Signaling and Operating System Support for Native-Mode ATM Applications

*R. Sharma (*rosens@cs.stanford.edu*)*

*S. Keshav (*keshav@research.att.com*)*

AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA

Abstract

Applications communicating over connectionless networks, such as IP, cannot obtain per-connection Quality of Service (QoS) guarantees. In contrast, the connection-oriented nature of the ATM layer and its per-virtual-circuit QoS guarantees are visible to a *native-mode ATM* application. We describe the design and implementation of operating system and signaling support for native-mode applications, independent of the semantics of the protocol layers or of the signaling protocol. The work was done in the context of a Unix-like operating system and the Xunet 2 wide-area high-speed ATM network. The IPC-based interface between an application and the signaling entity allows processes to request parameterized virtual circuits, and the signaling-kernel interface allows resources to be reclaimed from prematurely terminating processes. We also built a simple encapsulation layer over raw IP that allows any host with IP access to send AAL frames into the wide-area network with little performance degradation. Our design makes it simple to port existing TCP/IP socket applications to a native-mode ATM protocol stack and also enables interoperation of existing IP networks with our ATM network. Our experience has been positive - the design is robust, easily extendible and scales well with the number of open connections.

1. Introduction

Much current research in ATM networks has been at the datalink and physical layers of the protocol stack; the higher layers are typically assumed to implement the IP and TCP protocols. Since IP is connectionless, and it logically multiplexes TCP connections, it is impossible to specify per-connection Quality of Service (QoS) requirements, even though these can be provided by the ATM network. We would like to present the connection-oriented nature and the QoS guarantees provided by ATM to applications. We call a protocol stack that provides these features a *native-mode ATM stack* (Figure 1).

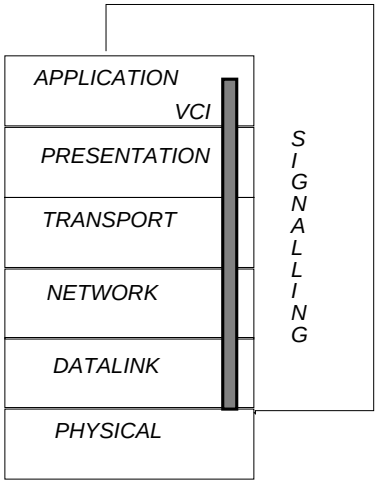

Figure 1: *Overview of the native mode stack.*

We view the design of a native-mode stack as having three independent components: semantics, implementation and support. The semantics of the stack specifies the abstract functionality provided by each layer, including signaling. An implementation is an instantiation of these semantics. Finally, the stack needs some support from the operating system (such as for actually moving data, notification of process termination etc.). In this paper, we present a scheme for the support of a native mode stack independent of its semantics or the details of its implementation. Thus, for example, we describe how an application may specify a QoS to the signaling entity, but we do not discuss the form that this specification takes, or how this specification maps to specific OS and network scheduling policies.

The semantics of the native mode stack are described in [12]. The key ideas here are that the stack has no logical multiplexing, and there is no duplication of AAL functionality in the

higher layers. A non-multiplexing protocol stack has several advantages over a conventional multiplexing stack [9]. First, per-application QoS requirements can be communicated both to the network and the local operating system, allowing intelligent scheduling of critical resources. Second, there are fewer multiplexing/demultiplexing overheads (since each demultiplexing point must compute a hash function). Third, the Virtual Circuit Identifier (VCI) provides a single index into a table of protocol control blocks, considerably simplifying the software structure.

We did our work in the context of the Silicon Graphics Inc. IRIX 4.0.1 kernel and the Xunet II wide area ATM network [10]. IRIX is a System V variant of Unix, with provision for BSD sockets [13]. XUNET II is an experimental wide-area ATM network that serves as a testbed for research on data networking. The network has connections to AT&T Bell Laboratories in Murray Hill, New Jersey, and to four universities: the University of California at Berkeley, the University of Illinois at Urbana-Champaign, the University of Wisconsin at Madison, and Rutgers University in New Brunswick, New Jersey. Long distance transmission is based on ATM using DS3 facilities (at 45Mbps) as well as optically amplified lines operating at 622 Mbps. Xunet II supports IP-over-ATM and quite a bit of the traffic over Xunet II is generated from IP-multicast based multimedia applications.

2. Scope of Our Work

IRIX provides a standard method for implementing a protocol stack inside the kernel using a BSD-style *protocol family* abstraction [13]. A protocol stack implementing this abstraction can access the BSD socket and device driver interfaces. When we began our work, the Xunet IRIX kernel already implemented a protocol family called PF_XUNET that provided simplex virtual circuits. A socket bound to this family allowed users to access the proprietary *Hobbit* ATM host-interface board [2] via the *Orc* device driver. There was also a daemon that implemented a proprietary signaling protocol, thus allowing an ATM endpoint to set up a simplex switched virtual circuit to any ATM destination on Xunet. However, there were two limitations. First, in order to set up a switched virtual circuit, an application had to be linked and loaded with the signaling entity. This was not acceptable for reasons of robustness and security. Second, switched virtual circuits were available only between ATM endpoints, that is, workstations with the Hobbit ATM host-interface card. Since Xunet only has eight of these, this limited the scope of the network.

