Design and Implementation of an RSVP-Based Quality of Service Architecture for an Integrated Services Internet

Tsipora P. Barzilai, Dilip D. Kandlur, *Member, IEEE*, Ashish Mehra, *Member, IEEE*, and Debanjan Saha

Abstract— **The Internet Engineering Task Force (IETF) is currently in the process of overhauling the architecture of the Internet to meet new challenges and support new applications. One of the most important components of that venture is the enhancement of the Internet service model from a classless best effort service architecture to an integrated services architecture supporting a multitude of classes and types of services. This paper presents the design, implementation, and experiences with a protocol architecture for the integrated services Internet. It is based on the emerging standards for resource reservation in the Internet, namely, the RSVP protocol and the associated service specifications defined by the IETF. Our architecture represents a major functional enhancement to the traditional TCP/IP protocol stack. It is scalable in terms of performance and number of network sessions, and supports a wide variety of network interfaces ranging from legacy LAN interfaces, such as Token Ring and Ethernet, to high-speed ATM interfaces. The paper also describes the implementation of this architecture on the IBM AIX platform and our experiences with the system. We then present a performance analysis of the system which quantifies the overheads imposed by all components of the QoS support, such as traffic policing, traffic shaping, and buffer management.**

Index Terms— **Communication protocols, multimedia, operating systems, resource management.**

I. Introduction

IT IS OFTEN difficult, if not impossible, to predict technological trends in the next millennium, except possibly that of the Internet. It seems certain at this point that the explosive growth of the Internet that began in the early parts of the 1990's will continue well beyond the year 2000. As more and more people join the Internet community and as more and more applications find their way onto the information superhighway, the Internet has to evolve faster than ever before to meet the higher demands and greater expectations. Anticipating this pressing need, the Internet Engineering Task Force (IETF) is developing a wide range of protocols and standards to meet the challenges of the next century. One of the most important components of this portfolio is a new and enriched service architecture for the Internet.

The Internet, as it exists today, is a best effort network. As audio and video annotations become common features on Web pages and multimedia applications become ubiquitous on the Internet, the need for "better than best effort" network services will become inevitable. Responding to this challenge, the IETF is on a course to enhance the Internet service architecture from a classless best effort network to an integrated services network supporting a multitude of classes and types of service. In the IETF's vision, and one that we share, applications will be able to request different classes of service and reserve resources in the network and at the hosts using an end-to-end resource reservation protocol (RSVP) [34], [8]. Appropriate resource management at the network elements will guarantee that these reservations are honored and the promised services delivered. The service classes currently under consideration by the IETF are guaranteed [28] and controlled load [32] services.

In order to support integrated services on the Internet [10], the network routers as well as end hosts need to be enhanced to perform classification of traffic on a per-flow basis, create and maintain flow-specific reservation soft states, and handle data packets from different flows in accordance with their service requirements. In this paper we focus on resource management, protocol stack extensions, and device support required at the end hosts to enable an RSVP-based quality of service (QoS) infrastructure in the Internet. More specifically, we concentrate on the design and implementation of QoS support on Unix variant Internet servers.

One of the primary goals in designing this service architecture is to blend the QoS support with the existing TCP/IP stack and socket API, such that the structure of the Unix networking subsystem is preserved. Applications that do not need QoS support for communication should continue to run as is, and yet the QoS extensions should allow new applications to benefit from QoS enhanced network services. The other important design objective is to ensure that the control overheads for QoS support do not have any detrimental impact on data path throughput. Our design is also influenced by the observation that with the rapid penetration of the Web, potentially any Internet site can be a content provider, and hence a source of multimedia data. It follows that not only must the design scale from small to large numbers of connections, it must accommodate differences in the capabilities of the attached network interfaces.

The heart of our QoS architecture is a new kernel module called the QoS Manager. It is entrusted with managing communication-related system resources at the end hosts. Applications or their designated agents can request reservations

Manuscript received January 10, 1997; revised September 29, 1997.

The authors are with the IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA (e-mail: tsipora@watson.ibm.com; kandlur@watson.ibm.com; mehraa@watson.ibm.com; debanjan@watson.ibm.com).

Publisher Item Identifier S 0733-8716(98)00643-X.

0733–8716/98$10.00 © 1998 IEEE

on a network session to the QoS Manager. Typically, applications would use the RSVP application interface for reserving network resources along the path of the data flow, and the RSVP process would act as the designated agent to request local resource reservations from the QoS Manager. We have extended the socket API and created a new protocol family for applications to avail the services of the QoS Manager. In response to a reservation request, the QoS Manager, in cooperation with the network device driver, memory allocator, and the network interface handlers (IFNET's), performs local checks on the availability of system resources. If adequate resources are available, the QoS manager establishes a local reservation state for the session. It also tags the data path with a session handle to enable handling of data packets commensurate with their service requirements. Note that the data and control paths are completely separate. This separation of control and data paths enables us to provide sophisticated control functions without sacrificing data path performance. Besides acting as the resource manager and the admission controller, the QoS Manager is also responsible for forwarding any network and data-link-related control information, such as changes in reservation state, from the network interface to the application concerned. This is done via asynchronous messages posted to the application.

We have also augmented the data path from the socket to the network interfaces to enable session-specific handling of data packets. As part of reservation establishment, data sockets are associated with a session handle. When data are sent on a socket for which a reservation exists, a protocol-specific send function is called. One of the responsibilities of this function is to obtain kernel buffers for the data. For QoS connections, these buffers are allocated in advance, and are managed by the QoS Manager. If a buffer is available (it is always available if the application conforms to its advertised traffic envelope), data from the user space is copied into this buffer which is marked with the session handle. As the data packet traverses through the protocol stack, the session handle carried in the buffer header is used as the classifier for session-specific handling of the packet. A similar enhancement to the receive data path is also possible. However, since our focus in this paper is on servers, and data transmission from a server dominates over data reception, we concentrate on the transmit part of the data path.

The QoS Manager and the supporting modules have been implemented on the IBM AIX platform. We have extended the socket interface to provide an API to the QoS Manager. We have enhanced the memory allocator for session-specific management of system buffers. The mbuf structure itself has been modified to act as the conduit for session-specific information for efficient data handling. We have taken care to ensure that the mbuf modifications maintain object code level backward compatibility to accommodate third-party network interfaces. We have enhanced the architecture for the network interface layer and network device drivers for packet classification and session-specific packet handling. For example, we have modified the IFATM layer (network interface layer for classical IP over ATM) [27], [11], [6] to establish separate ATM virtual channels (VC's) with appropriate QoS parameters for each RSVP session. We have also enhanced legacy LAN (Token Ring) drivers to support a service-class-based queuing structure.

The rest of this paper is organized as follows. Section II gives an overview of the design requirements that have guided our work. The various building blocks that comprise our QoS support architecture are presented in Section III. Section IV describes the prototype implementation on AIX that we have developed based on this architecture. Section V is devoted to application level evaluation of the proposed architecture. The overheads of different QoS components are presented in Section VI. Section VII discusses related work, and contrasts it with the present work. Finally, Section VIII concludes the paper.

II. System Overview

In this section, we present an overview of the RSVP protocol, and a brief description of different classes of services that are being standardized by the IETF. These descriptions are not complete by any means. The purpose of this discussion is to make the paper self-contained and to set the stage for the following sections. For a complete description of the RSVP protocol, refer to [8]. Details on different service classes can be found in [28], [32].

A. RSVP

There are two basic types of RSVP messages—PATH and RESV. PATH messages are sent by the source, and each PATH message is associated with a specific data flow. PATH messages are encapsulated in IP or UDP datagrams. As PATH messages traverse the network towards the destination(s), they are intercepted by RSVP-enabled IP routers on the path. The routers set up soft states for the data flows with which the intercepted PATH messages are associated. A PATH state includes the previous and next hops of the flow and its traffic characteristics. When a PATH message reaches its intended receiver(s), it is processed by the RSVP daemon. If the receiver wants to make a reservation for a specific flow, it responds with a RESV message. The RESV message traverses the reverse path back to the sender. On the way to the sender, it is intercepted by RSVP-enabled routers. If sufficient resources are available, a reservation soft state is established in the routers. Otherwise, an RESV ERROR message is generated and sent back to the receiver. The RESV ERROR message is also intercepted by RSVP-enabled routers, and the reservation states are deleted. An end-to-end reservation is successfully established when the RESV message reaches the sender and is successfully processed by the RSVP daemon on the sender. Note that the PATH and RESV messages are independent of the data flow from the sender to the receiver, although they follow the same route through the network. Hence, a reservation can be established before, or at any time after, the data flow starts.

Reservations can also be made on a multicast session. In this case, the sender sends PATH messages to a multicast group address. As in the case of unicast, the PATH messages traverse the network to all members of the multicast group, and

PATH states are established at all RSVP-enabled routers in the multicast tree. When PATH messages reach the receivers, each receiver independently decides whether it wants to request a reservation for the session. Each receiver can potentially request different reservations for the same session. As the RESV messages from the receivers traverse upstream to the sender, they are merged by the routers at the merging points. Eventually, a reservation tree is established with the sender as the root and the receivers requesting reservations as the leaves.

In the network, as well as at the hosts, an RSVP flow is uniquely identified by the 5-tuple ⟨`protocol,src address, src port, dst address, dst port`⟩. Filters are set up at the routers and at the hosts to classify packets belonging to an RSVP flow, and to treat them in accordance with the reservation made for the flow. Note that RSVP is just a signaling protocol that helps establish the reservation soft states at the end hosts and at the routers. Honoring the reservations and providing services in accordance to them requires, among other things, resource and traffic management at the hosts and at the routers. The resource and traffic management mechanisms depend heavily upon the service classes supported. The service classes to be supported in the Internet fall under the jurisdiction of the Integrated Services Working Group in the IETF. In the following section, we present a brief description of two service classes that are under consideration for standardization.

