Connection Closures
Adding application-defined behaviour to network connections

S. Rooney
University of Cambridge, Computer Laboratory
New Museums Site, Cambridge CB2 3QG.
Telephone: 44 1223 334650. Fax: 44 1223 334678.
email: `Sean.Rooney@cl.cam.ac.uk`

Abstract

New techniques in the implementation of out-of-band control in ATM networks are causing both industry and research laboratories to look again at the whole question of ATM signalling. These techniques devolve the control from the network devices into a higher level distributed processing environment, resulting in simpler network devices and more flexible control architectures.

This paper takes this idea one stage further and suggests that at least in some cases, the only place in which control can be exerted without inhibiting applications is within the applications themselves. We call the combination of an application defined control policy and a network connection a *connection closure*.

1 INTRODUCTION

A network control architecture is a set of policies and algorithms used to permit network devices to pass information between geographically separated end systems. Control architectures can be loosely divided into two groups: those that do not require the preallocation of resources to applications before allowing them to send information, and those that do*.

The first type of control architecture is often considered to be more flexible. Control decisions are taken on a hop-by-hop basis for each packet in the data stream and there is no need for a control protocol distinct from the data transmission protocol. On the other hand the latter approach allows stricter guarantees to be given to end systems about the likely behaviour of the network during data transmission. The set of resources allocated to an end-system across the network is normally termed a *connection*. IP is an example of a network having the first type of control architecture and ATM an example of the second. Over the last five years the Internet community has been working on adding a reservation protocol (RSVP) within the IP suite in order to allow soft real-time data such as video to be carried efficiently over the Internet [1]; with RSVP, IP will become an intermediate case.

ATM requires a connection control protocol, often called a signalling protocol, in order for the end-system to ask the control architecture for the creation, maintenance and liberation of connections. The exact nature of future ATM control is still to be decided. Flaws have been pointed out [2] in current ITU-T signalling protocols, such as Q.2931 [3] due to a lack of a clear distinction between application level services and their communication requirements.

*Although the situation is not quite as clear cut as suggested here and there are many intermediate cases, for the purposes of our discussion this binary division is sufficient.

More recently, there has been a move to devolve control from the physical switches into a distributed application level control architecture [4, 5]. The intention is to introduce greater flexibility into the management and control of the network by separating out-of-band control from the network devices; this will allow:

- service providers to introduce new services more quickly;
- network operators to optimise the use of the network devices for their particular network;
- the homogeneous control of heterogeneous network devices;
- the running of many different control architectures over the same devices simultaneously.

In these control architectures the application/control API is defined as a set of services within a Distributed Processing Environment (DPE), e.g. CORBA [6]. These services define the complete set of primitives that applications will use in order to create, maintain and release connections.

A connection type is defined by the nature, amount, time period and location of the resources that need to be allocated to it. At one extreme a primitive could be defined for the creation of each individual connection type; at the other, a single primitive could be defined taking as argument a well formed sentence in a connection description language. Real systems will use an approach that falls somewhere in between these two extremes.

For most applications this will probably be sufficient. However we see a role for allowing certain applications to pass their own *application specific* control policy for connections into the network and having that policy interact at a very fine level of granularity with the allocated resources during the life-time of each connection. This will allow users to take advantage of their high-level knowledge of the function of connections within an application to optimise their use of network resources.

We call the combined control policy and connection a *connection closure* following the usage in programming of a *closure* as a behaviour combined with a context over which that behaviour executes. In this paper we motivate the interest of connection closures and illustrate their use by way of several examples. The examples show the flexibility of the approach and its ability to provide fine control over resource use. We then discuss an implementation of connection closures in our ATM network. Finally we contrast our approach to that of other authors.

2 MOTIVATION FOR CONNECTION CLOSURES

Within existing ATM control architectures an application, after signalling its requirements for a connection, delegates its control over the connection to the control architecture. The connection belongs to the application in the sense that the resources comprising that connection can be unambiguously accounted to it. There is no fundamental reason why the application should not use its own resources in whatever way it wishes, even if those resources are scattered across the various network devices over which the connection exists.

It would be possible to build into the application/control-architecture interface a set of primitives which allow the application to interact with the connection during its life-time. However, this has two major disadvantages:

- it would involve a communication overhead which would limit the time-scales on which modification of the connection can be made;
- it would require the exact needs of present and future applications, in regard to their control over connections, to be built into this high-level interface. This is clearly impossible.

