The Hollowman
an innovative ATM control architecture

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

The current implementation of out-of-band control in ATM networks inhibits their successful exploitation. The confusion in signalling protocols between application services and their resource requirements results in the loss of one of the key advantages of ATM which is the ability of applications to decide the requirement of their connections. The control being immutably built into the switches results in switch vendors, rather than service suppliers, defining the management policy which best suits those services.

The World Wide Web has recently become one of the most important services on the internet but was unimagined five years ago. Clearly history should teach us of the impossibility of predicting the services that will be in common use in the near future. It is not at all obvious how one can define a priori the required control policy for these as yet unknown services.

This paper presents an innovative control architecture called *Hollowman*, which devolves control from the ATM switches into an application level distributed processing environment.

Keywords

ATM, delegated control, distributed processing

1 INTRODUCTION

As (Crosby, 1995) points out, current ITU-T signalling protocols, such as Q.2931 (ITU-T, 1994), are flawed due to the lack of a clear distinction between application level services and their communication requirements. A signalling protocol should specify how to establish connections with arbitrary requirements and let applications decide which connections they need in order to implement services, rather than trying to specify a complete set of connection types within the signalling protocol itself.

It is clearly impossible to try and define all future services; this means that signalling protocols will constantly need to be modified as new services are required and network users will be frustrated in their desire to implement these innovative services. These two points combined will seriously inhibit the introduction of ATM. The limitations of the standards are evidenced by work such as *OPENET*, (Cidon, 1995), which is seeking to extend the Private Network to Network Interface (PNNI) (PNNI, 1994), in order to make it usable for intra-networking.

Orthogonal to this is the fact that currently the control policy is typically implemented in software running on the physical switch. Just as service suppliers should be able to decide on the resource requirements of the connections used in their services, so they should be able to define the policy to control those connections. At the moment this is not possible.

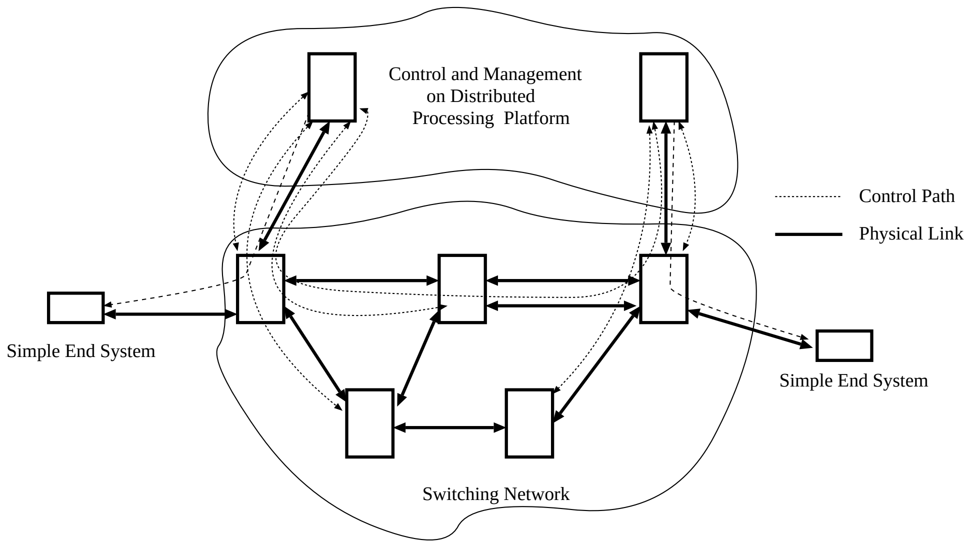

Figure 1 ATM Network with devolved management and control.

What we propose is to devolve control out of the physical switches into a distributed application level control architecture. Figure 1 shows the relationship between the control architecture and the controlled network. This control architecture opens up the management of the ATM switches and devices and enables service providers greater flexibility in the management of their services. At the Cambridge Computer Laboratory we have built an ATM control architecture called the *Hollowman*, based on these concepts.

