Semantics and Implementation of a Native-Mode ATM Protocol Stack

S. Keshav, AT&T Bell Laboratories, Murray Hill, NJ 07974, USA
H. Saran, Indian Institute of Technology, Hauz Khas, New Delhi, 110016, India

Abstract

We describe the semantics and implementation of a protocol stack that gives applications access to the Quality of Service guarantees available at the ATM level (a *native-mode ATM* protocol stack). The stack was designed using three principles: *minimal functionality in the critical path, no logical multiplexing,* and *specific targeting to ATM.* It allows applications to establish simplex or duplex virtual circuits with error control, leaky-bucket or feedback flow control and arbitrary size messages. In addition, a signaling component provides connection and Quality of Service management. The stack has been implemented in a simulator and two networks of Personal Computers interconnected respectively with Ethernet and ATM adaptors. We discuss our experiences with developing and porting the stack, and show how our design principles lead to rapid deployment as well as a small protocol overhead.

1 Introduction

A strong motivation for building ATM networks is that they can provide per-virtual circuit (VC) end-to-end Quality of Service (QoS) guarantees. This advantage is lost if the per-VC guarantees are hidden from applications due to interposition of connectionless layers (such as IP) or if streams with disparate QoS requirements are multiplexed together. We call a protocol stack that allows applications to access ATM level QoS guarantees a *native-mode ATM* protocol stack. This paper describes the semantics of native-mode ATM stack developed for the IIT, Delhi Low-cost Integrated testbed (IDLInet).

IDLInet is inexpensive testbed for studying protocol design issues in ATM networks. The testbed has been implemented on three platforms: a packet-level network simulator [13], a network of Personal Computers (PCs) running the MS-DOS operating system and interconnected by Ethernet, and a network of PCs running the Plan 9 [18, 19] operating system interconnected with Fore Systems' ATM adaptors and FORE switches. Since the system was first developed on the simulator and then ported to the other systems, the software on all three platforms is nearly identical. Hence, we will discuss only the simulator version in this paper.

The paper is laid out as follows. Section 2 presents the design principles used in our work, and Section 3 describes the services provided by the stack. This is followed by an overview of the operation of the stack in Section 4. Sections 5-9 describe each protocol layer in detail. Finally, Section 10 presents current status, and Section 11 discusses related work.

2 Design Principles

Our design is based on a few consistently applied design principles: *no logical multiplexing*, *minimal functionality in the data path*, and *specific targeting to ATM.* We discuss these in turn below.

Logical multiplexing refers to the mapping of multiple streams of layer *n* of the protocol stack into a single stream at layer *n-1* [6] (this is sometimes called *layered* multiplexing). IDLInet does not logically multiplex connections; there is only physical multiplexing of the virtual circuits at the lowest layer. This has several advantages [6, 8]. First, per-application QoS requirements can be communicated both to the network and the two endpoint operating systems, enabling intelligent per-VC scheduling of critical resources. Second, there are fewer header overheads since any header information that does not change per protocol data unit can be made part of call setup. Third, it is possible to customize the semantics of the protocol stack per virtual circuit, if that proves necessary (for example, the service given to a control VC could be different from that given to a data VC at every layer of the protocol stack). Fourth, the Virtual Circuit Identifier (VCI) provides a single small index into a table of protocol control blocks, considerably simplifying the software structure. Finally, an end-system supporting multiple QoS-sensitive streams need not be designed to support the most demanding QoS, since less demanding streams can be isolated from more demanding streams.

We believe that a protocol stack should have the minimal functionality necessary to provide a simple abstraction of the underlying communication network. The arguments in support of this position are similar to those used by operating system designers in support of a micro-kernel architecture, namely, a small, clean design leads to improved performance. In addition, the protocol stack becomes easy to specify, understand, implement and verify. In order to achieve the goal of a limited, clean design, we have designed the protocol stack and its service interface from first principles, selecting the best mechanisms from existing protocols. The key strategy is to design the stack as a whole, rather than layer by layer: this allows us to remove redundancy in layer functionality.

Finally, the protocol stack is designed to fit into existing ATM standards. The network layer is ATM Adaptation Layer (AAL) 5, so the stack does not re-implement AAL 5 functionality, such as data checksumming, in the transport layer. Also, the stack supports per-VC state and interfaces to ATM signaling. This gives the control plane access to the protocol stack at connection establishment and QoS renegotiation time. For example, a particular VC may require low delay and a high bandwidth data transfer. This information is passed to the transport and datalink layers by the signaling interface at the time of call setup. During data transfer, conflicting performance requirements can be resolved, and the QoS attributes of each VCI taken into account when scheduling critical system resources (this resolution has not yet been implemented).

3 Design Requirements

The protocol stack bridges the gap between applications and the network. It provides some set of services (such as duplex reliable data transfer) to applications, and makes some traffic guarantees (such as leaky bucket conformance) to the network. This set of services and guarantees together constitute the design requirements of the protocol stack.

The set of services provided by the protocol stack should match the anticipated application workload. We believe that ATM networks will need to support continuous media applications, which need QoS guarantees from the network (expressed in terms of deterministic or statistical guarantees of minimum bandwidth, priority, end-to-end delay and loss rate), while conforming to some traffic envelope [7]. We would also like to support data applications, which effectively need a zero loss rate. Still other applications may require a raw bit-stream abstraction upon which they can build custom flow and error control mechanisms.

