduling protocol processing and incorporation of implementation overheads and constraints.

Operating system support for QoS-sensitive communication: Real-time upcalls (RTUs) [23] are a mechanism to schedule protocol processing for networked multimedia applications via event-based upcalls [24]. Protocol processing activities are scheduled via an extended version of the rate monotonic (RM) scheduling policy [25]. Similar to our approach, delayed preemption is adopted to reduce the number of context switches. Our approach differs from RTUs in that we use a thread-based execution model for protocol processing, schedule threads via a modified earliest-deadline-first (EDF) policy [25], and extend resource management policies within the communication subsystem to account for a number of implementation overheads and constraints.

Similar to our approach, rate-based flow control of multimedia streams via kernel-based communication threads is also proposed in [26]. However, in contrast to our notion of per-connection threads, a coarser notion of per-process kernel threads is adopted. This scheme is clearly not suitable for an application with multiple QoS connections, each with different QoS requirements and traffic characteristics. Mechanisms for scheduling multiple communication threads, and the issues involved in reception side processing, are not considered. More importantly, the architecture outlined in [26] does not consider provision of 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 the notion of paths as a fundamental operating system structuring technique [27]. 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 [28], which is similar to Scout paths, provides a rich framework for development of real-time communication services, especially communication resource management, for distributed applications. Paths in [28] are envisioned primarily as a static, relatively coarse-grain mechanism, while Scout paths are not associated with communication resources or assigned priorities via admission control. 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 [6] have been combined with user-level protocol processing [4] for predictable protocol processing inside hosts [29]. However, no support is provided for traffic enforcement or the ability to control protocol processing priority separate from application priority. Several QoS-sensitive CPU scheduling policies have also been proposed recently [30–32]. These schemes can be utilized for network bandwidth allocation, but do not suffice for managing all available communication resources.

Receive livelock elimination: Recent efforts have also addressed an important problem associated with data reception, namely, receive livelock [33]. Receive livelock has been addressed at length in [34] via a combination of techniques (such as limiting interrupt arrival rates, fair polling, processing packets to completion, and regulating CPU usage for protocol processing) to avoid receive livelock and maintain system throughput near the maximum system input capacity under high load. Another approach is to schedule applications in a proportional share manner and use the cumulative rate to limit packet processing to solve the receive livelock problem [?]. Lazy receiver processing (LRP) [35], while not completely eliminating it, significantly reduces the likelihood of receive livelock even under high input load. In LRP, an incoming packet is classified and enqueued, but not processed, until the application receives the data.

While LRP works well for receive-livelock elimination for best-effort traffic, the architectural approach presented in this paper accommodates QoS-sensitive traffic. Similar to LRP, our approach also utilizes early demultiplexing and path or channel-specific queuing of incoming packets. However, packet processing and message reassembly is performed in a QoS-sensitive fashion via EDF scheduling of channel handlers, as and when communication capacity is made available. Demultiplexing incoming packets early and absorbing bursts in distinct per-connection queues is an attractive way to prevent receive livelock, an observation also made in the context of paths in Scout [27]. Our architectural approach facilitates provision of QoS guarantees while preventing receive livelock.

Dynamic QoS negotiation and adaptation: Since a broad class of multimedia applications are soft real-time in nature, i.e., can tolerate limited fluctuations in the delivered QoS, several research efforts have explored the issues involved in supporting QoS negotiation and adaptation functions at end hosts. The AQUA system [36] is one such effort which has developed QoS negotiation and adaptation support for allocation of CPU and network resources. Similarly, a QoS-adaptive transport system is described in [37] that incorporates a QoS-aware API and mechanisms to assist applications to adapt to fluctuations in the delivered network QoS. A scheme for adaptive rate-controlled scheduling is presented in [38]. QoS negotiation and adaptation support has also been developed for real-time applications [39], which provides support for specification of QoS compromises and supports graceful QoS degradation under overload or failure conditions.

While we do not consider dynamic QoS negotiation and adaptation in this paper, most of the architectural mechanisms and enhancements provided can be utilized for such scenarios. For example, our service architecture provides mechanisms to enforce application traffic contracts and generate notifications which applications can use to adapt. Future incarnations of our guaranteed-QoS service will include support for dynamic QoS negotiation and adaptation.

3 Real-Time Communication Service Architecture

Our primary goal is to provide applications with a service with which they can request and utilize guaranteed-QoS unicast connections between two hosts. In this section, we hilight the architectural components of unicast communication that, together with a set of user-specified policies, can implement several real-time communication models. The overall service is currently being utilized in the `ARMADA` project [40], which aims to implement a set of communication and middleware services that provide support for end-to-end guarantees and fault-tolerance for embedded real-time distributed applications.

3.1 Architectural Requirements

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 [41]. Earlier work in [41] 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 on microkernel-based operating systems with limited real-time support.