Such an approach is similar in principle to that of the TINA consortium (Barr, 1993) in as much as they both use the techniques of distributed computing. However as (Crosby, 1995) points out TINA remains compatible with existing control and management standards when it is their rigidity which is one of the key problems to overcome.

The *X-Bind* architecture described in (Lazar, 1996) is closer to that described here. Although there are key technical differences between the Hollowman and *X-Bind* which will be evidenced in the rest of this paper, the major difference is philosophical. We believe that there will be no one single, ubiquitous ATM control architecture. Our work is geared towards the development of a necessary infrastructure - called the *Tempest* - which allows the simultaneous execution of many different control architectures over the same physical network. Although the Hollowman is a fully functioning control architecture in its own right, we view it more as a set of components, techniques and algorithms which can be used and reused in other control architectures within this framework rather than as a future standard. The mechanics of how control architectures can be made to coexist is detailed in (van der Merwe, 1996), the rest of this paper concentrates on the Hollowman.

1.1 Terminology

A *Domain* is a logical node of the controlled network, which is defined by the set of resources that it contains. An *Application* is a schedulable entity within some *Domain* to which resources e.g. CPU time, may be assigned.

A *Service Type* is a well defined task that a given *Application* can carry out on behalf of another. An instance of a *Service Type* is called a *Service*. A *Service Offer* is the means by which the existence of a *Service* is advertised within the control architecture. A *Service Offer* defines: the type of the *Service*; the location of the *Service*; the protocols which may be used to access that *Service*.

The process of *Trading* is the act of matching the requirements of a service user for a *Service* with the set of available *Service Offers*. The process of *Reification* is the resolution of a *Service Offer* into a access point for that *Service*. *Reification* involves the reservation of sufficient resources within the control architecture to permit an *Application* to use the service.

A *Connection* is a set of resources allocated to two or more applications across the network in order to exchange information. A *Connection Type* is defined by the nature, amount, time period and location of the resources that need be allocated to a *Connection*.

1.2 Overview

First, the concept of *trading* is discussed and some extensions within the the control architecture are presented.

Second, the *soft switch* which is both the interface to the physical switch and the encapsulation of a switch control policy is explained.

Third, the *host manager* which manages the resources within a given domain is introduced and the means by which the host manager allocates one resource - virtual circuit identifiers - to applications is detailed.

Fourth, the means by which connections are created across the network by the *connection manager* is explained. The concepts of *caching* frequently used connections and *lazy evaluation* of connection end points in order to achieve better set-up times are introduced.

Fifth, the novel concept of an application specifiable *call closure* is presented. The interest of applications being able to take advantage of their high level knowledge in order to be able to optimise their resource usage is motivated.

Finally, we describe the set of Application Programmer Interfaces (API) that the Hollowman offers.

2 TRADER

A *Trader* is an application that maintains a set of available service offers for a given domain and whose location is well known within that domain. Domain applications register themselves with their trader at start-up and in doing so they obtain a name which is unique within the scope of their domain.

Trading is hierarchical within the control architecture, i.e. an attempt is made to find a match for a service request first of all within the same domain as the requestor and if this fails, within the scope of a higher level. Thus the use of service providers which are, in some sense *close*, is favoured; this is particularly advantageous if a domain corresponds to a single work-station, in which case optimised forms of communication between the service provider and user are employed.

Within the current implementation of the control architecture two levels of trading exist: a trader for each domain and a trader which federates all these traders. For convenience we term a trader that maintains offers for a given domain as a *domain trader* and the single encompassing trader as the *federating trader*. The domain traders maintain service offers for general applications

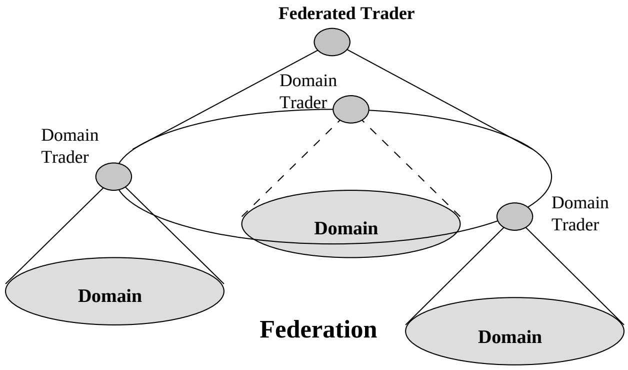

Figure 2 Example of three federated domain traders.

and the federated trader maintains domain trader service offers. Figure 2 shows a federation of three domain traders. At regular intervals the federated trader ensures that the domain traders are still active and if not removes them from the federation. All offers registered at a failed domain trader implicitly are no longer available within the federation.