We propose that the application should at connection creation, be able to pass into the control architecture an application-defined control policy as well as a connection description. The control policy is a dynamically loadable program in some appropriate programming language which the control architecture can read, load and run.

After the control architecture has successfully allocated the resources on the diverse network devices specified in the connection description, a handle on these resources is combined with the application-provided control policy to form a *connection closure*. This closure is then executed within the context of the control architecture.

The connection closure interacts with the network only via the handles on the resources it was allocated. Connection closures are free to manipulate the resources in whatever way they see fit; the role of the control architecture is simply to:

- allocate resources to applications;
- provide an interface to the resources;
- provide the context in which connection closures execute.

The utility of having application-defined entities executing at the heart of the control architecture is that they have high-level knowledge about the use that end systems are making of connections while interacting with the control architecture at a very low-level. We not arguing that this technique will replace the need for other control architectures - that is certainly not the case - but only that they allow applications a greater flexibility in the use of their connection resources and that for certain services this is very useful. Some examples of the potential uses of connection closures are given in the following subsections.

2.1 Optimising the Use of Resources

Suppose a security guard monitors video from two different locations, each with its own camera. Imagine further than the locations are adjacent and that the two cameras are connected to the same switch, call this *Switch 2*. Let the display be attached to a neighbour of *Switch 2* called *Switch 1*. It would be possible to establish two distinct connections from the cameras to the display of the security guard. However, the designers of this service may know that the guard will only ever observe one camera at a time and that therefore at any given moment one of the connections is redundant. Figure 1 shows this system diagrammatically.

In a traditional ATM control architecture the designers would not be able to take advantage of this fact, because there is no primitive for creating temporally multiplexed connections. However using connection closures they can define a control policy which creates one virtual channel to the display across *Switch 1* from a VCI on the output port of *Switch 2* and periodically interchanges the input virtual channel across *Switch 2* with which it is associated. This is effectively a time division multiplexed bus [7].

The result is a decrease in the amount of bandwidth that need be allocated to the application as well as a reduction in the total number of VCIs that it uses.

2.2 Reacting to Changes in Network State

Connection closures make the notion of a connection more dynamic and thus permit some of the flexibility normally associated with IP to be introduced into ATM networks.

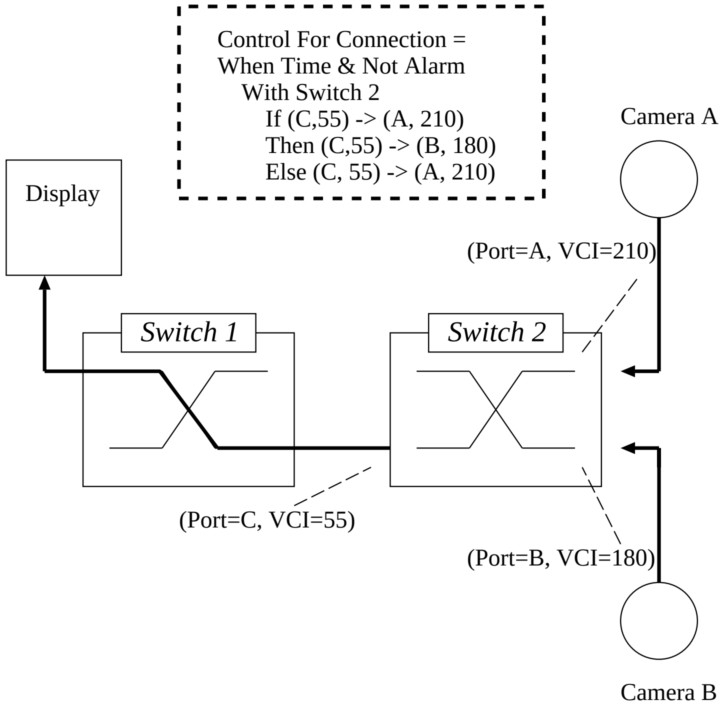


Figure 1 Example of Resource Optimisation

Within traditional IP, routing decisions are made for each packet at each router [8]. Routers periodically exchange information with their neighbours about the state of the network e.g. whether a given link is up or down. The routers can make decisions about the best next-hop on a per packet basis; thus two packets in the same message may take different routes to traverse the network, if for example a link fails or an administrator readjusts the weights assigned to the router links. The network can transparently alter the flow of data in response to changes in network state.