B. Service Classes

The two important classes of service that are currently under standardization are: 1) guaranteed service and 2) controlled load service. Guaranteed service guarantees that datagrams arrive within the guaranteed delivery time and are not discarded due to queue overflows, provided the flow's traffic stays within its specified traffic parameters. This service is intended for applications which need firm guarantees on lossless on-time delivery of datagrams. Some of the interactive audio and video applications, and applications with hard real-time requirements fall in this category. The end-to-end behavior provided to an application by controlled load service closely approximates the behavior visible to applications receiving best effort service under *unloaded* network conditions. That is to say: 1) a very high percentage of transmitted packets are successfully delivered by the network to the receiving end nodes and 2) the transit delay experienced by a very high percentage of delivered packets do not greatly exceed the minimum transit delay experienced by any successfully delivered packet. Clearly, the definition of controlled load service is less precise than that of guaranteed service. It is intended for the broad class of applications which have been developed for use in today's Internet, but are sensitive to overload conditions. Some of the important members of this class are the adaptive real-time applications such as vic,vat, nevot, etc.

To avail these services, a session must specify a traffic envelope, called Tspec. Tspec is carried in the PATH message, and includes a long-term average rate, a short-term peak rate, and the maximum size of a burst of data generated by the application. For example, for an application generating MPEG coded video, the average rate could be the long-term data rate, peak rate could be the link bandwidth, and the burst size could be the maximum size of a frame. Tspec also specifies the maximum and minimum packet sizes to be used by the application. For guaranteed service, traffic should be shaped to conform to the traffic specification. For controlled load traffic, shaping at the source is not mandatory. Violating packets belonging to a controlled load session are allowed to pass the conformance check point as best effort traffic. In addition to Tspec, each guaranteed session also has an associated Rspec that specifies the required rate of service and a slack term. The required rate of service should be at least as large as the long-term average rate specified in the Tspec. The slack term signifies the difference between desired delay and the delay obtained with the specified rate of service. The slack term can be utilized by the network to reduce the reservation level of the flow. For controlled load service, there is no separate Rspec. Each controlled load session is guaranteed a rate of service equal to the long-term mean rate specified in its Tspec. A session may receive better service if there is spare capacity in the system.

The Integrated Services Working Group is only responsible for defining the service specification. Realization of an end-to-end QoS architecture to support these services requires enhancements and extensions to the network elements involved in data transport. These include end hosts at the edge of the network, and bridges and routers inside the network. In this paper, our focus is on end hosts. In the rest of the paper, we present the design and implementation a prototype system that realizes an RSVP-based QoS architecture and our initial experiences with it.

III. Architectural Building Blocks

In this section, we first outline the key objectives and features, and then give an overview of our QoS support architecture. We also illustrate how the architecture accommodates traditional best effort traffic and network interfaces with differing capabilities.

A. Design Considerations

Our design strives to achieve several key objectives: 1) adherence to Internet standards; 2) clear separation of control and data paths; 3) efficiency in data handling for QoS; 4) transparent integration of network interfaces of varying capabilities; and 5) backward compatibility with existing applications. In the following, we elaborate on each of these design goals.

Adherence to Internet Standards: Interoperability is the Internet religion, and our design strives to maintain compatibility with the current and upcoming standards for integrated services on the Internet. These include the RSVP signaling protocol and the service classes for controlled load and guaranteed service. We also support the RSVP service mappings defined for the ATM interface [19], and plan to support the service definitions for LAN interfaces as they become mature.

Clear Separation of Control and Data Paths: In the traditional socket API, control and data are handled on the same

socket. Control parameters are usually passed using `ioctl` or `setsockopt` function calls. However, these calls do not accommodate the asynchronous upward flow of information from the protocol module to the application. More recently, in 4.4 BSD, the message structure used in the `sendmsg` and `recvmsg` function calls has been modified to include control fields, and these calls can be used to pass control information between the application and the protocol module. However, this multiplexing of control and data makes it difficult to construct applications in which the control information is handled by a different thread. For example, in a Web server, the control part may be handled by the HTTP deamon, while a separate data exporter is responsible for data handling. This allows data transfer mechanisms to be fast and dumb, while the control mechanisms can be as sophisticated as necessary. We exploit this feature to implement a full-function RSVP-based control architecture without sacrificing data path throughput.

Efficiency in Data Handling for QoS: One of the skepticisms about QoS support in the Internet stems from the concern that it will affect data transfer performance. We have taken special care to ensure that this is not the case in our system. We have optimized the data path by shifting some operations to the control path. It will be seen that data path latencies of QoS connections in our system are the same as, and at times better than, those for best effort connections.

Transparent Accommodation of Network Interfaces With Varying Capabilities: With the rapid penetration of the Web, potentially any Internet site can be a server. Hence, not only must the design scale from small to large number of connections, it must accommodate differences in the capabilities of the attached network interfaces. These network interfaces typically range from high-function ATM interfaces (those with explicit support for QoS guarantees) to relatively low-function (in terms of support for QoS guarantees) interfaces such as Ethernet and modems. Our design aims to transparently accommodate the available functionality of the attached interface, for a variety of LAN interfaces, and manage resources such as link bandwidth and buffers accordingly.

Backward Compatibility with Existing Applications: The QoS extensions to the protocol stack should not require changes to the existing applications and services. Since existing services such as FTP may perform better in the presence of QoS support, the design should facilitate easy integration into the QoS architecture, without requiring source code, and if possible object code, modifications. At the same time, the API must provide suitable mechanisms that can be used by new applications that are written to truly avail of QoS support. The socket API is one of the most popular API's for communication on the Internet, and it is desirable to allow applications to request and receive QoS guarantees for data originating and terminating on socket endpoints. Our design, therefore, provides extensions to this API for specification and negotiation of QoS parameters and requirements.

B. Architectural Overview

Fig. 1 shows the software architecture of an RSVP-enabled host. In this example, a number of applications are using

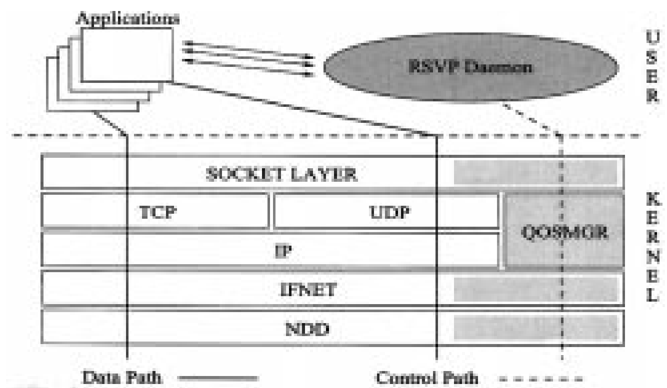


Fig. 1. Protocol stack architecture and extensions.

RSVP signaling for resource reservation. The applications use an RSVP API (RAPI) library to communicate with the user-level RSVP daemon running on the host. The RSVP daemon is responsible for translating the RAPI calls into RSVP signaling messages and local resource management function calls. For local resource management, the RSVP daemon interacts with the QoS Manager over an enhanced socket application programming interface.

The protocol stack in the kernel consists of a control plane and a data plane. The control plane is responsible for creating, managing, and removing reservations associated with different data flows. The data plane is involved in moving data from the application to the network and vice versa. The QoS Manager is the key component in the control plane of the protocol stack. It is entrusted with managing network-related resources, such as network interface buffers and link bandwidth. It is also responsible for maintaining reservation states of different flows and the association between the flows and their reservations. Moreover, it also performs data plane functions such as traffic policing and shaping unless the network interface adapters perform these functions in hardware.

The data plane of the protocol stack is an enhanced version of the classical Internet protocol suite. The extensions include changes to the socket layer, network interface drivers, and network device drivers for classification of network bound datagrams to the sessions to which they belong, and handling them in accordance with the reservation.

C. QoS Manager

As mentioned before, the heart of our resource management and control architecture is the QoS Manager. From the functional point of view, it is a control plane component primarily responsible for the creation, modification, and removal of reservations associated with different flows. As the resource manager of the system, it also gets involved in the data path of the QoS connections. The QoS Manager cooperates with other components in the data plane, such as the socket layer and network interface layer, to coordinate the management of all network-related resources. In the following, we present the architectural and functional details of the QoS Manager, and describe the enhancements and extensions required in other components to realize the overall QoS architecture.

The QoS Manager is responsible for: 1) creating and maintaining reservation states of QoS connections; 2) allocating and managing network buffers for these connections; and 3) policing and shaping of network bound traffic. It is implemented as a separate protocol module, and is accessed through the socket interface. Applications can access the services provided by the QoS Manager directly through the socket interface or via the RSVP daemon. For example, in order to set up a new QoS connection, an application can use the RAPI interface to the RSVP daemon, and communicate the endpoints of the connection and the traffic specification for the flow. The RSVP daemon uses the socket interface to communicate this information to the QoS Manager for creating the local reservation. Similarly, the QoS Manager is also invoked when the application decides to modify the reservation level or to remove the reservation altogether. The RSVP daemon and other applications can open multiple socket connections to the QoS Manager. We call these sockets `control sockets` since they are used to pass control information between the QoS Manager and its users. Multiple reservations can be created through a single control socket.

Maintaining Reservation States: The QoS Manager is responsible for creating and maintaining *local* reservation states for different flows. It is also responsible for creating an association between the data socket and the reservation state. When a request is received for a new reservation, the QoS Manager sets up a reservation state for the connection. It allocates buffers for the connection, performs admission control checks, and invokes the appropriate network interface level functions to set up link-level scheduling state for the flow. The reservation states are tied to the control socket through which they are created. A reservation state includes, among other things, a pointer to the associated data socket, a reservation handle, and a pointer to a preallocated buffer chain. The QoS Manager also tags the data socket for a reserved connection with the pointer to the associated reservation structure.

The QoS Manager performs various other control functions. For example, it is responsible for handling asynchronous notifications from the socket layer when data sockets are connected and disconnected. Since the control and data planes are completely separate, a reservation may be created before the data connection is actually established. Similarly, a data connection may be terminated without removing the associated reservation. The QoS Manager maintains the associations between the sockets and the reservations. This is achieved by registering handlers that are invoked by the socket layer in response to data socket connect and disconnect operations. The QoS Manager also keeps track of changes in connection state, and notifies the owner of the control socket by posting messages to it via the socket interface.