Since domain traders are recorded as service offers, they may themselves be the subject of service requests. In particular it is advantageous for a given domain trader - trader *T1* to establish a connection with another, trader *T2* if the service offers of *T2* are often required in the domain *T1*.

Trading is a well understood concept and has existed in Distributed Processing Environments such as ANSA (APM, 1995) for quite some time; the federation of traders and the garbage collecting of service offers for failed applications is a natural extension of the basic trading concept. The originality of trading in the control architecture comes from the use of trader knowledge for domain resource management (this is detailed in Section 4) and the employment of trader service offers in ATM signalling (this is detailed in Section 5).

3 SOFT SWITCH

The *Soft Switch* has the following roles within the control architecture: it defines a set of logical control interfaces to the switch; it implements a control policy for the switch and it encapsulates the precise method of interacting with the physical switch.

The relationship between the physical switches and the soft switches is one-to-one. A soft switch contains a representation of the physical switch's resources as well as a set of switch control services. The complete control policy for a given switch is partitioned across these switch services. The different control policies that each switch service implements are independent. This separation permits different aspects of control policy to be kept distinct and allows different control functionality to be manipulated using a specific and dedicated interface. In what follows a *switch* denotes the combination of the physical switch and its associated soft switch.

A soft switch holds state about the physical switch in order to perform its control functions. For example the connection management service needs to know about the current resource usage on the physical switch in order to determine if a demand for a connection should be satisfied or not. In consequence, the soft and physical switch have to resynchronise in the event that either of them is stopped and restarted.

X-Bind uses CORBA/OMG (OMG, 1991) as the underlying platform for communication between *all* entities including exchanges between the management layer and managed network elements themselves. In our opinion running CORBA/OMG on switches is unnecessary and restrictive. The Hollowman communicates with the physical switch using the *Ariel* (van der Merwe, 1996) switch management interface. An *Ariel* server runs on the physical switch and an *Ariel* client runs in the same address space as the soft switch. The interface is defined by a set of services. A minimal set of services is defined for a switch but different switches may extend/enhance this set. The precise services that the physical switch supports is determined by the soft switch at start-up time. Figure 3 shows a schema of a switch.

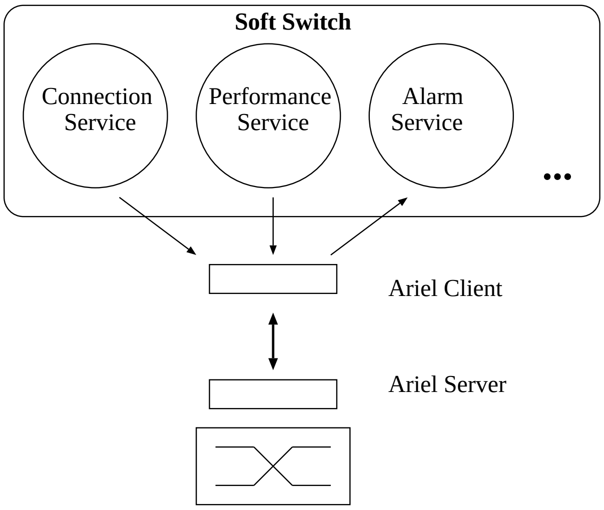

Figure 3 Example showing a soft switch containing three switch services.

It is important to stress that *Ariel* does not define a single wire representation for communication between the *Ariel* client and server. Many different mappings between the *Ariel* interface description and an underlying communication mechanism are possible e.g. RPC/UDP, CORBA/IIOP, SNMP/UDP. In the last case an SNMP daemon running on the switch takes the place of the *Ariel* server. A more detailed account of *Ariel* is given in (van der Merwe, 1996).