In our work, we extended the existing infrastructure in three ways. First, we allow any application to make RPC-like calls on the signaling software in order to set up and tear down simplex calls. Thus, the application does not have to be compiled in with the signaling entity. Second, we allow any endpoint with IP connectivity to get access to the ATM network by encapsulating AAL frames in IP packets. This lets us to move quickly and easily towards the goal of ‘ATM Everywhere’ [14]. Third, we extended the signaling mechanism to carry a QoS string from a client to a server, and the negotiated (possibly modified) QoS reply back to the client. QoS negotiation is proxied to the IP-connected endpoints. In the rest of the paper, we describe how these changes were made.

A note on terminology: We call machines with an ATM interface *routers*, since they perform the IP routing function when providing IP service on Xunet. Machines that do not have an ATM interface are called *hosts*. If a call originates from machine A, via routers B and C to machine D, we call A the *host*, B the *router*, C the *remote router*, and D the *remote host*.

3. Services

The first step in the design process was to determine what functionality should be provided to applications by our extensions. We classified applications into servers and clients. Servers should be able to register themselves with their local signaling entity. For example, a file server might advertise the name "file-service" with the signaling entity on host with ATM address "mh.rt".

A client application that wanted to access a file on this server would request the local signaling entity to initiate a connection to <"mh.rt", "file-service", QoS>. Here, *QoS* is a structure that represents the quality of service that the client wants from the network, such as <service class, bandwidth> as described in Reference [17]. In this paper, we will treat this structure as an uninterpreted string. Note that the client-to-server connection is simplex, so in our example, the server application would have to establish a return connection to actually return a file to the client. In addition to registering service names, and initiating connections to services, we wanted to allow clients and servers to cancel any outstanding requests.

The second set of functions extends native mode support to hosts without ATM host interfaces. We wanted to allow any host with IP connectivity to a router to be able to make requests to an Xunet signaling entity. Further, such hosts should be able to send and receive data on PF_XUNET sockets. Data sent over the IP path does not obtain any QoS guarantees - we view this strictly as a migration path to allow hosts without ATM host interfaces to access services on ATM networks.

4. Design Objectives

While meeting the service objectives in Section 3, we had the following design goals in mind.

Ease of modification and extendibility: We wanted to be able to modify the signaling protocol and applications without having to reboot the kernel. Since Xunet is an experimental network this was an important consideration.

Robustness: The extensions should work correctly independent of the behavior of the application programs. We did not want to crash the signaling entity or the kernel because of a misbehaving application. Thus, the system should protect itself from programs that crash, are malicious, or hold a half-open connection, i.e. to an application on a remote site that has failed.

Scaling: The number of active native mode ATM connections at an endpoint should not be limited by the design.

Frugal use of resources: Our extensions should not be a drain on the memory or CPU resources of an end point or the network. In particular, if an application reserved any resources and then crashed, the signaling protocol should

detect this and release any resources bound to that application throughout the network.

QoS Negotiation: Applications should be able to specify and negotiate their QoS requirements when setting up a connection. This QoS specification can be used for scheduling critical resources at endpoints and switches in the network. Of course, if ATM data is encapsulated in IP, no QoS claims can be made for traffic traversing the IP internet.

Orthogonality of implementation: Signaling and OS support should not make any assumptions about the functionality implemented by the protocol stack. This would allow us to experiment with different protocols in the stack (for example, implementing different congestion control schemes) without changing the support functions.

5. Major Design Decisions

The design considerations above led to some design decisions, which we discuss in this section.

5.1. Signaling entity in user space

The first major decision (which predates our work) was to implement signaling in user space, rather than in the kernel. In our experience, code in user space is far easier to develop and modify. Signaling state information is easily available and can be used by network management software. This decision does not have a severe performance overhead. First, note that signaling is invoked only during call setup, and does not impact the speed of data transfer. Second, the signaling protocol has to interact with the application in user space anyway (see Section 7.1), and so crossing the user-kernel boundary is unavoidable whether signaling is in user space or in the kernel. (However, with a user-space implementation, there would be four context switches, instead of two with an in-kernel implementation.) Third, while there is a gain in performance from an in-kernel implementation when a user process terminates abnormally, since the kernel can hand the termination message to the signaling entity without crossing the user-kernel boundary, this is not the common case. Keeping these costs and benefits in mind, we feel that the design objectives weigh the decision in favor of an implementation in user space.

5.2. TCP/IP sockets for application-signaling interface

The application-signaling interface allows clients to request a channel to a server from the signaling entity, and servers to register themselves. The traditional way of implementing this interface would be as a parameter to the `connect` or `bind` system call. The problem is that we would like to support QoS parameters that are not only passed into the kernel but also need to be negotiated with the peer endpoint and then passed back to the application. If QoS parameters are passed as part of the address structure or as a socket option, this would make QoS negotiation cumbersome (see Reference [4]).

Further, we wanted to allow an application on any host with IP connectivity to a router to be able to request a channel from the signaling entity. Since we had already decided to place signaling in user space, given these two requirements, we decided that applications should communicate with the signaling entity using inter-process communication (IPC) instead of system calls. We considered both BSD sockets and ANSA [1] RPCs for IPC. We wanted the IPC facility to be easy to use, ubiquitously available, and reliable. This ruled out ANSA and UDP, so we used TCP/IP for IPC, in essence building a special-purpose RPC facility.