Figure 1 illustrates the high-level software architecture of our guaranteed-QoS service at end-hosts. The

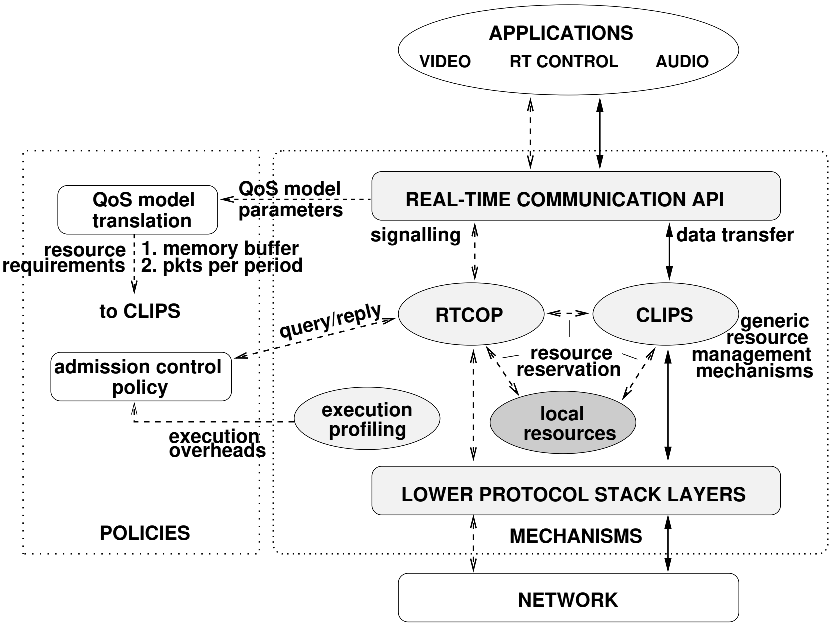


Figure 1: **Real-time communication service architecture**: Our implementation consists of four primary architectural components: an application programming interface (`RTC API`), a signalling and resource reservation protocol (`RTCOP`), support for resource management and run-time data transfer (`CLIPS`), and execution profiling support. Dashed lines indicate interactions on the control path while the data path is denoted by the solid lines.

core functionality of the communication service is realized via three distinct 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.

3.2 Architecture Components

Below we provide an overview of the components of the service architecture; Section 4 elaborates on the details of the core components (`RTC API`, `RTCOP`, and `CLIPS`) involved with actual connection establishment and communication. Section 6 describes the profiling component that captures and represents communication subsystem performance when the system is re-targeted on new or upgraded platforms.

Invocation via RTC API: Applications request establishment and teardown of guaranteed-QoS connections, and perform data transfer on these connections, by invoking routines exported by the `RTC API`. The design of the API has been significantly influenced by the structure of the sockets API in BSD Unix [42] and its variants. QoS parameters of real-time connections are translated into abstract memory, CPU, and network link bandwidth requirements for use in admission control and resource allocation.

Signalling via RTCOP: End-to-end signalling is performed by `RTCOP` to establish and teardown connections across the communicating hosts, possibly via multiple network hops. `RTCOP` provides reliable datagram semantics for signalling requests and replies between nodes, and implements consistent connection state management mechanisms at each node. `RTCOP` uses an admission control policy that depends on the particular service model, and invokes it on each host to verify the feasibility of QoS guarantees.

Data transfer via CLIPS: `CLIPS` implements a generic communication resource management mechanism. Given a set of abstract resource requirements, `CLIPS` facilitates real-time data transfer on established connections, and manages reserved CPU and link resources to maintain QoS guarantees.

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.

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 [39].

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 during data transfer 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.

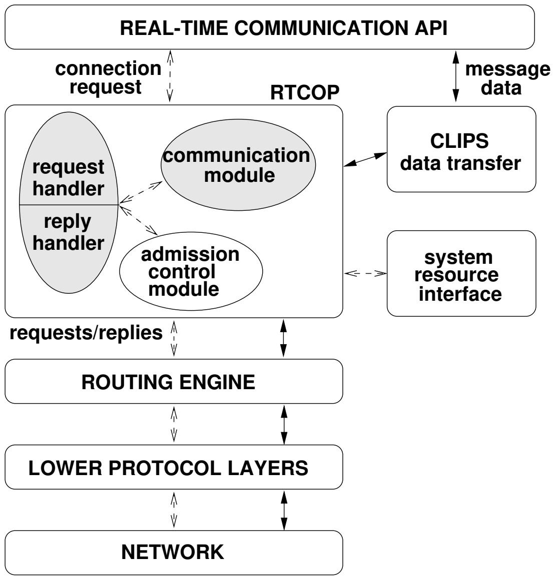
Routines	Parameters	Invoked By	Function Performed
Miscellaneous			
`rtcInit`	none	both	service initialization
`rtcGetParameter`	chan id, param type	both	query parameter on specified real-time connection
Signalling			
`rtcRegisterPort`	local port, agent function	receiver	register local port and agent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

Table 1: **Routines comprising** `RTC API`: This table shows the utility, signalling, and data transfer functions that constitute the application interface. The table shows each function name, its parameters, the endpoint that invokes it, and a brief description of the operation performed.

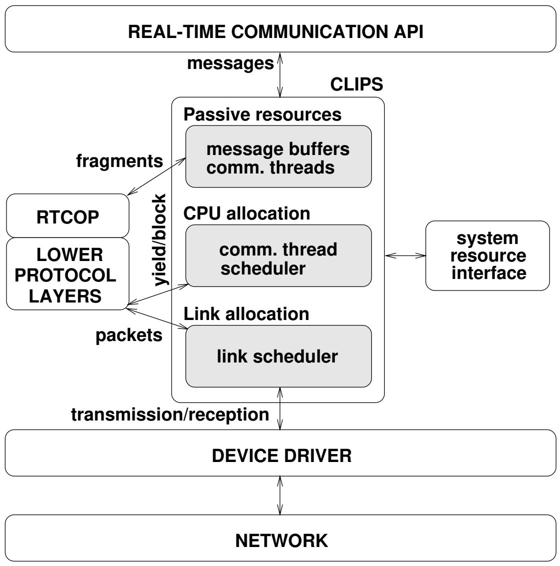
The design of `RTC API` is based in large part on the well-known socket API in BSD Unix. Each connection endpoint is a pair `(IPaddr, port)` formed by the IP address of the host (`IPaddr`) and an unsigned 16-bit port (`port`) unique on the host, similar to an INET domain socket endpoint. In addition 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.