4 HOST MANAGER

The *Host Manager* is an extension of ideas developed within the Nemesis real-time operating system (Leslie, 1996). Nemesis allows applications to manage the resources allocated to them at a very fine level of granularity and thus enables them to make precise guarantees about their behaviour.

The host manager is an entity which allocates resources to applications within a given domain. It is common that this scope corresponds to a single work-station but is not a constraint of the model. However all applications are resident in one and only one domain. We reserve our discussion of host managers to the features which are associated with the control of the ATM network, but it should be noted that the host manager is a much more general concept than presented here.

A connection is the means by which one application sends information to a set of others across

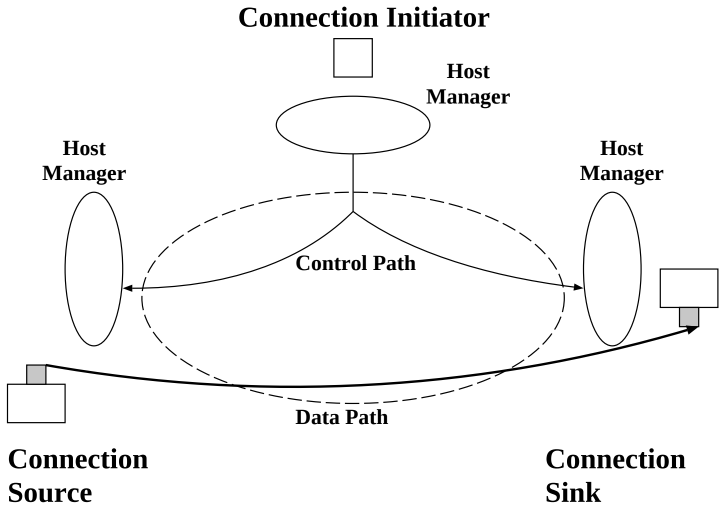

Figure 4 Example showing third party connection set-up.

the network. All connections have one sending application, called the source, and N receiving ones, called sinks.

The host manager has two interfaces: one which applications use to initiate connections and one which the connection manager uses to inform a host manager that another entity wishes to establish a connection with an application in the host manager's domain.

An application makes one of four requests in order to establish a connection:

(1) connect a sink service offer to a source service offer;
(2) connect itself as sink to a source service offer;
(3) connect itself as source to a sink service offer;
(4) connect a source service access point to a sink service access point.

In all four cases the host manager will check if the sink and source are actually both local and hence do not require a network connection in order to communicate. All but *(3)* may require joining a branch to a existing connection, in which case the host managers and connection manager recognise this and join at the appropriate place within the multi-cast tree. For the applications this is transparent.

It may be possible for applications to learn of the existence of an available service other than by using the control architecture trading mechanism which is why the *(4)* type request is supplied. For *(1)* and *(4)* the initiator of the request can be the sink or the source of that connection or neither. Figure 4 shows an example of third party connection set-up.

One managed resource is the virtual circuit identifiers (vci) that a given domain has at its disposal. The host manager at start-up obtains the set of Service Access Points (SAP) that have been allocated to that host within that control architecture. The SAP is the handle through which both applications and the host manager manipulate vci's. Host managers do not assign vci's directly to applications, rather they allocate them SAPs which the connection manager maps to a vci during connection creation.

An application wishing to establish a connection asks the host manager for a free SAP. If one exists the host manager sets the state of the SAP to *Reserved* and accounts it to the application.