5.3. Pseudo-device to kernel for exchanging state information

There are three situations in which the signaling entity needs to learn about application state, or inform an application about a change in network state.

- An application that was using a VCI terminated and the corresponding connection needs to be torn down.
- A connection was requested and established but the requesting process terminated before it could use it. Again, the connection has to be torn down.
- A connection was closed at the remote end. The signaling entity should inform the application at the local end that the connection was closed.

One approach we considered was to poll each client and server application and keep track of state information explicitly. While this would allow the signaling entity to know about process state, communicating network state to the application becomes hard, and would probably need a signal handler in every application. We decided that this was too cumbersome. Instead, note that process state information is easily available inside the kernel, and it is relatively simple to ask the kernel to modify socket state as the network state changes. Thus we decided that the information exchange about process and network state would be mediated by the kernel.

The mechanism to be used for communicating between the signaling entity and the kernel could have been either a 'special' socket or a pseudo device. Both allow two-way asynchronous communication, and have similar implementation complexity. Since we had source code for a pseudo-device already, we decided in its favor. The pseudo-device is used only for user to kernel space communication - communication between peer signaling entities uses a native-mode Permanent Virtual Circuit.

5.4. Link remote hosts with IP encapsulation

We could have moved data from the PF_XUNET protocol stack on a host to a router by setting up a proxy connection at any of the datalink, network, transport or application layers. Efficiency requires this connection to be as low in the protocol stack as possible. If we restricted hosts to ones that are on the same physical network as a router, then we could have modified the Ethernet or FDDI device driver to set up a proxy datalink layer connection with a new datalink protocol type. But this approach has the drawback that the source code for device drivers is neither easily available nor easily modifiable.

Instead, we decided to define a new encapsulation protocol as a raw protocol over IP. This allows PF_XUNET data (unsegmented frames without the AAL5 trailer) to be forwarded across an arbitrary internetwork without sacrificing too much performance. As a further advantage, adding a new protocol over IP to most UNIX-like kernels takes little effort. Though encapsulation above TCP is even easier, this is not only inefficient, but also could cause complex interactions between PF_XUNET flow control and TCP flow control, which we would much rather avoid.

Encapsulation above UDP buys us little functionality for the efficiency loss, and also was rejected.

Xunet implements a minor variant of the AAL5 adaptation layer, which guarantees that the receiving AAL can detect out of order frames and cell loss within a frame. We wanted to preserve these two guarantees for the path over IP. Since the encapsulation protocol does not segment AAL5 frames cell loss within a frame is not possible. All the encapsulation header needs to do is to detect out of order frames, which we do using a sequence number field.

6. Implementation

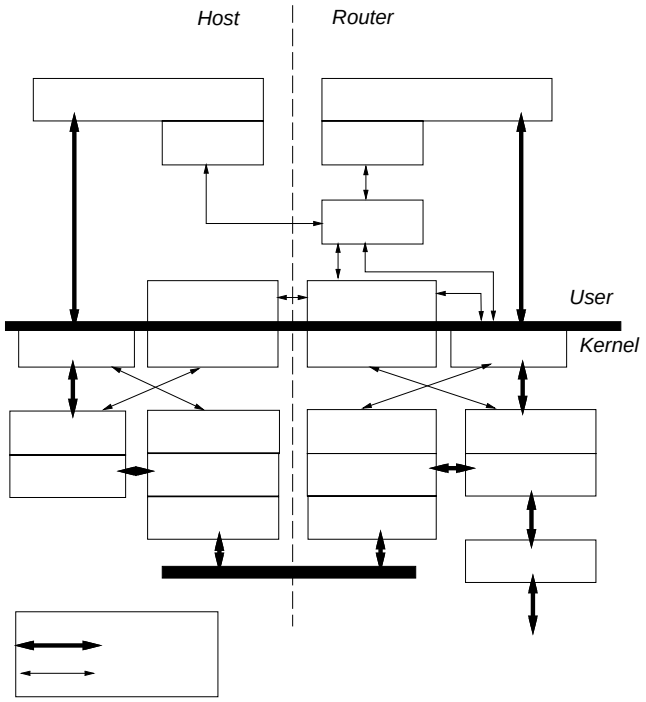

Figure 2: Overall design

In this section, we sketch out the implementation of the design discussed above (Figure 2). The major components in our implementation are the PF_XUNET protocol stack which actually implements the native-mode stack semantics, and the signaling entity on the router (called `sighost`). A single `sighost` serves applications running on the router as well as any number of applications running on hosts connected over IP. The `/dev/anand` pseudo-device provides the interface between the signaling-entity and the kernel. [†] A small client and server stub, (called `anand client` and `anand server` respectively) relay messages between the host kernel and `sighost`. Finally, the `PROTO_ATM` protocol provides ATM encapsulation/decapsulation as well as virtual circuit switching in the router kernel. In order to simplify the communication between a client or server application and `sighost`, we provide a library (marked *library*). The ATM host-interface on the router is called the *Hobbit* host interface, and it is controlled by the *Orc* device driver.