Within a traditional ATM control architecture this is not possible. Once established, a connection cannot be altered without the application being aware of the change. Thus, if an Available Bit Rate (ABR) [9] connection is established across a given set of switches and consequently those switches start dropping packets due to heavy congestion, there is no one way to alter the connection so that it takes a less congested route even though this may exist.

Running a part of the application - the connection closure - at the heart of the control architecture allows the application and control architecture to interact at a very fine level of granularity. Potentially, the closure, receiving information about dropped packets from the control architecture via alarms, might decide that the connection's quality is no longer adequate. It may then ask the control architecture for an alternative and replace all or part of the connection with that route. The closure could if necessary 'phone home' in order to inform the information producer to back-off during this operation.

In a similar way, an ATM connection may be made 'self-healing'; after a switch failure has been identified by the control architecture, connection closures having connections crossing that switch may adapt themselves in order to take into account this failure and find an alternative route. This is different to other methods of achieving self-healing in that applications decide the recovery policy e.g. drop connection, reroute, in reaction to failure, rather than some generic policy being used.

2.3 Mobility

Reference [10] identifies one of the challenges of mobile ATM as the necessity to distinguish within the mobile control architecture between different types of applications. Certain applications are keen to learn about changes in network conditions in order to be able to adapt their behaviours whilst others wish to operate transparently of the fact that the control architecture supports mobility. Reference [10] describes the SWAN ATM control architecture. It states that current ATM APIs are tailored for static environments and only allow basic control operations such as VCI establishment and tear-down; SWAN gets around this problem by allowing applications to register an interest in events - such as hand offs - at the MAC level.

The use of connection closures within the control architecture has obvious advantages when that control architecture supports mobility. The connection closure can communicate with the application that generated it in an application specific way in order to identify its current location and resource needs. The closure has the responsibility within the control architecture for modifying its application's connections. The closure, possessing application specific knowledge, may also modify the resources allocated to the application without having to communicate with it. Since the connection closure may be executing within the same address space as other parts of the control architecture this can be done very efficiently. The control architecture need not be aware of the precise needs of the application. Section 3 and Section 4 describe in detail our generic ATM control architecture and the proof of concept implementation of a basic mobile service.

3 ARCHITECTURE

In this section we give a general overview of the Hollowman control architecture. We describe how it allows connection closures to be loaded and executed and how those closures can manipulate network resources and communicate with their applications.

3.1 Hollowman Control Architecture

The Hollowman [5] was designed in order to demonstrate the feasibility of an ATM control architecture in which out-of-band control has been removed from the switches and devolved into a higher level distributed processing environment.

It was decided that the Hollowman should support different APIs adapted to different application needs. It was our realization that no single API could optimally encompass the network needs of all present and future applications that led to our experimenting with connection closures.

The Hollowman currently has three APIs:

- *a socket API* in which applications manipulate ATM Service Access Points (SAPs) in much the same way that BSD sockets use file descriptors;
- *a service API* in which applications trade and connect to service interfaces much as in a normal DPE;