Instead of providing a service corresponding to each anticipated application workload, we provide a set of orthogonal services which can be combined in order to match application requirements. The four major services are: 1) *simplex and duplex data transfer* 2) *error control* 3) *open-loop and feedback flow control* and 4) *multicast*. The first service is simply to move data, but this data may have errors, and is not subject to any flow control. The second service adds error control to the data movement. By this we mean that data stream seen by an application will have zero loss rate (possibly after retransmissions) and a corruption rate below some vanishingly small threshold. If the corruption rate is unacceptable, or if retransmissions are too slow, applications have the option of implementing Forward Error Control. The next service adds flow control. Open-loop flow control means that an application will have its traffic shaped to conform to some pre-specified envelope (negotiated during call setup). Feedback flow control implies that the transport layer will attempt to match the application's flow rate to the current bottleneck service rate in the network or

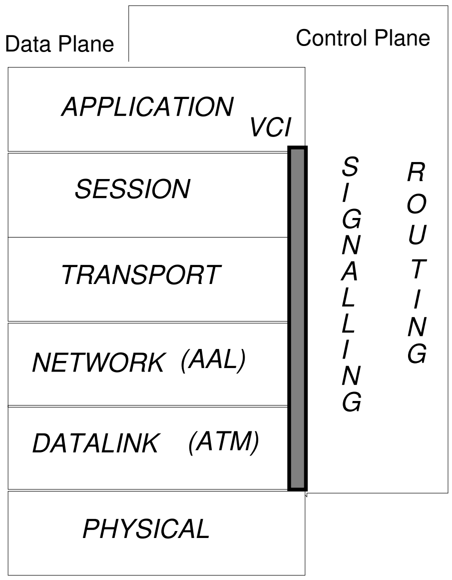

Figure 1: Overall view of the native mode stack. The vertical grey bar indicates that there is no logical multiplexing in the stack.

receiver. Finally, multicast support allows a source to send data to a more than one receiver. By putting together a combination of these services, an application can customize the service interface it wants from the protocol stack.

Currently, we support three combinations of the above services. These are "guaranteed-performance service", "reliable service" and "best-effort service". Guaranteed-performance service provides open-loop flow control, and simplex or duplex multicast connections. There is no error control. An application's QoS specifications are made available to the network, allowing it to reserve resources for each VC.

Reliable service provides error control, feedback flow control and simplex and duplex connections. Multicast is not allowed, nor are QoS guarantees supported. Best-effort service provides only the choice of unicast or multicast service. With this service, there is no flow control, error control or provision of QoS guarantees. We will develop other services derived from the four basic services above, as the need arises.

In addition to the four services described above, some other services are also available: 5) *arbitrary size application data units* 6) *a choice of blocking and non-blocking application interface* and 7) *a choice of byte stream and message transfer abstractions*. These services make it easier to port applications developed using the TCP/IP stack to the native-mode ATM stack.

We now describe the semantics and implementation of a protocol stack (Figure 1) that provides the services described above. We will start with bird's eye view of the entire stack, and then describe the semantics layer by layer.

Note that the description of each layer in this paper conforms to the OSI standard, instead of the ITU/TSS layering. From this perspective, the datalink layer is the same as the ATM layer, and the network layer is the same as the ATM Adaptation Layer. Note also that the flow control functionality of the transport layer is usually performed by the host adaptor or Network Interface Unit in most current designs.

