Technical Report

UCAM-CL-TR-451
ISSN 1476-2986

Number 451

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

The structure of open ATM control architectures

Sean Rooney

November 1998

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 1998 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-451*

Contents

List of Figures

List of Tables

Glossary

General

AAL	ATM Adaptation Layer
ABR	Available Bit Rate
ADM	Add and Drop Multiplexor
AFI	Authorising Format Identifier
AIN	Advanced Intelligent Network
ANSA	Advanced Network Systems Architecture
API	Application Programmer Interface
ASN.1	Abstract Syntax Notation
ATM	Asynchronous Transfer Mode
ATMF	ATM Forum
AVA	ATM Video Adapter
BER	Basic Encoding Rules
B-ISDN	Broadband Integrated Services Digital Network
B-ISUP	Broadband Integrated Service User Part
B-ICI	Broadband Inter-Carrier Interface
BIB	Binding Interface Base
CAC	Call Acceptance Control
CBR	Constant Bit Rate
CDV	Cell Delay Variation
CLR	Cell Loss Ratio
CMIP	Common Management Information Protocol
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DIMMA	Distributed Interactive Multimedia Architecture
DPE	Distributed Processing Environment
FIFO	First-In-First-Out
FRS	Frame Relay Service
GCAC	Generic Call Admission Control
GDMO	Guidelines for the Definition of Managed Objects
GIOP	General Inter-ORB Protocol
GSMP	General Switch Management Protocol
ICMP	Internet Control Message Protocol

IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IFMP	Ipsilon Flow Management Protocol
IIOP	Internet Inter-ORB Protocol
IISP	Interim Inter-Switch Signalling
ILMI	Integrated Local Management Interface
IN	Intelligent Networks
IP	Internet Protocol
IPOA	(Classical) IP Over ATM
ISDN	Integrated Services Digital Network
ITU	International Telecommunications Union
ITU-T	Telecommunications Standardisation Sector of ITU
JDK	Java Developers Kit
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
LAN	Local Area Network
MAC	Media Access Control
MBS	Maximum Burst Size
MCR	Minimum Cell Rate
MIB	Management Information Base
MO	Managed Object
MOC	Managed Object Class
NFS	Network File Server
NNI	Network-Network Interface
NSAP	Network Service Access Point
OAM	Operation and Maintenance
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Integration
OTS	Object Transaction System
PDU	Protocol Data Unit
P-NNI	Private Network-Network Interface
PSTN	Public Switched Telecommunication Network
PVC	Permanent Virtual Connection
RM	Resource Management
RMON	Remote Monitoring of Network Devices
RPC	Remote Procedure Call

RSVP	ReSerVation Protocol
SAAL	Signalling ATM Adaptation Layer
SCP	System Control Point
SNMP	Simple Network Management Protocol
SPANS	Simple Protocol for ATM Network Signalling
SRG	Systems Research Group
SSCOP	Service Specific Connection Oriented Protocol
SS7	Signalling System Number Seven
SSP	System Switching Point
SVC	Switched Virtual Connection
TCAP	Transaction Capability Part
TCP	Transmission Control Protocol
TINA	Telecommunications Information Networking Architecture
TINA-C	TINA Consortium
TMN	Telecommunications Management Network
TUP	Telephone User Part
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
UNI	User-Network Interface
VBR	Variable Bit Rate
VC	Virtual Connection
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VP	Virtual Path
VPI	Virtual Path Identifier
VPN	Virtual Private Network
WAN	Wide Area Network

Dissertation Specific

ARF	Automatic Resource Freeing
CA-OAM	Control Architecture — Operation and Maintenance
H-OAM	Hollowman — Operation and Maintenance
SAP	Service Access Point
SICI	Simple Inter-Control Interface
SIRI	Simple Inter-Routing Interface

Chapter 1

Introduction

Asynchronous Transfer Mode (ATM) control systems are complicated. This complexity is witnessed by the evolutionary approach that organisations such as the ATM Forum (ATMF) have taken to the introduction of ATM control standards. The objective of these organisations is to define a single monolithic control architecture which can handle the needs of all present and future services. Their general approach is influenced by the control techniques that have been successfully used for telephony. However, these techniques are too rigid to be used within networks in which new service types have to be quickly introduced. On the one hand by attempting to encompass *all* possible control functions within a single control system the standards are guilty of over-specification, forcing network operators to pay an overhead for functions they may not need, and at the same time forcing them to adapt their services to the functions that are available even if these are inappropriate. On the other hand the standards are guilty of under-specification: by not defining the interface between the control plane and the physical network they in effect allow switch vendors to sell control systems which are switch-specific and in consequence can only be maintained and evolved by the vendors themselves.

This dissertation shows how by defining a low-level interface between the control plane and the physical switch, control systems can be made less switch-dependent and therefore more portable and easily modified. These *open* control systems can be implemented by parties other than switch vendors allowing network operators the potential to implement and maintain the standard ATM control architectures themselves or alternatively to implement completely proprietary systems. Such an environment can support multiple control architecture allowing network operators to choose the control architecture best suited to controlling the services they wish to support.

1.1 Motivation

The transport and processing of network control information uses resources and therefore incurs some overhead. The cost is traded off against the increase in efficiency and predictability that the control engenders. As an example, the accuracy of resource-constrained routing is a function of the rate at which information is exchanged between network elements, but when this exceeds a certain frequency the amount of resource used in the transport and processing of the control information outweighs the gain in having accurate knowledge. For a given network, the balance struck between the overhead incurred and the gain achieved, reflects the needs of the services which that network is required to support. A *network control architecture* is the set of protocols, policies and algorithms used to control a network. Example control architectures include SS7 [**ITU-T93b**], P-NNI [**ATMF96**] and IP switching [**Newman97c**]. The nature of each of these control architectures is congruent with the types of services that the network they control is expected to support.

End-systems sending or receiving time-sensitive information require guarantees about the arrival process of units of data transported across the network. The guarantees typically relate to the number of units arriving in some fixed time, the delay between units, and the probability of the network dropping or corrupting a unit. Other than by over provisioning, the only way to be able to make these guarantees is through the reservation of resources on the set of network elements across which the data flows. The control architectures of networks that support the transport of time-sensitive data fulfil their obligations to end-systems by reserving resources for them. The set of resources allocated to an end-system across the network is normally termed a *connection*, and the exchange of reservation requests, *signalling*. As services which produce continuous media become more prevalent, the need for a widely deployed unified transport network capable of satisfying their diverse constraints will increase.

ATM has been designed to support the data transfer needs of a large and unspecified set of different services. The ATM data path is both flexible and efficient, making it a candidate for the transport protocol of high speed service-independent networks. The simplicity of the ATM data path contrasts with that of its control plane, which is more complicated than that of many other widely deployed networks because it is required to support connections with arbitrary resource guarantees. Although there is no guarantee that the future data transfer protocol will be ATM, whatever protocol is finally chosen, it will have to deal with the same problems that the designers of ATM networks have already considered.

Control in a packet[1] forwarding network involves determining, at each network node, whether and to where a packet is forwarded. The control policy can be viewed at two different levels of granularity and on two time-scales:

- at the network node level;
- at the control architecture level.

The first is termed *in-band* control; within an ATM network it covers functions, such as traffic policing, which directly manipulate cells in the data path. The second, termed *out-of-band* control, concerns the management of connections, and covers issues such as connection routing, signalling and call admission control. This dissertation focuses on the provision of out-of-band control.

The ATM standards have defined a set of control primitives for use between the end-system and the ingress/egress switch [**ATMF95a**] and between switches [**ATMF96**]. While the standards recognise that different services have distinct resource requirements for their connections, there is an implied assumption that the out-of-band control policy exercised over those connections can be made common to all services. The standards attempt to define a single ATM control architecture valid for all possibles services within planetary scoped ATM networks.

It is impossible to know the precise control requirements of all future services and in consequence ATM control architectures will be subject to constant evolution and revision as new services are introduced with distinct control needs. For example, the control needs of mobile ATM systems is still a topic of research[2]; as they are better understood their integration with the single fixed standard ATM control system will require further revision to the standards. Even if it were possible to define all the control needs of all possible services within a single control architecture, it is arguable whether it would be appropriate to do so; such a control architecture would be extremely large and complex, and all services would

[1]The word packet here refers the Protocol Data Unit (PDU) in any protocol in which data is transfered in discrete units. It covers both an IP packet and an ATM cell.

[2][**Ngoh97**, **Acharya97**, **Sfikas97**] all propose different protocols and algorithms for the handling of mobile ATM hand-offs.

have to pay the price for its general nature. For example, implementing complex error recovery strategies incurs a significant overhead, some services might prefer to trade greater efficiency against robustness, but all services will have to pay the price for the needs of the most demanding service.

Finally, the standards attempt to avoid influencing the nature of their implementation by saying nothing about the interaction between the controlling plane and the physical switch. This arises from a belief that this is forcibly switch-specific and properly the concern of the switch vendors. Currently, ATM switch vendors propose an integrated approach where both the cell forwarding and connection establishment functions are sold as one complete package. This obscures the fact that they are to a large extent independent and imposes architectural decisions on network operators. Although the ATM standards do not require this integrated approach, neither do they proscribe it; no standard interface is defined between the control and switching plane, allowing switch vendors to implement proprietary solutions. The consequence of this is that switch control software is difficult to port from one switch to another and is impossible for third parties to maintain and evolve.

The unsatisfactory nature of the ATM standards will inevitably impede the wide scale deployment of ATM. There is a need for control systems which are less monolithic and rigid than those proposed by the standards.

1.2 Contribution

This dissertation shows how a low-level, service-based switch control interface permits a sharp distinction to be made between the physical network and the control layer. Control architectures that allow this type of separation are termed *open*, as they can be implemented, modified and maintained by parties other than the switch vendor. Opening the switch's control has several advantages:

- control functions can be made switch-independent, so the same control architecture implementation can manage different switches in a similar way and the controlling and switching plane can evolve independently;
- the switch controller and the switch fabric can be physically separated, so the control architecture can be executed in an environment potentially more powerful than that available on the switch;

- since such a low-level interface can be controlled by any number of different control architectures, network operators are free to implement their own control architectures, either proprietary or standard, in the way best suited to the services they wish to support. In this way network operators can make decisions about the nature and implementation of their control architectures which are more congruent with the types of services that the network they control is expected to support.