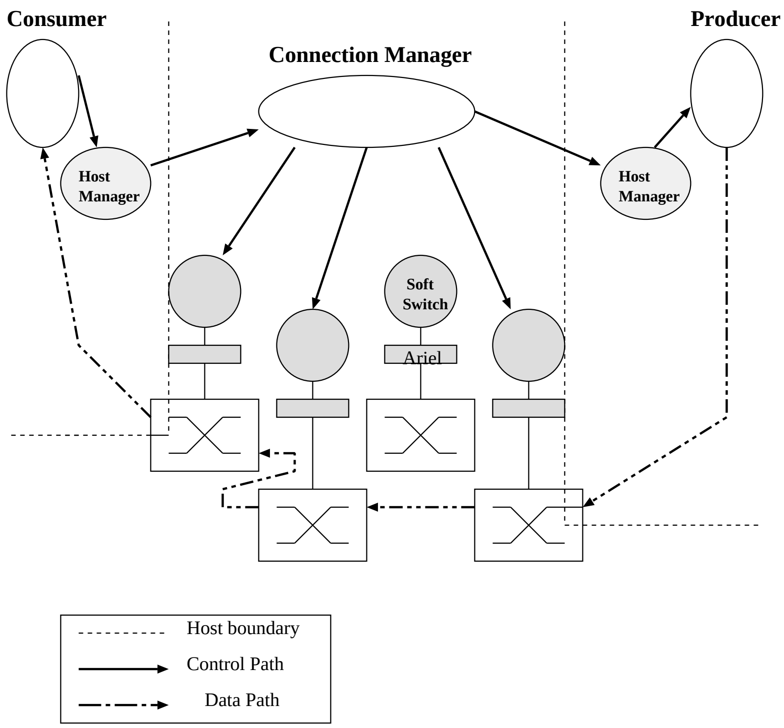


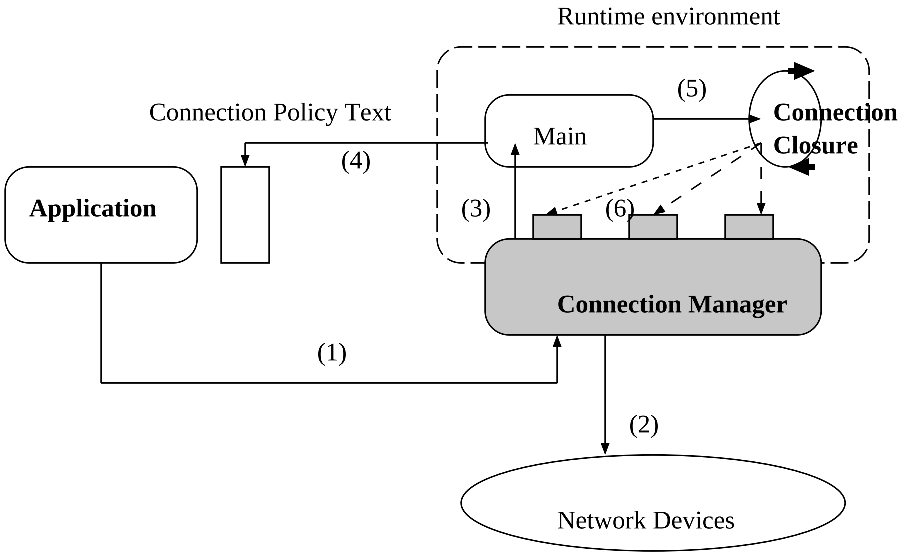
Figure 2 Connection establishment

- *a connection closure API* which applications use to request the loading and execution of closures to manage their connections within the Hollowman.

There is a trade-off between complexity and function in the decision as to which API to use. One of our overriding concerns was to ensure that the provision of advanced functions in the Hollowman would not impinge upon users wanting to do only simple things, i.e. 'you don't pay for what you don't use'. Applications requiring different APIs may use the Hollowman simultaneously.

The connection manager is a Hollowman service that handles the creation and liberation of connections across a set of switches. Each of the hosts connected to the network has a Hollowman host manager service. The host manager handles local host resources including the set of available Service Access Points (SAPs) on that host. To establish a connection to a remote host an application issues a request to its local host manager. The host manager allocates a SAP and relays the invocation to the connection manager which creates the ATM connection across the appropriate switches and invokes the remote host manager*. Figure 2 shows a schema for the creation of a simple connection between two hosts. The Hollowman communicates with the switches using a low-level switch interface called *Ariel* [11].

* This is only schematic; e.g. roll-back from failure, third party set up, multi-cast are all present in the Hollowman



(1) Call Control Architecture - createConnectionClosure
(2) Reserve resources for connection(s) on devices
(3) upcall into interpreted runtime system
(4) read and load connection policy
(5) Start-up Connection closure
(6) Read context and link to the beta-interface

Figure 3 Connection closure creation

3.2 Connection Closure

The connection manager has two interfaces: the α-interface for requesting the atomic creation and liberation of connections and the β-interface for more fundamental management of network resources. The first is used by the host manager and is typically implemented using the transport mechanism of the underlying DPE. The second can only be used by entities in the same address space and can thus be implemented very efficiently.

One of the operations in the α-interface is used to create a closure. The location of the connection closure behaviour is passed to this operation as a parameter. The connection manager loads the code from this location and starts executing it. The newly formed closure uses the β-interface for its interactions with the rest of the control architecture. Figure 3 shows a schema for the reading, loading and instantiating of connection closures. The closure is shown as being executed within an interpreted runtime environment, more information about the nature of this system is given in Section 4. The application's host manager is not shown for reasons of simplicity.

The β-interface contains operations which fall into the following classes:

- registration;
- resource management;

- virtual channel management;
- routing;
- notifying host managers;
- communication with applications.

These are now explained:

(a) Registration

Closures must register themselves before doing anything else. Non registered closures cannot obtain any resources and therefore cannot do anything useful. Within the Hollowman all applications are assigned a universally unique identifier. Closures use the identifier of their parent application in order to register. Registration, as well as allowing the closure to obtain resources, results in the creation of a mailbox. The mailbox is used for communication both between closures and between a closure and its parent application. This is explained in more detail in Section f.