Buffer Management: Each guaranteed and controlled load session is allocated a session-specific buffer pool. When a new reservation is established, the QoS Manager uses the application Tspec to determine the number of buffers and the size of each buffer required for the new reservation. The Tspec includes the peak rate of traffic arrival r_p, the mean rate of traffic arrival r_m, and the maximum allowable burst size b. It also includes the maximum packet size $L_{\max}$ and minimum packet size $L_{\min}$. To provide services in accordance with the specification, the buffer pool should be large enough to hold at least a burst size worth of data including the packet headers. Header overhead depends on the particular interface (or the media access control layer) and the protocol (UDP/TCP) used. Let us assume that L_{header} is the size of the header. In general, it is possible to allocate buffers of any size that is a power of two. In order to keep the allocation simple, we populate the buffer pool with buffers of a single size. This design is based on the expectation that applications will perform path MTU (maximum transfer unit) discovery[1] and use the appropriate value for the maximum packet size so as to prevent fragmentation. Since we allocate buffers of only one size, for each packet we have to allocate buffers sufficient to hold the maximum size packets. Hence, the size of each buffer is $L_{\max} + L_{\text{header}}$ rounded to the next higher power of 2. The number of buffers allocated would be $b/L_{\min}$ rounded to the next integral value. The QoS Manager sets up the preallocated buffers as externally owned cluster `mbuf`'s.[2] It retains the responsibility for freeing them at an appropriate time.

This chain of preallocated buffers is tied to the reservation structure associated with the QoS connection. Another approach would be to associate these buffers directly with the data socket. However, the second approach does not facilitate sharing of reservations across data sockets. We set up the free routine of each `mbuf` cluster via the `mbuf` extension header[3] to allow reclaiming buffers from a reservation at the time the reservation is released. Due to traffic shaping and/or buffer shortages, an application thread may be blocked waiting for buffers to be freed. Hence, the free routine wakes up blocked threads, if any. The free routine also triggers the final close of a QoS connection if all of the buffers are back and the `delete connection` request is pending.

The QoS manager provides an interface for the socket layer to request buffers. When an application makes a transmission request on a reserved connection, the socket layer requests the QoS Manager to allocate buffers from the reserved pool. Based on the reservation handle presented by the socket layer, QoS Manager returns a buffer, if one is available, from the pool of preallocated buffers. If preallocated buffers are not available, the application thread is either blocked or returned a best effort buffer (if one is available), depending on the reservation type. For nonblocking sockets, the QoS Manager returns an error code under all circumstances where blocking would occur.

Traffic Policing: All network bound datagrams belonging to reserved connections are checked for Tspec compliance by the QoS Manager. For that purpose, it maintains two auxiliary variables in the reservation state of each connection. Let us call these variables t_m and t_p. Informally, t_m is used to check if the source is conforming to the long-term average rate specified in the Tspec. Similarly, t_p is used to enforce the short-term peak rate. Both t_m and t_p are set to $-\infty$ (a large negative number)

[1] It is highly recommended, if not required, to discover end-to-end MTU for QoS connections.

[2] Mbuf's are kernel managed buffers.

[3] We have added an extension header to the `mbuf` structure. This is explained in detail later in the paper.

at the time of reservation setup. When an application makes a transmission request call with a data buffer of size L, t_m and t_p are recomputed as follows:

$$t_m = \max(t - b/r_m, t_m) + \max(L, L_{\min})/r_m$$
$$t_p = \max(t - L_{\max}/r_p, t_p) + \max(L, L_{\min})/r_p.$$

In the expressions above, t is the arrival time of the packet, and $L_{\max}$ and $L_{\min}$ are the maximum and minimum packet sizes, respectively. Intuitively, t_m is the expected arrival time of the next packet assuming a rate of arrival of r_m and a burst size of b. Note that even if the application sends packets of size smaller than $L_{\min}$, it is charged for packets of size $L_{\min}$. This is in accordance with the IETF specifications, and is required for proper accounting of the fixed per-packet processing overhead at the host as well as in the network. Similarly, t_p is the expected arrival time of the next packet assuming a rate of arrival r_p and burst size of $L_{\max}$. It helps maintain minimum distance between consecutive packets in a flow. If current time t is less than the higher of t_m and t_p, then the application thread is in violation of its declared Tspec. Tspec violations are handled differently, depending on the type of reservation. In the case of guaranteed service, the thread is blocked until it is conformant (shaped). Nonconformant control load connections can send excess traffic as best effort data.

Traffic Shaping: If the application specifies that traffic on a connection be shaped, QoS Manager must *shape* (i.e., delay) noncompliant traffic until compliance. In general, our architecture provides two mechanisms for traffic shaping: *session-level shaping* and *packet-level shaping*. In session-level shaping (effectively performed at the session layer), QoS Manager blocks the corresponding application thread for $\delta = \max(t_m, t_p) - t$ time units. Note that this is transparent to the socket layer, which essentially sees a delay in allocation of a buffer to that packet. Session-level shaping is not required if the network interface has the capability to shape per-connection traffic, as in ATM networks, or such capability is provided in the network device driver (NDD), as is possible with other LAN technologies. The corresponding thread is still flow controlled by the QoS Manager as per the availability of buffers for that connection.

An alternative to session-level shaping is packet-level shaping, in which the NDD buffers noncompliant packets until compliance before transmitting them into the network. With packet-level shaping, QoS Manager only manages per-connection buffers and flow controls the send thread as per availability of buffers. While session-level shaping controls the execution of application threads, packet-level shaping controls the transmission of packets.

In our architecture, traffic policing and shaping are closely integrated within the buffer allocation service, especially when session-level shaping is used. The QoS Manager checks for both Tspec compliance as well as the availability of preallocated buffers. For guaranteed connections, the application thread is blocked if either of these checks fail. On success, reserved buffers are returned from the preallocated buffer pool. For control load connections, when the checks fail, the application thread is allocated best effort buffers, if available. Otherwise, the thread is blocked. On success, reserved buffers are allocated from the preallocated buffer pool.

There are several advantages to this approach. 1) Performing traffic policing and shaping at a layer just below the socket layer makes it more efficient to block and wake up the application threads. The application need not worry about conforming to the traffic specification; the kernel blocks the write thread automatically as per buffer availability and Tspec compliance. 2) Blocking the application write thread as soon as buffers become unavailable ensures that data that cannot be sent right away reside in the application's address space; this is consistent with the handling of buffer overflows in the socket layer in most Unix variant systems. 3) Since the socket layer only copies as much as can be sent on the associated reservation, shared resources such as kernel buffers are kept available for other flows.

D. Socket Layer

The socket programming interface has been enhanced in several dimensions. First, it has been extended to support a new protocol family. Sockets created with this protocol family are interfaced with the QoS Manager, and are referred to as control sockets. Control sockets are used to avail services offered by the QoS Manager, specifically to create, modify, and delete reservations made on data connections. An alternative way of extending the socket interface would have been to extend the `getsockopt` and `setsockopt` kernel services. We chose to use the control socket mechanism over that of socket options because of its flexibility and architectural richness. Unlike socket options, the control socket interface can be used for: 1) asynchronous upward control flow; 2) third-party control on data flows; and 3) sharing of reservation between multiple data sockets.

Second, since QoS connections require traffic policing/shaping and buffer management, the socket layer must use the QoS Manager memory allocator for all sockets associated with QoS. Not only must it be able to efficiently determine that a data socket is associated with a QoS session, it must also ensure that for all data transmissions on this socket, data are moved into pinned kernel buffers only when the buffer can be transmitted according to the associated QoS.

Third, we have extended the protocol switch table to include protocol-specific send routines. This delegates the responsibility of moving data from the application space to the protocol-specific send routine. For example, a TCP specific send routine would move in data only when it recognizes an opportunity for transmission. In contrast, in most socket layer implementations, data are moved into the kernel regardless of when they are sent out by the corresponding protocol. Buffer space (mbuf's) is also consumed on a first-come, first-serve basis, which suffices for best effort traffic, but is unsuitable for QoS connections. The use of QoS Manager for buffer allocation, in conjunction with the protocol-specific send routine, ensures that when the QoS Manager does return a buffer, there is a high probability that the packet is sent synchronously, thus allowing for reasonable traffic shaping. Since the QoS

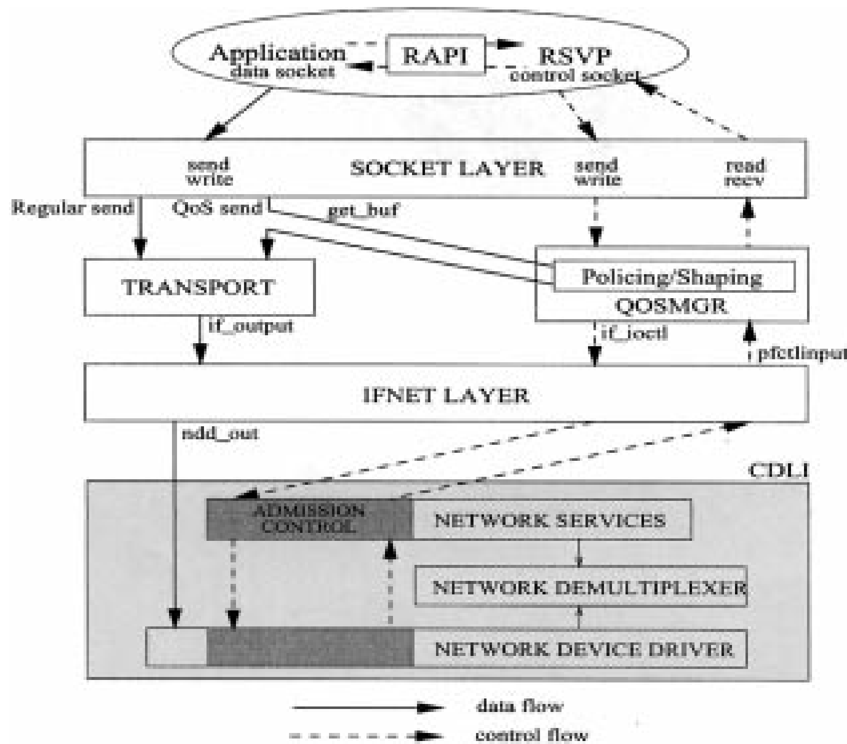


Fig. 2. Call structure at host.

Manager memory allocator is responsible for traffic shaping, no other changes need be made to the socket layer.