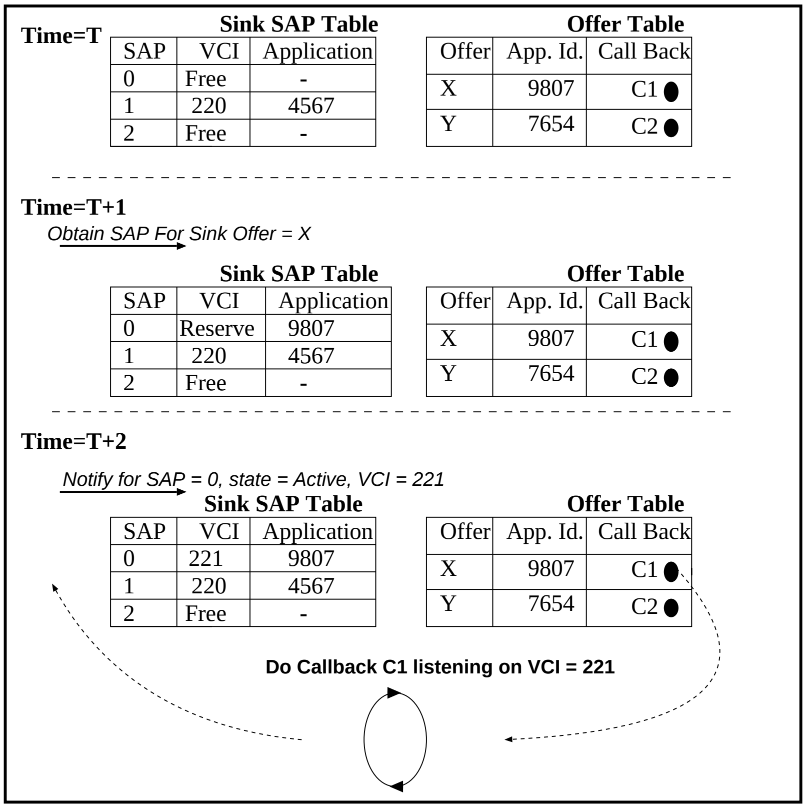

Figure 5 Example showing the host manager state changes during the establishment of a connection.

The SAP can be viewed as a token for a vci which will redeemed during connection creation; the connection manager having a more complete knowledge of the current network connections can decide what is an appropriate vci to map that SAP to.

Each time that a connection is established to an application within the scope of a host manager, the state of the SAP is set to *Active* and the SAP/vci table mapping is updated. The freeing of a *Reserved* SAP simply sets its state back to *Free*, the freeing of an *Active* SAP causes the host manager to ask the connection manager to release the connection and to notify the other host managers involved about the change in state. If an application fails before releasing a SAP then at some subsequent moment the domain trader will realize that the application has disappeared and tell the host manager to release any resources associated with that application.

An application registering a service offer associates a call-back with that offer in the host manager. The call-back is invoked when an attempt is made to reify that offer. After a successful invocation of the call-back the service provider application should be able to guarantee the service. Before an attempt is made to reify an offer no resources at all may have been allocated to it. Figure 5 shows a simplified version of the sequence of events which occur when a host manager receives a request to reify an offer as a sink in a connection.

The connection manager requests the host manager to find a free SAP and to reserve it for the offer X. The host manager verifies X is an offer present in its offer table, finds the application that supports X and allocates it SAP $= 0$. At a later stage the control manager notifies the host manager that a connection has been established to SAP $= 0$ and that the vci associated with it is 221. The host manager finds out that X is the offer waiting on SAP $= 0$, and executes the call-back for X.

5 CONNECTION MANAGER

Within the control architecture the *Connection Manager* has responsibility for establishing and tearing down connections between applications in distinct domains. In order to achieve this the connection manager has knowledge of the topology of the ATM network. It acquires this knowledge during the bootstrapping of the network or the network elements. From the point of view of the connection manager the network is made up of: host locations, switch locations, device locations, e.g. a camera.

The initiators of requests to the connection manager for connection creation are always domain host managers. The connection manager identifies the sink and source domains of the connection and determines a sequence of switches which constitutes a route between them. It then uses the host manager of the sink and source domain and the soft switches in order to reserve the required resources and establish the connection.

In the general case many routes may exist between two domains and the connection manager uses a routing algorithm to distinguish a 'best' route. Currently the default algorithm is a variant of the weighted spanning tree algorithm. The weights associated with each of the switches in a route are defined as a function of the current resource usage on a switch and the necessary resources required for a connection. The exact formula used to turn these two pieces of information into a weight is switch dependent and an intrinsic part of the control policy of the connection soft switch. However, within the framework of the Hollowman any of the diverse techniques described in (Lee, 1995) for resource constrained routing could be used.