Other research [**Barr93**, **Lazar96**] has also recognised the limitations of the current standards and attempted to rectify them in a similar way. What distinguishes these approaches from that described here is that they attempt to replace the current single monolithic standard control architecture with another, perhaps more modular, one. This dissertation shows that much more flexibility is attained by accepting that no single ATM control architecture can ever be optimal for all services and defining a framework, called the *Tempest*, in which multiple control architecture, both standard and proprietary, can be supported. There is no longer any need to define one single 'best' control architecture.

The issues involved in open control in general and the Tempest framework described here in particular, are best revealed through an examination of a working control architecture. This dissertation describes the structure of an advanced switch-independent ATM control architecture, which executes in the framework. The feasibility and desirability of open control are demonstrated by showing that this control architecture is:

- **Realistic**: the control architecture, although simple, is fully operational and used by applications in the University of Cambridge Computer Laboratory. It implements control functions more advanced than those currently *required* by the ATMF signalling standards, e.g. sink initiated joins to multicast groups. Moreover, the performance of the control architecture compares favourably with that of several commercial implementations of the ATMF signalling standards. This dissertation uses the description of the structure of the control architecture to highlight some of the general problems in implementing control systems for multi-service networks.
- **Flexible**: the example control architecture described in this dissertation can be modified quickly and independently of the physical network, expediting the evolution of out-of-band control and allowing network operators to customise the control architecture for their own needs. The flexibility of open control manifests itself at two other levels:

  - firstly, within the overall Tempest framework many control systems can coexist allowing network operators to choose the control architectures best adapted to the services they wish to support;
  - secondly, advanced control architecture can be extended by dynamically loading code into them. Executing application-specific code within a control architecture allows applications to take advantage of their application-specific knowledge in order, for example, to optimise resource usage.

- **Scalable**: the usefulness of a particular control architecture cannot be judged only on its parsimony in the allocation of network resources and the flexibility it affords network operators and applications. The number of services that can be signalled to by using that control architecture is of fundamental importance, i.e. being able to communicate with the rest of the world is as important as being able to communicate with a small domain efficiently. It is likely that some widely deployed standard ATM control architecture will emerge. This dissertation shows how a control architecture of the type described here can take advantage of this by interoperating with standard ATM control architectures. Control operations which do not cross the domain boundary of the control architecture can take full advantage of its flexibility, while still allowing applications to signal to services offered outside its domain.

- **Robust**: a network's robustness is governed in part by its control systems ability to detect and react to unexpected changes in network state. This can be viewed on two different time-scales:

  - firstly, the control architecture automatically adapts its behaviour to take account of the unexpected change of state, e.g. port failure;
  - secondly, on a longer time-scale, a human network operator attempts to determine the origin of the incident and to modify the network if necessary, using tools offered by the management system.

  The clear separation between switching and controlling planes that characterise open control has an effect on the way the network is maintained.

  In regard to the first time-scale, the adaptation techniques specified by the standards assume that all control entities can read Operation and Maintenance (OAM) cells sent in the ATM data path. In the environment described here this is no longer the case as control entities may be physically separated from the switches they manage. The automatic adaptation of a open control architecture in response to network failure is considered by examining a number of approaches for dealing with port failure. This dissertation suggests that the standard OAM techniques should be extended in order to allow OAM messages to be exchanged between control entities, as well as network elements. Furthermore, it shows how allowing messages to be executable, i.e. to carry code as well as data, is beneficial in permitting more adaptive and flexible failure management.

In regard to the second time-scale, the fact that several distinct network operators may be running their own control architecture simultaneously within the Tempest framework means that it may not be appropriate to give a network manager full access to the complete state of the switch. This dissertation shows how existing management techniques can be adapted to permit access control at a finer level of granularity, while taking into account the trend in network management research for allowing network managers to dynamically add code to the management servers running on the network devices.

1.3 Outline

The organisation of the rest of the dissertation is as follows.

Chapter 2 presents an overview of the research context and describes the switch control interface. This chapter explains how multiple control architectures, all of which use this interface, can coexist on the same physical network.

Chapter 3 describes the components of a switch-independent control architecture which executes in the environment explained in Chapter 2, while Chapter 4 shows how the components described in Chapter 3 cooperate to allocate, maintain and liberate ATM connections. Chapter 3 and 4 together allow the flexibility enabled by open control to be examined through the description of an advanced working ATM control architecture. At the same time this description permits the demonstration of how some problems general to all ATM control architectures, such as atomic synchronisation of resources allocation and call admission control, are influenced by open control.

Chapter 5 details experiments carried out using the control architecture. Experiments related to connection caching and parallel connection establishment allow these techniques to be evaluated and the flexibility of the open control architecture to be further demonstrated. The description of some of the applications that the control architecture supports evidences its generality. Finally, the performance of the control architecture is evaluated and shown to be as least as efficient as control architectures more closely bound to the switch.

Chapter 6 discusses how applications can be hindered by high-level APIs. Extending the architecture described in Chapter 3 allows applications to add application-specific code to it. This increases the flexibility of the control architecture by allowing applications to take advantage of their application-specific knowledge.

Chapter 7 shows how two instances of the control architecture can interoperate to allow signalling beyond the control architecture domain. The generality of this solution is demonstrated by describing how the open control architecture interoperates with an implementation of ATMF UNI 4.0.

Chapter 8 explores the problem of fault detection and recovery within switch-independent control architectures. It shows how using mobile code allows adaptive and decentralised fault management.

Chapter 9 identifies some weaknesses in using standard management protocols within the environment described in this dissertation. It proposes a general purpose management system which addresses these issues and describes its implementation.

Chapter 10 summarises the research, identifies areas for future work and concludes the dissertation.

Related work is reviewed in each chapter.

Chapter 2

The Tempest Environment

This chapter describes the context in which the research detailed in this dissertation took place. The *Tempest* environment allows many different ATM control architectures to execute simultaneously over the same physical network. A requirement for achieving this is a mechanism which allows the resources of a switch to be partitioned between the control architectures[1].

2.1 Motivation for the Tempest

Schematically, an ATM switch contains:

- a number of ports through which cells are received and transmitted;
- a switching fabric across which cells are forwarded;
- a control module which creates and deletes connections;
- an Operation and Maintenance (OAM) module which ensures the general health of the switch.

In most commercial ATM switches, the control module communicates with the rest of the switch using a proprietary interface. Although this allows communication to be very efficient, the control plane is tightly coupled to the switching

[1]Jacobus van der Merwe originated this idea and was responsible for the implementation of the partitioning mechanism [**van der Merwe97**]; [**van der Merwe97**] calls the Tempest the more prosaic: *Open Service Support Architecture.*

plane. This means that it is neither portable nor readily modifiable by parties other than the switch vendor.

As [**Chen94**, **Alles95**, **Veeraraghavan95**, **Kant97**] all point out, ATM control is extremely complex. For example, [**Kant97**] states:

The stringent reliability and performance requirements for the current PSTN and ATM Networks have resulted in a rather complex set of signalling protocols for call set up and network management, and this complexity is expected to continue growing as new services and functionalities are introduced.

This complexity is witnessed by the evolutionary approach the ATM Forum (ATMF) has taken to the standardisation of ATM signalling, with standards — for example that of User Network Interface (UNI) [**ATMF93**, **ATMF94a**, **ATMF95a**] — being revised as experience is gained and new functions are introduced. It is impossible to know the precise control needs of services that have yet to be invented. Since one of the important motivations behind the use of ATM is that it will be able to support the resource guarantees of new continuous media services, the constant evolution of the signalling standards is likely to continue. Mobile ATM is one example of a new service whose control needs are distinct from those in the current standards. Moreover, even if it were possible to define a complete and final set of control functions adequate for all present and future services, including them within a single generic control architecture would both complicate the implementation of that control architecture and at the same time force even very simple services to pay the price, in terms of efficiency, for the complexity of the implementation.

As new versions of the ATMF's signalling system are standardised and implemented the software running on network nodes needs to be upgraded. This must be achieved without disturbing services written using earlier versions. So, for some period while the new version of the standard signalling software is introduced and tested, it will coexist alongside the older version. There is already a need for multiple control architecture, even if only for the migration path of the implementation of the existing standards. At the same time, new techniques, such as IP switching [**Newman97c**] or Xbind [**Lazar96**], make use of the ATM data path, while requiring a distinct control architecture from that defined by the standards, further demonstrating the need for the coexistence of multiple control architectures.

In summary, the complexity of ATM control coupled with the need to introduce new services quickly means that monolithic solutions are inappropriate and other solutions must be found. Multiple control architectures are already required and any workable solution must take this into account.

The control plane can be made less dependent on the switching plane by defining a non-proprietary interface through which the controller and the fabric communicate. A control architecture can then be implemented without detailed knowledge of the working of the switches it controls, allowing parties other than the switch vendors to implement the required control functions. Control architectures can then evolve independently of the physical network. The immediate advantage of this is that different switches can be controlled in the same way, enabling heterogeneous networks to be controlled homogeneously. For example, versions of UNI signalling developed for one network can quickly be ported to another, thereby expediting the update of signalling software.

Perhaps more importantly, making a clear distinction between the switching and control planes allows network operators to implement their own control architectures in the way best suited to their needs. Architectural decisions forced on network operators by vendors can be rethought. For example, a given switch controller can manage many switch fabrics. Conversely a given fabric can be managed simultaneously by many controllers if the resources of the switch can be partitioned between them. [**van der Merwe97**] calls such a partition a *switchlet*, because to each control architecture the partition resembles a small switch.