E. Network Interface Layer

The network interface layer is responsible for implementing link-layer adaptation functions for different subnetwork types such as Ethernet, Token Ring, ATM, etc. We have extended this layer to provide local reservation services for a subnetwork. The local reservation services are provided as control path functions to higher layers (QoS Manager) through the I/O control (`ioctl`) interface. Although this control interface is common across all interface types, the capabilities of specific interfaces may differ substantially based on the characteristics of the network and the level of sophistication of the network interface device. We have partitioned the space of network interface capabilities into three broad categories.

NO_QOS: The interface does not support any reservation capabilities. Unmodified legacy LAN interfaces fall in this category.

STD_QOS: The interface supports admission control and scheduling, but does not provide traffic shaping. This implies that the interface supports, at a minimum, a priority queue mechanism that is capable of differentiating between the different service categories as shown in Fig. 5. The interface may also provide a more sophisticated scheduling mechanism within each service category.

HIGH_QOS: In addition to admission control, the interface supports per-session traffic shaping. Typically, this traffic shaping would be accomplished with hardware support as in the case of the timing wheel shapers found in ATM network interface devices.

These levels of service are, in turn, derived from the capabilities of the network device. The QoS Manager is cognizant of these levels of service, and makes appropriate reservation and flow control decisions. The QoS Manager cannot make any reservations for sessions directed to an interface in the first category. For sessions directed to interfaces in the second category, the QoS Manager assumes the responsibility of policing and traffic shaping. As discussed earlier, the traffic shaping is optional, and applies to specific service categories. A HIGH_QOS device absolves the QoS Manager from the potentially expensive operation of traffic shaping. Moreover, by virtue of the buffer preallocation mechanism, the flow control mechanisms continue to operate, and an application would be blocked due to a lack of buffers when it tries to send data at a rate far exceeding its reserved rate.

Extensions are also required in the network device drivers. However, the device driver modifications are very specific to the particular network device concerned. In the next section, we describe the changes made to the ATM and Token Ring drivers.

IV. Prototype Implementation

The QoS architecture described in the last section has been implemented on RS/6000–based servers running AIX release 4.2 and equipped with ATM adapters and IEEE 802.5 Token Ring Adapters. Fig. 2 illustrates the structuring of the software layers, and how they interact in response to RSVP and application requests. We first present the implementation of the QoS Manager, and then discuss control and data flows in the case of the high-function ATM network interface (Section IV-B). This is followed by a description of the extensions required

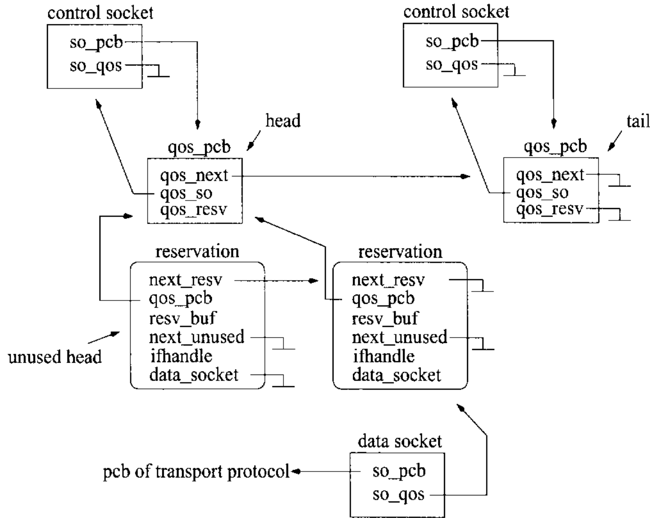


Fig. 3. QoS protocol control block and reservation structures.

to support the low-function Token Ring network interface (Section IV-C).

A. QoS Manager

The QoS Manager maintains two kinds of associations: one between control sockets (PF_QOS domain sockets) and the reservations created via them, and the other between data sockets and the reservations that apply to the corresponding socket endpoint(s). These two associations must be kept relatively independent of one another since an application could close data sockets and control sockets at different times. For example, closing a data socket should not affect the established reservations, which are released either in response to an explicit request by the application, or when the control socket is closed.

Fig. 3 illustrates the data structures used by the QoS Manager. When a control socket (`qos_so`) is created, it is allocated a QoS protocol control block (`qos_pcb`) which points back to the control socket. The `qos_pcb` thus allocated is inserted into a list (pointed to by `head` and `tail`) of protocol control blocks, similar to the mechanism employed for other protocol control blocks. When a new reservation is established, a reservation structure is allocated and inserted into a list of reservations pointed to by the `qos_pcb`. At this time, the entire list of Internet protocol control blocks is searched to find a data socket that should be associated with this reservation. If such a socket exists, the address of the reservation structure is stored in a new `so_qos` field of the socket structure, and the reservation is marked as used. This address is used during data transfer to efficiently identify the reservation structure to use for policing and shaping traffic on that data socket. Note that the `so_qos` field is always null for control sockets.

If a connected data socket matching the reservation cannot be found, the newly allocated reservation is marked unused, and is appended to the list of unused reservations pointed to by `unused head`. The QoS Manager receives socket connect/disconnect notifications from the socket layer via functions `qos_isconnected` and `qos_isdisconnected`

MBUF HDR	MBUF PKTHDR	MBUF EXTHDR	MBUF QOSHDR
m_next m_nextpkt m_data m_flags	pkthdr.len pkthdr.rcvif	ext_buf ext_size ext_free = qos_free() ext_arg = qoshandle	qosifhandle qos_priority

Fig. 4. Mbuf header extensions used by QoS Manager.

registered by the QoS Manager when it is initialized. In response to a data socket connect notification, the QoS Manager searches the list of unused reservations for a matching reservation, and if one is found, stores the address in the `so_qos` field as before.

Fig. 4 shows the new QoS extension header needed to associate QoS on a per-packet basis. The new *m_qoshdr* header is only valid if the M_PKTHDR, M_EXT, and M_QOS flags are set in the `m_flags` field of the `mbuf` header. Ideally, this information should be added to the `pkthdr` field because it must be in the first mbuf of a packet. For binary compatibility reasons, however, the new fields must be after the `exthhdr`. This preserves the location of the `pkthdr` and `exthdr` structures. The new `m_qoshdr` structure defines two new fields: a field for specifying the reservation handle associated with this packet, and a priority field that indicates to the network interface layer the level of service required for this packet. All appropriate `mbuf` services that copy `pkthdr` fields must now copy `qoshdr` fields as well.

Note that with this design, only the control path incurs the overhead of associating data sockets with reservations. Since the reservation handle is passed in the `mbuf` extension header, there is no additional packet classification [33], [31] overhead in the data path, thus keeping it efficient.

B. ATM Network Interface

The Turboways ATM adapter used in our system is an example of a high-function network interface that supports shaping and scheduling of traffic in hardware. Consequently, no device driver extensions are required to support QoS connections on this interface. However, we had to extend the network interface layer (IFATM) so that RSVP sessions are not multiplexed on the virtual connection (VC) established for default IP; rather, separate ATM connections are created for each RSVP session. The IFATM is responsible for translating the QoS specifications for controlled load and guaranteed service into appropriate parameters for the ATM VC's. This translation is conformant to the guidelines defined by the Integrated Services over Specific Lower Layers (ISSLL) Working Group of the IETF.

Creation of an RSVP session is initiated by the RSVP daemon on behalf of the application by making a reservation request to the QoS Manager. As mentioned earlier, the QoS Manager is responsible for managing interface buffers for reserved connections. If sufficient buffers are available, the reservation request is passed on to IFATM using the I/O control (`if_ioctl`) interface. We have extended the `if_ioctl` interface with a new command code (`IFIOCTL_QOS`) for reservation-related requests. The data argument passed with this command code contains, among other things, a control opcode identifying the specific action requested. Currently,

TABLE I
REQUEST AND RESPONSE OPCODES USED BY QoS MANAGER FOR COMMUNICATION WITH OTHER LAYERS

Opcode	Description
ADDRESV	Ask for a resource reservation setup
MODRESV	Change existing resource reservation
DELRESV	Close an existing reservation
STATRESV	Request to return status of session
RESVRESP	Positive/negative response for a reservation
DELRESP	Response to closing existing reservation
STATRESP	Response for the status of current session

four opcodes (listed in Table I) are supported. ADDRESV, MODRESV, and DELRESV are used to request creation of a new reservation, modification of reservation level of an existing reservation, and removal of a reservation, respectively. The STATRESV opcode is used to inquire the status of a reservation. Table I also lists the asynchronous responses to these requests.

If adequate resources are available, ADDRESV triggers a VC setup procedure for the RSVP session. This involves the ATM call manager. The QoS Manager is notified of the success or failure of session establishment (VC setup) asynchronously via the `pfctlinput` upcall. Note that we can utilize the upcall mechanism to deliver asynchronous notifications since the QoS Manager is implemented as a protocol module. Creation of separate VC's for different RSVP sessions requires modifications to the ARP (address resolution protocol) table structure maintained by IFATM. In order to accommodate more than one connection to an IP destination, ARP entries to a particular destination are linked together and hooked off the IP destination entry in the ARP hash table.

RSVP allows dynamic modification of reservation levels. The MODRESV opcode is used to request this service from IFATM. Since the current standards for ATM signaling (UNI 3.1) do not support renegotiation of resources for existing connections, MODRESV essentially requires setting up of a new connection, and then tearing down the old one. A more efficient solution is possible with UNI 4.0 which supports renegotiation of QoS on an existing connection. DELRESV for an existing RSVP session initiates tearing down of the VC serving the session. STATRESV is used to enquire the status of the VC dedicated to a reservation.

On the data path, when a packet comes to IFATM, it checks for the QoS Header (see Fig. 4) in the mbuf carrying the packet. If no QoS header is present, the ARP table is searched for the default IP VC handle to the packet's next hop destination. If the QoS header is present, IFATM extracts the `ifhandle` field from the QoS header. No ARP table search is required in this case since `ifhandle` is the VC handle for the RSVP session to which the packet belongs. To send a packet on a VC, IFATM calls `ndd_output` with VC handle as one of the parameters.