The connection manager frees the resource associated with a connection when a host manager asks it to do so. The host manager may have been explicitly asked by an application or it may have decided that the application involved in the connection had failed.

The connection manager may decide not to remove a given connection between two domains if there are frequent requests for connections of that type between those domains, i.e. the connection may be *cached*. The connection can then be re-used the next time an appropriate connection request is received, thus reducing the latency in the set-up time. It should be noted that only the Hollowman makes a distinction between connections that are in use and connections that are cached; as far as the physical switch is concerned they are indistinguishable. Which connections are likely to be reused is highly application specific. This makes the use of connection caching problematic within a generic control architecture, but highly promising for application specific control architectures.

The demand for connection creation involves modifying state in at least four places: the source host manager, the sink host manager, the connection manager and the switches. It is possible that the attempt to create a connection fails after state has been already updated in one or more of the above. In this case all the updates must be undone and the original state before the creation restored. Thus a connection creation is in fact a distributed transaction which can be rolled back if the creation fails at any stage. This problem is made more complicated by the fact that the state in the connection and host managers should be locked for as short a time as possible to optimise concurrency and so we cannot, for example, simply lock the whole of the connection manager during an operation.

We have experimented with making the communication with the switches for creation and deletions of connections asynchronous in order to minimise the amount of time an operation occupies in the control architecture. Since the control architecture has a complete view of the state of resources in the switch, once the control architecture has decided that, say, a create operation is valid, then the only way the switch can refuse the connection is due to switch failure. The network connection is only marked complete when each switch has returned successfully. An application after asking for the creation of a connection will be forced to wait until the connection has been marked complete, however the application will not cause another application to block because the network connection belongs to it alone. In addition the create operation is executed in parallel on all the switches that it involves as well as with the modifications in state within the host and control manager, further optimising the connection creation time. The price of this is the introduction of asynchronous communication and hence some additional complexity. The above type of connection creation/deletion we denote as *lazy* in analogy to lazy evaluation within programming languages (Bird, 1988). We have experimented with this technique within the Hollowman and noted the expected factor of decrease in connection set-up time, as the N switches do their processing in parallel. The drawback of this technique is that it supposes that once a connection has been authorised by the control architecture that the physical switch cannot refuse it, if the switch does refuse it then recovering from the failure is further complicated.

The control architecture has complete knowledge of the topology of the network and maintains information about the current resource usage in its nodes. When the network is interconnected with a larger network it is neither desirable or feasible to have such information for the unified network. The interconnection of a network managed using the control architecture described here and other networks outside its control is a subject of on-going research.

6 CALL MANAGER

There are advantages to being able to group logically associated connections together into a higher conceptual entity. For example, it is likely in the bi-directional communication case that if one connection fails then the other should automatically be removed as well. We term this logical grouping of connections a *Call*.

Applications are free to associate any group of connections into a call. This avoids attempting to define all call types that all applications will ever want. We have adopted a similar approach to that defined in (Minzer, 1991), having a language in which an application may define a complex mesh of connections for, say, a video-conferencing application. What makes the concept of a call powerful is by allowing an application to create the control behaviour that is to be used within it. The associations of a group of connections with the control behaviour to manage those connections we term a *Call Closure*. Using call closures, applications can take advantage of their high level knowledge about how connections are to be used within a service in order to optimise the use of their resources. The call manager is the environment in which these call closures are loaded and executed.