The Tempest framework supports a low-level service interface between control architectures and the physical network and provides a mechanism for creating switchlets, allocating them to control architectures and ensuring their integrity. A Tempest *virtual network* is composed of a number of related switchlets located on different physical switches. It is possible to run both standard and non-standard control architectures simultaneously within the Tempest environment. The Tempest solves the problem of allowing the different versions of UNI signalling to coexist, while also permitting operators to run other standard control architectures, e.g. IP switching [**Newman97c**], or even completely proprietary control architectures, within the same framework. The ability to support many control architectures means there is no need to try to define one single all-encompassing 'best' control architecture.

2.2 The Tempest Environment

This section gives an overview of the Tempest environment. First some terminology is introduced, followed by a description of:

- the testbed ATM network and Distributed Processing Environment (DPE);
- the *Ariel* switch control interface;
- the network builder;
- the *Prospero* switch divider;
- the control architectures.

2.2.1 Terminology

This section introduces and defines the terminology used in the rest of the dissertation.

The *Domain* of a control architecture is the part of the network it controls. A *Host* is a logical edge node of the controlled network. The type of a host is defined by the set of resources that it contains. An *Application* is a schedulable entity within some host 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*; when there is no ambiguity a service type may simply be called a service. An application that offers a service is called a *server* and the applications that use it are called *clients.* The set of operations that define the client's view of the service is called an *interface.* A given service may have many interfaces. The means through which clients and servers exchange information is called a *transport.*

An *Interface Reference* is an entity containing the information required by a client to locate and establish communication with a server. An interface reference typically contains information about the service type and the possible transports that can be used to communicate with a server. The client communicates with a server by associating some local resources to an interface reference, e.g. a TCP socket. The set of client resources allocated to an interface reference is termed an *Invocation Reference.*

A *Service Offer* is the means by which the existence of a service instance is advertised within some scope. Service offers encapsulate interface references. 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.

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

2.2.2 The Testbed Network

At the time of writing, the ATM test network at the Cambridge Computer Laboratory consists of three commercial ATM switches, two Fore ASX-200 switches and one Fore ASX-100 [**Fore95a**]. Attached to these switches is a variety of ATM capable workstations: Sun Ultras under Solaris 2.5, HPs under HP-UX 9.0 and DEC 3000/400 Sandpiper workstations running the experimental Nemesis multi-service operating system. In addition, a number of ATM cameras — Fore AVA 200 and 300 — [**Pratt94**] are connected to the test network.

An IP network is overlaid across the ATM network. This uses RFC 1577 classical IP-over-ATM (IPOA) [**Laubach94**] based on UNI signalling [**ATMF95a**] or the Fore proprietary SPANS (Simple Protocol for ATM Signalling) [**Fore95b**] signalling protocol for connection control. It supports services, such as NFS, used for the normal operation of the general purpose workstations.

The Fore ATM libraries are used on the Solaris and HP-UX machines. These allow applications running on ATM-enabled workstations connected to Fore switches to create ATM connections and to send and receive data across those connections. The libraries support a high-level operational API. The ATM connections can be established using either SPANS or UNI signalling. The work described in this dissertation makes use of the data transmission and reception part of the Fore API, but makes no use of the Fore signalling primitives. This is possible because the API also allows for Permanent Virtual Channels (PVCs) to be created other than by signalling; the connections created by Tempest-aware control architectures appear to the Fore API to be PVCs.

2.2.3 Distributed Processing Environment

A Distributed Processing Environment (DPE) is the framework in which distributed entities cooperate to achieve some purpose. The OMG/CORBA DPE [**OMG95b**] specifies the language for defining interfaces, transports between clients and servers, and the behaviour of clients and servers during information exchanges. Also specified within CORBA [**OMG95a**] — although not widely available in existing implementations — are generic services, such as persistence, which are useful within the DPE. CORBA, both because of its position as a standard environment and the large number of free and commercial implementations is a convenient DPE for building the Tempest. The CORBA implementation used was APM's Dimma [**Li95**].

In the current implementation of the Tempest, the CORBA Internet Inter-ORB Protocol (IIOP) [**OMG95b**] is used as the reliable transport for signalling, as opposed to the ATMF's reliable transport SSCOP [**ITU-T94a**], and the signalling requests are carried over the IP-over-ATM (IPOA) network. This is extremely useful in an experimental environment, as it avoids having to directly set up and maintain signalling channels[2]. Although the Tempest's DPE is a CORBA implementation, care has been taken to ensure that the Tempest's dependency on that implementation is minimised.

2.2.4 The Ariel Switch Control Interface

The *Ariel* switch control interface is the means by which control architectures communicate with a switch. The intention is that Ariel be as low-level as is consistent with it being independent of any given switch.

Ariel clients communicate with the Ariel server using some convenient transport, for example CORBA's IIOP transport, and the Ariel server translates Ariel control operations into a form that can be understood by the physical switch. Since the internal interfaces of most commercial switches are not in the public domain the Ariel servers use whatever standard protocols are offered by the switch in order to accomplish their control functions, e.g. the General Switch Management Protocol (GSMP) [**Newman96**] defined for IP switching. The operations in the fundamental Ariel interface are divided into six groups:

[2]Q.Port [**Bellcore97**] allows SSCOP messages to be carried over UDP rather than AAL5 for the same reason.

- configuration operations;
- port operations;
- context operations;
- connection operations;
- statistics operations;
- alarm operations.

Switch vendors may extend the Ariel interface to take into account special features of a particular switch, but all Ariel servers are required to support at least the operations defined here. These are now explained.

The **Configuration operations** allow an Ariel client to determine the switch configuration. The minimum amount of information the Ariel server should be capable of returning is: an identifier indicating the make and version of the switch; the type of the Ariel interface it supports (extended Ariel servers support a subtype of the basic Ariel interface); the number of ports on the switch; the VPI/VCI range for each port and the ATM service categories that each port can support, e.g. ABR, VBR.

The **Port operations** allow a client to examine and change the administrative state of a port. The port states can be active, inactive or cell loop-back.

The **Context operations** allow clients to build resource contexts which can be associated with connections during connection creation. Making a distinction between the allocation of VPI/VCI space and other network resources has the following advantages:

- it enables Ariel clients requiring only best-effort connections to avoid using the context interface altogether;
- it allows standard context types to be defined, avoiding the need for their descriptions to be repeatedly passed across the Ariel interface;
- it allows the more complex context interface to evolve and be extended independently of the more fundamental connection interface.

The context contains four delay and loss parameters: Cell Delay Variation, Maximum Cell Transfer Delay, Mean Cell Transfer Delay, Cell Loss Ratio. In addition, there are six traffic parameters: Peak Cell Rate, Sustainable Cell Rate, Cell Delay Variation Tolerance, Maximum Burst Size, Minimum Cell Rate and the feedback algorithm for flow control. Different combinations of these parameters will be significant for the different ATM service categories of the connection.

The parameters follow those defined by the ATMF in [**Sathaye95**] and therefore should be adequate for many applications and supported by most commercial switches. This avoids requiring knowledge about the precise implementation of the fabric, e.g. queueing algorithms, which is unlikely to be available for commercial switches.

The **Statistics operations** allow a client to examine information about virtual channels, virtual paths and ports. This includes the number of received and transmitted cells, the number of erroneous cells, the number of dropped cells, and for virtual paths and virtual channels, the time elapsed since creation.

The **Alarm operations** allow clients to register an interest in changes of state in ports, for example in order to determine when a port has failed.

The **Connections operations** allow clients to create and delete virtual paths and virtual channels and to determine which virtual paths and channels are in place at any given moment.

Ariel does not specify the means by which the Ariel server communicates with the switch fabric; it is a programming interface not a protocol specification. Many different implementations have been carried out. These are discussed in Section 3.9.

2.2.5 The Network Builder

Figure 2.1 shows the relationship between the network builder, the Prospero switch dividers and the control architectures within the Tempest framework.

The *Network Builder* is a Tempest service which control architectures use to create, modify and release virtual networks. The control architecture either builds its network by specifying the resources it requires and the switches on which it requires them, or it is given a predefined virtual network. The network builder, after allocating a virtual network, informs the control architecture about

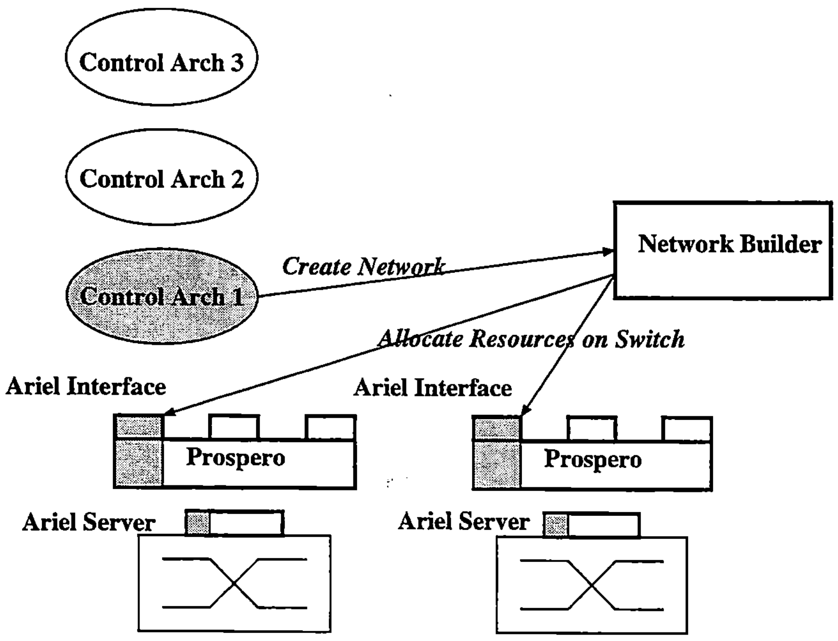


Figure 2.1: The Tempest infrastructure

the resources that have been assigned to it and passes it the Ariel interface references through which they can be accessed. When a control architecture finishes executing, it informs the network builder which liberates its virtual network. The resources which constitute that virtual network then become available for use by other control architectures.