C. Token Ring Network Interface

To understand the complexity involved in supporting QoS in non-ATM LAN's, we have also developed an implementation

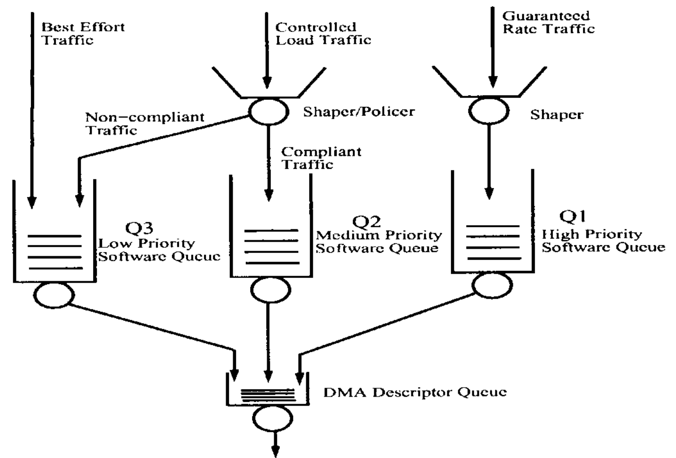


Fig. 5. Shaping and scheduling of traffic.

for the IEEE 802.5 Token Ring network adapter, which is an example of a `NO_QOS` interface. A similar approach can be used for Ethernet and other legacy LAN interfaces. The AIX Token Ring NDD does not provide any support that could be utilized by the QoS Manager to reserve link bandwidth for individual connections. Accordingly, we have enhanced it from a `NO_QOS` level to a `STD_QOS` level by providing the traffic queueing and scheduling structure in Fig. 5. These enhancements for reservation and scheduling functions are realized via extensions to the Network Services (NS) component of Common Data Link Interface (CDLI) [20] and the NDD (see Fig. 2). We do not address the problem of resource reservation on multiaccess LAN's. In that sense, our solution is more appropriate for switched Token Ring or Ethernet.

Buffer management and traffic shaping are now provided by the QoS Manager, as illustrated in Fig. 2. However, the local admission control procedure during reservation establishment is split between the QoS Manager and our NS extensions. While the QoS Manger performs the buffer requirement checks, NS (with possible support from NDD) performs the link bandwidth checks. The control and data flow is similar to that with the ATM network interface, except that the `if_ioctl` notifications are now relayed to NS. The request and response opcodes are the same as the ones listed in Table I.

As shown in Fig. 5, we enhanced the Token Ring NDD to support three packet queues, $Q1$, $Q2$, and $Q3$, where $Q1$ has nonpreemptive transmission priority over $Q2$ and $Q3$, and $Q2$ has nonpreemptive priority over $Q3$. Traffic from connections belonging to the guaranteed service class is queued in $Q1$. Conformant traffic from controlled-load connections is queued in $Q2$. Nonconformant traffic from controlled-load connections and best effort traffic is queued in $Q3$. The link scheduler services $Q1$ first; if there is a packet to transmit, it is dequeued from $Q1$ and prepared for transmission by insertion into the DMA descriptor queue on the adapter. If there is no traffic in $Q1$, $Q2$ is served. $Q3$ is served when both $Q1$ and $Q2$ are empty. Note that once the network adaptor is busy, the link scheduler is invoked in response to transmission completion interrupts. This is similar

to the invocation of packet transmissions in most BSD-derived Unix versions.

For the purpose of bandwidth management, NS keeps track of the maximum, current,[4] and minimum bandwidth reserved for each of the three queues. The maximum and minimum bandwidth levels are set at system configuration time, and reflect the desired bandwidth sharing between different traffic classes on an interface. For example, one could set maximum bandwidth for $Q1$ and $Q2$ to zero; the interface would the provide only best effort service. Alternately, one could set the minimum bandwidth for $Q3$ to, say, 2 Mbits/s to ensure that best effort traffic is guaranteed to have a bandwidth of at least 2 Mbits/s available at all times.

The data path for packets destined to the Token Ring interface is almost the same as it is for packets going out of an ATM interface, except that QoS Manager performs traffic shaping, if required. Another difference is that IFTR does not demultiplex packets based on the ifhandle carried in the QoS header. Demultiplexing of packets belonging to different sessions is done at the device driver using the priority field in the QoS header.

V. Application Level Performance

In this section, we demonstrate the efficacy of our QoS architecture in providing applications with the requested QoS. We have implemented the QoS architecture described in the previous section on RS/6000 servers running AIX release 4.2 and equipped with ATM and IEEE 802.5 Token Ring adapters. In the rest of the section, we focus on the ATM network interface, although much of the discussion is also applicable to the Token Ring interface. Since our ATM adapters provide QoS support in the form of VC setup/teardown, traffic policing, and scheduling, the QoS Manager has the option of not performing traffic policing and shaping for sessions using the ATM interface. The QoS Manager may, however, perform traffic policing and shaping to control the buffer usage and handle traffic that is in excess of the specified TSpec. For our experiments, we have extended `netperf` [26] to interact with the QoS Manager and create QoS sessions. We have instrumented `netperf` to permit the creation of sessions with different options for local traffic control and collect user-level statistics for the packet transmission time. The traffic control options provided can be used to selectively enable traffic policing and/or shaping for a reserved session. This allows us to incrementally assess the efficacy of each traffic control function performed by QoS Manager.

All of our experiments are performed between two machines—1) `tapti`—RS/6000 model 42T with 120 MHz Power PC 604 CPU, and 2) `tista`—an RS/6000 model 530 with a 33.4 MHz POWER CPU. Both machines run AIX version 4.2, and are equipped with 100 Mbits/s Turboways100 ATM adapters connected by an IBM 8260 ATM switch. The purpose of using the RS/6000 model 530 is to observe the impact of the new communication architecture on the vast existing base of older and slower servers. Although the POWER machine runs at a much lower clock speed, its overall performance is not proportionately worse than that of the PowerPC machine. The POWER and PowerPC CPU's are architecturally different, and the differences in their clock speed are not directly reflective of their relative processing power.

In the following, we examine the overall efficiency of the three different mechanisms used in the QoS Manager, namely, buffer preallocation, buffer preallocation and traffic policing, and buffer preallocation and traffic shaping. The first mechanism represents the case when only datalink-level shaping is used, whereas the third represents the case when session-level shaping is used. We first describe the operation of these three mechanisms over an ATM network, and contrast them with the default behavior (best effort without reservation and preallocation).

Default IP traffic is handled over the ATM network using the classical IP over ATM standards [21]. In order to send IP packets to a destination across the ATM network, a best effort VC is set up to that destination. It is possible to configure the maximum bandwidth for these connections, which sets the peak rate for the ATM VC. In all of our experiments, the best effort bandwidth was set to 10 Mbits/s.

When a reservation is established, a new virtual connection is created using the traffic parameters specified for the reservation following the mapping rules specified in [12], [19]. In our implementation, when neither traffic policing nor shaping is requested, the session data are directed to the VC that was created for the session (the session VC). On the other hand, when traffic policing is requested, the QoS Manager directs compliant packets to the session VC and noncompliant packets to the best effort VC. When traffic shaping is requested, since all packets are compliant after shaping, the session data are directed to the session VC.

While the architectural changes we made to integrate the QoS Manager are not coupled to any transport protocol, the data path mechanisms can have different impacts on the transport protocols. In the following, we first measure the performance of the various datapaths through the QoS Manager, and then observe the goodput obtained for UDP and TCP sessions.

Fig. 6 shows a plot of the measured data rates at the user level for different paths through the QoS Manager as a function of the reserved data rate. For this purpose, we ran an UDP stream through the QoS Manager, and measured the traffic load at the user level. Since UDP is a relatively lightweight protocol, traffic load at the user level provides a good measure of the efficiency of the QoS Manager vis-à-vis the best effort data path.

The top curve shows the measured performance for the case where a reservation is created, but traffic policing and shaping are disabled. Here, it can be seen that allocating buffers out of a preallocated pool results in lower latency than the best effort path (the second curve from the top). Note that this figure does not show the goodput achieved, and packets may be dropped at the source or in the network. This also explains the apparent anomaly in the top curve, where the measured rate decreases slightly with increases in the reserved rate. Since the ATM

[4] The current bandwidth reflects the present level of reservation at the interface.

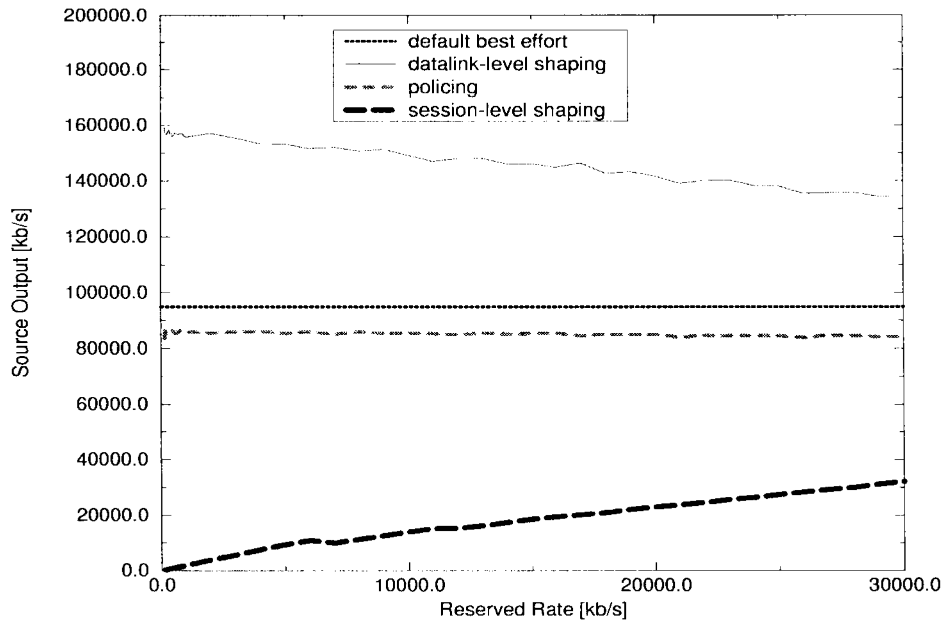


Fig. 6. Overall QoS manager performance.

interface card performs traffic shaping in accordance with the reserved rates for the VC, it can only send out packets at the reserved rate. If packets from the IP layer arrive faster than they can be transmitted, they create a buffer backlog and have to be discarded. The path length in the ATM driver for discarded packets is shorter since it does not have to perform the transmit processing operations. As the reserved rate increases, the number of packets that are successfully transmitted increases, which results in a slight increase in the execution time for the `send` operation.