(b) Resource management

A closure is a combination of state and behaviour. In order to be well formed a closure must obtain some network resources. The required resources are described in a string in a simple resource language. In the current implementation two types of resources exist, namely VCIs on switch ports and Service Access Points (SAPs) on hosts. A SAP is the means by which an application on a host uses a network connection. Resources such as bandwidth and buffer space are not currently handled, but would easily fit into the architecture.

Allocated SAPs are accounted to applications. For a SAP to be allocated to an application, the application must already have registered itself with the host manager at that host.

Handles to the allocated resources for the connection are returned to the closure which controls it. The closure uses these handles in order to manipulate the underlying resources. Closures are restricted to using only those resources allocated to them, thus offering a certain level of protection from interference between closures. A distinction is made between source and sink VCIs due to the asymmetric nature of ATM connections resulting from multi-cast. A similar distinction, and for similar reasons, is made between source and sink SAPs.

(c) Virtual Channel management

A source VCI on a port of a switch may be associated with a sink VCI on a port of the same switch. This creates a virtual circuit across that switch. This is a much more fundamental operation than the creation of a virtual circuit from host to host. It may be instructive to think of the resource allocation operation issuing the closure a set of dots across the network and the association operation managing the joining of these dots.

(d) Routing

A closure may know the switch ports on which it needs to obtain VCIs in order to create its network connections, in which case it does not need a routing function. However, if it doesn't know the network topology or if it wishes to reserve resources as a function of the current network state, it asks the connection manager to ascertain a best route between the two hosts. The connection manager calculates the best route using two functions: the cost function which is used to give a value for a connection passing through a given switch and a routing function which uses these values in order to determine the best route.

These functions are passed into the connection manager from the closure. Defaults exist for both functions; the default for the cost function is shortest path, i.e. it returns unity for every

State \ Statement	Allocated to application	Associated with VCI	Involved in end-to-end connection
Free	✕	✕	✕
Reserved	●	✕	✕
Active	●	●	●
Suspended	●	●	✕

● *Statement is true for SAP State*
✕ *Statement is NOT true for SAP State*

Figure 4 Summary of the semantics of SAP states

switch traversed and for the routing function the default is the weighted spanning tree. However, a closure could define any cost function and any routing function. Whether in a real system applications should be allowed to define their own cost functions is questionable, but it is useful in our experimental environment.

(e) Notifying host managers

The ability to associate VCIs is not in itself enough to create an application-to-application connection. The applications must associate their allocated Service Access Point (SAP) with a VCI on the port of the switch to which they are attached. A SAP is the application level end-point for a connection. An application must know of the current state of the SAP, in order to make use of a connection. Four states exist: *Free*, *Reserved*, *Active* and *Suspended*. Three factors determine the state of a SAP; these are whether the SAP is

- allocated to an application;
- associated with a VCI;
- involved in an end-to-end connection.

The combination of these three distinguishing factor for each state is summarised in Figure 4.

For example, an application having an *Active* sink SAP can expect to be receiving information on that SAP. If that SAP is set to *Suspended* it can expect that no more information will be received until it is reset to *Active*.

The closure can ask the connection manager to notify the host manager about a change in state in a SAP. As well as doing its own book keeping the host manager executes an application specified call-back to inform the application about the change in state. Application responses to changes in SAP state are application specific.

(f) Communication with applications

As a side effect of both host managers being informed of changes in SAP state and applications being 'call backed' when the SAP changes state, closures can prompt applications to start, stop and suspend receiving and sending information. For many applications this will probably be enough, but to take full advantage of the power of closures we require more complex communication. For example, an application should be able to prompt its closure to ask for more resources, as the needs of the application evolve.

In order to do this we allocate a mailbox to each closure when it registers. A mailbox is simply a pair of FIFOs and two event channels. When a closure sends a message it writes into its send FIFO and notifies its send event channel. When a receive event is signalled the closure reads the next message from its receive FIFO.

The parent application, on being notified of a receive event, reads the message across the network using a mailbox DPE service. It writes to the distant mailbox using the same service. Although the application must communicate by reading and writing over the network, all the reads and writes for the closure are local. This reflects the desire to make the closures as simple as possible.