Each of the signalling and data transfer routines in Table 1 has its counterpart in the socket API. For example, the routine `rtcRegisterPort` corresponds to the invocation of `bind` and `listen` in succession, and `rtcAcceptConnection` corresponds to `accept`. Similarly, the routines `rtcCreateConnection` and `rtcDestroyConnection` correspond to `connect` and `close`, respectively.

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. It may also perform authentication checks based on the requesting endpoint specified in



(a) RTCOP structure



(b) CLIPS structure

Figure 2: **Internal structures and interfaces:** In this figure we show the internal functional structure of RTCOP and CLIPS along with their respective interfaces to other components. In (a), data and control paths are represented with solid and dashed lines, respectively.

the signalling request. This is unlike the establishment of a TCP connection, for example, which is completely transparent to the peer applications.

The QoS-parameters passed to `rtcCreateConnection` for connection establishment 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 Signalling and 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. As illustrated in Figure 2(a), RTCOP is composed primarily of two relatively independent modules. The *request and reply handlers* manage signalling state and interface to the admission control policy, and the *communication module* handles the tasks 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.

The request and reply handlers generate and process signalling messages, interface to RTC API and CLIPS, and reserve and reclaim resources as needed. When processing a new signalling request, the request handler invokes a multi-step admission control procedure to decide whether or not sufficient resources are available for the new request. As a new connection request traverses each node of the route from source to destination, the request handler invokes admission control which decides if the new connection can be locally admitted. Upon successful admission, the handler passes the request on to the next hop. 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. As the notification is received at each hop, the reply handler commits connection resources. These resources include packet and message buffers, and a periodic connection handler thread with a specified execution budget. The reply handler notifies admission control of connection establishment so that it may account for the corresponding CPU and link bandwidth consumption in its future decisions. Note that whether these resources are set aside for the connection, or multiplexed among several connections, is a policy decision implemented by admission control independent of `RTCOP`.

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. As noted earlier, signalling messages are transported as best-effort traffic, but are delivered reliably using source-based retransmissions. Reliable signalling ensures that a connection is considered established only if connection state is successfully installed and sufficient resources reserved at all the nodes along the route. The communication module implements duplicate suppression to ensure that multiple reservations are not installed for the same connection establishment request. Similar considerations apply to connection teardown where all nodes along the route must release resources and free connection state. Consistent connection state management at all nodes is an essential function of `RTCOP`.

`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 for Data Transfer

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. `CLIPS` implements a traffic policing mechanism, as well as its own default admission control policy that can be overridden by a user-specified alternate admission control policy. Note that `CLIPS` interface parameters correspond precisely to the abstract resource requirements that are relayed by the API from the application. The additional deadline parameter is equal to the period by default, unless its value is set by admission control in accordance with particular QoS model-specific requirements. We use CLIPS to provide connection endpoints with QoS-sensitive allocation of CPU and link resources. The real-time communication service described in this paper uses a subset of CLIPS features: the complete library includes support for QoS adaptation and resource monitoring as detailed in [43].

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(b).

QoS-sensitive CPU allocation: 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. The scheduler then 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.

Communication handlers (implemented by CLIPS) execute a user-defined protocol stack, then return to CLIPS code after processing each message or packet. Ideally, each clip should be assigned a *CPU budget* to prevent a communication client from monopolizing the CPU. Since processor capacity reserves are not available on most operating systems, the budget is indirectly expressed in terms of a maximum number of packets to be processed within a given period. The handler blocks itself after processing the maximum number of packets allowed within its stated time period.

Policing and communication thread scheduling: As mentioned above, communication threads eligible for CPU allocation are multiplexed on the CPU by a communication thread scheduler which supports dynamic handler priorities. To police non-conformant sources, the handler is blocked when its budget expires and there are pending messages. Thus, associating a budget with each connection handler facilitates traffic enforcement. This is because a handler is scheduled for execution only when the budget is non-zero, and the budget is not replenished until the next (periodic) invocation of the handler. These mechanisms together ensure that high-priority misbehaving connections do not consume excessive system resources at the expense of lower priority connections. We do not assume that the underlying operating system kernel supports fully preemptive scheduling. Instead, we implement a "cooperative preemption" mechanism that prevents handlers with large periods and budgets from inflicting unacceptable jitters on the execution of handlers with smaller periods.

Each handler participates in cooperative preemption by voluntarily yielding the CPU after processing a certain (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 CPU budget, or when it yields the CPU.

QoS-sensitive link bandwidth allocation: Modern operating systems 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 in a prioritized fashion. 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 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 [44,45]. 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. The algorithm determines relative connection priority so that QoS requirements for all admitted connections are satisfied. Details on `D_order` and subsequent extensions to account for CPU preemption costs and the relationship between CPU and link bandwidth are available in [45] and [46], respectively.

5 Service Implementation

In this section we describe how the architectural components described in the preceding sections can be implemented on a realistic platform. 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.