From the figure, it can also be seen that when traffic policing is enabled, it increases the path length slightly, reflecting the cost of compliance checking. In this case, the gain due to buffer preallocation is offset by the overhead of compliance checking. More detailed measurements of these overheads are presented in Section VI. In the case when traffic shaping is enabled, the application thread is blocked from sending at a rate that is greater than the reserved rate. This is demonstrated clearly in the figure as the measured rate is very close to the reserved rate.

A. Goodput of UDP Streams

We now measure the goodput for UDP streams between two stations using different levels of reservation. The measurements were obtained using `netperf`, and show the average number of bytes received by the receiver. As mentioned earlier, the bandwidth allocated to the best effort VC was 10 Mbits/s. Fig. 7 shows the goodput for different reservation rates under various forms of source control using QoS Manager.

In the absence of any reservation, the goodput observed is approximately 9 Mbits/s. In this case, all packets are sent on the best effort VC which is policed at 10 Mbits/s, which limits the packet rate delivered to the receiver. The slight difference in observed goodput is due to packet overheads and the precision of the ATM card's peak rate policing. For the other three cases, an end-to-end reservation was established and buffer preallocation was performed. When policing is enabled, compliant packets are transmitted on the reserved VC created for the connection, whereas noncompliant packets are transmitted on the best effort VC. Clearly, for a null reservation, all of the packets are sent over the best effort VC. Increasing the reservation rate increases the goodput, but at a rate that is smaller than the sum of the best effort rate and the reserved rate. This is due to inefficiencies in the receive path, which result in a loss of packets at the receiver interface.

It can be seen that the goodput observed for session-level shaping, where the QoS Manager performs shaping, and datalink-level shaping are almost identical. In the previous experiment, we have seen that `netperf` can generate UDP packets at a rate of close to 150 Mbits/s. Since the actual throughput over the ATM network is limited to the reservation rate, the goodput obtained is very close to the reserved rate. In the case of session-level shaping, the application data rate is limited to the reserved rate by the QoS Manager. Shaping blocks noncompliant requests to send data packets until they are compliant, and therefore allows other processes to use the CPU more efficiently. In the case where the underlying link-level protocol does not perform any policing as is the case for many common link layer protocols that are in use today (Token Ring, Ethernet, etc.), session-level shaping is actually the only means of making end systems compliant with a negotiated traffic specification.

B. Goodput of TCP Connections

While UDP is not reliable and does not implement any flow control mechanism, TCP is more sophisticated. Not only does TCP segment the byte stream that it receives from an application into packets of a size that can be accepted by the network, it also implements flow control. It increases the flow when more bandwidth is available in the network, and it can significantly reduce the flow in case of congestion.

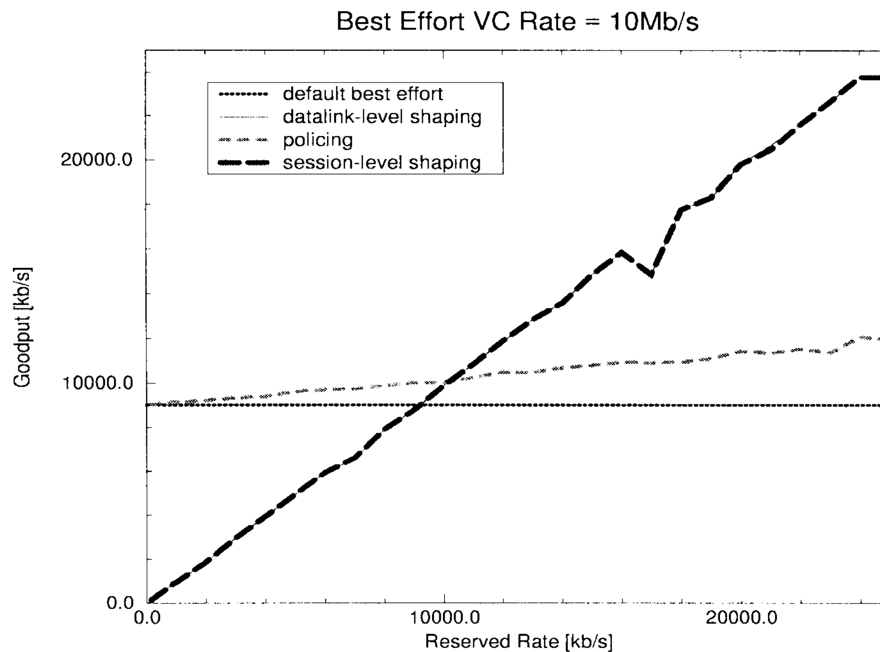


Fig. 7. Goodput for policed/shaped UDP.

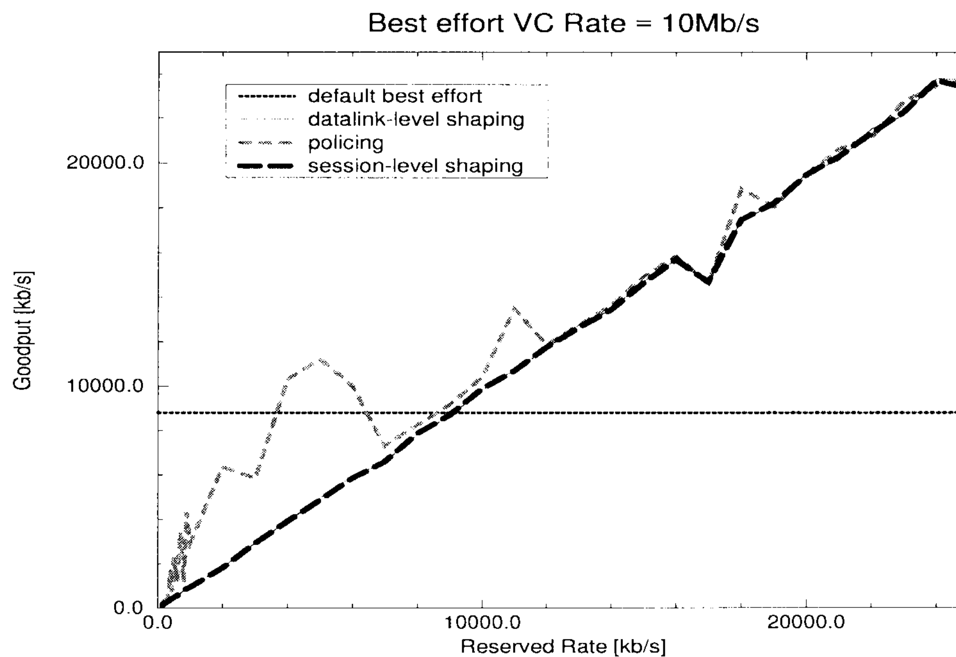
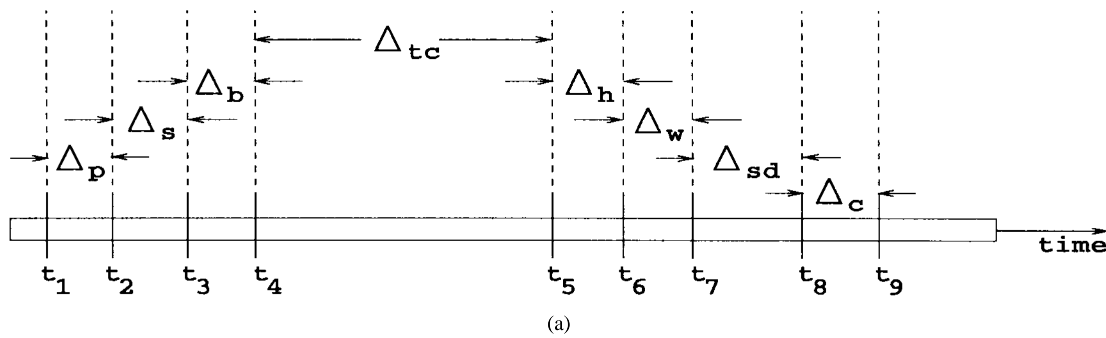


Fig. 8. Goodput for policed/shaped TCP connections.

Although our architecture is designed in a way that does not interfere with TCP's flow control (policing and shaping are done between the socket layer and the transport layer), we need to make sure that the goodput is not affected by the combination of policing/shaping and flow control.

Fig. 8 shows the goodput we observe for different reservation rates and various QoS Manager mechanisms over a TCP connection. Here, again, the allocated rate for the best effort VC is 10 Mbits/s. The horizontal curve shows the default case where no reservation is made (and therefore no buffers are preallocated) and traffic is sent over the best effort VC. It can be seen that the TCP flow control mechanism adjusts itself to the policed rate, without a significant drop in goodput. In fact, the observed throughput is almost identical to that of the UDP stream.

When a reserved session is created, and traffic policing is used, the data transmission of the TCP connection is split between the reserved VC and the best effort VC. In this case, we observe that the goodput is much less predictable than that of the UDP stream, where it increased proportionally with the increase of the reserved rate. Since TCP does not transmit individual, noncorrelated datagrams over a network, but instead guarantees a reliable delivery of a byte stream, splitting up the transmission on two VC's with different



(a)

Time	Event Description
t_1	packet arrival
t_2	time to compliance computed
t_3	shaping timer set
t_4	thread blocked
t_5	shaping timer expires
t_6	shaping timer handler invoked
t_7	blocked thread woken up
t_8	thread scheduled to run
t_9	shaping timer cancelled
$> t_9$	thread continues processing packet

(b)

Symbol	Overhead Component
Δ_p	policing overhead
Δ_s	overhead to start shaping timer
Δ_b	overhead to block thread
Δ_{tc}	time-to-compliance of this packet
Δ_h	overhead to handle shaping timer
Δ_w	overhead to wake up blocked thread
Δ_{sd}	delay in scheduling thread
Δ_c	overhead to stop shaping timer

(c)

Fig. 9. Timeline of events during shaping and associated overheads. (a) Timeline of events in QoS Manager data path. (b) Time instants and corresponding events. (c) Overheads associated with shaping.

policed peak rates causes the segments to arrive out of order at the receiving end. This means that at the receiving end, temporary "holes" appear in the byte stream when packets are sent more slowly on one of the VC's. Such a hole can trick the flow control mechanism into believing that there is congestion on the network, and that the transmission rate needs to be throttled down. Therefore, sending compliant packets over a reserved SVC and using any free bandwidth available in the best effort VC does not work well with TCP.