The overall flow of control is as follows. A server program on a host or router registers itself with `sighost` and awaits calls from a client. A client application uses the user library to contact the nearest `sighost` and requests a connection with a particular QoS to a destination and service name. This is then routed to the server, which is free to accept or deny the call and also modify the QoS parameters. The modified QoS and the accept/deny decision is returned to the client. Once the call is set up, data is sent from the client to the server using a PF_XUNET socket, possibly with IP encapsulation of AAL frames.

7. Implementation Details

In this section, we describe the details of the implementation sketched out in Section 6. The signaling entity at the router is central to our design. It only acts in response to messages received from the user library, the local or remote kernel, or the peer signaling entity. Thus, we describe our implementation in terms of these message exchanges.

7.1. Signaling-Application Interface

All applications are expected to use the user library to interact with the signaling entity. We first consider the messages exchanged by the signaling entity and a server application (see Figure 3). A server sends an EXPORT_SRV message when it wishes to advertise a service name. This message contains the port number on which the server will be listening to be informed of incoming requests from a remote client. The signaling entity acknowledges this message with a SERVICE_REGS message. When the signaling entity receives a call from its peer, it notifies the server with an INCOMING_CONN message.

In order to provide a measure of security, the INCOMING_CONN message carries an identifier, or *cookie* with it. A cookie is a 16 bit capability that gives the holder the right to access a socket bound to a particular VCI. `sighost` maintains a per-VCI table of cookies. When an endpoint does a `connect` or an `accept` on a socket, it must supply the cookie provided to it during call setup. The PF_XUNET socket layer passes up the cookie and VCI to `sighost` for these two calls, and `sighost` can then authenticate the call. If authentication fails, the call is torn down, and the socket marked unusable. A malicious process on the server machine cannot access a VCI opened to a server, since it would not be able to guess the cookie. A similar cookie is handed to the client as discussed below, and prevents unauthorized clients from communicating with a server. A cookie can be handed to a child of the server application or any third party. Cookies last for the lifetime of a connection, and are lost due to termination of an application, its peer or an intermediate router.

The INCOMING_CONN message also carries the QoS requested by the client. The server can now accept the connection by sending an ACCEPT_CONN message carrying the modified QoS that is acceptable to the server. If a server does not want to accept the call, it sends a REJECT_CONN message instead. The signaling entity replies to an ACCEPT_CONN message with a VCI_FOR_CONN message containing the VCI for the connec-

[†] In honor of the evergreen star of the Indian screen, Dev Anand.

tion.

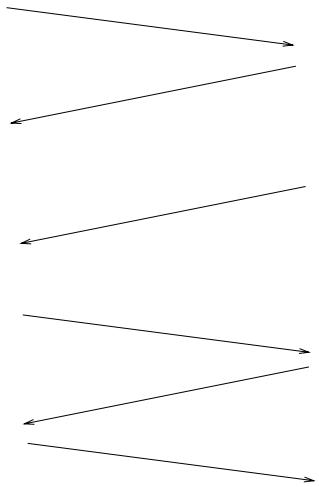

Figure 3: Messages exchanged when an echo server registers itself

We now discuss the messages exchanged between a client application and the signaling entity (see Figure 4). A client sends a CONNECT_REQ message to the signaling entity when it wishes to connect to a server. It has to specify the destination, the service name, the QoS required and the port on which it should be informed about connection establishment or rejection (the port number is used only to support our RPC-like mechanism). The server replies to a CONNECT_REQ message with a REQ_ID message. This has a cookie identifying the connection that will be established on the client's behalf. When the connection is actually established, a VCI_FOR_CONN message is sent to the client with the modified QoS and the VCI for the connection. A client may cancel its connection request with a CANCEL_REQ message.

7.2. Signaling-Kernel Interface

This interface is implemented by the `/dev/anand` pseudo-device and the `anand` client-server pair. The pseudo-device is implemented as a character device and the client and server communicate using TCP/IP sockets. A call downwards is made when `sighost` receives a termination indication from a peer, or authentication on a socket fails, and `sighost` needs to inform the kernel. `sighost` sends a message to `anand server` which either does a write on the router's pseudo-device, or passes it on to `anand client` which then does a write on the host's `/dev/anand/` pseudo-device. In either case, the pseudo-device's write routine calls the socket layer's `soisdisconnected` routine, marking the socket as being unusable.

The kernel passes messages upwards when a process terminates, or when it binds or connects to a PF_XUNET socket. The termination indication is needed to allow `sighost` to inform the remote router (or host) that the client (or server) no longer exists, and the connection can be torn down. The bind or connect indication is needed since a process might request a VCI, but not use it, or terminate before binding (resp. connecting) to it. Since no bind (resp. connect) occurred, the kernel will not pass a termination message to `sighost` on process termination, and network resources would stay blocked indefinitely. To prevent this, `sighost` keeps a per-VCI timer that is loaded when a VCI is handed to an application. If no bind (resp. connect) indication is received before timeout, the connection is torn down.