An example illustrates the point: a security guard monitors video from two different rooms each with their own camera. Suppose that the rooms are adjacent and that the two cameras are connected to the same switch. We could establish two distinct connections from the cameras to the display of the security guard. However, the guard will only ever observe one camera at a time and that therefore at any given moment one of the connections is redundant. Knowing this we build a call closure which contains a connection from each camera to the display and which multi-plexes the two connections every 3 seconds. The call closure creates one connection to the display from a vci on the output port of the camera connected switch and periodically interchanges the input vci with which it is associated. Figure 6 is a schema for this example. (Ravindran, 1996) describes architectural and protocol techniques for optimised multi-cast transport, allowing many sources

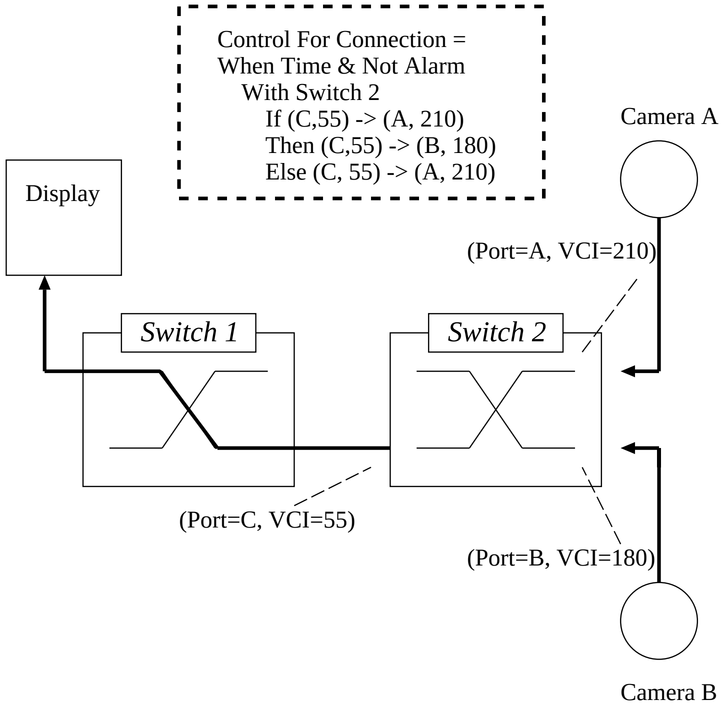

Figure 6 Example showing how call closures allows efficient resource usage in an application.

for example, to share the same distribution tree multiplexing them temporally. Call closures allow a means by which an application can define how this is to be done on *a per-application basis*.

The control defined by the application within the call request may only manipulate the connections allocated to that call, preventing calls interfering with each other; (Rooney, 1996) examines the motivation for call closures and their implementation within the Hollowman in greater detail.

In summary, the Hollowman allows application programmers to create a behaviour - defined in a dynamically loadable programming language - for the connections that make up their services and have that behaviour execute at the heart of the control architecture during the lifetime of those connections. This allows applications great flexibility over the network resources allocated to them.

7 HOLLOWMAN APIS

Applications at start-up, connect with their host manager and request the creation of an API instance appropriate for its needs. Three different types of API are currently present in the Hollowman:

- a BSD-socket like API;
- a service offer API;
- a call closure API.

The BSD-socket like API allows applications to do the normal: *open*, *listen*, *connect*, *receive*, *close*, type primitives on the Hollowman SAPs, no notion of trading or services is involved.

Applications which do not wish to use the notion of service are not obliged to. The second API involves importing service offers from the Hollowman trader and reifying those offers into SAPs, the reification is achieved by the use of the host manager and connection manager as described in Sections 4, 5. Thus applications can be unaware of the location of the entities they are communicating with and details such as joining to multi-cast trees are handled by the control architecture transparently, thus simplifying the application.

The third API is simply a gateway in which user defined call closures can be loaded and executed within the call manager.

8 IMPLEMENTATION

The Hollowman is written in C/C++ and uses Dimma (Li, 1995) for application-to-application communication. Dimma is a framework ORB, on which real-time ORBs can be constructed. Currently all of our communication is using a Dimma implementation of the standard CORBA protocol IIOP. The call closures and call manager are written in Java and interface to the rest of the control architecture using the Java-to-C API.

