Experience and Results from the Implementation of an ATM Socket Family

Richard Black and Simon Crosby
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge, CB2 3QG, U.K.

27th October 1993

Abstract

This paper describes the implementation of an ATM protocol stack as a protocol family within a 4.3 BSD derived Unix. A novel approach to the implementation of the management and control functions for the ATM protocol stack has been adopted. The data path is implemented within the kernel but all control and management functions are implemented by a user space daemon. An encapsulation of IP on the ATM protocol is provided by means of a logical IP interface. The mapping of IP addresses to ATM addresses is performed by the user space daemon.

1 Introduction

The Cambridge ATM environment is heterogeneous, and includes networks such as the Cambridge Fast Ring (CFR) [8], Cambridge Backbone Ring (CBN) [6], and Fairisle [2]. The CFR and CBN are respectively 50Mb/s and 500Mb/s slotted rings which predate current ATM standards and use a 36 octet cell size, whereas Fairisle is a switch-based ATM network with a standard B-ISDN and ATM Forum 53 octet cell size. To permit easy interconnection of ATM networks and services the ATM protocol is also carried across the Ethernet in the form of "Fat Cells", in which an Ethernet frame is filled with ATM cells in a format which enables rapid cell forwarding at Ethernet-ATM gateways. The software environment is also heterogeneous and supports Unix and experimental micro-kernels. To this environment will be added standards compliant ATM equipment purchased from third parties, and in the near future the network will be connected to the national wide area ATM network provided as part of Super-JANET.

To facilitate the management of this complex system amid constantly changing protocol standards, and to permit both experimental and service use of the network, we decided to implement the management and control functions for the ATM network in a single body of code which could be ported to all of our operating platforms, including 64 bit architectures. This was accomplished by implementing the code as a user space daemon. This paper describes the features implemented in the 4.3 BSD based Ultrix kernel to support the management and control functions in user space, and discusses the design and implementation of the user space manager.

The management code can be extended to accommodate emerging international standards for ATM networks, such as the ATM Forum signalling protocol and the IETF suggested standard for address resolution for IP on ATM [10]. Such extensions can be carried out without modifying the kernel, as all management and control functions exist in user space.

Problems resulting from the asynchronous nature of the control interface are described. We identify several shortcomings of the BSD socket interface for the ATM protocol domain, including the difficulty of describing parameters required for the ATM protocol such as Quality of Service (QoS) specifications and adaptation layer requirements.

2 The ATM protocol

In the ATM protocol domain the socket interface provides an application with direct access to an ATM virtual connection. The ATM protocol is offered as a new address family, `AF_ATM`. Each Protocol Data Unit (PDU) to be transmitted over the connection is handed directly to the ATM adaptation layer for segmentation into ATM cells and subsequent transmission. Similarly, received PDUs are handed up to the socket layer by the adaptation layer on completion of reassembly. Draft standards for ATM UNI signalling [3] provide a mechanism for the selection of the ATM adaptation layer (AAL) and Quality of Service (QoS) to be used for an ATM connection. However, there is no mechanism in the socket interface to permit an application to specify these parameters to the `connect()` system call in a clean manner. Consequently, for this implementation all ATM connections make use of AAL5, and a default "best-effort" QoS specification is used.

At the ATM layer each connection is represented by an *association* record, and each socket is associated with a single association. In the case where the connection is local, no association record is required and each of the two sockets corresponding to the endpoints of the connection contains a reference to the other. This is similar to the local connection case in `TCP`. During connection setup the user space manager instructs the ATM protocol code in the kernel to build the data path for the socket. When the connection is complete the protocol control block (PCB) associated with the socket contains a reference to the ATM association record for the connection. Final authority as to the state of a connection lies with the manager. It can unilaterally decide to terminate a connection at any time.

Additionally the ATM code provides logical interfaces to the IP code within the system and will set up tunnels over the ATM network to carry IP traffic to its destination. In this case the upper layer for the ATM protocol is a tunnelling / logical interface engine rather than the socket code. Figure 1 shows a diagrammatic overview of the system components.