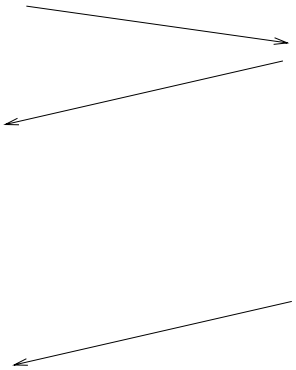

Figure 4: Messages exchanged when a client establishes a call

Messages from the kernel are queued in the read queue of the pseudo-device. The device supports the `select()` system call. Thus, `anand server` (or `anand client` on a host) simply blocks on `select()`, and when unblocked, passes the message on to `sighost` (via `anand server`, if on a host).

7.3. Signaling Entity Internal State

As the messages described in Sections 7.1 and 7.2 are processed, `sighost` maintains internal state in one of five lists. The *service_list* maps service names to server ports. This list is looked up when an incoming call arrives. The *outgoing_requests* list stores client requests waiting for a reply from a server. This allows the `sighost`-client interface to be non-blocking. The *incoming_requests* list stores connections awaiting acceptance or rejection by the server associated with the specified service name. This is symmetric to the *outgoing-requests* list. The *wait_for_bind* list identifies connections that have been established by the signaling entity, but have not yet been bound to by an application. The *VCI_mapping* list contains the current set of VCIs that have been bound and their corresponding sockets. If `sighost` receives a close message from its peer, this list is used to inform the kernel to mark the socket as disconnected.

7.4. AAL over IP

We designed a way to carry ATM data by encapsulating it in IP, thus allowing arbitrary IP hosts to send AAL frames into Xunet. In order to proxy signaling at the host, we could have run a copy of `sighost` at both the router and a host. However, this would require quite a few changes to `sighost`, since peer

`sighosts` would need to communicate information such as IP forwarding addresses between themselves. Instead, `sighost` runs only on the router, and the `anand` client-server pair manage IP specific information (described below).

Data written by a client to a PF_XUNET socket is encapsulated in an IP packet and sent to an Xunet router. Here, the AAL5 trailer is added, and the AAL5 frame is segmented into cells and sent over Xunet to a remote router. The remote router re-encapsulates the data in IP and sends it to the server on a remote host, where it appears at a receiving PF_XUNET socket. We now describe the details of this scheme.

We first discuss the implementation at a host. Note that signaling messages from the user library to `sighost` are sent over TCP/IP, so there is no change to the user library. We modified the host kernel in two ways (see Figure 2). First, we added the `/dev/anand` pseudo-device, the PF_XUNET protocol family and the Orc device driver to the kernel. As in a router, writes to a PF_XUNET socket are handed to the Orc device driver (which would normally control the Hobbit host interface), and reads on the socket are translated to reads on the device driver. (Since the Hobbit board computes AAL5 trailers, the data passed down from the Orc on a send is simply a pointer to an mbuf chain, which is the same as the interface to raw IP.) Second, we added an encapsulation/decapsulation protocol layer (marked PROTO_ATM) over a raw IP socket. The device driver's output routine calls the encapsulation routine and its input routine reads from the decapsulation routine. Thus, the existing PF_XUNET implementation on a router could be ported to a non-router host with no change, other than to replace calls from the device driver to the Hobbit board with calls to the encapsulation/decapsulation layer.

The encapsulation layer attaches a header to the data, and puts it in the data portion of a IP packet. The header has three fields:

Source Address	ATM address of the sending node
Sequence Number	To detect out-of-order packets
VCI	VCI on which to send the encapsulated data

We do not currently have a header checksum field, since our IP links are over reliable FDDI links. A header checksum could be added to the encapsulation header if needed.

The IP header of the encapsulated packet needs to have its protocol type and destination field filled in. The IP protocol type field is set to a new protocol type called IPPROTO_ATM, to allow demultiplexing at the router. The IP address field contains the IP address of the closest router. This is configured by `anand client` on startup by writing a message on a socket bound to the IPPROTO_ATM protocol. This message has the router's IP address as its destination address. The socket send routine for IPPROTO_ATM assumes that a write to it is a configuration message. It sets the IP forwarding address for IPPROTO_ATM to the destination address of this message, and simply discards the message. This allows a host to reconfigure its target router easily. The default forwarding decision can be set by putting `anand client` in the boot sequence.

At a router, a decapsulation protocol is placed above the IP layer. When it is handed an encapsulated packet (of protocol type IPPROTO_ATM) by IP, it checks that the data is in-sequence and then simply hands the mbuf chain to the Orc driver along with the VCI. The AAL5 trailer computation, segmentation and data transmission is handled by the Hobbit ATM host interface [2]. Since these data intensive operations happen only at the router, there is little overhead in providing the PF_XUNET interface to non-ATM hosts. Computing the encapsulation header is inexpensive (roughly the same time as for computing the UDP header).

At the remote router, the Orc driver needs to hand over data received on a VCI either to the IP encapsulation routine or to the router's PF_XUNET protocol. To do this demultiplexing, the router kernel maintains a table that contains a pointer to the handler procedure for each VCI. In addition, for those VCIs that need to be forwarded to a server on a remote host, it maintains a per-VCI IP destination address table. When a server on a remote host binds to a VCI, the bind message is handed to `anand client` by the remote host kernel, as described earlier. `anand client` sends a message to `anand server`, which then knows the destination IP address for incoming data packets, and also the VCI on which this data will arrive. It saves the IP address in a table, and sets the handler for that VCI to be the IPPROTO_ATM encapsulation routine. By maintaining an explicit per-VCI IP destination address table, one can trivially route the IP path over one of multiple IP interfaces (such as Ethernet and FDDI). The handler routine for a VCI owned by a process running on the router is automatically set to the IP packet handler by PF_XUNET.