The network builder obtains information about the network resources available on the switches from the switch dividers, explained in Section 2.2.6. After each virtual network is created, the network builder communicates with the switch dividers in order that the assigned resources be accounted to the requesting control architecture.

2.2.6 The Prospero Switch Divider

An Ariel client uses the interface references passed to it by the network builder to perform control operations within the scope of its virtual network. The client does not communicate with the Ariel server directly, but through the *Prospero* switch divider. The Prospero switch divider supports an interface which is identical to that of the Ariel server. The switch divider ensures that the resources an operation is trying to use belong to the client; if they do not, then the switch divider refuses the operation; otherwise the switch divider forwards the operation to the Ariel server. In this way, the Tempest constrains the control architecture to its own virtual network. A fuller account of the network builder and switch dividers is given in [**van der Merwe97**] and is not repeated here.

2.2.7 Control Architectures

The Tempest provides the framework in which control architectures can be allocated resources and the means within that framework to communicate with those resources. Control architectures which make use of this infrastructure are termed Tempest control architectures. The relationship between control architectures and the Tempest infrastructure is similar to that between applications and an operating system.

In general, the communication between the Tempest entities will take place across the ATM network that they control. Some primal bootstrapping control architecture must be created by means other than the Tempest itself. For example, the network builder must be able to communicate with the divider servers before the first Tempest control architecture is started.

In the current implementation, IP is used as the primal network. This has a number of advantages:

- IP is available on most ATM networks already;
- it is simple to bootstrap the IP network if it is not already in place;
- issues such as naming and routing are handled by IP and all the existing services which presuppose the availability of Internet protocols are obtained for 'free', e.g. NFS, IIOP, SNMP.

The running of IP over ATM requires some signalling protocol that the IP network can use to establish ATM connections. The solution used in the current implementation is simply to make use of the testbed's IP-over-ATM (IPOA) network. IPOA requires UNI signalling, so the presence of one control architecture — UNI — is presumed and this control architecture is used to create others. The bootstrapping problem is one which is common to all control architectures. In standard ATM control architectures the problem appears in the need to establish signalling channels without signalling. Similarly, the Tempest needs to create the bootstrapping Tempest control architecture without using the Tempest. In both instances, the solution depends on there being some in-built knowledge. The use of IPOA, although useful in the experimental environment, is questionable as a general solution, as UNI signalling is too heavyweight. LAN Emulation (LANE) [**ATMF95c**] is also excluded for the same reason. Other simpler methods of running IP over ATM, such as IP switching [**Newman96**], are promising; their implementation within the Tempest is the subject of future work.

Where Tempest control architectures execute is dependent on the location of the switch dividers. If the switch dividers are running off-switch, then the control architectures which use them are required to do so as well. If the switches allow the execution of foreign code, then control architectures and switch dividers can be executed on the switches themselves allowing all communication between the two to be performed using some efficient local transport. In the current proof-of-concept implementation of the Tempest, in order to take advantage of the rich development environment offered by the workstations, the control architecture entities and switch dividers are executed off-switch.

Q.Port [**Bellcore97**] is an implementation of the UNI signalling standard that has been slightly modified by the author so that it executes in the Tempest environment. Q.Port achieves some switch-independence by defining a *Fabric* interface that Q.Port switch controllers use to communicate with the actual switch. The fabric interface is tightly coupled to the Q.Port control architecture and is not as general as Ariel. The execution of Q.Port within the Tempest environment is achieved by implementing the fabric interface using the more fundamental Ariel interface. Q.Port is a commercial implementation; the fact that it can be treated in the same manner as in-house control architectures supports the assertion that *any* control architecture can be run within the Tempest.

The generic Tempest control architecture described in the next two chapters was used to test the Tempest environment.

2.3 Summary

This chapter has outlined the Tempest environment which allows many control architectures to run simultaneously over the same ATM network. The work in the rest of this dissertation was developed in the context of the Tempest.

Chapter 3

The Hollowman: An Open Control Architecture

This chapter introduces the core services within the Hollowman switch-independent control architecture. The Hollowman is a simple but fully functional and efficient ATM control architecture. The goal in designing and implementing the Hollowman was not to replace the ATMF's standard control architecture, but to demonstrate that ATM control can be made more open and that this openness allows greater flexibility and reduced complexity. Standard control architectures could usefully profit from these techniques, for example, by standardising a control interface similar to that of Ariel. The description of the Hollowman's structure in this and the next chapter elucidates these advantages, while also permitting some issues that have yet to be fully resolved to be considered.

Although it is likely that a small number of generic ATM control architectures will eventually be adopted and deployed by industry, this dissertation argues that some services require more advanced control than those that can be offered by generic control architectures. In particular, generic control architectures by their nature cannot allow *application-specific control*; Chapter 6 shows how a flexible control architecture, such as the Hollowman, can enable application-specific control.

The Tempest permits standard control architectures to coexist alongside more advanced proprietary ones and removes the need to define a 'best' control architecture. With that in mind, the Hollowman can be viewed not simply as a control architecture, but as a set of fundamental components with which network operators can build their own control architectures within the Tempest framework.

3.1 Introduction

Chapter 2 described how the Tempest can simultaneously support many running control architectures. The Hollowman is a Tempest-aware control architecture, used in the first instance to test the Tempest framework.

The Hollowman is a realistic ATM control architecture both in terms of the functions that it offers and the efficiency with which it performs those functions. The core Hollowman functions enable end-user applications to create and delete point-to-point, point-to-multipoint, and third-party connections between ATM capable workstations and devices using an Application Programmer Interface (API) which is small and simple to use. As Section 5.5 shows, the Hollowman's signalling latency compares favourably with a number of commercial implementations of the ATMF standards.

The only requirement that the Hollowman makes on the switches over which it executes is that they support Ariel, meaning that it could easily be ported to other environments. The Hollowman was implemented without detailed knowledge of the switches' implementation showing how parties other than switch-vendors can build ATM control systems. The Hollowman, while simple, supports the control needs of a number of different applications running in the Cambridge Computer Laboratory, demonstrating how realistic services do not necessarily require the complexity of standards-based signalling. Those applications have in turn served as a means of throughly testing the Hollowman. Moreover, the Hollowman serves as a flexible platform on which experiments related to ATM control can be carried out.

The structure of a switch-independent control architecture is elucidated through a detailed description of the Hollowman implementation. First, the following *core* Hollowman services are described:

- the soft switch, through which the rest of the Hollowman interacts with the physical switch;
- the host-manager, which manages the resources of a host;

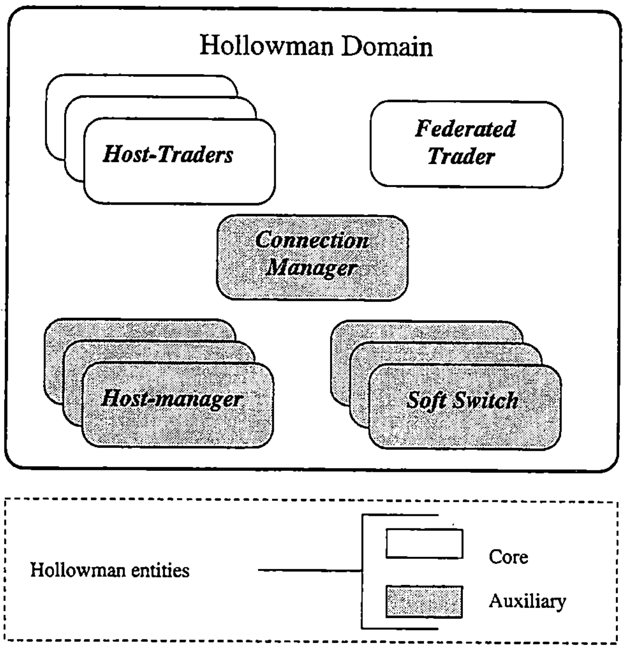


Figure 3.1: Hollowman entities

- the connection-manager, which synchronises the creation, modification and deletion of connections.

Thereafter several *auxiliary* issues are considered in relation to host addressing, application naming, trading and end-user applications. Figures 3.1 shows the core and auxiliary Hollowman entities.

Details of the implementation are given and a survey of related work concludes this chapter. Chapter 4 describes how the services introduced in this chapter interact to perform complete Hollowman control functions and Chapter 5 describes some of the applications that the Hollowman supports and some experiments carried out using it.

3.2 Soft Switch

The *Soft Switch* is the entity through which the rest of the Hollowman interacts with the physical switch. For every physical switch within the Hollowman virtual network, one soft switch is instantiated within the control architecture. Figure 3.2 shows an example Hollowman instance.

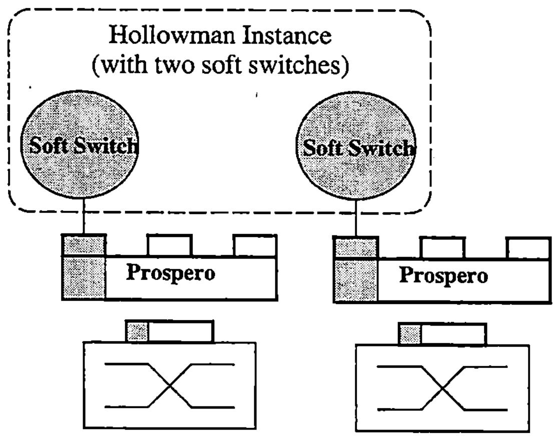


Figure 3.2: Example Hollowman instance

The soft switch has the following roles:

- managing the Hollowman's view of the switch resources;
- defining a set of logical control interfaces to the switch;
- encapsulating the precise method of interacting with the physical switch.

The Hollowman soft switch, implementing as it does typical switch control functions such as Call Admission Control (CAC)[1], is as much a part of the switch as the switching fabric itself. The *switch* can be viewed as the combination of a soft switch with an actual physical switch. Soft switches can easily be modified and replaced without affecting the underlying physical switch demonstrating the ability of both to evolve independently. The different roles of the soft switch are now considered in turn.