4 Overview

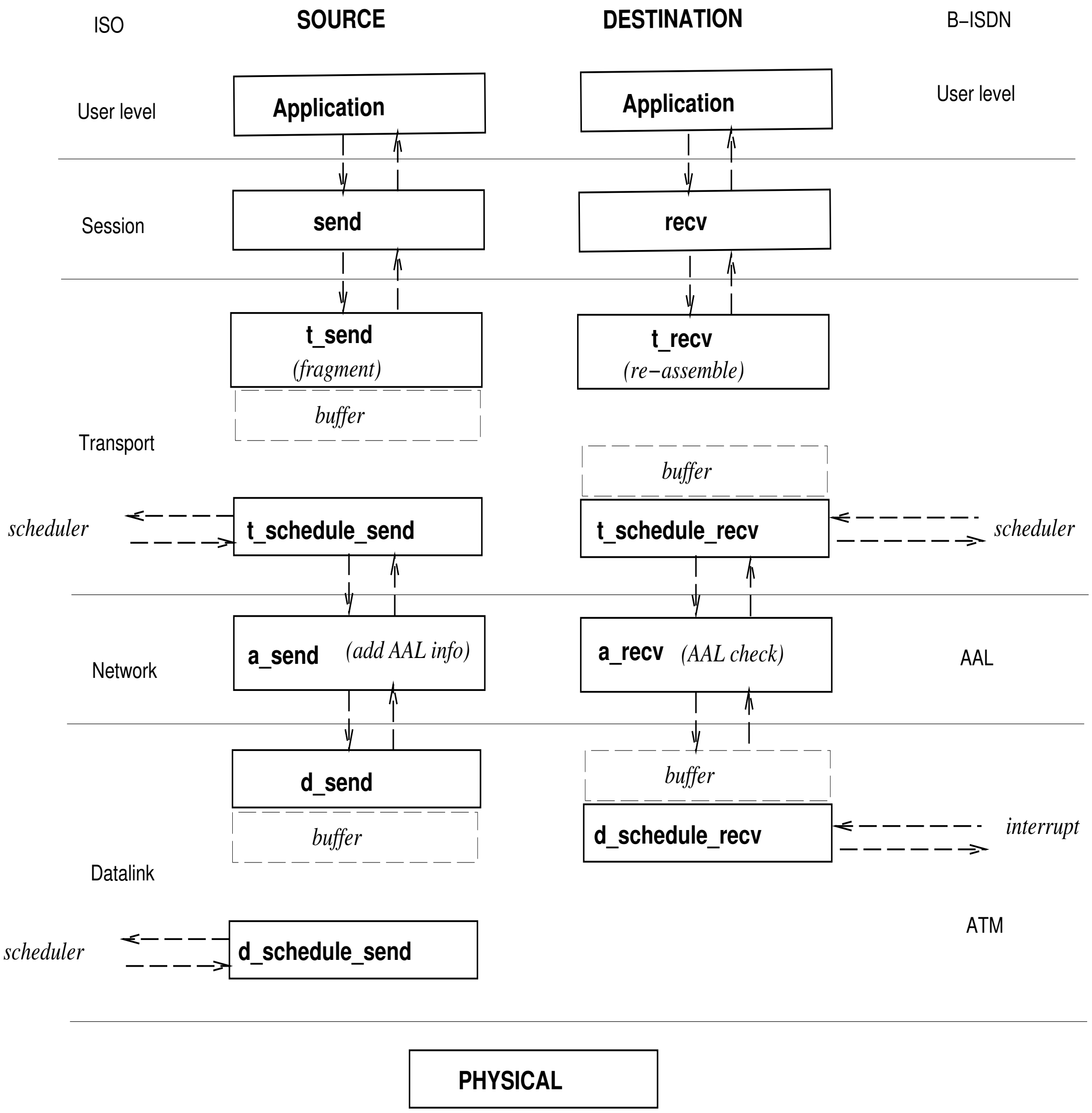

Figure 2: Control flow in data plane.

In this section, we will present an intuitive understanding of how the stack works. We describe how data moves through the protocol stack (see Figure 2), and how applications can use the stack to set up QoS provisioned virtual circuits.

It is easiest to understand the operation of the stack by looking at its building blocks: *procedures, tasks* and a *task scheduler*. A *procedure* is identical to a C procedure. A *task* is a non-blocking and re-entrant piece of executable code that shares an address space with the other tasks in the system. The *task scheduler* registers tasks and launches ready tasks one by one. The called task may, in turn, register other tasks or timeouts with the scheduler. Thus, the basic control loop in the system is for the scheduler to increment time, check for timeouts, and then schedule the next ready task. Each layer in the protocol stack is implemented as a set of procedures and tasks, as described below.

Let us trace the movement of data when an application wants to send data on a virtual circuit. An application sends data by calling the session layer's `send` procedure, which in turn calls the transport layer's `t_send` procedure. This copies the application data into local buffers (crossing the user-kernel boundary if necessary), schedules the transport layer's `t_schedule_send` task, and returns. At some point, the `t_schedule_send` task is launched by the task scheduler, and it decides which transport data units (TPDUs) are ready for transmission (based on open or closed loop flow control). If some TPDU is available, the AAL layer's `a_send` procedure is called, which appends an AAL trailer and calls the `d_send` procedure to actually send the data. If the output line is free, `d_send` sends the data right away, otherwise it queues the data and returns. When the line becomes free, the `d_schedule_send` task is called, which dequeues data and transmits it. If the host adaptor has its own CPU, it can poll the datalink layer queue, eliminating the need for `d_schedule_send`. Note that there are only two copies in the stack, once from application space to the stack's internal buffers, and once from these buffers to the physical link.

On the receive side, when data is received, the interrupt routine copies data from the link into a buffer and schedules the datalink layer's `d_schedule_recv` task. This assembles cells into per-VC buffers, and if the last cell of an AAL 5 frame is seen, schedules the transport layer's `t_schedule_recv` task. When the transport task is run, it calls the AAL layer's `a_recv` procedure, which removes an AAL frame and checks its correctness. If it is a valid frame, the data is placed in the transport layer's buffers. When an application wants to receive data, it calls the session layer's `recv` procedure, which in turn calls the transport layer's `t_recv` procedure. This simply copies out fully received messages into application buffers. Note again that there are only two copies in the receive path - once from the link to the network buffers, and once from the network buffers to the application.

If the transport layer cannot send data due to flow control or lack of tokens in the leaky bucket, it schedules a `t_timeout` task. This is called by the task scheduler when the timer expires. By providing cheap timers to all the protocol layers, the protocol stack scheduler considerably simplifies the implementation.

So far, we have only discussed how data moves in the stack. Before this can happen, a virtual circuit must be established on behalf of the application. We now briefly describe how this happens, details can be found in [22].

The key player in setting up calls is the *signaling entity*. This entity allows servers to register services with it. When a client wishes to set up a VC to a server, the signaling entity creates a signaling message and sends it to a peer in the switch controller. The switch controller performs admission control and forwards the call to the next switch controller on the path to the destination. Eventually, the signaling entity at the remote end is contacted, and if the application accepts the call, the VC is set up between the client and the server. A client passes a QoS specification to the signaling entity, and this is made available to both local and remote protocol stacks, so that per-VC QoS can be provided by both end systems.

Having presented an overview of the stack, we now describe each protocol layer in turn.

5 Datalink Layer

The datalink layer corresponds roughly to the ITU/TSS ATM layer. It is responsible for transferring ATM cells between end-systems over a series of links and switches. At each switch, the datalink layer, implemented in hardware, is responsible for cell switching, cell scheduling and buffer management. By intelligent scheduling of the bandwidth and buffer resources, the datalink layer can provide per-VC QoS at each switch.

On the transmit side, the datalink layer is called from its `d_send` procedure. This first checks if the output line is free. If so, it segments an AAL frame into 48 byte cells and adds a standard 5 byte ATM header (in the ITU/TSS framework, this is done by the AAL layer). If ATM Adaptation Layer 5 (AAL5) is used, then the last cell header in the frame has the ATM-layer-user to ATM-layer-user (AUU) bit set [4]. The datalink layer also adds a per-cell header checksum.

If the output line is busy, then incoming data is queued. When the data is actually sent depends on the available hardware. If the hardware interrupts after cell completion, then the interrupt routine can schedule the datalink send task. If the hardware supports polling, then the scheduler must spin testing for the busy to idle transition. If the hardware has an onboard processor, then this processor can simply poll the datalink queue and asynchronously transfer data.

On the receive side, the interrupt routine copies the data from the hardware into network buffers and schedules the datalink layer's `d_schedule_recv` task. The datalink layer maintains a number of per-virtual circuit reassembly queues. When the receive task is run, it checks the incoming cell's header checksum, and if it is invalid, discards the cell, or else adds it to its corresponding queue. The transport layer's receive task is scheduled when a cell with the AAU bit is seen.