It can be seen that the curves representing goodput for session-level shaping and datalink-level shaping are almost identical, and the goodput corresponds very precisely to the reservation rate. When datalink-level shaping is used, the TCP flow control adapts itself to the reserved rate. Hence, in this simple scenario, it is possible to disable session-level shaping without any loss in performance. In more complex network scenarios, where the TCP connection crosses several links and the round-trip delay is significant, traffic shaping at the source offers some advantages by presenting a well-spaced out-flow to the network [17].

VI. QoS Component Overheads

The results of the previous section demonstrate the efficacy of the QoS architecture in providing QoS to applications. In this section, we identify the overheads associated with the QoS components, and experimentally quantify them using our prototype implementation.

A. Data Path Overhead

Fig. 9(a) illustrates the timeline of events as a packet passes through the QoS Manager. An outgoing packet "arrives" at the QoS Manager at time t_1. The packet's time-to-compliance ($\Delta_{\rm tc}$) is computed as part of the policing function, and by time t_2, it is known whether the packet is compliant or not; the overhead incurred due to the policing function is Δ_p. For a noncompliant packet (and hence one that needs to be shaped), a shaping timer is started at time t_2, incurring an overhead Δ_s. Subsequently, the calling thread is blocked on a semaphore, incurring an overhead Δ_b; at time t_4, the CPU can be reallocated to another thread. At time t_5, after a delay of $\Delta_{\rm tc}$ from time instant t_3, the shaping timer expires, and an overhead of Δ_h is incurred by the kernel in searching for the timer block and invoking the corresponding handler. The handler simply delivers a wake-up to the blocked thread and exits. Waking up a thread incurs an overhead of Δ_w, and at time t_7, the thread is residing in the CPU run queue, i.e., has been made runnable. The operating system's CPU scheduler allocates the CPU to this thread (i.e., schedules the thread for execution) at time t_8 after a scheduling delay of $\Delta_{\rm sd}$. The thread first stops the shaping timer, which incurs an overhead Δ_c, and continues processing the now-compliant packet from time t_9 onward.

We have instrumented the network subsystem of the modified AIX kernel for detailed profiling of the data path of reserved sessions using the native tracing facility. The trace facility captures a sequential flow of timestamped system events, providing a fine level of detailed system activity. A trace event can take several forms, and consists of a hookword, optional data words, and an optional timestamp. The hookword is used to identify the specific event being traced. To minimize tracing overhead, our event records use only the hookword and timestamp. Timestamps are taken by reading a real-time clock. The real-time clock is an integral part of POWER and PowerPC CPU's used in RS/6000's, and can be read directly

from the kernel as well as from the applications. We use a two-instruction assembly language routine to read two 32-bit clock registers with minimal overhead. The timestamps are of microsecond granularity.

Referring to Fig. 9, we measure Δ_p, Δ_s, Δ_w, and Δ_c by taking timestamps before and after the corresponding kernel calls, computing the difference, and averaging over a large number of invocations. For this purpose, we utilized the functionality of the QoS Manager (which is realized as a protocol module, as mentioned earlier) to send it appropriate user-level commands and trigger our profiling code in the kernel. Computation of Δ_h is slightly more involved. Suppose a timer is set for δ time units, where a time unit corresponds to the duration of the system timer tick. If t_b is the timestamp taken before starting the timer, and t_a is the timestamp taken immediately upon entry into the timer handler, then $t_a - t_b = \delta + \Delta_h$ holds, and Δ_h can now be computed.

Table II lists the measured values of these component overheads on `tapti` and `tista`. The policing overhead (Δ_p) includes the cost of retrieving the appropriate session state variables, computing the timestamps, and checking whether a packet is compliant by comparing its expected arrival time with the current system time. The current implementation uses two 4-byte words to represent each timestamp, one for the seconds field and the other for the nanoseconds field. The policing computation can be further optimized by using only the nanoseconds field for comparing and manipulating timestamps, and handling the (rare) rollover as a special case.

The overheads of shaping noncompliant packets can be broken up into timer operations (Δ_s: set timer, Δ_h: handle timer, Δ_c: cancel timer) and thread operations (Δ_b: block thread, Δ_w: wake-up thread). For a modern machine such as `tapti`, timer operations are reasonably small relative to the base best effort (UDP) path latency. The timer overheads correspond to the case with only one shaping timer outstanding, with perhaps a few system (kernel) timers active concurrently. For good scalability with the number of active connections, the OS timer support must scale well with the number of active timers. It is observed that the overhead of blocking threads is significant, notwithstanding faster processors, with context switching being the dominant component.

Allocation of buffers (for noncompliant packets) from the shared `mbuf` pool (overhead Δ_a^b) is significantly more expensive than allocation of buffers (for compliant packets) from the preallocated session-specific reserved pool (overhead Δ_a^r). Thus, the savings in buffer allocation serves to offset some of the overheads of traffic policing and shaping. Likewise, the cost of freeing a best effort buffer is significantly higher than the cost of freeing (i.e., returning to the session-specific buffer pool) a preallocated buffer. In effect, our architecture keeps the data path efficient while increasing the control path latency slightly.

Both policing and shaping insert additional operations in the data paths for reserved connections relative to the best effort data path. One would therefore expect to see higher data path latencies for reserved paths over that of the best effort data path. However, the best effort path differs from the reserved data path in that it

TABLE II
OVERHEADS OF DIFFERENT QoS COMPONENTS

Component Overheads			tapti	tista
Policing	Δ_p	compute compliance time	16.0μs	23.0μs
Buffer management	Δ_a^r	allocate reserved buffer	6μs	18.5μs
	Δ_f^r	free reserved buffer	15μs	33μs
	Δ_a^b	allocate best-effort buffer	18μs	35μs
Timer operations	Δ_s	set timer	7.4μs	14.0μs
	Δ_h	handle timer	7.1μs	30.1μs
	Δ_c	cancel timer	6.5μs	9.6μs
Thread operations	Δ_b	block thread	37.4μs	78.8μs
	Δ_w	wakeup thread	14.4μs	23.8μs
ARP search	Δ_{arp}	search for ARP entry	10μs	17μs

TABLE III
DATA PATH LATENCIES ON TAPTI (IN μS)

Connection Type	Message Size (in bytes)						
	64	128	256	512	1024	2048	4096
Best Effort	133	137	163	189	215	223	288
Reserved	159	162	172	186	217	231	298

- uses the standard mbuf allocator for packets, as opposed to using a preallocated private pool of buffers, and
- incurs a search for the ARP (address resolution protocol) entry in the ARP cache (Δ_{arp}); in the reserved path, the session handle maps directly to the appropriate virtual connection identifier, thereby eliminating this search.

As revealed by the measurements listed in Table II, these differences taken together constitute a significant reduction in the data path latency for the reserved path, and partially offset the overheads due to policing and shaping.

For high-function network interfaces (NIC's) that perform traffic shaping, such as our ATM adapter, the only overheads incurred are those of traffic policing and buffer management, which are necessary to support the different service classes such as controlled load. If $\mathcal{C}_{\text{base}}$ represents the base latency of the best effort path, the latency seen by a compliant packet is $\approx \mathcal{C}_{\text{base}} + \Delta_p - (\Delta_a^b - \Delta_a^r) - \Delta_{\text{arp}}$, and that seen by a noncompliant packet is $\mathcal{C}_{\text{base}} + \Delta_p$. For `tapti`, $\Delta_{\text{arp}} \approx 10$ μs (Table II) for an entry that is resident in the cache. Thus, the effective increase in data path latency for a compliant packet is ≈ -6 μs, i.e., the data path is slightly *faster*. Similarly, the reduction in data path latency on `tista` is $\approx$10 μs.

Table III shows application-level latency measurements for UDP traffic on best effort and reserved data paths for a range of message sizes. These results indicate that for 512-byte packets, compliant packets do experience a slight ($\approx$1.5% decrease in latency. The same experiments also reveal that for other packet sizes, compliant packets see an increase in latency. While additional experiments are needed to investigate this further, we suspect that this is primarily due to cache- and OS-related effects. A noncompliant packet experiences a latency increase of $\approx$12.5% since it incurs policing overhead and uses best effort buffers.

Session-level traffic shaping incurs significant additional overheads in the form of timer and thread operations. Further,

TABLE IV
CONTROL PATH LATENCIES

Machine	ADDRESV	DELRESV
tapti	16.50ms	0.27ms
tista	27.00ms	3.40ms

session-level shaping may also incur a variable (CPU load-dependent) delay in scheduling a blocked thread on the CPU ($\Delta_{\rm sd}$).

B. Control Path Overhead

Table IV shows the application-level latencies in creating (ADDRESV) and removing (DELRESV) a reservation between `tapti` and `tista`. The latency involved in creating a reservation includes the time taken to create local reservation states, as well as perform signaling across the ATM network to set up a VC for the RSVP flow. The removal of a reservation includes cleaning up the local reservation state and tearing down the VC associated with the RSVP flow. However, the latency seen by an application in DELRESV does not include the time taken to tear down the VC across the ATM network. This explains the large difference between the latencies in ADDRESV and DELRESV. Note that the most significant part of ADDRESV latency is contributed by ATM signaling overhead.

VII. RELATED WORK

We have focused on QoS support in TCP/IP protocol stacks for Internet hosts that produce QoS-sensitive network data for output (such as servers or any content provider on the Web). Our work builds upon and relates to several recent research efforts, as discussed below.

Integrated Services on the Internet: The service model for integrated services on the Internet and expected requirements of applications are discussed in [10], [7], [18], and the service classes under consideration by the IETF outlined in [28], [32]. Various issues in supporting QoS in IP–ATM networks are also being examined [11], [27], [6], including the use of RSVP for end-to-end signaling across ATM networks [4] and for integrated services using RSVP over ATM [12].