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

Most of the experiments have been carried out using Ariel/SNMP for communication with the switches. This is not ideal in terms of performance, but allows *any* switch running an SNMP daemon to be managed. Currently in the unoptimized version of the control architecture connection set up across a single switch requires approx. 200 milli-seconds. This breaks down to 10 % for application-to-application communication, 40 % for application processing and 50 % for communication with the switch. This is of a similar order of magnitude to that defined in (Veeraraghavan, 1995) for an implementation of B-ISUP. We confidently expect to be able to reduce this latency by an order of magnitude as more efficient implementations of *Ariel* become widely available and by the use of lighter inter-application communication mechanisms. Caching the connection effectively removes the part of the connection set-up time required for communication with the switch; lazy connection evaluation allows the partial parallelisation of the communication and processing with the different switches, thus is most useful when there are many switches.

9 CONCLUSION

This paper has presented the innovative Hollowman control architecture which devolves control out of the physical switches and into a distributed processing environment.

We have detailed the techniques by which:

- applications learn of the existence of services;
- physical switches are managed within the control architecture;
- connections are established between applications;
- resources associated with connections are managed.

We have showed how the flexibility of the Hollowman allows us to experiment with techniques such as connection caching and lazy end point evaluation. The concept of a *call closure* as a bundle of logically associated connections together with the control to manage those connections has been introduced.

The Hollowman demonstrates that it is possible to delegate many control functions out of ATM switches and that by doing so we permit the full exploitation of ATM. We do not however believe that in the future that there will be one, single, standard, ATM control architecture and that in consequence we are building an infra-structure in which many control architectures: Hollowman, X-Bind, TINA, Q-2931 may run simultaneously over the same network elements.

10 REFERENCES

Architecture Projects Management Limited (1995) ANSAware/RT 1.0 Manual *ANSA project*.

Barr, W. J. Boyd, T. Inoue, Y. (1993) The TINA initiative *IEEE Commun. Mag. March 1993*

Bird, R. Wadler, P. (1988) Introduction to Functional Programming *Prentice Hall*.

Cidon, I et al (1995) The OPENET Architecture *Sun Microsystems Laboratories SMLI TR-95-37*.

Crosby, S. (1995) Performance Management in ATM Networks *Cambridge University PhD dissertation, available as technical report TR 393*.

ITU-T (1994) 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*.

Lazar A, and Lim, K.S. (1996) Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture *IEEE Journal on Selected Areas in Communication, Vol 14, Sept. 1996*.

Lee, W. Hluchyj, M. Humblet, P. (1995) Routing Subject to Quality of Service Constraints in Integrated Communication Networks *IEEE Network July/August 1995*.

Leslie, I. et al (1996) The Design and Implementation of an Operating System to Support Distributed Multimedia Applications *IEEE Journal on Selected Areas in Communication, Vol 14*.

Li, G. (1995) Dimma Nucleus Design *APM Technical Report, APM 1553.00.05*.

Minzer, S. (1991) A Signaling Protocol for Complex Multimedia Services *IEEE Journal on Selected Areas in Communication, Vol 9, Dec. 1991*.

OMG (1991) The Common Object Request Broker: Architecture and Specification *Document Number 91.12.1, revision 1.1*.

PNNI (1994) ATM Forum contribution, Draft Specification *94-0471 R7*.

Ravindran, K. (1996) Architectural and Protocol Frameworks for Multicast Data Transport in Multi-service Networks *ACM SIGCOMM Computer Communication Review, Jan. 1996*.

Rooney, S. (1996) Connection Closures: Adding application defined behaviour to network connections *Submitted to Computer Communication Review, Oct 1996*.

van der Merwe, J.E. and Leslie, I. (1996) Switchlets and Dynamic Virtual ATM Networks *Procceding's IM'97, San Diego*.

Veeraraghavan, M. La Porta, T. Lai, W.S. (1995) An Alternative Approach to Call/Connection Control in Broadband Switching Systems *IEEE Communications Magazine, Nov. 1995*.

11 BIOGRAPHY

Sean Rooney received the B.Sc and M.Sc degree in Computer Science from The Queen's University Belfast in 1990 and 1991 respectively. After three years at the research center of Alcatel Alsthom at Marcoussis working in the field of network management, he started working for his PhD degree at Cambridge University.