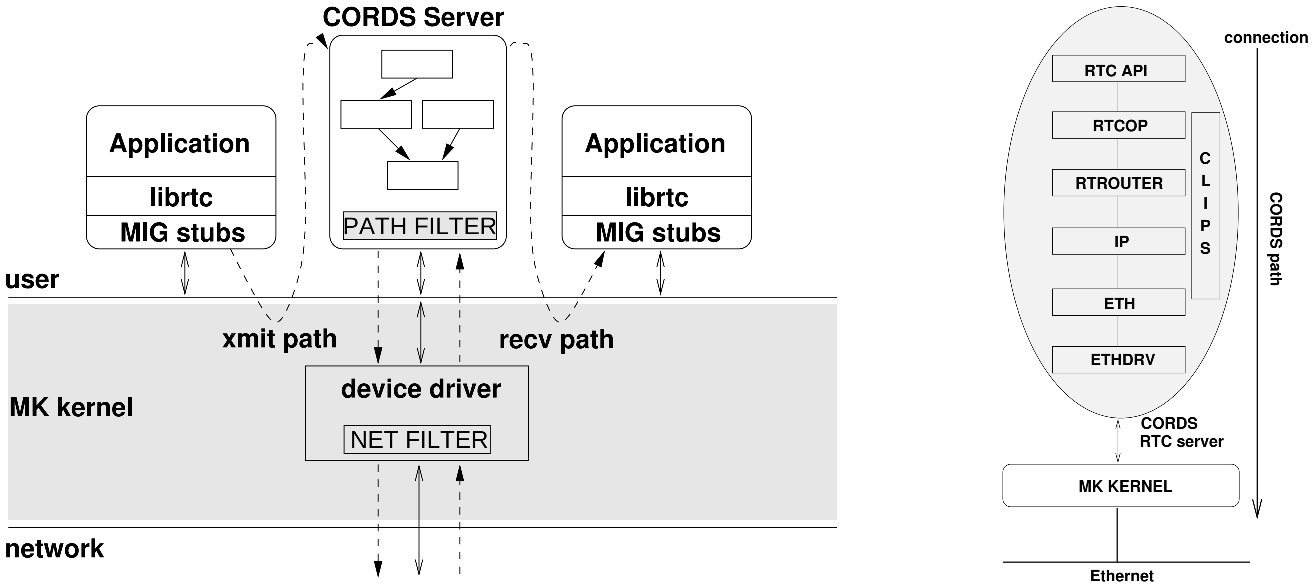
While not a full-fledged real-time OS, MK 7.2 includes several features that facilitate provision of QoS guarantees. Specifically, though it provides only preemptive fixed-priority scheduling, the 7.2 release includes the `CORDS` (Communication Object for Real-time Dependable Systems) protocol environment [28] in which our implementation resides. `CORDS` is based on the x-kernel object-oriented networking framework originally developed at the University of Arizona [47], 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.

We focus on our experiences with the utilization of available kernel-level features to realize real-time communication. We identify limitations common to contemporary microkernel systems, and describe our solutions to mitigate their effect on our ability to provide real-time guarantees. In addition, we note features that aided in realization of the service along with how they were used in the implementation.

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 of which is 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 the we currently use default IP routing, we provide `RTROUTER` as a go-between protocol to keep the routing interface independent of IP. `RTROUTER` is intended to allow `RTCOP` to 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 `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.



(a) Server configuration

(b) CORDS protocol stack

Figure 3: **Service implementation as CORDS server:** The left figure shows the communication path used by applications wishing to use guaranteed-QoS service. The right figures illustrates the configurable protocol stack implemented in the server which handles admission control, run-time resource management, and connection state maintenance in a single protection domain.

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 the effects of deficiencies 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 application 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 [4, 5]. 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 the large data transfers performed via the guaranteed-QoS communication service, although it may affect connection admissibility.

Network device interface: A server-based implementation presents a number of significant 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 a link scheduler implemented by `CLIPS` as close as possible to the device driver without being in the kernel. However, being in user space, the link scheduler cannot be invoked directly by the kernel device driver in response to transmission completion interrupts unless the underlying 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 [28]. 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 as early as possible, 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 underlying kernel has support for 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 has two main disadvantages. First, it does not respect connection QoS requirements, since urgent packets can suffer unbounded priority inversion when preceded by an arbitrary number of less urgent packets in the queue. Second, 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 kernel-level IPC queues (Mach port queues in our implementation); our strategy is to drain them as fast as possible, transferring messages to connection-specific queues in the communication server. These queues 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.

Global path name space: Since our real-time communication service aims to provide QoS guarantees on a per-connection basis, it is natural to assign a distinct path to each connection. In order to realize our end-to-end service architecture, however, traffic on a particular path must be serviced according its QoS by the communication subsystem on *all* hosts and routers. This in turn requires that the path name space be global across all hosts participating in the real-time communication service. To realize a global path name space, we concatenate the unique host name and a sender-based connection identifier to construct unique path identifiers for real-time connections. Note that though there is a one-to-one mapping of path identfier to connection identifier, applications have no knowledge of paths; they use only connection identifiers in their operations.

Dynamic path creation/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

Anchor Parameter	Anchor Routine	Message Size 1 byte	1k bytes	10k bytes	30k bytes
C_a	Send message	301	331	606	1191
$C_a^{r,w}$	Receive message (msg waiting)	335	350	1097	2653
$C_a^{r,nw}$	Receive message (none waiting)	54	54	53	54
$C_a^{r,cb}$	Receive message callback	95	92	96	96

Table 2: Data transfer overheads in RTC API anchor (in μs).