When the IDLInet datalink layer is implemented over a an Ethernet link, a cell to be transmitted is encapsulated into an Ethernet frame and handed to the Ethernet MAC layer for transmission. On the receive side, the Ethernet header is stripped and the cell is handed to the datalink layer for normal processing. In order to provide a worst-case delay bound, peer data link layers coordinate access to the shared link by exchanging tokens. The datalink layer is allowed to transmit only after receipt of an Ethernet frame. If there is nothing to send, then a token is transmitted. Each endpoint of the link sets a timer every time it sends a frame, and the endpoint with the lower Ethernet address regenerates a token if no token or data frame is received before the timeout. This ensures robust operation of the link even if tokens are corrupted or lost. A link failure is detected if there is no activity for three consecutive timeouts.

6 Network Layer

The network layer corresponds to the ITU/TSS ATM Adaptation Layer (AAL). At the moment, IDLInet supports only AAL 5 [4]. AAL 5 provides the abstraction of an end-to-end pipe that can detect lost or corrupted data. This is done by adding a trailer containing a 32 bit data CRC and a 16 bit length field.

On the transmit side, the network layer is called from its `a_send` procedure. This procedure takes a Transport data unit (TPDU), adds the AAL trailer, and calls the datalink layer `d_send` procedure. On the receive side, the network layer's `a_recv` procedure is called from the transport layer's receive task only when it is known that an AAL 5 frame is ready to be received. Thus, the procedure simply picks up the completed AAL frame from the datalink layer and checks it for correctness. Correct frames are handed to the transport layer. In the current version of the semantics, incorrect AAL frames are discarded, though we planning to pass up errored AALs for error-tolerant applications.

Though IDLInet implements the datalink and network layers in software so that cheap non-ATM interfaces can be used to emulate an ATM network, in practice, these two layers would be implemented in hardware on a host adaptor board. We have found it straightforward to provide a software wrapper (at least for the FORE 200 series ATM card) that provides the same interface to the transport layer as the IDLInet network layer. Thus, the transport layer and layers above can be ported with little extra work.

7 Transport Layer

The transport layer provides *simplex* virtual circuits, error control, and flow control. In addition, it segments application layer buffers into TPDUs and reassembles them on the receive side. Here, we present the mechanisms required to provide these semantics.

The transport layer is implemented by the `t_send` and `t_recv` procedures and the `t_schedule_send` and `t_schedule_recv` tasks. `t_send` and `t_recv` are called either by the socket layer or an application to send and receive data. The `t_schedule_send` task is scheduled by the `t_send` procedure, and the `t_schedule_recv` task is scheduled by the datalink layer, as described earlier.

7.1 Segmentation and Reassembly

There are two reasons why the transport layer may want to fragment an application message into TPDUs. First, there is a limit on the size of an AAL frame (44 bytes for AAL 3/4 and 64K for AAL 5). Thus, if the message is larger than this size, it will need to be fragmented.

A more compelling reason has to do with error control. The unit of error detection and retransmission is a TPDU. If this is large, then each loss is reflected in a large retransmission overhead. By keeping TPDUs small, the retransmission inefficiency is minimized. Thus, the TPDU size can be chosen to deal with per-fragment overheads, the connection's error characteristics and the available timer resolution. Indeed, this is the choice of 'Multiplexing Block' in reference [6].

On the transmit side, the transport layer's `t_send` procedure copies an application buffer into a chain of TPDUs, and then schedules the `t_schedule_send` task. In order to preserve message semantics, the TPDU header has a message number, fragment number, and an end-of-message flag.

On the receive side, the `t_schedule_receive` task picks up TPDUs from the network layer and queues them in per-VC queues. If the VC supports message boundaries, fragments are reassembled by the receiving transport layer and the `t_recv` procedure returns only complete messages to an application. If not, `t_recv` returns all available in-order fragments in the receive queue. For both cases, out of order TPDUs are queued awaiting arrival of the missing fragments.

7.2 Error Control

While the AAL 5 checksum detects corruption and loss within an AAL frame, this, by itself, is not sufficient for error control. For a reliable connection, not only must corruption and loss be detected, but lost data must also be retransmitted. This is done at the transport layer.

The fundamental mechanism for dealing with losses is for correctly received data to be acknowledged by the recipient (thus a reliable transport connection requires a pair of simplex VCs - see Sections 8 and 9). If an acknowledgment is not received by the sending transport layer for a sufficiently long time, it retransmits the TPDU.

To allow a receiver to detect duplicate data from retransmissions (which may be arbitrarily delayed, perhaps extending beyond virtual circuit closedown), sequence numbers are necessary. The transport layer uses a standard three way handshake at startup to choose the initial sequence number correctly [23]. We do not use a two-way handshake for termination, since termination is handled by the signaling layer.

The transport layer uses per-TPDU cumulative acknowledgments for redundancy. Cumulative acknowledgments have the added benefit that if an acknowledgement sequence number is repeated, the source can guess with high probability that the packet with a sequence number one larger than this sequence number was lost.

In our scheme, the acknowledgment also carries the sequence number of the TPDU that generated the acknowledgment, allowing sources to additionally determine which sequence numbers have been correctly received [12]. A retransmission is triggered either by a repeated cumulative acknowledgment (fast retransmission) or by a retransmission timeout. In either case, the entire current transmission window is scanned for possible retransmissions (as in go-back-n). During a fast retransmit, only packets not already retransmitted or not correctly received are retransmitted. During a timeout, only packets not correctly received are retransmitted - thus packets retransmitted by a fast retransmit but subsequently lost are retransmitted a second time by the timeout. This scheme combines the efficiency of selective retransmission with the robustness of go-back-n retransmission. They allow a sender to quickly fill a gap in the error-control window without stalling while waiting for a timeout, or paying the overhead and complexity of a selective acknowledgment scheme.