The two mappings (i.e. VCI to handler, and VCI to IP address) are configured by writing a VCI_BIND message from `anand server` to a IPPROTO_ATM socket (the same mechanism as is used at a host to set its IP destination address field). The VCI_BIND message contains the IP address of the host running the server, and the VCI on which it will be receiving data. The socket send routine for IPPROTO_ATM sets the handler procedure pointer for that VCI to the IPPROTO_ATM encapsulation routine and also sets up the VCI's address mapping to the address of the remote host. If `sighost` receives a termination message for a VCI, it passes this on to `anand server`. The server then writes a VCI_SHUT message to an IPPROTO_ATM socket so that no more data is forwarded to the remote host on that VCI. The two mappings are cleared, and the Orc driver is told to discard any more data arriving with that VCI.

8. Application code example

In this section we present code fragments for an example client and server that uses our extensions. Our goal was to make it easy for an application developed over TCP/IP and BSD sockets to be ported to PF_XUNET. This is achieved by hiding the message exchanges between the application and the signaling entity in a user library. Figure 5 presents a code fragment for a server. A server calls the `export_service` routine to register itself, then awaits incoming connections using `await_service_request`. When a connection arrives, the server is unblocked, and it can accept a call using `accept_connection`. At this point, the server can spawn off a child to do the actual work. Thus, the three additional calls hide the signaling process from the application.

Figure 6 shows a code fragment for a client. A client application needs only one additional call, to `open_connection()`. In the current version of the library, this is a blocking call, and when the client is unblocked, it can

`connect` to a PF_XUNET socket. It would be straightforward to provide a non-blocking version of `open_connection` if that were needed. When either client or server closes a PF_XUNET socket, the signaling entity will automatically tear down the associated call.

```
export_service("traffic", TCP_PORT);

/* Initiate exchange of messages with the
 * signaling entity as shown in Figure 3 */

listen_sock = create_receive_connection(TCP_PORT);

/* Create a regular TCP socket for doing an
 * accept. Sighost will inform the program
 * using this socket  when it receives a
 * connect request */

cookie = await_service_request(listen_Sock,
                               comment, QoS);
VCI = accept_connection(cookie, comment, QoS);

/* This VCI is for the incoming connection request,
 * with QoS parameters given in 'QoS'. A server may
 * modify the QoS and return it to the client. */

recv_sock = socket(AF_XUNET, SOCK_DGRAM, 0);
add.family = AF_XUNET;
addr.VCI = VCI;

bind(recv_sock, addr);
/* Directs the protocol stack to pass any data
 * received on this VCI to the given socket */

/* Now the server can receive data on this socket */
```

Figure 5: Code fragment for a server using signaling extensions

```
VCI = open_connection("mh.rt", "echo", TCP_PORT,
        "this is a comment", traffic_class, QoS);

/* Returns a VCI with proper QoS */

send_sock = socket(AF_XUNET, SOCK_DGRAM, 0);

add.family = AF_XUNET;
addr.VCI = VCI;

connect(send_sock, addr);
/* This binds the VCI to the socket */

/* Now the client can send data on this socket */
```

Figure 6: Code fragment for a client using signaling extensions

9. Measurements

We used the technique similar to that of Clark et al [7] to count the number of instructions to send and receive packets over PF_XUNET at a host. We looked at assembly code for the protocol-specific routines, ignoring procedure call overheads and calls to memory management routines. Thus, the instruction count is what is needed for packet protocol processing, assuming that procedure calls were replaced with inline code, and independent of the details of the operating system's memory management, socket and interrupt masking routines. We used the IP send count of 61 and receive count of 57 from Reference [7].

On the receive side, the number of instructions executed from software interrupt of IP by the device driver to enqueueing the message in the socket layer is 194 + 8 * (# of mbufs in the message). This is broken down into 57 for IP, 36 for IPPROTO_ATM, 2 for the Orc device driver, and 99 + 8 * (# mbufs) for PF_XUNET. On the sending side, the total number of instructions to send a message (from socket layer to device) is 119 + 8 * (# of mbufs). This is broken down into 58 + 8 * (# of mbufs) for IPPROTO_ATM, and 61 for IP. On the send side, the PF_XUNET and Orc send routines simply call the next layer down without touching the data or the header, thus incurring zero cost (since procedure call overheads are not counted). At a router, switching an encapsulated packet adds 39 instructions to the overhead for FDDI/Ethernet driver input, IP switching and Orc driver output. This is summarized in Table 1.