As we have seen, applications communicate with their connection closures using mailboxes. Connection closures can communicate with each other using the same mechanism. This opens up the possibility of closures actually being able to trader resources amongst themselves with minimal interaction with the rest of the control architecture. For example, in an environment supporting mobile connections, a connection may be modified simply by adding a new virtual channel to the end of the existing one. This, although suboptimal in terms of resource usage, allows fast updates. At some point the connection may have to be modified in order to release the resources being wasted. If each of the mobile connections is controlled by a closure, then a closure might be prompted to 'slim down' its connection when another signals that it has failed to obtain its required resources. This is just a tentative suggestion and has not yet been implemented.

4 IMPLEMENTATION

In this section we describe the implementation of the architecture described in the previous section. We note some of our objectives, mention some problems and give some results. We conclude by mentioning proof of concept applications.

Fast connection set up is important to many applications so the vast majority of the code for the Hollowman is written in C/C++ [12] and compiled into native object code. The connection manager executable can be created and run in two ways:

- by linking the appropriate Hollowman's libraries with a C *main* and running the resulting binary as a 'stand-alone' application;
- by loading and linking those same Hollowman libraries into an interpreted runtime environment.

The latter gives us an extra degree of flexibility since the runtime environment allows the dynamic integration of new modules of interpreted code. Simple control operations behave exactly as if the code were being executed independently, outside this runtime environment. There is no cost for the fact that they are embedded inside an interpreted runtime system. Other entities in the control architecture make no distinction between the 'embedded' and 'stand alone' connection manager.

Currently we have implemented the connections closures in Java byte code [13] and they are executed in a Java virtual machine. Originally we had thought of defining a dedicated control language with only control related primitives. This approach would make the security issues easier to resolve albeit at a cost of restricting what the users could do. However, since initially our concern was not primarily one of security we decided to allow applications the greatest amount of freedom by permitting them to define control policies in a general purpose programming language. Java was the obvious candidate due to its ubiquity. In addition the Java Native Interface (JNI) permits complete interaction between Java and C, permitting us to call Java entities from C and pass C entities into the Java virtual machine. The fact that we are using Java byte code for the implementation of the closure's policy, does not mean that we are forced to write their high-level code in Java as well. Java byte code compilers now exist for many diverse languages [14], so connection closures are effectively language independent.

Hollowman uses a CORBA implementation called Dimma [15] for its application-to-application communication requirements. Having the source code for Dimma allowed us much greater flexibility than if we had used a commercial ORB.

The connection manager's Java *Main*, when executed in a virtual machine loads the appropriate native code libraries using the Java *Runtime* class. We then initialise our connection manager service, such that it exports its interface reference to our trader and starts listening on a socket for an invocation. This connection manager service is run as a single Java thread. Since Java threads are non-preemptive the connection manager must explicitly yield from time-to-time in order that the rest of the virtual machine can execute. This is achieved by timing out on the *select*, using the JNI to up-call into Java and yielding the connection manager thread. This is the only modification that we needed to make to Dimma.

When a simple connection request is received by the connection manager service, it is processed in the normal manner without any interaction with Java. However, when a request is received for a connection closure creation, an up-call is made into Java. Our Java code then reads the connection closure from the network location identified in the invocation using a modified form of the Java *ClassLoader* class. The behaviour of the closure is defined as a subclass of the Java *Runnable* class, so after reading in the behaviour we can instantiate it and call the *start()* method. The closure can now use the interface described in the previous section to create, maintain and liberate connections.

Due to the fact that the Java virtual machine in Java Developer's Kit (JDK) 1.1 is a single kernel thread, we cannot use a threaded version of Dimma as the virtual machine becomes 'confused' at the creation of new kernel threads. To get around this problem we used a single threaded version of Dimma and ensure that two Java threads are not calling into Dimma at the same time.

The test-bed in which we experiment with the control architecture contains a small set of Fore Switches attached to HP, DEC Alpha, and Sun Sparc machines, and some Fore AVA ATM Cameras.

4.1 Proof of concept applications

We decided that a challenging problem would be to implement a connection closure that was capable of establishing an ATM connection from an ATM camera to a display application and updating that connection each time the application moves to another host.

When and how to modify the connection is application specific; the closure may be used to add a new branch to the connection when the application moves, modify a part of the connection, or even remove the whole of the connection. We experimented with various strategies, showing that all of them could be realised within our architecture.

A key advantage of our architecture is that it allows us to load and execute multiple closures simultaneously; thus we could run the security closure of Section 2.1 at the same time as the mobility closure.