To allow retransmissions, a source must keep a copy of the outstanding data, and the size of this buffer is limited by an error-control window. Since a receiver will also need to keep a copy of delivered data to assure in-sequence delivery of data, the error-control window size must be negotiated by the peer transport layers during call setup. At the moment, IDLInet does not implement this negotiation.

Note that the transport layer does not do checksumming to detect corruption, since this is already taken care of at the network layer.

The reliable service is simplex, so the sender sends only data, and the receiver sends only acknowledgments. This makes their state machines rather simple. (The abstraction of duplex reliable service is provided by the session layer, described in Section 8.)

7.3 Feedback Flow Control

Flow control allows an endpoint to regulate the data transmission rate to match the maximum sustainable flow by that VC in the network. The transport layer provides both open-loop and feedback flow control.

If the scheduling discipline at the switches is round-robin like, feedback flow control is based on the Packet-pair flow control scheme [14]. In this scheme, all TPDUs are sent out in back to back pairs, and the inter-acknowledgment spacing is measured to estimate the current bottleneck capacity (the bottleneck may be in the network or the receiving end-system). This time series of estimates is used to make a prediction of future capacity, and a simple predictive control scheme is used to determine the source sending rate. It has been shown that for a wide variety of scenarios, Packet-pair flow control performs nearly as well as the optimal flow control scheme, that is, a scheme that operates with infinite buffers at all bottlenecks [12]

Note that this scheme cleanly separates flow control and error control. Windows are used for error control and to size buffers at the transmitter and receiver. Flow control is used to match the source transmission rate with the current bottleneck capacity. When windows are used both for flow control and error control, packet losses will trigger a slowdown in the sending rate [20, 9], which may not be warranted by the current congestion level. This becomes important in high-speed networks where the bottleneck service rate is a rapidly changing quantity.

If the network does not support round-robin scheduling, the transport layer uses a standard dynamic-window flow control scheme similar to TCP flow control [9].

7.4 Open-loop Flow Control

The transport layer provides open-loop flow control based on leaky bucket semantics. Usually a leaky bucket is placed at the datalink or physical layer. We believe that the traffic shaping function should be as close to the application as possible to allow it to quickly get feedback about its allowed flow rate. Since there is no multiplexing in the protocol stack, we are able to move this functionality up to the transport layer. Thus, an application sending data faster than its leaky bucket rate would fill its session layer input buffer, and when this happens, either the application could be throttled by the operating system, or a signal can be handed to the application informing it about the overflow situation. This control is much easier to implement at the transport layer than at the host adaptor or a remote Network Interface Unit, as is usually the case.

The implementation of leaky bucket is trivial - for each virtual circuit, the transport layer remembers the time that the last TPDU was sent. On arrival of a session layer PDU, flow control compares the current time with that time to determine how many tokens must have arrived in the interim. This is sufficient to know how many TPDUs can be sent right away, and the earliest time that the next TPDU can be sent. The transport layer then extracts the largest number of fragments that can be sent at the current time from the TPDU chain, and hands them to the network layer. A timer is also set for the earliest time the next TPDU can be sent, based on the leaky bucket arrival rate.

8 Session layer

The session layer provides two services. First, it creates an abstraction of a duplex service over simplex transport streams. Second, it provides maps the transport layer's data movement interface and signaling layer's connection establishment interface to a BSD-socket procedure call interface.

Some applications might want to both read and write from a transmission endpoint. Since transport service is simplex, the session layer coordinates two transport endpoints to provide a duplex service abstraction. During data transfer, when an application does a `send` to the session layer, this is translated into a `t_send` to the corresponding simplex transport connection. Similarly, `recv`s are translated into a `t_recv` on the corresponding simplex connection. The signaling required to set up duplex connections is described in Section 10.

Since there is a large existing base of applications based on BSD socket semantics [16]. In order to allow easy porting of these applications to the native mode stack, the session layer translates BSD calls into transport layer and signaling layer calls. The BSD `socket` call sets up an entry point into the protocol stack to provide a handle into the read and write VCIs. The `bind` call is used by a server to register with the signaling entity. The `connect` call translates to a message sent to the signaling entity requesting a connection. The server uses the `listen` and `accept` calls to accept this incoming connection. The `read` and `write` calls map into the `t_send` and `t_recv` calls on the appropriate simplex VC.

9 Control Plane Functionality

In a network where the network and datalink layers are connectionless, such as those based on IP or CLNS, an endpoint does not need to communicate with switching points before initiating data transfer. However, in connection-oriented ATM networks, an endpoint must explicitly request switches to set up a connection before the start of data transfer. This request may be qualified with a Quality of Service specification, thus allowing switching points to reserve bandwidth and buffer resources on behalf of the connection. Conceptually, the communication between an endpoint and the network is orthogonal to the data transfer layers, and thus is in the control plane (see Figure 1). In this section, we discuss the semantics of the control plane for the native mode ATM stack.

The control plane is responsible for four major functions: establishing channels with a specified QoS, routing, OS interaction for process management and multicast support. We discuss these below.

9.1 Establishing provisioned virtual circuits

The first function of the control plane is to allow an application at an endpoint to set up a QoS provisioned virtual circuit, and to inform data transmission layers at end systems and intermediate switches about the QoS parameters. This function is provided by two entities- the QoS broker [15] at each endpoint, and the signaling entity at each endpoint and switch. The QoS broker translates from application QoS specification to network QoS specification and passes this information on to the signaling entity. The signaling entity cooperates with its peers in order to set up QoS-provisioned channels. We describe these two functions next.

9.1.1 QoS Broker

A user's view of QoS is very different from the network view of QoS. For example, the user of a video application may parameterize QoS by the size of the display (width and height) and the quality of the picture, specified using a mean opinion score (MOS). The video application works with the media parameters: the quantization level and frame rate. Finally, the network deals with yet another view of QoS: bandwidth, jitter, loss rate etc. A video application thus needs to be able to translate between three different views of QoS which we call the user QoS, media QoS and network QoS respectively. We have built a QoS broker [15] to help with this translation.