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 hop-by-hop signalling of teardown messages from connection source to destination.

6 System Profiling and Parameterization

To provide predictable QoS guarantees on a given platform, the system costs and overheads (parameters) must be determined accurately for effective admission control. In this section we describe the detailed profiling of our service implementation to obtain system parameters. Our profiling methodology is to conduct all measurements on two hosts connected by an isolated Ethernet segment. The service library librtc and the CORDS protocol stack are instrumented appropriately to perform the desired measurements. Only one set of measurements are performed at a time in order to minimize the perturbation induced by the profiling code.

Given our primary focus on run-time resource management, we concentrate on the data transfer performance of our prototype implementation, for both incoming and outgoing data. As mentioned in Section 4.4, we use real-time channels as the service model. For all the results reported here, a single real-time channel is created from a sending client on one host to a receiving client on another host. Figure 4 shows various costs incurred during data transfer on the sending host as well as on the receiving host. These overheads can be categorized into three components as illustrated in the figure. We first present profiling results for RTC API, followed by the results of profiling the remaining protocol stack layers, and finally show results for link input and output overhead. The results are averaged over 500 samples.

6.1 Profiling the RTC API

In this set of measurements, we profile routines in RTC API that handle message sending and receiving. The overheads of these routines constitute API cost in Figure 4. RTC API routines are invoked either in response to application send/receive requests, or in response to message arrival. Since the application invokes the receive primitive asynchronously with message arrival we distinguish between the case where some message is already waiting at the time the application calls the receive primitive, and the case where the message queue is empty.

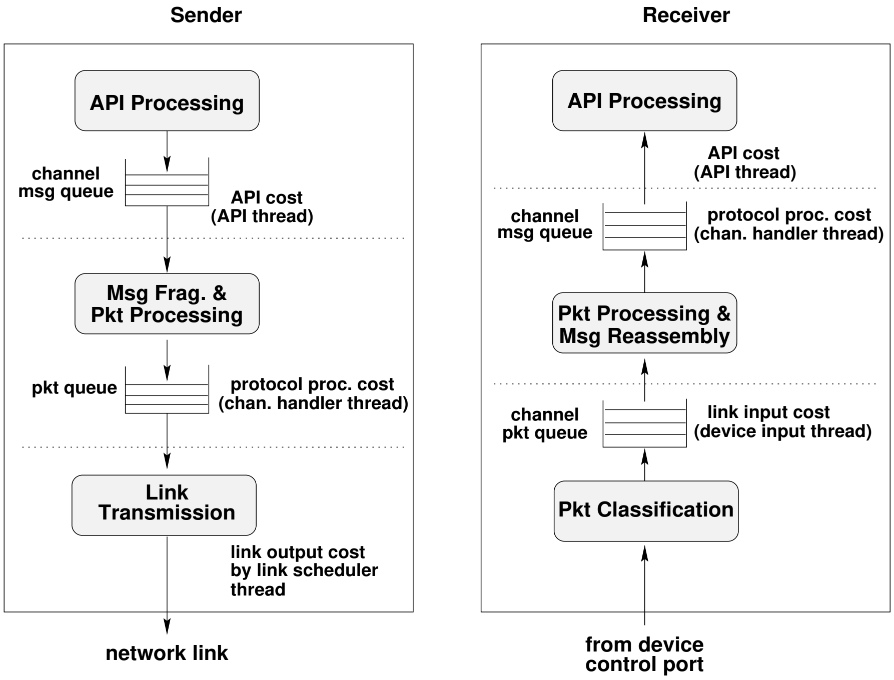


Figure 4: **Processing overheads source and destination hosts:** As shown in this figure, the costs at both sender and receiver is divided into three components, (i) API cost, (ii) fragmentation or reassembly and associated protocol processing, and (iii) link access overhead.

Table 2 lists the data transfer overheads of these anchor routines for several messages sizes and defines symbols to refer to each cost component. Note that $C_a^{r,nw}$ only includes the time from entry into `RTC API` till the time that the application thread is blocked waiting for a message to arrive. When a message arrives later, the channel handler thread will invoke the receive message callback function. This overhead is denoted by $C_a^{r,cb}$.

For each routine, the measurements for a message size of 1 byte essentially correspond to the fixed overhead introduced by the routine. Beyond the fixed overhead, the time spent in this routine increases with message size. `RTC API` copies application data into path-specific message buffers in order to preserve application data integrity in the worst case. For our platform, lmbench [48] reports a memory copy bandwidth of ≈ 40 MB/second; The increase in C_a can be largely attributed to the time to copy in application data.

An anomalous trend is observed for $C_a^{r,w}$. If the message is already waiting, it is dequeued and copied into path-specific message buffers before returning it to the application. Once again, the overhead increases with message size. For message sizes of 10K bytes and 30K bytes, the overhead cannot be attributed only to the cost of copying the message. We believe this is partially due to the overhead introduced by the memory allocation primitive provided in `CORDS`.