3 Communication with the Manager

To permit the ATM connection manager to be implemented in user space, a mechanism was required to permit communication between the kernel socket layer and the manager, and between the ATM layer and the manager. Two possible solutions were identified, namely the provision of a special device in `/dev` or a "magic" control socket type. We chose to implement the latter option, adding a socket type `SOCK_RAW` to `AF_ATM` which can have only one instance per machine. The reasons for this choice are:

- For concurrency reasons it was preferable for the manager to interface directly with the networking code rather than via the file system. This would also make implementations on kernel threads, such as on OSF/1, easier. This is particularly important because most of the information for the manager is generated as a result of calls from the socket layer.
- The user space manager is responsible for the transmission and reception of ATM signalling messages. Since these must be presented to the ATM layer in the form of queues of `mbufs` and the mechanism for the encapsulation of data in `mbufs` is already provided by the socket layer, it is preferable to transmit and receive signalling via the socket interface rather than via the file system. The fundamentally asynchronous nature of interaction between the kernel and the daemon maps well onto the asynchronous nature of socket communication.

The control socket is used to exchange three types of information: socket layer requests, ATM layer requests and ATM signalling. Socket layer requests to the manager include, for example, requests to `bind()` an ATM address to a socket, `connect()` a socket to a destination ATM address, and `close()` a connection. Responses from the manager indicate the status of outgoing connections and include notification of incoming connection requests. The manager uses the

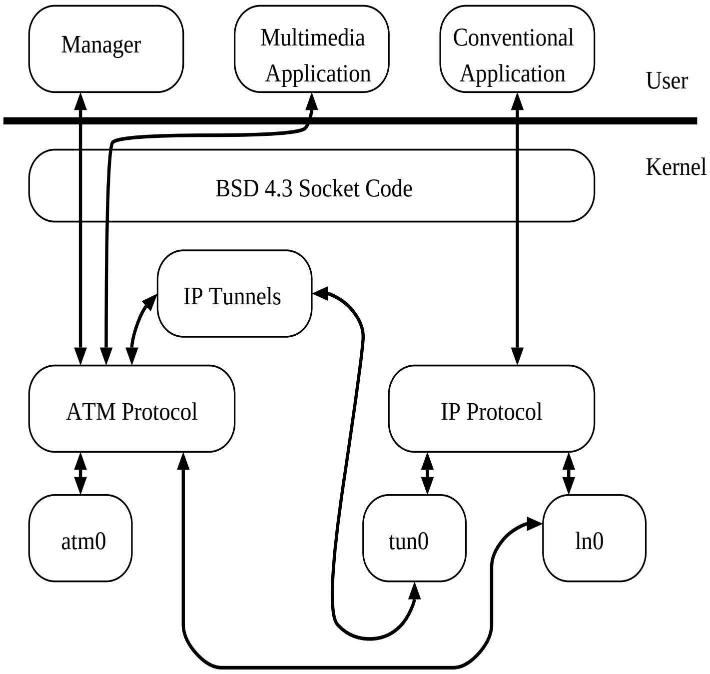

Figure 1: Overview of System Components

control socket to issue commands to the ATM layer to build an association for each connection, provide it with a virtual circuit identifier (VCI), and monitor its status. In the control plane the manager transmits and receives ATM signalling messages via the control socket. Implementation of the ATM control plane in user space has the advantage that modification of the signalling protocol to comply with the latest ATM signalling standards involves only the user space daemon and not the kernel ATM layer code. In addition, development can be aided by the use of programming tools such as debuggers.

Messages exchanged between the kernel and the manager are normally in the form of fixed length control messages. Some of these message blocks, however, may be followed by ATM signalling messages for transmission or which have been received.

A few management operations cannot be performed asynchronously and are transferred using the `ioctl()` system call. In this case the kernel can synchronously modify the block to return the required response.

4 Kernel Modifications

Two key issues were identified which dictated the design of the kernel code. First, although the user space manager is a part of the operating system, its failures should be localised. If the manager is not running, or crashes, the kernel code should remain consistent and stable. Existing sockets should be closed and additional requests at the socket layer for `AF_ATM` should be denied. Malfunctions on the part of the manager should be incapable of crashing the kernel.