	Receive	Send
Component	Instructions	Instructions
PF_XUNET	99+ 8*#mbufs	0
Device driver	2	0
IPPROTO_ATM	36	58 + 8*#mbufs
IP	57	61
Total	194 + 8*#mbufs	119 + 8*#mbufs

Table 1: Instruction counts for the send and receive paths at a host

We measured the time it takes for a client to establish a connection and for a server to register a name and accept a connection. The measurements were taken on a testbed consisting of two routers (SGI 4D/30 workstations), with a three hop (two switch) ATM path between them. The time to register a service was 17-20 ms, and most of the time was due to the four context switches performed in completing this RPC (application to kernel, kernel to `sighost`, `sighost` to kernel and kernel to application). The time to accept an incoming call was also around 20ms, reflecting the dominant cost of context switches. The time to establish a call between processes on two different routers was about 330 ms. This was mainly due to the large amount of maintenance information logged per call by the signaling entities. Since connection establishment can be made non-blocking, we do not think that this poses a serious problem. However, there is ample scope for optimization and performance tuning.

The overhead for encapsulation/decapsulation is small (39 instructions), and so we expect throughput between a host and a router to be comparable to that of UDP. Since this path is not traffic-controlled, the throughput and delay on this path is highly variable, and we did not measure it.

Table 2 gives the sizes of the principle software

components at a host in lines of C code (with comments) and text, data and BSS size. The code size is fairly small compared to the kernel size of ˜1.75 MB.

Component	Lines	Text	Data	BSS
Sighost	1204	104.7	26.8	77.9
User lib	373	2.5	1.4	0
/dev/anand	382	1.95	1.15	9.6
PF_XUNET	463	2.4	.37	16.4
IPPROTO_ATM	164	.64	.20	0
Orc	96	2.5	.18	22.5

Table 2: Code sizes for principal components at a host (KB)

10. Experience and discussion

The design described in Section 7 has been implemented at AT&T Bell Laboratories, Murray Hill. The signaling extensions were completed in August '93, and AAL5 over IP in December '93. Our experience with the design has been positive. In this section, we discuss the extent to which the design goals stated in Section 4 have been satisfied.

Ease of modification and extendibility: We were able to easily modify the signaling entity since it runs in user space. An example: the first cut at signaling did not support QoS parameters, but we were able to add them with only a few hours of work.

Robustness: Routers with the modified kernel have stayed up even when thousands of calls have been setup and torn down. We designed an intensive workload in which a hundred calls were initiated as fast as possible. Each call was held for one second, then torn down. This workload has been run successfully between routers as well as between a host and a router. We also ran tests where clients and servers were terminated during various stages of the call setup process. The network and signaling state were always correctly restored. Based on this experience, we are satisfied that our software is robust enough to be placed in the field.

Scaling: There were two subtle scaling problems. First, the size of the message buffer in the pseudo-device restricts the number of messages that the kernel can enqueue for the signaling entity. Initially we configured the device with only eight buffers, which led to problems when a large number of connections were simultaneously opened by the test workload, and some bind indications were lost. Our current implementation has eighty buffers, which has proved to be adequate. In any case, each message is small (4 bytes), so it cheap to increase the size of this buffer.

The second problem is that the maximum number of file descriptors that can be accessed by a server restricts the number of simultaneously establishing connections. Each incoming connection from a client is forwarded by the signaling entity to the server over a TCP/IP socket, and is associated with a corresponding file descriptor. This descriptor is kept open for the duration of connection establishment and then immediately closed. However, even after the socket is closed by both ends, TCP keeps the descriptor in the table for two Maximum Segment Lifetimes to handle packets that arrive after long delays. This uses up table space, and restricts the number of clients that can establish a connection to a particular server simultaneously. Since the table size is typically around twenty, this created problems for the test workload which opened a hundred connections to the same server. To get around this problem, we increased the kernel's per-process file descriptor table size to 100. With this change, and the increase in the message buffer size, we were able to establish and keep open two hundred connections between two routers.

Frugal use of resources: We were able to recover reserved resources throughout the network when either the client or the server application terminated.

QoS negotiation: As discussed earlier, the user library allows client programs to specify a QoS specification that can then be modified by a server application and returned to the client application. We believe that this provides a minimal framework in which to provide QoS to applications in an integrated network. At the moment, the QoS string carries only a service class and a bandwidth request, as discussed in Reference [17]. We plan to extend this framework as we get more experience with application needs.

Orthogonality of implementation: Our implementation does not make any assumptions about the functionality provided by the PF_XUNET stack. In fact, the stack currently implements only a UDP-like functionality. An outline of the semantics we plan to support in the stack is presented in Reference [12]

11. Related work

Our work is most similar to that of Biagioni, Cooper and Sansom [3] at Fore Systems. They have integrated an ATM Application Programmer's Interface and the proprietary SPANS signaling protocol to allow users to send ATM cells into a Fore ATM network. They locate the signaling entity in the device driver, making it hard to implement and modify SPANS signaling. While this may be reasonable for a commercial offering, it is unacceptable for a research environment such as ours. (However, putting signaling in the kernel makes it much easier to free up resources from terminated applications.) The ATM API they describe bypasses the socket layer. This makes it harder to port existing BSD-socket based applications. Finally, unlike our design, they do not allow applications on machines without ATM host interfaces to access their API and send encapsulated cells into their ATM LAN.

Black and Crosby at the University of Cambridge describe support for the MSNA ATM protocol stack in Reference [4]. A key design decision that they made was to use a special socket for communication between the signaling entity and the kernel. This leads to numerous problems with concurrent accesses to per-VCI data structures. Further, the kernel needs to be aware of signaling messages interspersed with control messages on the special socket. We sidestep most of these problems by using a pseudo-device.