We use the interval [0,1] to represent a quality range, 1 corresponding to perfection. Given a particular media parameter (say frame rate) we assume that there is a function that maps this to a quality value. The overall picture quality is obtained by multiplying the individual quality value for each medium parameter. Conversely, we assume that given the quality value of a medium parameter, there is a function to map back to the required media value.

The forward translation process is to map user QoS to media and then network parameters. However, this is a one-to-many function, as one may achieve the same MOS with many combinations of quantization level and frame rate. As a heuristic, given a user MOS (which we scale to the [0,1] interval), we divide the quality value equally along each dimension (thus a user MOS of 0.81 implies that the quality in the frame rate parameter is 0.9 and in the quantization level parameter is 0.9). Using the mapping functions we obtain the appropriate frame rate and quantization level. This immediately gives us the network QoS requirements. These are communicated to the signaling system during call establishment.

In case a call setup request fails, the broker is informed about best possible network QoS values. It must then map this to the best possible user QoS values and determine the operating parameters to use. The reverse translation is also non-unique and requires a heuristic decision. The QoS broker tries to choose parameters so that the quality values in each dimension are roughly equal. Thus, the reduced network QoS are translated back to reductions in user perceived QoS, and these can them be implemented by the application.

9.1.2 Call Establishment

Once the application has determined the appropriate QoS using the broker, it must then set up a call. This is done by contacting the signaling entity and requesting a call to a destination and a service at that destination [22]. When the signaling entity at an endpoint or switch receives a call setup request, it uses a routing database to compute the next hop. It then gives the call a temporary identifier and contacts its peer signaling layer. It also checks the state of local resources and performs admission control. If there are insufficient resources, the call is rejected, and the application is informed. Otherwise the call request is passed from peer to peer, and each does admission control, allocating temporary identifiers if the call is accepted. This is the forward pass. At the destination, the signaling layer contacts the application that had registered to provide the service. The application is allowed to accept or reject the call, or modify the QoS parameters requested by the sender. In the reverse pass, if the application accepts the call, the QoS requested is confirmed at each intermediate point, and temporary identifiers are converted to VCIs. Finally, the client application is informed about the success of the call, and the VCI to be used. The local data transport layers are also informed about the new connections and their QoS parameters to allow them to make scheduling decisions.

When an application requests a reliable stream, the situation is somewhat more complicated. We will refer to the signaling entity that initiates the call as L, the local signaling entity, and to the signaling entity that receives the call as R. When L receives a request for a reliable connection, it passes this request to R via the intermediate peers as before. The session layer associated with L also creates a unique dummy service name and passes this to R in the call-setup packet. At the remote end, if the application accepts the call, R sends back a confirmation of the forward simplex connection and also initiates a connection to the dummy service. When L receives a connection request on the dummy service, it accepts the incoming call and returns the confirmation. This sets up the two simplex circuits. What remains is to inform the data transport layers about the forward and reverse VCIs. This is done by entering the two VCIs in the appropriate protocol control blocks.

If an application asks for a duplex stream (either unreliable or reliable), a similar set of actions is performed by the session layer. The two corresponding transport connections are managed by the session layer, and are unaware of each others existence.

9.2 Routing

If an endpoint is connected to multiple switches, it needs to know which switch to use for contacting a given destination. To solve this problem, the signaling layer periodically exchanges linkstate update packets on best-effort channels with all its peers [17]. These packets are forwarded by each receiving signaling entity on every outgoing link, thus flooding the network. By combining the linkstate information received through flooding, a signaling entity can build its own copy of network topology, on which a standard shortest-path algorithm is run in order to construct a routing table.

If a signaling entity does not receive updates over a link for a long period of time, it marks that peer dead, and the revised link state is broadcast, triggering an update of the distributed routing database. Thus, future calls will be routed around the failed link. At the moment, we do not perform failure recovery for existing calls.

9.3 OS interaction

The signaling entity needs to interact with the operating system in order to clean up after applications that terminate without releasing network resources (i.e. without initiating circuit teardown) [22]. If an application (either client or server) terminates abnormally, network resources can be locked up. To prevent this, the operating system must inform the signaling entity when an application terminates. On receiving a termination indication from the operating system, the VC is torn down and resources are released. When the signaling entity receives a close indication from its peer, the session layer marks the VC as closed, so that on a subsequent read or write from that VC, an error indication is returned. The IDLInet implementation of these actions is discussed in [22].

9.4 Multicast

This is the fourth major function provided by the control plane. For implementing multicast we use the shared tree approach. We require each multicast virtual circuit to be bidirectional. Each switch controller maintains a list of virtual circuits that are in the same multicast group and when a message comes on any one of these virtual circuits it is sent out by the switch on all associated VCs other than the virtual circuit on which the message arrived. We use single master approach where all members who wish to join the group send message to the group master, which uses knowledge of the network topology (from routing) to decide a multicast tree. Signaling messages are then sent by the master to the appropriate switches for the establishment of this tree. All VCs in a multicast tree must have the same QoS specification. Since the single master coordinates joins and leaves, these are easy to implement.

A centralized multicast master does not scale well, and in future work, we plan to explore strategies for distributing this functionality.

9.5 IDLInet implementation of the control plane

IDLInet supports the Real-time Channel Administration Protocol from UC Berkeley [1], and the SPANS protocol from Fore Systems. RCAP has been extended to allow duplex channels, multicast, a QoS manager and QoS renegotiation. SPANS has been extended to allow duplex channels.