Secondly, both the socket level code and the manager will asynchronously issue requests pertaining to sockets in the ATM protocol domain. Because they operate concurrently it is impossible to ensure that both have a consistent view of the state of all ATM connections at any one time. Special measures must be taken to ensure that errors resulting from this inconsistency are avoided.

To solve these two problems the kernel protocol code keeps in the PCB for each socket its own notion of the state of the socket[1]. For example the protocol code will call `soisconnecting()` and even `soisconnected()` to cause the socket code to perform the required operations even when the protocol is not necessarily either connecting or connected. Likewise the PCB continues to exist even when the socket code detaches the socket until the manager has acknowledged that it has finished dealing with the signalling for that PCB.

The kernel code and the connection manager identify individual ATM connections by means of their associated PCBs. To the manager the PCB is an opaque identifier which uniquely identifies an ATM connection. Within the kernel PCBs are kept in an open hash table which is used to verify every PCB reference passed to the kernel by the manager. Each PCB contains a pointer to a set of methods for dealing with the higher level protocol. It also contains methods which implement the transmission path and permit the socket layer to destroy the data path, as well as an opaque value which, in the case of a local connection, is a pointer to the PCB for the peer socket, and in the case of a remote connection, is a pointer to the association structure at the ATM layer.

The association structure contains various control fields such as the VCI and a pointer to the receive routine. For normal connections the receive routine performs AAL processing on the received cells before passing up AAL Service Data Units (SDUs) to the socket layer. For each ATM network interface a special association exists which is used for meta-signalling. For these associations the receive routine passes the entire payload of each meta-signalling cell up to the socket layer for forwarding to the manager via the control socket. The manager transmits meta-signalling via the control socket.

5 Kernel Implementation Problems

In the following sections we address several of the implementation problems which were encountered, some of which resulted directly from the design choices described above.

5.1 Socket Addresses

The BSD 4.3 socket code requires the implementation of the `bind()`, `getsockname()`, and `getpeername()` system calls for all protocols to be non-blocking. Many applications perform a `getsockname()` immediately after a `bind()` in order to publish or export their service address. This unfortunately means that the implementation of the ATM service address allocator must remain within the kernel. The addresses allocated are reported to the user space manager. Whilst it would have been preferable to locate this function with the rest of the control plane functions in the manager, this restriction is a minor inconvenience.

5.2 Interrupt levels

BSD 4.3 is structured such that all interaction with the socket level code must be performed while operating at `splnet` whereas device interrupts and the internals of the `mbuf` system operate at `splimp`. BSD 4.3 assumes that there will always be a queue structure between the device driver and the protocol handler. For this reason the ATM device driver cannot manipulate the association data structures, for example during reassembly. Instead it enqueues complete blocks of data on a per VCI basis for the ATM protocol layer. The ATM protocol layer runs at `splnet` and removes blocks from the queue asynchronously, adding them to the receive buffer on the appropriate socket. This additional layering represents a significant bottleneck in the protocol stack and significantly slows down the implementation.

[1]or IP tunnel

5.3 Accepting Semantics

The ATM protocol implemented on the Cambridge networks, MSNA [12], allows the passive end of a connection to consider both the address of the peer and the quality of service requested before deciding to accept the connection. This is directly at odds with the BSD 4.3 semantics which require that a connection be declared "soisconnected" before the receiver is informed of the connection. In our implementation an incoming connection request is forwarded to the manager which in turn presents the connect request to the kernel ATM layer. On receipt of the connect request the ATM layer declares to the socket layer that the connection is complete (soisconnected), and prevents subsequent transmission on the socket associated with the connection by manipulation of the socket transmit buffer. It then instructs the manager to accept the connection. Once the manager has accepted the connection and built the data path, it informs the ATM layer that the connection is complete and transmission and reception are enabled.

Although a Unix server is still prevented from vetting its peer before accepting a connection, our implementation preserves the expected ATM protocol semantics. The inability of the socket interface to permit an application to vet its peer and to accept connections conditional on the availability of sufficient resources to meet their QoS requirements should be addressed to provide the full flexibility of ATM to applications.