The work of Cranor and Parulkar, who have designed and implemented the COIP-K protocol domain in the BSD Unix kernel [8], is similar in that they also provide an infrastructure for protocol development and testing. They have extended the protocol domain to support multiple COIP protocols simultaneously.

However, they do not discuss the interaction of the signaling entity and the kernel, which we believe is critical to providing per-application QoS.

Robin, Coulson, Campbell, Blair and Papathomas at Lancaster University have described an ATM stack implementation in the Chorus microkernel [16]. Their work is much more ambitious in that they consider both network and CPU scheduling for performing admission control of continuous media streams. They describe the semantics of the stack, its implementation, as well as the support needed from the system. However, since all their implementation is in user space, they do not face the user-kernel interaction problems that are common in Unix.

Our work can be viewed as complementary to that of Campbell et al at the Lancaster University [5, 6] and by Fry et al at the University of Technology, Sydney [11]. They have designed the semantics of a protocol stack that provides per-application QoS. We hope to learn from their work in implementing suitable protocols in the PF_XUNET domain.

The QoS parameters passed by a client or server application to the signaling entity can be used to schedule resources at the end system, as described in [15] or in the network (see Reference [18] for a partial survey). This is an area rich in research possibilities, and we hope to use our testbed in exploring some of them.

12. Conclusions

We have a clean design for implementing a native mode ATM stack in a Unix environment (Figure 2). We have implemented this design in the IRIX kernel. Our work allows user applications to establish a connection parameterized by QoS parameters over ATM networks. A user library and Berkeley socket compatibility makes the task of porting existing applications to the new framework quite straightforward. The design is robust and easy to extend. A novel technique allows AAL frames from IP-connected hosts to be encapsulated in IP packets and thus enables them to access ATM networks. We believe that our research will enable future work in areas such as designing connection-oriented services that can specify QoS requirements to the network, and efficient scheduling of end-systems to support QOS. This is essential in any future multimedia network.

13. Acknowledgments

We would like to thank M.J. Dixon for the original implementation of the PF_XUNET stack and signaling protocol, and Dave Presotto and Joann Ordille for comments on an earlier draft of the paper. Dimitri Pendarakis helped us collect performance measurements on our testbed. Thanks also to the anonymous referees for their detailed and insightful comments.

14. References

1. *ANSA Reference Manual Release 1.00*, APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, March 1989.
2. A. Berenbaum, M. J. Dixon, A. Iyengar and S. Keshav, Design and Implementation of a Flexible ATM Host Interface for XUNET II, *IEEE Network Magazine*, July 1993.
3. E. Biagioni, E. Cooper and R. Sansom, Designing a Practical ATM LAN, *IEEE Network Magazine*, March 1993.
4. R. Black and S. Crosby, Experience and Results from the Implementation of an ATM Socket Family, *Proc. USENIX '94*, 1994.
5. A. Campbell, G. Coulson, F. Garcia and D. Hutchison, A Continuous Media Transport and Orchestration Service, *Proc. ACM SIGCOMM*, 1992.
6. A. Campbell, G. Coulson and D. Hutchison, A Multimedia Enhanced Transport Service in a Quality of Service Architecture, *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.
7. D. D. Clark, V. Jacobson, J. Romkey and H. Salwen, An Analysis of TCP Processing Overhead, *IEEE Communications Magazine*, June 1989, 23-29.
8. C. D. Cranor and G. M. Parulkar, An Implementation Model for Connection-oriented Internet Protocols, *Internetworking: Research and Experience 4*, 3 (September 1993).
9. D. C. Feldmeier, Multiplexing Issues in Communication System Design, *Proc. ACM Sigcomm 1990*, October 1990, 209--219.
10. A. G. Fraser, C. R. Kalmanek, A. Kaplan, W. T. Marshall and R. C. Restrick, Xunet 2: A Nationwide Testbed in High-Speed Networking, *Proc. IEEE INFOCOM 1992*, May 1992.
11. M. Fry, A. Richards and A. Seneviratne, Framework for Implementing the Next Generation of Communication Protocols , *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.
12. S. Keshav, Semantics of a Native-Mode ATM Protocol Stack, *Submitted to ACM Multimedia '94*, March 1994.
13. S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, in *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.
14. I. M. Leslie, D. R. McAuley and D. L. Tennenhouse, ATM Everywhere?, *IEEE Network Magazine 7*, 2 (March 1993).
15. K. K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser and W. Duso, Operating System Support for a Video-On-Demand File Service, *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.
16. P. Robin, G. Coulson, A. Campbell, G. Blair and M. Papathomas, Implementing a QoS Controlled ATM Based Communications System in Chorus, Internal Report MPG-94-05, March 1994. Available by anonymous FTP from `comp.lancs.ac.uk:/pub/mpg/MPG-94-05.ps.Z`.
17. H. Saran, S. Keshav, C. R. Kalmanek and S. P. Morgan, A Scheduling Discipline and Admission Control Policy for Xunet II, *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.
18. H. Zhang and S. Keshav, Comparison of Rate-Based Service Disciplines, *Proc. ACM SigComm 1991*, September 1991.