Where possible (i.e when communicating to a switch controller running IDLInet), the signaling layer uses the duplex reliable service of the session layer to communicate with its peers (we do not, at the moment, implement the ITU/TSS SSCOP transport layer). Since this service requires the peer transport endpoints to exchange messages for a three way handshake during initialization, there is a potential bootstrapping problem, where signaling needs the transport layer to provide reliability, and the transport layer needs signaling in order find out which VCI to use for the forward and reverse connections. To solve this, at the time of initialization, the signaling layer loads the protocol status block with both the forward and reverse VCIs initialized with the values discussed above. The transport layer simply uses these values in doing its three way handshake. Subsequently, signaling peers can communicate through these signaling channels using `t_send` and `t_recv`. The result of this is that all retransmissions are done by the transport layer and the signaling entity can assume in order and guaranteed delivery.

Our experience in writing signaling implementations over both unreliable and reliable channels is that signaling over reliable links is far easier to implement and debug. Unfortunately, many current signaling standards run over reliable transport layers, but still repeat all the work done in the transport layer.

The procedure interfaces support by the control plane for signaling are as follows

```
   /* Port register/unregister */
r-register(uint port, uint queue-length);

    /* call establishment */
 r-receive-request(uint port, Param-Block *param, rcapAddress *source, uint *conn-id);
 r-establish-request(Param-Block *param, uint *conn-id,uint source-port);
 r-establish-confirm(uint conn-id);
 r-establish-return(uint conn-id, uint result,u-int reasonCode,rcapUser Control *control);

   /* Call close/teardown */
rcap-close-request(uint conn-id, uint reasonCode);
```

Typically, the server would register itself using `rcap-register`, and then wait for calls using `rcap-receive-request`. The client would establish a new connection using `rcap-establish-request`, and periodically check status of the call using `rcap-establish-confirm` (a spin lock). The client's call is accepted by the server using `rcap-establish-return`. Finally, `rcap-close-request` is called to tear the connection down.

10 Current Status

A first version of the IDLInet stack on the REAL simulator and DOS PCs is available for public distribution. This distribution includes all the features discussed in this paper except OS support, which has not yet been incorporated into the simulator version of the stack. Table 1 shows the size of each component. Note that the entire stack is only around 7500 lines of C code, including comments. The data transfer path (the datalink, AAL, transport and session layers) adds up to about 45% of the code, and about 27% is devoted to signaling. Using 80386 based MS-DOS PCs, we have measured user-to-user data transfer rates of 1.5 Mbps over a 10 Mbps Ethernet.

Layer	Lines	Percent
Datalink	708	9.2
AAL	243	3.2
Transport	1521	19.7
Session	995	12.9
Scheduler	387	5.0
Memory management	543	7.0
Link state routing	458	5.9
Signalling	2072	26.9
Multicast	784	10.2
Total	7711	100.0

Table 1: Number of lines in each layer of the protocol stack.

A port of the stack to the Plan 9 operating system [18, 19] is substantially complete. This port contains the OS support functionality, but not yet the transport layer. With a null transport layer, we have achieved a user-to-user data rate of 65 Mbps between two 80486 PCs running the Plan 9 operating system and connected by Fore Systems' HPA-200 cards and an ASX-100 switch. We can establish and tear down connections using SPANS signaling. We do not expect there to be any performance degradation with the addition of the transport layer since the bottleneck is the EISA bus on the PC, and we already account for the data copies in this test. [1]

Our experience with porting IDLInet to other platforms has been very positive. The stack has been designed so that machine-dependent and machine-independent subsystems are clearly separated. Thus, a port only involves rewriting the machine-dependent subsystems. For example, the port to DOS took two weeks, and the port to Plan 9, once the device driver was written, has taken a little over a month (and is still in progress).

Another gain from our approach to protocol stack development is that new features can be extensively tested and debugged *before* they are introduced into an OS kernel. This has proved to be extremely useful for debugging the transport layer semantics.

11 Contributions of our work

IDLInet was designed using a few principles: minimality, no logical multiplexing, and ATM specificity. The criterion of minimality led to a design from first principles, cleanly separating the functionality of each layer and each service. Our service interface allows applications to mix and match from a small number of orthogonal services. By eliminating logical multiplexing, we present a direct path from an application to its associated ATM VC. This path can be customized per-VC using the service interface. Targeting to ATM allows the stack to be easily integrated into future ATM networks. For example, the AAL does not have to

[1] The final version of this paper will have performance figures for the transport layer.

adapt to the network layer, since it *is* the network layer. Further, the signaling layer allows the endpoint to explicitly request an ATM connection from the network.

There are several novel aspects to our work. First, all the layers of the stack were designed at the same time. This allowed us to minimize data copying overheads and keep the design simple. We have optimized the distribution of tasks between layers to maximize parallelism and eliminate repeated functionality. Second, the task-based non-multiplexed implementation makes it easy to provide per-VC QoS. The central scheduler can examine all the QoS requirements in deciding which task to run next. It can also coordinate resource allocation with the end-system CPU scheduler. Third, the stack was first implemented on a simulator, then moved to a real system. This allowed us to understand the design and debug it before committing it to the field. Improvements to the protocols are tested in the simulator before they enter the kernel. This speeds up the design and implementation process. Fourth, since this is a new design, with no need for backward compatibility with existing implementations, we have been able to incorporate the best available flow and error control mechanisms into the stack. Finally, the stack's external interfaces are compatible with the existing TCP/IP and BSD socket interface. Thus, existing application programs can be migrated to the IDLInet stack with little effort.

12 Related work

It is useful to contrast the semantics of IDLInet with standard BSD socket/TCP/IP/Unix device driver semantics. At the socket layer the semantics are similar (including duplex, error-controlled and flow-controlled connections), except that sockets are bound to VCIs instead of TCP ports. This allows easy portability of existing code. At the transport layer, TCP's error control uses similar mechanisms. However, the flow control mechanisms differ considerably since we propose rate based flow control independent of the error control window size. Further, unlike TCP, data checksumming is done at the network layer.