3.2.1 Resource Management

During its instantiation, the soft switch is told about the resources that have been allocated to the control architecture on the corresponding physical switch. The soft switch updates its view of the state of the switch after every relevant state changing operation performed by the control architecture. For example, it

[1]Within the Hollowman this might be more properly called Connection Admission Control but the term more widely used in the literature is preferred.

records which virtual channel identifiers are currently being used in connections on the switch.

Within a control architecture's virtual network all control operations are performed by the control architecture, therefore during normal operation keeping the control architecture's view of the state of the physical switch consistent with the actual state is straightforward. Achieving consistency after network failure is considered in Chapter 8.

3.2.2 Control Interfaces

In the current implementation of the Hollowman there are three control interfaces: the *Call Admission* interface, the *Connection* interface and the *Alarm* interface. The interfaces that a particular instantiation of a soft switch is required to support are determined from a configuration file during the initialisation of the Hollowman. All soft switches must have at least one call admission service and one connection service. The alarm service is optional. Other switch control services, e.g. an accounting service are the subject of future work.

The call admission interface takes a connection creation request description and returns a boolean value indicating whether or not the request can be accepted. If it can, then an indication of the cost of that connection is also returned. In the current implementation the cost is given simply as a real number in the range zero to one, where one is the highest cost and zero is the lowest. For example, the soft switch can calculate the cost of the connection as a function of the amount of resources it would leave free for further connection requests. The cost is used by the connection-manager when deciding whether to accept a connection and through which soft switches to route it. Any algorithm could potentially be used within the soft switch for deciding whether a connection should be accepted or not, and network operators can refine this algorithm, for example, in order to privilege certain connection types over others. By its nature, the soft switch is independent of the implementation of the switch and this means that its knowledge of the capacity of the switch is restricted; the effect that this has on CAC is considered in Section 3.2.4.

The connection interface allows resources to be reserved on the switch, connections to be created on the physical switch using reserved resources, connections to be removed from the switch, and resources to be liberated. In the current implementation of the connection service the only resources the connection service

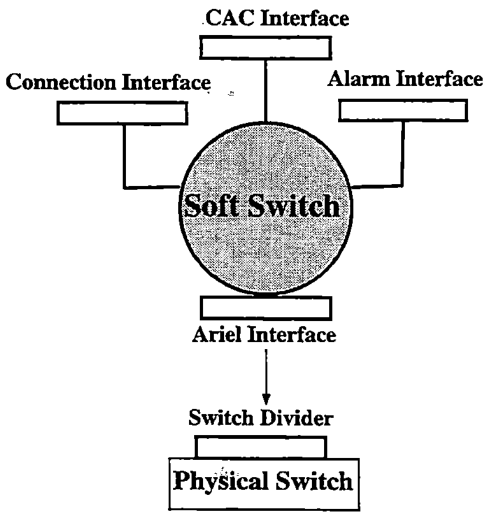


Figure 3.3: Soft switch

manages are the virtual channel and virtual path identifiers of the switch ports.

The alarm interface allows an interest to be expressed in the notification of an alarm, and the required response to that notification to be defined. Chapter 8 has a more detailed discussion about failure recovery within the Hollowman.

3.2.3 Encapsulation of Switch Interface

At its instantiation, the soft switch receives an Ariel interface reference and uses this to communicate with the switch. The set of Ariel interface references assigned to a control architecture are obtained from the network builder during the creation of the virtual network. The operations of the connection interface are mapped onto the Ariel connection operations; interest in alarms expressed using the alarm interface are forwarded to the Ariel server. No other entity in the Hollowman communicates with the switch directly. Figure 3.3 represents the soft switch diagrammatically.

The soft switch can determine if the Ariel server supports an extended version of the Ariel interface by examining the identifier of the Ariel interface obtained using the Ariel configure operation. For example, if an Ariel server offered the ability to support the *abstract switch model* defined in [**Newman97a**], then the soft switch could take advantage of that fact to implement the CAC function.

The Ariel ports, context and statistics operations are not used in the current Hollowman implementation.

3.2.4 Discussion of Call Admission Control

Call Admission Control (CAC) is the act of deciding if there are sufficient resources available to satisfy a request for the creation of a connection without affecting existing connections. Other factors, such as charging, may also influence the CAC policy. Commercial ATM control systems make use of very detailed knowledge of switch implementations in order to perform CAC. For example, to accurately determine whether statistical multiplexing gain is sufficient to permit a given mix of VBR connections, whose total *peak* bandwidth requirements are greater than that of the output port through which they pass, requires a knowledge of the cell queueing algorithms used by the switch.

Open control architectures, such as the Hollowman[2], are independent of any particular switch implementation. Ideally, decisions should be made by the control architecture independently of the switch about whether a connection with a given traffic profile should be accepted. This means that the open control architecture should implement its own CAC algorithms. Section 3.2 described how within the Hollowman each soft switch has its own CAC control interface. The CAC algorithms implemented in the Hollowman's soft switches can be:

- less conservative than that of the switch;
- identical to that of the switch;
- more conservative than that of the switch.

If the algorithm is less conservative, then the control architecture may determine a route which will be refused by the switches during establishment. The

[2]In the current implementation of the Hollowman, the only resources managed by the control architecture are the virtual channel identifiers and virtual path identifiers. Applications are not given guarantees about delay, loss or traffic parameters. Ariel uses the standard set of ATMF parameters — see Section 2.2.4 — to define the resource context for a virtual channel across a single switch. A traffic descriptor structure could easily be added to the relevant operations in the Hollowman's API and the connection-manager could transfer the request to the switches using Ariel. Some traffic parameters, e.g. peak cell rate, can be checked for each switch in isolation; others, such as total transit delay, require coordination amongst a number of switches.

control architecture must take this into account, e.g. allowing for rolling back part of the creation and attempting another route (crank-back).

Using the same algorithm as the switch requires special knowledge about the switch's cell scheduling policy. Such information is not generally available. It is possible [**Lazar96**] to map out the schedulable region for a given type of switch for a given set of call types. This has the disadvantage that it limits users to using only those call types and needs to be done for each different version of each switch.

The third possibility is to use algorithms which are more conservative than those used by the switch, for example mapping Variable Bit Rate traffic parameters onto Constant Bit Rate at peak rate[3]. Calls may be refused when in fact the switch can support them. If the algorithms used are only slightly more conservative, then the number of unnecessarily blocked calls may be sufficiently low that this technique is acceptable. In fact, this is the technique that P-NNI [**ATMF96**] uses to source route a connection, while having only incomplete knowledge about remote switches. In P-NNI, nodes periodically exchange information about their current resource usage with nodes of the same group. P-NNI specifies two algorithms — the Complex Generic CAC algorithm (C-GCAC) and the Simple Generic CAC algorithm (S-GCAC) — which a node can use to determine if a foreign node is likely to accept or refuse a given connection. Switch vendors are free to develop their own CAC algorithms, but they are constrained to being less conservative than the GCAC.

By their nature switch-independent control architectures, such as the Hollowman, have less knowledge of the capacity of the switches that they control than more switch-specific ones. The problems arise simply because switch manufacturers are understandably loath to reveal the mechanisms, e.g. cell scheduling algorithm, that their switches use; in that sense it is a commercial rather than a technical issue. Nevertheless, open signalling must address this problem, to be considered a practical technique. Mapping out schedulable regions is a useful solution for situations in which the call types are predefined and well known, but it is not generally applicable. Furthermore it reduces the flexibility of open control as the control plane is again tightly and statically bound to switch imple-

[3]It is interesting to note that [**Kalmanek97**] questions the utility of VBR for services that are as bursty as variable bit rate video. The problem is that it is difficult *a priori* to define reasonable values for the burst tolerance, sustainable rate and peak rate parameters. VBR may not be needed in many multi-service networks, significantly reducing the problem of Call Admission Control.

mentations. Unless there is some way of dynamically determining the capacity of commercial switches, open control architecture when performing CAC must either be:

- overly optimistic, thereby complicating the task of connection creation (to take account of rollback) and removing the possibility of performing effective resource constrained routing;
- overly pessimistic, and in consequence not take full advantage of the capacity of the switch.

The former is acceptable only in extremely undemanding contexts and the latter option is preferred. The use of the ATMF's GCAC as the conservative CAC algorithm has the advantage that switch vendors will in all likelihood ensure that their switches' capacity is at least that stipulated by the GCAC and that commercial switches designed to support P-NNI will be capable of supplying the necessary values to allow its calculation.

However, recent work suggests that it may be possible to dynamically determine a switch's capacity.

Firstly, [**Newman97a**] describes a proposed extension to GSMP [**Newman96**] which allows a switch to describe its scheduling and policing policy to a controller in terms of an abstract switch type. In the best case — in which the abstract switch model accurately models the capacity of the underlying switch — this will allow the controller to avail of the full capacity of the switch; in the case that the capacities of the underlying switch cannot be completely defined by the model, then the controller can implement a CAC algorithm which is conservative, but probably less so than GCAC. Future work will look at the possibility of extending Ariel to take into account this new version of GSMP. To some extent the success of this approach depends on the willingness of switch vendors to support enhanced GSMP servers; this again is a commercial rather than a technical question.

Secondly, the *Measure* project [**Crosby96**] is attempting to predict the effective bandwidth of a switch port by measuring the current traffic patterns and applying some results from large deviation theory [**Duffield95**]. Resource management in a network which uses statistical multiplex, such as ATM, depends on knowing the probability of rare events, e.g. cell loss. The probability of rare events is determined by the *rate* function of the traffic classes. Normally, a model is used to define this rate function; the models must characterise the expected traffic types making it non-general and difficult to evolve. The Measure project has adopted the novel approach of trying to measure the rate function. The ability to do this is based on the similarity between large deviation rate functions and thermodynamic entropy. The mathematical theory behind Measure is beyond the scope of this dissertation.