6 IP Logical Interface

Within the kernel a logical IP interface is provided which makes use of ATM connections as "tunnels" – each tunnel is a logical link in the IP network. The IP tunnelling code is responsible for generating requests for the establishment of ATM connections and for the transfer of IP datagrams over these connections. It makes direct use of the protocol code and provides the PCB for each of its tunnels with methods which permit it to monitor the state of the connection.

The tunnelling code sets up an ATM connection to every active IP destination, which are torn down after a period of inactivity. It would be preferable to use one ATM connection for each TCP connection or UDP session, however this would require extensive modification of the BSD 4.3 IP code[2]. An alternative, which has not yet been implemented, would be to identify the higher layer TCP and UDP associations within the ATM driver in a similar manner to that used in SLIP header compression [11].

6.1 IP Address Resolution

To set up an ATM connection for tunnelling IP packets, the IP tunnelling code must somehow resolve the IP address to form an ATM address. It was a particular choice of our design that this mapping would be performed by the manager, and so we permit `AF_INET` addresses to be specified as the destination for an ATM connection. The manager performs the necessary mapping.

Address mapping schemes used previously in the implementation of IP over X.25 use either a fixed algorithm [1] (which requires a private X.25 network), or a manually administered configuration file read at initialisation time such as [13]. In contrast in the scheme we have adopted, any mapping is possible and the management daemon is free to solicit or receive updates or enhancements to its knowledge at any time. For example at boot time the manager may know a single IP to ATM address mapping for its default router. The router could then inform it of optimisations that could be made based on traffic analysis.

[2]This approach has been followed in our local micro-kernel IP implementation

6.2 IP MTUs

Recent debate in the internet community has addressed the problem of choosing a suitable Maximum Transfer Unit (MTU) for the encapsulation of IP on ATM. The MTU determines the maximum size of an IP datagram which will be transmitted on the ATM network, and its choice is influenced by several factors. In order to exploit the high bandwidth of the ATM network the MTU should be as large as possible, for example 64KB. On the other hand, some routers, switches and hosts may be unable to support such a large MTU, and fragmentation of IP datagrams into smaller units will result. In addition, if a single ATM connection is used to carry all IP traffic between two hosts then performance of higher layer protocols may be degraded if the MTU is too large because of the delay in processing each large packet as a single unit. If one ATM connection is used per TCP connection then this can be avoided. In a LAN environment a single IP datagram is likely to arrive as a large burst of ATM cells. Care should be taken to ensure that buffer space in the receive interface does not overflow, taking particular account of the interrupt response time of the system under realistic loads.

Clearly some mechanism for the negotiation of the MTU per ATM connection should be provided to permit individual hosts to achieve maximum performance from the ATM network. Our ATM signalling protocol permits the negotiation of the maximum sizes of the encapsulated forward and return PDUs, thereby permitting the MTU to be negotiated per tunnel. In our current implementation we provide four logical IP interfaces, each with a different MTU. This will permit us to experiment with different MTU sizes without modifying the generic IP code. The IP routeing tables on each host are configured to use the IP interface with the best MTU for the attached ATM network.

7 Design of the User Space Manager

The user space manager implements the control and management planes for the ATM protocol. It is responsible for ATM network signalling and communicates with the ATM protocol family code in the kernel. It also performs mapping of IP to ATM addresses for the IP tunnelling code.

At the socket layer requests issued to the manager include `bind()`, `connect()`, `listen()`, `accept()` and `close()`. Each request specifies a unique identifier (the PCB pointer) which identifies the ATM connection to which it refers. Requests are received asynchronously on the control socket. For each call reference the manager maintains a record of the state of the connection, the association fields such as the VCI, and ATM addressing information. The manager issues requests to the protocol layer to process incoming connection requests and to inform the socket layer of the state of each connection. The manager is responsible for building the data path for each connection at the ATM layer. It issues requests via the control socket to build associations, assign VCIs and monitor the status of each connection. Signalling traffic is transmitted on the appropriate ATM interface by the manager to request the setup of connections on behalf of the protocol layer.