Network and Protocol Support for QoS: The Tenet protocol suite [3] provides real-time communication support in wide-area networks (WAN's). While they also develop architectural enhancements for a sockets-based communication subsystem, the protocol suite adopted does not conform to IETF standards for integrated services. Moreover, they do not provide support for network interfaces with widely differing capabilities. The native-mode ATM transport layer described in [1] also performs traffic policing and shaping while copying application data into kernel buffers. However, our design is applicable to general TCP/IP protocol stacks, including legacy LAN and ATM interfaces, participating in an integrated services Internet. Furthermore, the decision to shape traffic on a particular reservation is based on the service class of that reservation.

Protocol Processing and Data Transfer Optimizations: Several recent efforts have focused on optimizing the performance of the data transfer path in TCP/IP protocol stacks, including protocol processing latency [25], [5], [30] and user-level handling of network data [22], [29], [14], [15], [9] to increase throughput via data copy minimization. All of these efforts are geared toward traditional best effort traffic, and as such, are complementary to our work, which focuses on the performance impact of supporting QoS in TCP/IP protocol stacks. More specifically, we have examined the overheads imposed by *new* data-handling components in the protocol stack.

Efficient Packet Filters: Recall that we use a statically compiled packet filter for traffic classification during reservation setup signaling. While this provides us with very efficient classification for outgoing traffic at the sending hosts, more general mechanisms are needed to classify packets at intermediate routers and receiving hosts. A general classifier for real-time packet forwarding is described in [31]. Packet filters [23], [33], [2] provide general and flexible classification of incoming packets to application endpoints. More recently, dynamic code generation techniques have been applied to realize very efficient packet filters [16].

Packet Handling on Reception: While we have focused on the transmission path, for end-to-end QoS, packets arriving at a destination host must be classified as above and processed as per their associated QoS. The processing of an arrived packet can be delayed until the application receives the data, as in LRP [13]. While this works well for best effort traffic, appropriate OS support is still needed to ensure that the application is scheduled to run in a QoS-sensitive fashion. An alternative approach, which also utilizes early demultiplexing of arriving packets, but performs packet reassembly in a QoS-sensitive fashion via EDF scheduling of protocol processing, is described in [24].

VIII. SUMMARY

We have described the design and architecture of a framework for communication resource management for providing QoS support on Unix-like Internet servers, these being the typical source of multimedia data on the Internet. The heart of our architecture, which embraces emerging Internet standards for end-to-end resource reservations, is a new kernel module called QoS Manager. This module controls several important network-related resources, namely, bandwidth and transmission priorities on network interfaces and kernel buffer space (mbuf's). We have also augmented the sockets layer to enable session-specific handling of data packets. The QoS Manager and the sockets layer together provide a novel combination of buffer management and traffic shaping to provide a synchronous feedback mechanism for applications. These extensions preserve binary and API compatibility for sockets applications, while providing significant new functionality.

We have developed a prototype implementation of this architecture for the IBM AIX platform. We have implemented the QoS Manager and network interface support for ATM and

Token Ring networks. This implementation is one of the first implementations of the RSVP protocol over an ATM network. When operating on an ATM network, our implementation provides QoS guarantees for TCP/UDP/IP applications with only a small increase in latency, thereby achieving our goal of efficiency.

We have also measured the performance impacts of the QoS architecture and its various components using detailed kernel profiling. Our main conclusions can be summarized as follows. Of the different QoS overheads considered, traffic shaping presents the most challenges due to its interaction with the OS CPU scheduler. Traffic policing and shaping overheads are partially offset by savings due to preallocation of per-session buffers. Further savings can be obtained for ATM networks due to a faster path through the network interface layer. While policing overheads can be further optimized in a straightforward manner, reducing traffic shaping overheads would require improvements in OS timer and thread operations. Potential load-dependent variations in the actual shaping latency can be accommodated via appropriate adaptation and buffer management, or largely eliminated via integration with QoS-sensitive CPU scheduling policies.

Acknowledgment

The authors gratefully acknowledge the contributions of J. Chu, I. Chang, and R. Engel at the IBM T. J. Watson Research Center, and S. Wise, S. Sharma, D. Badt, and W. Hymas at IBM Austin.

References

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

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PATHFINDER: A pattern-based packet classifier," in *Proc. ACM SIGCOMM*, London, U.K., Aug. 1994, pp. 115–123.

[3] 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, pp. 1–11, Feb. 1996.

[4] A. Birman, V. Firiou, R. Guerin, and D. Kandlur, "Provisioning of RSVP-based services over a large ATM network," *IBM Res. Rep.* RC 20250, Oct. 1995.

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

[6] M. Borden, E. Crawley, B. Davie, and S. Batsell, "Integration of real-time services in an IP-ATM network architecture," *Request for Comments RFC 1821*, Aug. 1995.

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

[8] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource reservation protocol (RSVP)—Version 1, functional specification," *Request for Comments RFC 2205*, Sept. 1997.

[9] V. Buch, T. von elicken, A. Basu, and W. Vogels, "U-net: A user-level network interface for parallel and distributed computing," in *Proc. ACM Symp. Operating Syst. Principles*, Dec. 1995, pp. 40–53.

[10] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. ACM SIGCOMM*, Aug. 1992, pp. 14–26.

[11] R. Cole, D. Shur, and C. Villamizar, "IP over ATM: A framework document," *Request for Comments RFC 1932*, Apr. 1996.

[12] E. Crawley *et al.*, "A framework for integrated services and RSVP over ATM," work in progress, July 1997.

[13] P. Druschel and G. Banga, "Lazy Receiver Processing (LRP): A network subsystem architecture for server systems," in *Proc. 2nd OSDI Symp.*, Oct. 1996, pp. 261–275.

[14] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. ACM SIGCOMM*, London, U.K., Aug. 1994, pp. 2–13.

[15] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Clamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," in *Proc. ACM SIGCOMM*, London, U.K., Aug. 1994, pp. 14–24.

[16] D. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. ACM SIGCOMM*, Oct. 1996, pp. 53–59.

[17] W.-C. Feng, D. Kandlur, D. Saha, and K. Shin, "Understanding TCP dynamics in an integrated services Internet," in *Proc. 7th Int. NOSSDAV Workshop*, May 1997.

[18] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, Aug. 1995.

[19] M. W. Garrett and M. Borden, "Interoperation of controlled-load and guaranteed services with ATM," work in progress, July 1997.

[20] G. Joyce, "OSF NETSIG: Common data link interface," design specification, Apr. 1993.

[21] M. Laubach, "Classical IP and ARP over ATM," *Requests for Comments RFC 1577*, Jan. 1994.

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

[23] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. Winter USENIX*, 1993.

[24] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring communication software for quality-of-service guarantees," in *Proc. 17th Real-Time Syst. Symp.*, Dec. 1996, pp. 144–154.

[25] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley, "Analysis of techniques to improve protocol processing latency," in *Proc. ACM SIGCOMM*, Oct. 1996, pp. 73–84.

[26] Netperf homepage, http://www.cub.hp.com/netperf/Netperfpage.html.

[27] M. Perez, F. Liaw, A. Mankin, E. Hoffman, D. Grossman, and A. Malis, "ATM signaling support for IP over ATM," *Request for Comments RFC 1755*, Feb. 1995.

[28] S. Shenker, C. Partridge, and R. Guerin, "Specification of guaranteed quality of service," *Request for Comments RFC 2212*, Sept. 1997.

[29] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. Lazowska, "Implementing network protocols at user level," in *Proc. ACM SIGCOMM*, San Francisco, CA, Oct. 1993, pp. 64–73.

[30] R. van Renesse, "Masking the overhead of protocol layering," in *Proc. ACM SIGCOMM*, Oct. 1996, pp. 96–104.

[31] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Floyd, "Implementing real-time packet forwarding policies using streams," in *Proc. USENIX Winter 1995 Tech. Conf.*, Jan. 1995, pp. 71–82.

[32] J. Wroclawski, "Specification of controlled-load network element service," *Request for Comments RFC 2211*, Sept. 1997.

[33] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large messages," in *Proc. Winter USENIX*, 1994.

[34] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource reservation protocol," *IEEE Network*, pp. 8–18, Sept. 1993.

Tsipora P. Barzilai received the M.Sc. degree in information system engineering from Polytechnic University, Brooklyn, NY.

She is a member of the Network Control and Architecture Group at the IBM T. J. Watson Research Center, Yorktown Heights, NY. She has been with IBM since 1982, and was involved in developing a tool for performance evaluation of computer networks. In 1984, she joined the team responsible for the architecture and implementation of APPN (advanced peer-to-peer networking). Since 1987, she has been involved in analyzing and developing communication protocols for high-speed networks. She is currently pursuing research and development of the next-generation Internet. Her primary research interest is in distributed algorithms for communication protocols and communication support for multimedia applications.

Dilip D. Kandlur (S'90–M'91) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1985, and the M.S.E. and Ph.D. degrees, also in computer science and engineering, from the University of Michigan, Ann Arbor, in 1987 and 1991, respectively.

Since joining the IBM T. J. Watson Research Center, Yorktown Heights, NY, he has worked on the design and implementation of the Multimedia Multiparty Teleconferencing (MMT) System, a high-quality desktop videoconferencing system which was deployed as part of the CNRI Aurora Gigabit Testbed. His research has focused on several aspects of multimedia systems such as video/audio support for desktop collaboration, multimedia networking, and multimedia storage management. He currently leads the Network Control and Architecture Group whose focus is on providing networking support for multimedia applications.

Dr. Kandlur is a member of the IEEE Computer Society, and he currently cochairs the Multimedia Computing and Networking Conference (MMCN98) at the SPIE Symposium on Electronic Imaging.

Ashish Mehra received the B.Tech. (Bachelor of Technology) degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1989, and the M.S.E. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1992 and 1997, respectively.

He is currently a Research Staff Member in the Network Control and Architecture Group at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His primary research interests are in operating system and networking support for application quality of service requirements, Internet-based network computing, code mobility and security, high-speed networking, and performance evaluation.

Dr. Mehra is a member of the IEEE Computer Society, ACM, and USENIX.

Debanjan Saha received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1990, and the M.S. and Ph.D. degrees in computer science in 1992 and 1995, respectively, from the University of Maryland, College Park.

He is currently with the Network Control and Architecture Group at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His current research interests include high-speed and high-function networks and network security.