If Measure proves applicable to commercial networks, then open control architectures could dynamically determine the capacity of switches by *measuring* their behaviour. The Measure approach has the great advantage that it does not require much support from switch vendors; only the ability to obtain some basic traffic measurements. Although this work is still on-going, [**van der Merwe97**] reports some success in the use of an implementation of Measure's effective bandwidth estimator in determining the resources that need be allocated to Tempest virtual networks running on commercial switches. The author suggests that further work needs to be done: in investigating the measuring time required before the effective bandwidth estimates become usable and the application of Measure to switches which, unlike those used in [**van der Merwe97**], are not simple FIFO output queued and therefore less easily mappable on the Measure model of a single shared server. These issues are being addressed by the Measure project.

3.3 Host-Manager

For an end-user application to send and receive a continuous media stream it needs to reserve network resources on the devices across which that steam will be transported. However, both sending and receiving applications also need to be allocated operating system resources if true end-to-end resource guarantees are to be given. It is pointless, for example, to acquire enough resources to carry a video stream across the network to a workstation if the application cannot guarantee that enough of the framestore will be available to allow the video stream to be rendered. An advanced control architecture for multi-service networks must take this into account.

In addition, the signalling of end-user applications needs to be managed. For example, some access control policy must be enforced, stopping a given application from deleting the connections of another. This is best done as 'close' to the end-user applications as possible, since communication between the end-user applications and the signalling manager can be efficiently performed and the signalling manager can take advantage of any artifacts offered by the environment for the management of applications.

Within the Hollowman the allocation of resources on a host and the management of end-user signalling are both handled by an entity called the *host-manager*. There is one host-manager per host within the Hollowman domain. A host-manager normally runs on the host that it manages, but when managing dumb devices such as a camera, e.g. the Fore ATM Video Adaptor (AVA) [**Pratt94**], it is convenient to run it on a separate workstation. An end-user application uses the host-manager for signalling to other applications. The simple host-manager API is described in Chapter 4.

In the current implementation only a small number of resources relevant to the creation of ATM connections is managed. Future work will investigate the issues in allocating a complete set of both network and operating system resources to Hollowman applications. The Nemesis multi-service operating system [**Leslie96**] offers an application process the ability to specify precisely the operating system resources it requires. Nemesis is the ideal platform on which to run the Hollowman, and the host-manager and host-trader have been designed with this in mind[4].

3.4 Connection-Manager

Within the Hollowman soft switches manage virtual channels across the part of the switch that they control and host-managers manage the resources allocated to applications on hosts. In order to establish an end-to-end connection some entity must coordinate the action of the soft switches and host-managers. The Hollowman entity responsible for performing this coordination is called the *connection-manager*. Figure 3.4 shows the entities involved in the creation of a Hollowman connection.

In the description of the operation of the connection-manager given in this chapter and the next, it is assumed that there is only one connection-manager per Hollowman domain. This is simply for reasons of convenience and other configurations are possible, e.g. a connection-manager per switch. From the point of

[4]An exploratory port of part of the Hollowman to the Nemesis operating system has already been carried out by the author. A version of ANSAware [**APM95**] was used as the DPE, since Nemesis, at the time, did not support CORBA.

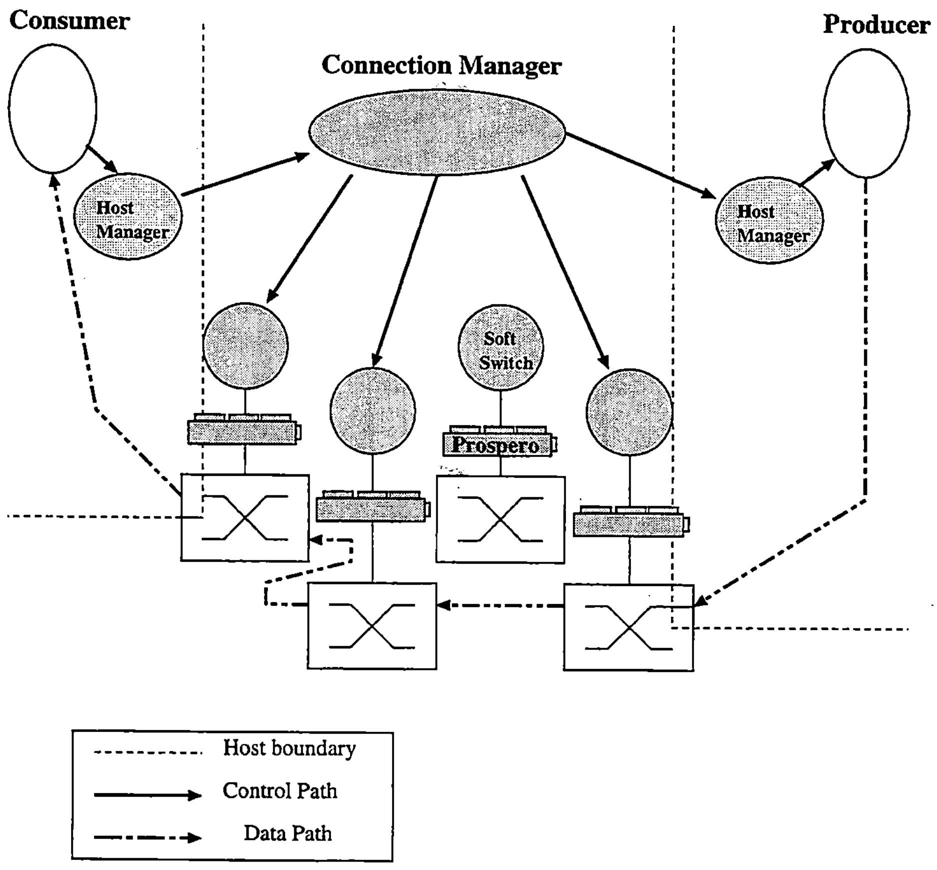


Figure 3.4: Hollowman control path

view of the connection-manager the network is made up of host locations, switch locations and gateway locations. The host locations are instances of the host-manager, the switch locations are soft switches and the gateway locations are instances of a gateway manager which allows Hollowman connection-managers (and as Chapter 7 shows, control architectures) to be interconnected. The explanation of how the function of the connection-manager can be distributed using gateways is reserved until Chapter 7.

The connection-manager has complete knowledge of the topology of its Tempest virtual network. In the current implementation it acquires this knowledge from a set of configuration files; an automatic discovery mechanism is a subject for future work. The Hollowman connection-manager has much the same role as the connection module in a normal switch, except that there is no need for the connection-manager/switch fabric mapping to be one-to-one. This allows network operators to trade-off the risks of centralisation, e.g less robustness, against the reduction in complexity that it permits.

An application uses its host-manager to perform control operations. The host-manager, after doing some local verification, forwards this request to the connection-manager, which then contacts the other entities concerned in the operation and synchronises their activity. For example, when a host-manager forwards a connection creation operation, the connection-manager determines the source and sink hosts and the sequence of switches which constitute a route between them. It then uses the host-managers and soft switch entities to establish that connection.

In order to route, the connection-manager needs accurate information about the state of the virtual network. When there is a single connection-manager per domain, then this state can be determined simply from the complete set of soft switches that the connection-manager coordinates. When there are many connection-managers, they must exchange routing information amongst themselves. This is explained in Chapter 7. In ATMF signalling it is the ingress switch which determines the complete set of nodes to the egress switch, all switch control software must be capable of performing routing functions. All switches must be updated in order to introduce a new routing algorithm, influencing the time-scale at which innovation can take place. In the Hollowman, the routing algorithms are separated from both the physical and soft switch and new algorithms can easily be introduced and tested.

Currently the default routing 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 an intrinsic part of the control policy of the soft switch. Within the Hollowman any of the diverse techniques described in [**Lee95**] for resource constrained routing can 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 that owns the connection has failed.

The demand for connection creation necessitates the modification of state in at least four places: the source host-manager, the sink host-manager, the connection-manager and the switches. When an attempt to create a connection fails after state has been already updated in one or more of the above, then all the updates must be undone and the original state restored. The creation of a connection is a distributed transaction which can be rolled back if the procedure fails at any stage. This problem is 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. Section 4.6 has a fuller discussion of the issues involved in ensuring coherence in control operations.

3.5 Host Addressing

An address identifies a network location. In the ATM standards [**ATMF95a**] a distinction is made between public ATM addresses which are 15-digit E.164 [**ITU-T91**] format telephone numbers and private addresses which are 20-octet ISO Network Service Access Point (NSAP) [**ISO/IEC95a**] formated. There are three different classes of private address corresponding to different authorising entities as distinguished by the Authorising and Format Identifier (AFI) field in the address. Other classes are possible by mutual agreement between involved parties. The complexity of addressing within the standards is as a direct result of their all encompassing nature.

Hollowman addresses are byte arrays; the Hollowman places no constraints on the structure of the data contained in the address other than that the array be non-empty. During the initialisation of the Hollowman a unique address is assigned to every host in the Hollowman's domain. In the current implementation, uniqueness is guaranteed by using the host-name of the machine as its address[5].

Hollowman addressing is much less elaborate than that in the ATMF standards because the Hollowman is not a single ubiquitous control architecture. This demonstrates how simplifying assumptions, appropriate in certain contexts, allow control architecture complexity to be reduced. Moreover, this absence of structure means that any format can be used for a valid Hollowman address. Chapter 7 describes how this helps the interoperation between the Hollowman and other control architectures.

[5]Note that IP addresses would serve equally well for Hollowman addressing and would be *universally* unique.

3.6 Application Naming

The name of an application instance identifies that instance uniquely in the context of a Hollowman domain. In the rest of this dissertation an application instance's name is termed an *application identifier.* Application instances are not tightly bound to hosts and therefore application identifiers are kept distinct from host addresses.

Application identifiers are used within the Hollowman to distinguish between application instances during control operations. They also permit resource bookkeeping and some limited access control. In the current implementation, an application identifier is a single integer. The application instance obtains this unique integer from the host-trader application explained in the next section.