Since the control socket delivers requests asynchronously and since the manager needs to be able to process several requests (from all layers) simultaneously, it is implemented as a multi-threaded process. The implementation uses the POSIX Pthreads package [9] to implement several threads of control to manage the following concurrent tasks:

- receipt and transmission of socket layer requests,
- management of the ATM layer and its associated network interfaces,
- receipt and transmission of ATM signalling, and
- generation of timeouts and performing consistency checks.

These independent threads are scheduled by the Pthreads package, which also implements concurrency control primitives for shared data. As the performance measurements in section 8 show, the concurrency control primitives have a high associated cost, and result in some inefficiency.

An important consideration when designing the manager was the potential inconsistency between the manager and socket layer views of the state of each connection. For example, an application could issue a `listen()` and then immediately `close()` the socket. Due to the asynchronous nature of the control socket, the manager might attempt to complete an incoming connection to the listening socket after the application had already issued the `close()`. To prevent this, the kernel code checks all instructions issued by the manager to ensure that they pertain to valid PCBs. Similarly, the manager will destroy any socket for which the kernel issues an invalid request. Final authority as to the state of a connection lies with the manager.

8 Performance

In this section we present performance measurements for the implementation of the ATM protocol. The performance of the control path was investigated by measuring the time taken to set up connections over the ATM network and the Ethernet. In addition, the manager was profiled to determine which parts of the code were critical in determining its performance.

8.1 Data Path

The implementation of the data path in the kernel is crucial to the performance of the ATM protocol. Comparison of the performance achievable with that for the micro-kernel demonstrate that it is limited by the current structure of the BSD 4.3 protocol stack. Nevertheless, high data transfer rates can be achieved. Performance measurements using a very simple cell based interface with little buffering [7] indicate that a raw ATM throughput of 20Mb/s can be achieved between two DS5000/25 hosts. Using an MTU of 2.5 KB a throughput of 15Mb/s for a TCP connection over IP on ATM was achieved.

8.2 Control Path

The performance of the control path is affected by several factors: scheduling of the manager as a user space process, concurrency within the manager, and communication between the manager and the kernel. Table 1 gives typical connection setup and teardown times over the Ethernet and a direct ATM link between two lightly loaded DS5000/25 hosts running the kernel and user space manager, and for local IPC on a single host. Measurements are also given when the hosts are connected through an ATM switch. The performance measurements were made using `etp` [4].

Network	Setup (ms)	Teardown (ms)
Ethernet	21.7	8.4
ATM	20.4	8.1
Local	21.8	3.2
ATM (switched)	42.1	8.7

Table 1: Connection setup and teardown times

The measurements indicate that our implementation performs worse than an earlier implementation of a (simpler) version of the control plane within the Unix kernel, for which average connection setup times of 3.5 ms were measured between two Unix hosts over the Ethernet during periods of low network activity [5]. Although it would be unreasonable to expect our implementation to perform as well as a more mature, though less functional, kernel implementation, we might have hoped for better results. The results can be only partially explained by competition between the client and connection manager (and in the local case, the server), all of which were running at the same, default, user priority level. To determine why connection setup took longer than expected, we profiled the connection manager.

The profile results showed that the manager typically spends just over half of its active time (i.e. time not blocked in the `select()` call) executing code in the Pthreads library. In all of our measurements, the 10 most frequently called functions were in the Pthreads library. Most of these functions are concerned with thread context switching and concurrency control. Table 2 lists these functions, their associated frequencies, and the percentage of the total active time of the manager for which they account, for a typical profile during which 20 connections were set up. Clearly the concurrency control primitives have a high cost. The manager could be made more efficient by implementing it as a single threaded process. On an operating system platform which supports lightweight threads, such as OSF/1, it would be hoped that the multi-threaded design of the manager would not impact its performance, and the connection setup time would be proportionately reduced. We believe that this, in conjunction with some further optimisations identified, would enable a reduction of approximately 50 % in the connection setup time, reducing it to the order of 10 ms.