The mobile connection closure is about 200 lines of Java and compile to 4 KiloBytes of Java byte code. After the closure is loaded and resources are allocated, the time to establish and modify connection is mainly determined by the overhead of communication with the switches. Although the Java code is the top layer it is a very thin layer and does not have much influence on connection set up time.

The time for connection creation within the Hollowman can be divided into three components:

- processing within the control architecture ;
- communication in the DPE;
- communication with the switch

A connection across a single switch requires less than 3 ms of control architecture processing. The code for the control architecture is as yet unoptimised and we expect to be able to radically reduce this figure. Each Dimma invocation using the CORBA IIOP protocol requires about 5 ms; three invocations are required in order to establish an end-to-end connection. Hence DPE communication currently accounts for about 15 ms. The literature [16] has shown that latencies of between 1-2 ms for CORBA invocations with less than 100 bytes of information are achievable for commercial implementations. We have a variety of ways of communicating with the switch. Communication for the creation of a connection across a single switch using GSMP [17] takes 6-8 ms. Better results have been achieved using proprietary methods. Connection creation using SNMP takes more than 100 ms. The reasons for this is that the request for a single virtual channel requires the modification of many SNMP tables. This lead us to conclude that SNMP is inadequate as the basis for our control architecture/switch interface*.

We believe that we can achieve connection set up times at least as fast as the published figures for systems implementing standards based ATM signalling, e.g. Q-Port [18], requires approx. 9 milliseconds for connection set up across a single switch. This will be the subject of future work.

Some additional overhead results form the reading and loading of the Java code, but the types of connections which we believe can benefit from application defined behaviour are those which have connections which are long lived e.g. a multi-cast tree for a video-conference, and hence the initial cost of loading is not prohibitive. Applications simply wanting quickly to set up and tear down connections should not use closures but one of the other APIs that the Hollowman offers.

5 RELATED WORK

X-bind [4] is a middleware toolkit for builing open programmable networks which allows the creation of open ATM control architecture. This approach is similar to that of the Hollowman, however rather than supporting application definable loadable programs in order to extend the API, X-Bind instead offers a very rich application programming interface. The drawback of doing this have been elaborated in Sections 1 and 2.

Many attempts have been made to make network nodes e.g switches, programmable, by adding interpreters or dynamic linkers to the node software, [19, 20, 21]. Programs are sent along the data path and tagged for interpretation. The switches detect these programs and execute them. It is possible to distinguish two different aim for doing this:

- to deploy and upgrade control software at network nodes;
- to allow data streams to change the policies and algorithms controlling them.

---

*We are currently using SNMP because virtually all commercial switches support it.

The first happens infrequently and stops or inhibits the normal function of the node, until the transfer and loading are complete. The Hollowman follows the approach that [22] terms the fundamental axiom, namely that the impact of adding network control to managed nodes must be minimal, reflecting a lowest common denominator. We are therefore opposed to adding interpreters to switches unless it can be demonstrated that they are less costly in terms of processing power and memory than a static low-level switch interface such as our switch interface *Ariel* [11]. With advances in 'Just In Time' (JIT) compilers this may become feasible, in which case we would view it as an extremely attractive option for enhancing and updating *Ariel* itself.

The second, of which Active Networks [21] is an example, happens on a time scale which is at, or close to, the speed with which data units are transferred across the network node. For example within an ATM switch, the switch must distinguish these 'active' cells from normal data, assemble them and interpret them. In a sense this is what Operation And Maintenance (OAM) cells within the ITU-T forum standards [23] already do, except the languages in which these cells are programmed may be completely defined by a finite set of sentences stipulated in the relevant standards: e.g. *ok*, *not ok*. The use of OAM cells is restricted to ensuring the health and proper functioning of the entire network rather than optimising the resource usage for specific applications. The problem is our view is *significantly* more complicated when the programs, as in Active Networks, are:

- written in a general purpose programming language e.g Tcl, Java;
- sequenced across an arbitrary large number of cells, which must be identified as special and assembled into a program;
- can manipulate the virtual channel on which these cells arrive;
- can do this with out interfering with other users;
- can do this for switches with different capabilities and from different vendors.

We await experimental evidence that these active programs can be executed in a time scale that warrants their presence in the data path. The approach we have described in this paper restricts itself to the control path.

Reference [24] describes another solution to the limitations of trying to define a single generic control API for all present and future services. It proposes the creation of *service-specific* control architectures. This is made possible by allowing many control architectures to execute simultaneously over the same set of switches. This is complementary to the solution outlined here, i.e. we may within a service-specific control architecture use connection closures.