3.7 Trading

A trader is an application that maintains a set of available service offers within a given scope. Trading [**ITU/ISO97**] is a well understood concept and has existed in Distributed Processing Environments such as ANSA [**APM92**] for some time. Traders are not core Hollowman services, because they are not directly involved in the management of connections, but they are a useful auxiliary service. Traders serve two distinct functions within the Hollowman:

- they enable the Hollowman core services and Hollowman end-user applications to get in contact with each other;
- they allow end-user applications to learn about the ATM information producing and consuming services currently being offered by other applications.

The former is an artifact of the current implementation and is transparent to end users. The latter is a facility offered to end-users which they are not obliged to use. For reasons of simplicity, in both cases service offers are used to advertise the services. Service offers used for ATM signalling can be distinguished from the others by the fact that they have ATM as one of the allowed transports.

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 scope as the requester 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. Within the current implementation of the control architecture two levels of trading exist, a trader for each host — *host-trader* — and a trader which federates all these traders — *federated-trader*. The host-traders maintain service offers for end-user applications and the federated-trader maintains service offers for the host-trader services themselves. The host-trader and host-manager are the only two services directly addressed by an end-user application. Applications advertise their services and learn of the existence of other services through their host-trader.

Within the ATM standards end-points can register the fact that they support an *anycast* [**ATMF95a**] address; an application can signal to an anycast address without knowing the precise end-point on which the corresponding service is implemented and the UNI control architecture routes to the closest. The distinct concepts of location and service are mixed in an anycast address. Within the Hollowman, when an application receives a service offer for a given service type from the trader, it is not aware of where that service is implemented. Moreover, because of the hierarchical nature of the traders, it will receive a reference to the closest available offer.

3.8 End-User Application

An end-user application, after obtaining an application identifier from its host-trader, may perform control operations by communicating with its host-manager. In every control operation the application supplies its application identifier, allowing the host-manager to account the resources to the application and perform some basic access control.

The direction in which data flows in a connection is independent of the location that initiates the connection creation. A data producer, a data consumer or another party may request the creation of a connection. Within the current implementation all connections are uni-directional; setting up bi-directional communication requires the creation of two separate connections. This accords with the principle of a small useful set of functions described in Section 3.1. Section 4.4 discusses how more complex connections (or calls) can be supported within the Hollowman.

An application may assume one of two roles during connection creation: ser-

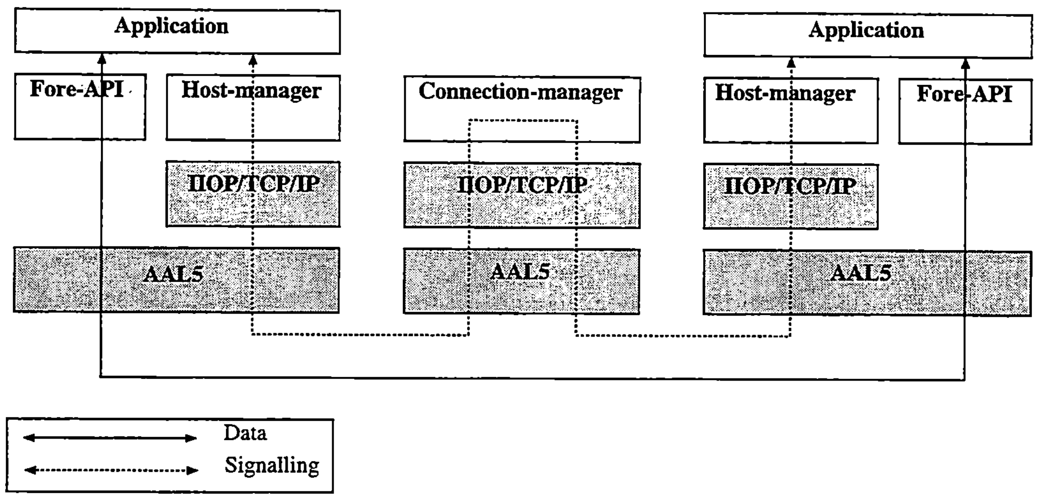


Figure 3.5: Protocol stack

vice user or service provider. Service users may identify the service with which they wish to communicate by supplying a service offer. The application may have received this service offer from its trader or it may have another origin. A service offer must have ATM as one of its allowed transports if the host-manager is to use it to create an ATM connection.

Service providers can publicise the existence of an ATM service to the other applications in the Hollowman by registering offers with their host-trader. A typical service would be the ability to produce a video stream from an ATM camera or to display a video stream on some monitor. The direction of the data stream from the service is determined by whether the offer is a source or sink. When an application registers an offer, it supplies an interface reference of a *callback* service, which is then associated with the offer in the host-manager. The callback service is the means by which the Hollowman informs applications about the state of a connection. The callback interface contains an operation — do-It — which is invoked when a connection is established to a service offer. The implementation of the do-It operation is the application-specific response to a connection creation.

An application can make a service offer without that service being operational. For example, a service provider that controls an ATM camera can offer a video service without initiating a video stream from the camera; it is only when the do-It operation is called that the video service is started. The do-It operation could be extended so that additional information about the connection, e.g. cell rate, could be passed to the application. This would help the application to estimate the operating system resources required to accomplish its function. This is the subject of future work.

If the connection-manager and host-manager are thought of as protocol layers then the protocol stack for communication between applications is represented by Figure 3.5.

3.9 Implementation

The Hollowman is written in the C/C++ programming language. It executes on Sun Ultras under Solaris 2.5, HPs running HP-UX 9.0 and DEC Alphas running digital UNIX. Applications use the Fore libraries for data transfer, but not for signalling.

A variety of implementations of Ariel have been achieved by Jacobus van der Merwe [**van der Merwe97**] including:

- Scotty [**Schoenwelder96**]— a Tcl implementation of SNMP;
- a GSMP [**Newman96**] implementation;
- a CORBA [**OMG95b**] implementation;
- a proprietary message passing implementation.

These varied in efficiency and generality. The author performed one implementation using a native implementation of SNMP [**Hardaker97**]. It was hoped that this would allow more efficient connection set up than that performed using Scotty. Although this implementation was twice as fast as the Scotty one, it was still extremely slow compared with other implementations. SNMP was useful for the initial implementations of Ariel, since nearly all switches support it. However it is too slow for signalling, both because the creation, transmission and interpretation of ASN.1 PDUs is slow, and because many SNMP operations are required to perform simple control functions, such as the creation of a virtual channel. GSMP is the preferred implementation for Ariel when the switch controller is executing remotely from the switch. Ariel is easily mapped onto GSMP, and GSMP is both efficient and likely to be available on many commercial ATM switches. Although GSMP currently lacks support for allocating a complete range of resources to a connection, proposed extensions to GSMP [**Newman97a**] will resolve this problem[6].

The Dimma framework ORB [**Li95**] is used for the communication between distributed entities. Dimma supplies a set of general purpose interfaces for distributed computing which can be mapped onto a variety of distributed computing systems. Dimma 2.0 comes with an implementation of an Internet Inter-ORB Protocol (IIOP) protocol stack and a Basic Object Adaptor (BOA) [**OMG95b**]. The traders, host-managers and connection-managers are all applications within the DPE. The interfaces of the services they support are defined using the CORBA Interface Definition Language (IDL) and communication exchanges take place using IIOP.

The connection-manager, host-manager and traders each require under 3.5 megabytes of virtual memory; most of this is due to the DPE. The Hollowman has been used to create several million connections in one session; the memory occupancy does not rise. The performance of the Hollowman is considered in some depth in Section 5.5.

3.10 Related Work

This section examines the relationship between Hollowman and other control architectures. Within the broader framework of the Tempest all these control architectures can coexist.

3.10.1 Signalling System No. 7 (SS7)

Signalling System No. 7 (SS7) [**ITU-T93b**] is a network of signalling channels used in the Public Switched Telecommunication Network (PSTN) and a suite of protocols for sending control information across this network. The nodes of this network are:

- Service Switching Points (SSP), where data circuits are switched;

[6]Some other ways of extending GSMP have also been proposed. [**Murthy97**] suggests the addition of the standard ATMF resource parameters to the GSMP `ADD BRANCH` message, while [**Adam97b**] extends GSMP through the use of the schedulable region concept, explained in Section 3.2.4.

- Signalling Transfer Points (STP), for the routing and transfer of signalling messages;
- Service Control Points (SCP), servers which perform advanced control operations.

The SS7 suite of protocols includes the Telephone User Part (TUP) for Telephone Signalling, the ISDN User Part (ISUP) for ISDN signalling and the Transaction Capability Part (TCAP). TCAP is used for the exchange of non-circuit related data, e.g. routing information for free-phone numbers, and is mainly used for Intelligent Network (IN) services.

The ITU-T and the ATM Forum have used SS7 as their model for ATM signalling. For example, ATM signalling in the PSTN is performed using B-ISUP which is an extension of ISUP. While this is understandable in the context of the PSTN, this dissertation argues that telephony signalling is not a suitable model for general purpose multi-service networks.

3.10.2 Intelligent Networks (IN)

Intelligent Networks (IN) [**ITU-T92a**] allows the introduction of services other than basic telephony into the PSTN. Examples of IN services include free-phone, call forwarding and universal personal telephone numbers. Initially these services were amalgamated with the switching elements, but the additional complexity in call processing that such services engender and the need for faster introduction meant that a distinction had to be made between the switching plane — sets of Service Switching Points (SSPs) in IN terminology — and the IN control plane — Service Control Points (SCPs). The resulting architecture is called AIN (Advanced Intelligent Networks) [**Garrahan93**]. The interfaces between the SCPs and SSPs are standardised and to an extent the SCP logic can be modified without changing the SSPs.

The aim of making a clearer separation between the switching and controlling plane is shared by the work in this dissertation. However, the AIN model still requires the SSP to have a significant amount of service logic. For example, the ingress SSP has to determine that a call is special, e.g. an 0800 number, and to delegate the treatment of that call to an SCP; the SCP uses the SSP to perform the required intelligent control. This dissertation argues that by defining a lower level interface between the controlling and switching plane, most service logic can be removed from the switching plane. Any call model can then be implemented without modifying the network elements.