Function	No. of calls	% CPU time
exc_push_ctx	3037	6.411
_setjmp	3037	2.046
exc_pop_ctx	3019	0.716
cma__int_mutex_unblock	948	0.471
cma__int_mutex_block	948	0.426
cma_init	922	0.178
pthread_mutex_unlock	831	0.705
pthread_mutex_lock	829	0.672
cma__queue_dequeue	829	0.605
cma__attempt_delivery	696	0.410
Top 10 calls combined		12.64
Total for Pthreads		52.72
Manager		47.28

Table 2: Most frequently called functions

8.3 Influence of System Load

In the following sets of results, the influence of system load on connection setup time is studied. Three sets of measurements were taken. In each set the system load was varied by running a number of CPU-bound processes in competition with the manager. The competing processes each consisted of a shell pipeline[3]which required both user space and kernel processing. The competing pipelines were run at the default user priority of 0. The number of interfering pipelines was varied from 0 to 3. For a given system load the performance of the manager was measured by profiling a client which repeatedly set up a connection, exchanged a single message with a remote server, and then tore down the connection. In each experiment, the client set up 100 connections with a random interval between each. In all of the experiments, connections were set up from a DS5000/25 over the ATM network, via 2 Fairisle ATM switch ports, to a second

[3]`while true; do a=‘echo $PATH | grep wanda | sed -e "s/wanda/foo/g"‘; echo $a >/tmp/foo.$$; done`

DS5000/25 also running our code. Both workstations were fitted with a simple, cell-based ATM interface [7].

In the first set of measurements, the manager was run at a priority of 0. As expected, the connection setup time degraded as the system load increased. Figure 2 shows an initial sharp increase in connection setup time with 1 competing pipeline, followed by a linear increase with increasing load. In the second set, the priority of the manager was increased to -1. As can be seen from the graph, the connection setup time initially increased rapidly with increasing load, but the rate of increase slowed thereafter. In the third set of experiments the priority of the manager was set at -6, higher than that of the automount daemon. This served to stabilise the performance, in spite of an initial sharp increase with a single competing pipeline.