Unlike IP, the network layer does not do fragmentation and reassembly, instead, this is done at the transport layer. Also, the network layer does not do routing, which is done by the signaling layer. In contrast, a considerable complexity in IP is in implementing fragmentation and routing correctly. By removing this burden from the network layer, it becomes faster to implement and execute. Finally, the network layer does not multiplex multiple transport connections (unlike IP), allowing the QoS from the ATM layer to be visible at the higher layers. Thus, we believe that our protocol stack semantics are more tuned to ATM networks than stock TCP/IP.

Our work differs from IP-over-ATM in many ways. In the IP-over-ATM approach, the application sees only the IP interface, which does not provide any QoS guarantees. Thus, any guarantees available from the ATM network are hidden. Second, the IP-over-ATM subsystem has to make signaling requests on behalf of the application, which adds considerable complexity to the kernel. Third, IP routing assumes a broadcast medium in the local area, which is critical for the ARP protocol. IP-over-ATM has to spend a lot of effort emulating this over the point to point ATM network. By using a native mode ATM stack, all these problems are automatically eliminated.

Our work is closely related to that of Campbell et al [3] who have proposed a multimedia enhanced transport service and a Quality of Service Protocol Architecture [2]. As in our stack, they have placed flow regulation at the transport layer, and have no logical multiplexing of streams. However, they have decomposed the service interface into guaranteed-performance and best-effort flows. This hides the orthogonal aspects of flow control, error control, QoS specification and simplex versus duplex circuits that our stack explicitly presents. As a consequence, their interface does not allow for some combinations that may prove to be useful, such as non-error controlled but feedback flow controlled flows, which is an alternative way to carry Variable Bit Rate video traffic [11]. Second, their stack provides a complex API - for example, applications are expected to provide dummy upcalls for computing the average time taken for a user task. We think that this complexity can result in poor performance. Finally, their choice of a QoS specification seems premature since much is unknown about what is really needed. session layer semantics.

Greaves and McAuley at Cambridge [8] have also implemented a non-multiplexed protocol stack over ATM. While interesting, their work has mainly been at the network layer (MSNL), and, to the best of our knowledge, do not extend to transport and session layer semantics. In contrast, IDLInet provides a fairly complete stack that is compatible at the user-interface level with BSD sockets and TCP/IP.

There have been numerous attempts to design transport layer protocols in the past. Of these, one of the more complete attempts is the TP++ project [5]. TP++ has several interesting features such as forward error correction, timer-based connections and congestion control based on backpressure. While per-VCI backpressure can require considerable feedback from switches, which is hard to legislate in a multi-vendor framework, several of their ideas are orthogonal to our innovations, and we plan to consider them in future work.

13 Future Work

We are still building the stack. The first version of the stack has been released, but extensions of the work for better flow control and multicast support are needed. One area that has attracted much recent attention is in defining the semantics of CPU scheduling for end-systems that support per-VCI QoS [21, 10]. As we have discussed, the IDLInet stack provides a good testbed for research in this area. We also plan to investigate the semantics of fault recovery in more detail, particularly with reference to re-routing.

14 Acknowledgments

We would like to acknowledge the contributions of R.N. Moorthy and R.P. Rustagi in the detailed design of the stack and in coordinating its first implementation. This implementation was revised and extended by R. Ahuja, J. Sarbadhikari and S. Vijay. The first draft of this manuscript benefited from comments by C.R. Kalmanek, J. Crowcroft, and P.P. Mishra. Our thanks to all of them.

References

[1] A. Banerjea and B. Mah. The real-time channel administration protocol. *Proc. Second International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 160–170, November 1991.

[2] A. Campbell, G. Coulson, and D. Hutchison. A quality of service architecture. *ACM Computer Communications Review*, April 1994.

[3] 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.

[4] CCITT. Draft specification i.363. *Available for anonymous FTP from datanet.tele.fi*, 1993.

[5] D.C. Feldmeier. An overview of the tp++ transport protocol project. In A. Tantawy, editor, *High Performance Networks-Frontiers and Experience*, pages 157–176. Kluwer Academic Publishers, Boston, MA, 1993.

[6] D.C. Feldmeier. Multiplexing issues in communication system design. *Proc. ACM Sigcomm 1990*, pages 209–219, October 1990.

[7] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE J. on Selected Areas in Communications*, April 1990.

[8] D.J. Greaves and D. McAuley. Atm network services for workstations. *I.95 deliverable documents*, 1993.

[9] V. Jacobson. Congestion avoidance and control. *Proc. ACM SigComm 1988*, pages 314–329, August 1988.

[10] M.B. Jones. Adaptive real-time resource management supporting modular composition of digital multimedia services. *Proc. 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[11] H. Kanakia, P.P. Mishra, and A. Reibman. An adaptive congestion control scheme for real-time packet video transport. *Proc. ACM SigComm*, 1993.

[12] S. Keshav. Packet-pair flow control. *Submitted to ACM Trans. on Networking*, 1994.

[13] S. Keshav. Real : A network simulator. *CSD TR 88/472 , UC Berkeley*, December 1988.

[14] S. Keshav. A control-theoretic approach to flow control. *Proc. ACM SigComm 1991*, September 1991.

[15] J.M. Smith K.Nahrstedt. The qos broker. *IEEE Multimedia*, 1(1), Spring 1995.

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

[17] R. Perlman. *Interconnections*. Addison-Wesley, Reading, MA, 1992.

[18] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A distributed system. In *EurOpen, Proceedings of the Fall 1991 Conference*, Tromso, Norway, May 1991.

[19] D. Presotto and P Winterbottom. The organization of networks in plan 9. In *Usenix, Winter 1993 Conference Proceedings*, pages 271–280, San Diego, CA, January 1993.

[20] K K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks. *TOCS*, 8(2):158–181, May 1990.

[21] 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.

[22] R. Sharma and S. Keshav. Signaling and operating system support for native-mode atm applications. *Proc. ACM Sigcomm 1994*, 1994.

[23] C.A. Sunshine and Y. Dalal. Connection management in transport protocols. *Computer Networks*, 2:454–473, 1978.