Consistent behavior is observed for $C_a^{r,nw}$. As expected, $C_a^{r,nw}$ is independent of message size, since `RTC API` effectively blocks the application if no message is waiting. For $C_a^{r,cb}$ the overhead measurements are also independent of message size. This is completely consistent since the receive message callback function simply signals the blocked API thread, if there is one. Note that the reported measurements correspond to the case

Service Library Routine	Message Size			
	1 byte	1k bytes	10k bytes	30k bytes
rtcSendMessage	1170	1210	1480	2070
rtcRecvMessage (msg waiting)	870	894	1660	3210

Table 3: Application-level send and receive latencies (in μs).

where the application is blocked waiting for a message, which is the worst-case scenario for $C_a^{r,cb}$.

We also measure application-level latencies for the `rtcSendMessage` and `rtcRecvMessage` routines. These latencies include the cost of an IPC call to the server and executing the corresponding anchor routine discussed above. As shown in Table 3, the application-level latency for `rtcSendMessage` follows a pattern similar to that of C_a, except that the fixed overhead is significantly higher. This extra overhead is the cost of a send IPC to the `CORDS` server. Comparing `rtcSendMessage` latency to C_a, the average extra overhead is $\approx 875\ \mu s$.

A similar observation can be made for the `rtcRecvMessage` latency, which follows a pattern similar to that of $C_a^{r,w}$. We observe that the latency for `rtcRecvMessage` reveals an average extra overhead of ≈ 563 μs, which accounts for the cost of a receive IPC from the application to the `CORDS` server across the MIG interface.

While the `RTC API` overheads are relatively high, these measurements are for an unoptimized implementation and can be improved substantially with careful performance optimizations. With appropriate buffer management and API buffering semantics [49, 50] it may even be possible to completely eliminate the copying of data within `RTC API`. However, more immediately we are concerned with ensuring that the overheads incurred in `RTC API` do not result in QoS-insensitive handling of data. We address this concern by accounting for the costs with appropriate admission control. While the application-level latencies are also high, this is primarily due to significant IPC overhead, which may be reduced with a colocated in-kernel `CORDS` server.

6.2 Protocol Processing Costs

For the other layers (`RTCOP`, `RTROUTER`, `IP`, `ETH`, `ETHDRV`) we only measure the aggregate overhead. Table 4 lists the profiling results for all layers other than data link-level transmission and reception, which is profiled separately in Section 6.3 below. The overheads of these layers correspond to the protocol processing cost in Figure 4.

Messages are enqueued (dequeued) at the top of the protocol stack for (by) a communication handler thread. The thread then executes the protocol stack. The average message enqueue/dequeue cost is $\approx 5\ \mu s$. For the send path, we distinguish the measured overhead for the first packet of a message from that for the other packets. Note that the aggregate protocol processing cost for the first packet is significantly higher than that for the other packets. This difference can be partly attributed to cache effects, which have been shown to affect protocol stack execution latency significantly [51, 52].

In the receive path, the message reassembly cost is only incurred during the processing of the last packet of a message. The average reassembly cost increases roughly linearly with message size, from 17 μs for message

Data Path	Protocol Stack Layer	Packets in Message	
		First	Other
Send	fragmentation	53	30
	RTCOP + RTROUTER + IP + ETH + ETHDRV	128	47
Receive	RTCOP + RTROUTER + IP + ETH + ETHDRV	260	

Table 4: Protocol stack latencies for send and receive paths (in μs).

size of 1.4K to 239 μs for message size of 28K. The receive path overhead listed in Table 4 shows the aggregate per-packet protocol processing cost, excluding message reassembly, which is the same for all packets of the message.

6.3 Link Input/Output Overhead

For a complete parameterization of the communication subsystem, we also profile packet transmissions by the link scheduler at the sending host, and packet reception by the CORDS server at the receiving host. These overheads are shown in Figure 4 as link output/input cost.

Table 5 lists the packet transmission latencies measured in the link scheduler as a function of the packet size. Once again, the latency measurement for 1-byte packets roughly corresponds to the fixed overhead. This overhead includes the cost of a user-kernel context switch, invocation of the device driver transmit routine, handling of the transmission-complete interrupt, delivering of an I/O-complete notification to the Mach device control port (which in turn wakes up the waiting link scheduler), and another context switch to resume execution of the link scheduler thread. Since the measured latency includes the time to transmit the entire packet on the wire, larger packets incur higher latencies.

Table 6 lists the overhead incurred by the CORDS device input thread to receive an incoming packet from the device control port, classify it to determine its path, locate the corresponding buffer pools, enqueue the packet in the path input packet queue, and signal the CLIPS CPU scheduler to wake up the input communication handler associated with this path. Note that the input overhead for the first packet is significantly higher than that for the subsequent packets. We have verified that this is primarily due to the high overhead to signal the handler on the arrival of the first packet.

The CORDS device input thread overhead does not include the in-kernel cost of fielding the packet arrival interrupt, accepting the arrived packet, applying the generic net filter and dispatching the packet to the appropriate device control port. Preliminary in-kernel measurements reveal this overhead to be $\approx$ 650 μs for an Ethernet MTU-sized packet.

7 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

ETHDRV Layer	Packet Size		
	1 byte	500 bytes	1416 bytes
Packet transmission by link scheduler	673	1510	1775

Table 5: Link scheduler packet transmission latencies (in μs).

ETHDRV Layer	Packets in Message	
	First	Other
Packet input by `CORDS` device input thread	120	20

Table 6: `CORDS` device input thread overhead (in μs).

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 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. 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.

7.1 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 5, 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.