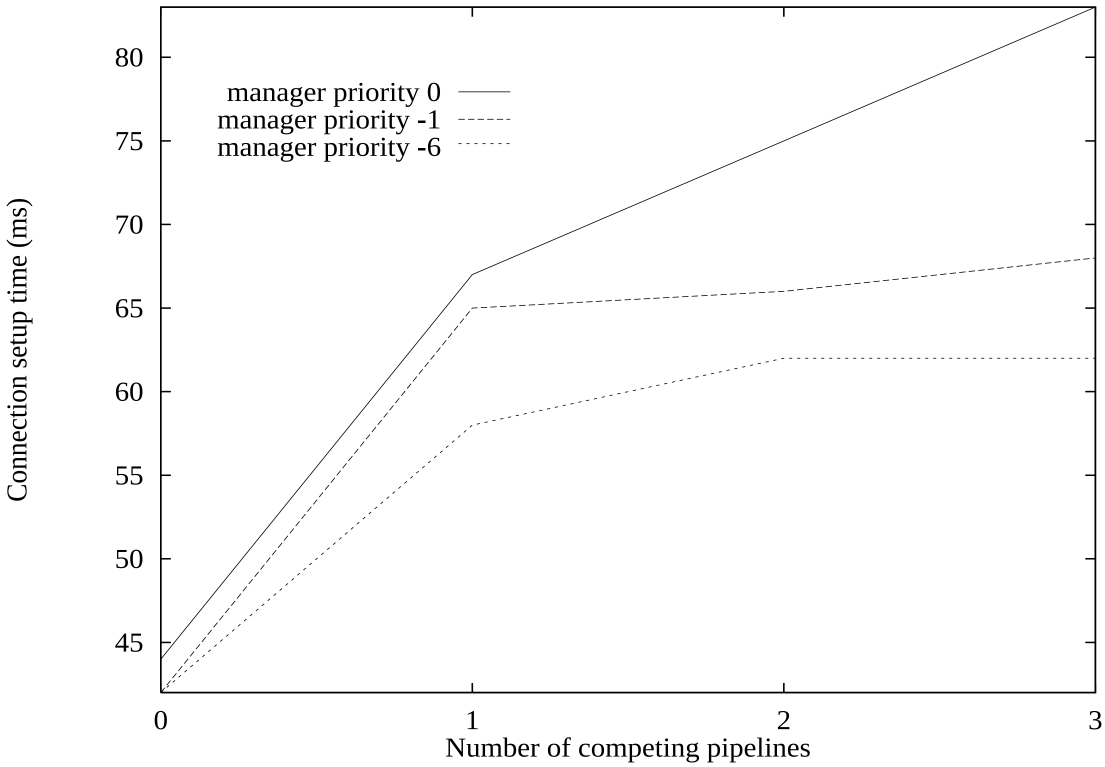

Figure 2: Influence of system load on mean connection setup time

The results suggest that the manager should be run at a high priority, to ensure that an increase in system load does not affect its performance excessively.

8.4 Code Size

An important consequence of our design is that the complexity of the system, in terms of code size, has been significantly reduced. The connection manager compiles to approximately 0.5 MB, almost all of which is due to the Pthreads library. The size of the kernel ATM code has been reduced by about 30 % to a mere 27 KB of text segment including the ATM device driver.

9 Conclusion

This paper has described the design and implementation of an ATM protocol stack within the Unix kernel. A novel approach to the implementation of the control plane has been adopted: the management and control functions are located in a user space daemon. Communication between the kernel networking code and the daemon takes place via a special control socket. The user space daemon is responsible for signalling, socket layer management and control of the ATM protocol code. It is implemented as a multi-threaded process using the Pthreads package.

We conclude that our design is both viable and beneficial. The performance costs are low, and mostly result from a poor Pthreads implementation on Ultrix. The benefits are that the complex and fluid control plane need not be buried in the kernel, and that the resulting code is more portable.

References

[1] A. Malis, D. Robinson, R. Ullman. Multiprotocol Interconnect on X.25 and ISDN in the Packet Mode. RFC-1356, August 1992.

[2] ACM. *Fairisle: An ATM Network for the Local Area*, Computer Communicatations Review, SIGCOMM, September 1991.

[3] ATM Forum. *ATM User-Network Interface Specification Version 2.1*, 1993.

[4] M. Burrows. A Prototype Elapsed Time Profiler for Alpha OSF and MIPS Ultrix. Technical report, Digital Equipment Corporation, Systems Research Center, 1993. Yet to appear.

[5] M. J. Dixon. System Support for Multi-Service Traffic. Technical Report 245, Cambridge University Computer Laboratory, January 1992. Ph.D. dissertation.

[6] European Fibre Optic Conference. *The Cambridge Backbone Network*, Amsterdam, June 1988.

[7] L. French, D. Greaves, and D. McAuley. Private ATM Networks and Protocol and Interface for ATM LANs. Technical Report 258, Computer Laboratory, May 1992.

[8] A. Hopper and R. M. Needham. The Cambridge Fast Ring Networking System. *IEEE Transactions on Computers*, 37(10), October 1988.

[9] IEEE Computer Society Draft Standard. *Extension for Portable Operating Systems P1003.4a / D6*, 1992.

[10] Internet Working Group. Draft 5, Classical IP and ARP over ATM, October 14th 1993.

[11] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC-1144, February 1990.

[12] D. R. McAuley. Protocol Design For High Speed Networks. Technical Report 186, Cambridge University Computer Laboratory, January 1990. Ph.D. dissertation.

[13] SUN Microsystems. *SunNet X.25 System Administration Manual*, October 1990. Version 7.0 beta.

Richard Black (Richard.Black@cl.cam.ac.uk) obtained a Bachelor's degree in Computer Science from the University of Cambridge in 1990. He is currently in his third year as a Ph.D. student at the University of Cambridge Computer Laboratory. He has worked on the hardware and software of the Fairisle ATM switch, and the implementation of the Wanda micro-kernel.

Simon Crosby (Simon.Crosby@cl.cam.ac.uk) is a Ph.D. student at the University of Cambridge Computer Laboratory, studying control and management of ATM networks. He holds an MSc degree in Computer Science from the University of Stellenbosch, South Africa, and a BSc (Hons) degree in Computer Science and Mathematics from the University of Cape Town, South Africa.