6 CONCLUSION

In this paper we motivated the need for the devolution of control out of the physical switches into a distributed application level control architecture. This enables open - or at least *more* open than the standards - signalling. Although we are in agreement with this approach we have identified a limitation: it is not possible to take advantage of application level knowledge within a generic control architecture without adding application specific primitives to the generic control architecture API. We believe that this is impossible to achieve as it would require us to define the control policies of as yet unimagined applications. Instead we propose the loading and execution of application defined code within the control architecture itself. We call the combination of this policy and the sets of connections that it manipulates a *connection closure*.

We have motivated the use of connection closures for reasons related to:

- optimisation of network resources,
- reacting to changes in network state and
- mobility.

We have outlined an implementation of connection closures within the Hollowman control architecture achieved at the University of Cambridge Computer Laboratory. We stress that applications using the Hollowman which do not require connection closures pay no price in terms of efficiency for the fact that the architecture supports them.

7 REFERENCES

[1] L. Zhang, "RSVP: a new resource ReSerVation protocol," *IEEE Network, vol. 7, pp. 8–18, Sept. 1993*, 1993.

[2] S. Crosby, "Performance Management in ATM Networks," *Cambridge University PhD dissertation, available as technical report TR 393*, May 1995.

[3] ITU-T, "Draft Recommendation Q.2931, Broadband Integrated Service Digital Network (B-ISDN) Digital Subscriber Signalling Systems No. 2, User-Network Interface layer 3 specification for basic call/connection control," *ITU publication*, November 1994.

[4] A. Lazar and K.-S. Lim, "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture," *IEEE Journal on Selected Areas in Communication, Vol 14*, September 1996.

[5] S. Rooney, "An Innovative ATM Control Architecture," *Proceeding's IM'97*, May 1997, San Diego.

[6] OMG, "The Common Object Request Broker: Architecture and Specification," *Document Number 91.12.1, revision 1.1*, December 1991.

[7] W. Stallings, "Data and Computer Communications, Fourth Edition," *Macmillian Publishing Company*, 1988.

[8] J. Moy, "OSPF Version 2, IETF," *URL* `http://www.ietf.org/ids.by.wg/ospf.html`, May 1997.

[9] ATM-Forum, "Traffic Management Specification," *Version 4.0*, April 1995.

[10] P. Agrawal and al, "SWAN: A Mobile Multimedia Wireless Network," *IEEE Personal Communications*, April 1996.

[11] K. van der Merwe and I. Leslie, "Switchlets and Dynamic Virtual ATM Networks," *Procceding's IM'97, San Diego*, May 1997.

[12] B. Stroustrup, "The C++ Progamming Language," *Addison-Wesley ISBN 0-201-53992-6*, 1994.

[13] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification," *Addison-Wesley ISBN 0-201-63451-1*, 1996.

[14] R. Tolksdorf, "Programming languages for the java virtual machine," *URL* `http://grunge.cs.tu-berlin.de/ tolk/vmlanguages.html`, 1997.

[15] G. Li, "Dimma Nucleus Design," *APM Technical Report, APM 1553.00.05*, October 1995.

[16] D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance End System Architecture for Real-Time CORBA," *IEEE Communications Magazine*, Feb. 1997.

[17] P. Newman, G. Minshall, T. Lyon, and L. Huston, "IP Switching and Gigabit Routers," *IEEE Communications Magazine*, Jan 1997.

[18] Bellcore, "Q.Port Portable ATM Signalling Software," *Product Information*, 1997.

[19] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie, "SwitchWare: Accelerating Network Evolution," tech. rep., CIS Department, University of Pennsylvania and Bell Communications Research, June 1996. White Paper.

[20] Y. Yemini and S. da Silva, "Towards Programmable Networks," in *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, (L'Aquila, Italy), Oct. 1996.

[21] D. Tennenhouse and D. Wetheral, "Towards an Active Network Architecture," *ACM SIGCOMM CCR, Volume 26, Number 2*, Apr. 1996.

[22] M. Rose, "The Simple Book," *Prentice-Hall ISBN 0-13-812611-9*, 1991.

[23] ITU-T, "ISDN-Maintenance Principles, B-ISDN Operation and Maintenance Principles and Functions," *ITU-T Recommendation I.610, ITU publication*, 1993.

[24] K. van der Merwe and I. Leslie, "Service Specific Control Architectures for ATM," *University of Cambridge work in progress*, 1997.