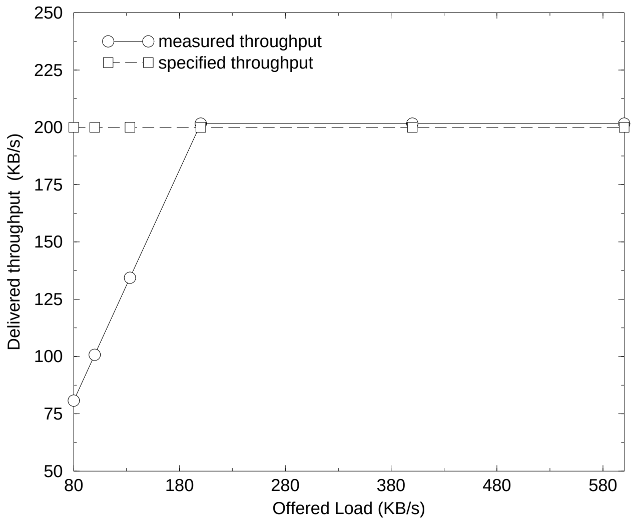


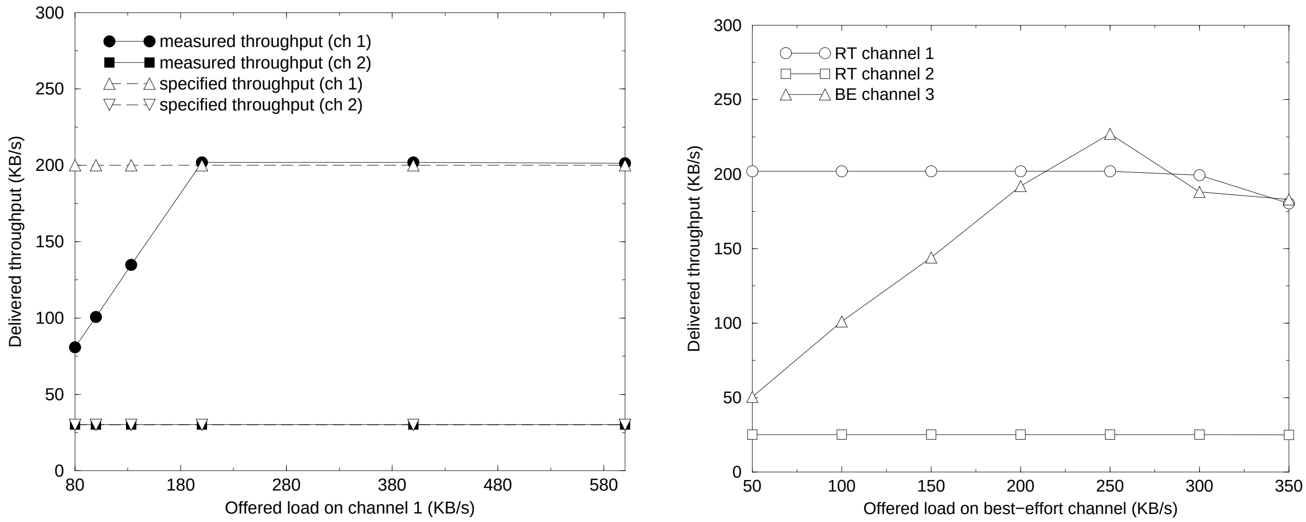
Figure 5: **Traffic enforcement on a single real-time channel:** The enforcement mechanism prevents the delivered throughput from exceeding the specified rate, even as inter-message generation time goes to 0 ms. The channel has a traffic specification of M_{max} = 40 KB, B_{max} = 10, R_{max} = 5 messages/second, and deadline of 200 ms.

7.2 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 6(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



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

Figure 6: **Traffic isolation:** The left graph shows that traffic specification violation on real-time channel 1 does not affect the QoS for the other real-time channel 2. Channel 1 has the same traffic specification as in Figure 5 and channel 2 has a traffic specification of B_{max} = 10, M_{max} = 15 KB, R_{max} = 2 messages/second, and deadline of 100 ms. The right graph shows that increasing best-effort load does not interfere with real-time channel throughput. In this graph channel 1 has a specification of B_{max} = 10, M_{max} = 20 KB, R_{max} = 10 messages/second and deadline of 200 ms and channel 2 has B_{max} = 10, M_{max} = 5 KB, R_{max} = 5 messages/second and deadline of 100 ms.

best-effort channel (channel 3) is varied from 50 KB/s to 350 KB/s by controlling the message generation interval.

Figure 6(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.

7.3 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

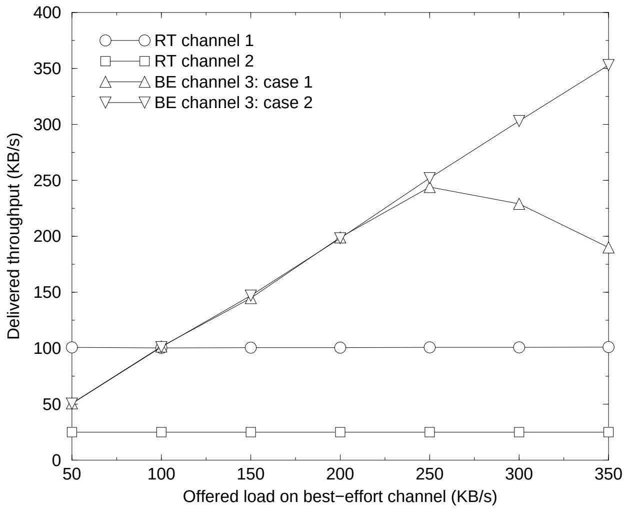


Figure 7: **Traffic isolation and unused capacity utilization:** When real-time connections do not used their specified capacity, best-effort traffic should utilize the excess bandwidth. Channels 1 and 2 have the same parameters as in Figure 6(b) but channel 1 underutilizes its reservation. In case 1, channel 1 generates 20 KB messages at 5 msg/s; in case 2, it generates 10 KB messages at 10 msg/s. When traffic is very bursty (case 1), packet loss at the receiver prevents full utilization of unused capacity. We see in case 2, though, that best-effort traffic is able to fully utilize unused bandwidth with reduced real-time traffic burstiness.

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 7 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 6(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 7, 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).

7.4 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. 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.

8 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. In realizing this service, 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.

Though we implemented the service architecture on a specific platform, many of the methods employed to circumvent kernel limitations are applicable to other microkernel and traditional monolithic operating systems. Our experimental results demonstrate that the architectural features provided in the service are effective in providing QoS guarantees to individual real-time connections while maintaining fairness to best-effort traffic. These results also reveal deficiencies of a server-based implementation, especially at the receiving host, that could be largely resolved by migrating the service to the kernel.

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. We are also currently extending the communication architecture to allow for QoS-adaptation to available host and network resources. We have implemented an end-host architecture for adaptive-QoS communication services [43] and plan to use components of the architecture presented in this paper to implement an end-to-end adaptive QoS communication scheme. In Section 6 we described the complex process of parameterizing the overheads of the communication subsystem and target platform. The efforts involved in detailed manual profiling on each target platform 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 [53].

References

[1] H. Custer, *Inside Windows NT*, Microsoft Press, One Microsoft Way, Redmond, Washington 98052-6399, 1993.

[2] D. G. Korn, "Porting UNIX to windows NT," in *Proc. USENIX Winter Conference*, January 1997.

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

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

[5] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," *IEEE/ACM Trans. Networking*, vol. 1, no. 5, pp. 554–565, October 1993.

[6] 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.

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

[8] 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.

[9] 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.

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

[11] 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*, vol. 4, no. 1, pp. 1–11, February 1996.

[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*, pp. 14–26, August 1992.

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

[14] J. Wroclawski, "Specification of the controlled-load network element service," *Request for Comments (RFC 2211)*, September 1997.

[15] 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.

[16] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 4, pp. 365–386, August 1995.

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

[18] 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.

[19] K. Nahrstedt and J. M. Smith, "Design, implementation and experiences of the OMEGA end-point architecture," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1263–1279, September 1996.

[20] K. Nahrstedt and J. M. Smith, "The QoS broker," *IEEE Multimedia*, vol. 2, no. 1, pp. 53–67, Spring 1995.

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

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

[23] 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*, p. 231, December 1995.

[24] D. D. Clark, "The structuring of systems using upcalls," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 171–180, 1985.

[25] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in hard real-time environment," *Journal of the ACM*, vol. 1, no. 20, pp. 46–61, January 1973.

[26] 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.

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

[28] 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.

[29] 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.

[30] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, PhD thesis, Technical Report, MIT/LCS/TR-667, Laboratory for CS, MIT, September 1995.

[31] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time time-shared systems," in *Proc. 17th Real-Time Systems Symposium*, pp. 288–299, December 1996.

[32] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *Proc. 2nd OSDI Symposium*, pp. 107–121, October 1996.

[33] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, February 1993.

[34] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Winter USENIX Conference*, January 1996.

[35] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd OSDI Symposium*, pp. 261–275, October 1996.

[36] L. Krishnamurthy, *AQUA: An Adaptive Quality of Service Architecture for Distributed Multimedia Applications*, PhD thesis, University of Kentucky, 1997.

[37] A. T. Campbell and G. Coulson, "QoS adaptive transports: Delivering scalable media to the desktop," *IEEE Network Magazine*, pp. 18–27, March/April 1997.

[38] D. K. Y. Yau and S. S. Lam, "Adaptive rate-controlled scheduling for multimedia applications," in *Proc. of ACM Multimedia*, November 1996.

[39] T. Abdelzaher, E. Atkins, and K. Shin, "QoS negotiation in real-time systems and its application to automated flight control," in *Proc. Real-Time Technology and Applications Symposium*, pp. 228–238, June 1997.

[40] 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*, pp. 11–18, San Francisco, CA, December 1997.

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

[42] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, May 1989.

[43] 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.

[44] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368–379, April 1990.

[45] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[46] 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*, pp. 130–138, June 1996.

[47] N. C. Hutchinson and L. L. Peterson, "The *x*-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[48] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proc. USENIX Winter Conference*, pp. 279–295, January 1996.

[49] J. C. Brustoloni and P. Steenkiste, "Effects of buffering semantics on I/O performance," in *Proc. USENIX Symp. on Operating Systems Design and Implementation*, pp. 277–291, October 1996.

[50] B. Murphy, S. Zeadally, and C. J. Adams, "An analysis of process and memory models to support high-speed networking in a UNIX environment," in *Proc. USENIX Winter Conference*, January 1996.

[51] T. Blackwell, "Speeding up protocols for small messages," in *Proc. of ACM SIGCOMM*, pp. 85–95, October 1996.

[52] E. Nahum, D. Yates, J. Kurose, and D. Towsley, "Cache behavior of network protocols," in *Proc. of ACM SIGMETRICS*, pp. 169–180, June 1997.

[53] 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.