3.10.3 ITU-T & ATM Forum

The International Telecommunications Union (ITU) and the ATM Forum (ATMF) are the most important organisations concerned with the standardisation of ATM. The concerns of the ITU's Telecommunication Standardisation Sector (ITU-T) are focused on ATM within the PSTN, while the ATMF has concentrated more on private networks. Their respective standards are aligned. The ATMF and ITU-T have standardised the interaction between an end-system and the network in the User Network Interface (UNI) [**ATMF95a, ITU-T94b**], between different network elements in the Private Network Network Interface (P-NNI) [**ATMF96**] and between public networks in the Broadband Inter-Carrier Interface (B-ICI) [**ATMF95b, ITU-T96**].

The UNI describes two different things: the functions that an end-user can perform and the means by which they can be achieved. The functions have evolved over the successive versions of UNI, the latest being UNI 4.0. The Hollowman's control functions are comparable to the entire suite defined by UNI 4.0. A UNI 4.0 implementation is only *required* to implement point-to-point connections, the rest — leaf initiated join, proxy signalling, anycast, etc. — are optional. UNI/NNI signalling takes place using a special signalling protocol stack, across a virtual channel well known to both the end-system and the network. The signalling protocol's topmost layer is called Q.2931. This defines both the formats and actions of a set of control primitives, and the patterns of exchanges between end-systems and switches required to perform some entire control function. Q.2931 makes use of a Signalling Adaption Layer (SAAL) containing a reliable network-to-network transport protocol called Service-Specific Connection Oriented Protocol (SSCOP). The signalling part of P-NNI is an extension of UNI 4.0. P-NNI also defines how routing information is exchanged between P-NNI nodes.

UNI/NNI signalling is the definitive ATM control architecture and comparisons between it and the Hollowman are made throughout this dissertation.

3.10.4 TINA-C

The Telecommunication Information Network Architecture (TINA) [**Barr93**] Consortium, active since 1993, is a group of software manufactures, hardware vendors and network operators dedicated to defining an integrated management and control system for the B-ISDN. TINA's objective is to have a single model within which the interaction between *all* elements in a public switching multi-service network can be defined. [**Nilsson95**] states that:

The goal of the TINA Architecture is to provide a set of concepts and principles to be applied in the design, processing and operation of telecommunication software.

TINA tackles issues as diverse as the integration of Intelligent Networks (IN), computer operating system resource control and Telecommunication Management Networks (TMN). TINA's *large vision*, while having the advantage of addressing all the concerns of its diverse participants, means that there is a strong emphasis on evolution from existing techniques to avoid losing capital investment, e.g. the integration of SS7 signalling [**ITU-T93b**], Telecommunication Management Networks (TMN) [**ITU-T92b**], Intelligent Networks (IN) [**ITU-T92a**], etc. For example, [**Bloem95**] states in relation to the TINA Connection Management architecture that:

Since Connection Management does not differ in a major way from TMN and OSI Management Developments, easy inter-working and migration from TMN-based management systems is guaranteed.

TINA is all encompassing. This is in contrast with the work described in this dissertation, which argues that trying to embrace all present and future control functions within a single, albeit modular, control architecture is too inflexible for a multi-service network in which new services must be quickly introduced, and too complex to be easily implemented and maintained. However, much of the justification for TINA, e.g. greater flexibility and quicker introduction of services, is resonant with the motivation behind the Tempest. For example, [**Rublin94**][7] states that:

[7][**Rublin94**] defines an antecedent of TINA called INA.

The architecture must enable the network to support a multiplicity of call models and signalling protocols.

Although this aim is shared by the work presented in this dissertation, the means of attaining it is different. TINA assumes that generality is achieved by defining many high-level building blocks and then defining complete models of how these building blocks will interact to perform all required functions. By separating the description of the function of these building blocks from the method in which they communicate, greater modularity is achieved than is presently possible within the ATMF standards. However, the patterns of interactions between the building blocks defined by TINA restricts the call models and signalling protocol that it can support. The Tempest shows that true generality is attained by giving both network operators and end-users fine grained access to network resources.

The Hollowman — a specific instance of a Tempest-aware control architecture — shares some features in common with TINA, for example the use of distributed computing for network control. However, the purpose of the Hollowman — the implementation of a small useful set of control functions to demonstrate the practicality of open signalling — is not common with TINA's.

3.10.5 Xbind

[**Lazar96**] describes a framework — called a Binding Model — for the creation of multimedia services within ATM broadband networks. This framework offers a set of open interfaces — the Binding Interface Base (BIB) [**Adam97a**] — that developers can use for developing open services. *Xbind* is an implementation of these interfaces which uses CORBA[8].

The work presented in this dissertation shares with Xbind both its motivation — greater flexibility in ATM control — and the method of achieving this — a switch-independent interface between the physical network and the control plane. The Tempest approach differs from that of Xbind in the manner in which generality is achieved. The only constraint that the Tempest places on a control

[8]It is interesting to note that while both Xbind and TINA implicitly assume that there is a distinct IP control architecture running in their environment — as both specify the use of standard CORBA transports for communication between control entities — neither of them explains how this fits with the rest of their control model.

architecture is that it must use Ariel to perform its control operations; everything else is up to the control architecture, e.g. the method of communication between control entities, the application API. A control architecture built using Xbind is constrained by the interfaces of the Binding Interface Base (BIB) in how it interacts with the physical network. For example, the *SchedulableRegion* BIB interface assumes that the control architecture will be performing Call Admission Control (CAC) in the way specified by the Binding Model. As Section 3.2.4 has explained, the *SchedulableRegion* interface may not be suitable for certain control architectures. In addition, the requirements of the BIB are such that existing implementations of control architectures, e.g. UNI signalling, would need to be completely rewritten to conform to the model.

The Binding Model's objective is to be so general that any user can implement his desired control within that model; the Tempest approach is to allow users controlled access to their own resources at a low enough level that they can define their own model. Since one model of signalling, that of the ATMF, is already well established and others have been implemented both commercially [**Newman97c**] and in research laboratories [**Arango93**, **TINA-C97**, **Kalmanek97**, **Hjálmtýsson97**, **Rooney97a**] as well as Xbind itself, it is likely that there will be many different ATM control architectures. Acceptance of open signalling systems will be dependent on their ability to take these multiple models into account.

3.10.6 Xunet

Xunet 2 [**Kalmanek97**] was a prototype nationwide ATM network — active from 1991 until 1996 — linking many laboratories in the USA. It investigated areas related to LAN interconnection over wide area ATM, traffic management in a multi-service network and control and management of large ATM networks.

The Xunet switch controller was designed as a set of distributed services, e.g. signalling, routing and resource management, which communicate amongst themselves using the ANSAware [**APM92**] DPE. For convenience, the switch controller ran remotely from the switch and communication between controllers took place across an Ethernet, using a fabric-independent protocol. This fabric-independent protocol was based on an abstract model of the switch, that is to say it defined a set of functions general to many switches, e.g. for modifying entries in the header translation tables, assigning weights to virtual circuits serviced by a weighted round-robin scheduling algorithm. Its implementation depended on knowing the capacity of the switch and being able to map the abstract operations onto its internal interfaces. The switches that Xunet used were built in-house and so this posed no problem. In contrast, Ariel controls commercial switches and the development of the Hollowman required no special knowledge of the switches' implementations.

In Xunet, the relationship between switch controller and switch is one-to-one. and adjacent switch controllers exchange control messages across dedicated PVCs established at start-of-day. The Hollowman allows a looser coupling between the control plane and the physical network, e.g. a given Hollowman connection-manager can manage many switches if it is appropriate to do so. Moreover, Xunet treated only simplex connections. The basic Hollowman control architectures allows both unicast and multicast connections, established by a source or sink or some other third-party. The extensions to the Hollowman, explained in Chapter 6 allow end-users to dynamically introduce their own call types into the control architecture.

Xunet demonstrated the practicality and utility of using distributed processing techniques in a large network. The authors of [**Kalmanek97**], while recommending these techniques, note that overuse of RPCs leads to an unacceptably large overhead in the creation of connections. In Xunet, *each* switch controller was comprised of processes for signalling, routing, resource management, trading and acting as hardware proxies, as well as some other processes for Operation and Maintenance. All of these processes communicated amongst themselves using RPCs in order to establish a virtual circuit across a given switch. The Hollowman is more frugal in its use of RPC and this is reflected in the performance figures given in Section 5.5.

The authors of [**Kalmanek97**] recognise that many ATM control architectures, differentiated by the environment in which they are required to run, will be needed. They state that:

> *It is clear that such complex protocols [as those defined by the ATMF] are not needed everywhere. For example, a desk area or home network, might use a lightweight signalling protocol, with a concise encoding and good support for the common case.*

The Tempest elegantly manages the coexistence of multiple control architectures.

3.10.7 IP Switching

IP switching [**Newman97c**] is a commercially developed control architecture of IP routers running over ATM switches. The routers can optimise the throughput of long lived IP flows by mapping them onto ATM connections. The packets are switched in the ATM switch rather than in the IP router, increasing efficiency. The IP header information can be removed from the packet as it enters the ATM connection and added when it leaves, reducing the amount of information that needs to be transported. Advantages are also accrued from being able to map IP multicast directly onto ATM multicast. [**Rekhter96**] proposes a similar technique called Tag switching.

IP switching requires two protocols: the General Switch Management Protocol (GSMP) which allows the IP switch controller to create and remove virtual channels on the switch and the Ipsilon Flow Management Protocol (IFMP) which allows adjacent switch controllers to exchange information about the IP flow/virtual channel mapping. An IP switch controller, after identifying a long lived IP flow, requests, using an IFMP message, the downstream controller to start sending the IP packets of that flow on a given virtual channel, rather than the default one. If the next upstream controller does likewise then the switch controller can use GSMP to create a dedicated virtual channel for that flow on its ATM switch.

IP switching demonstrates the flexibility that is engendered by the clear separation of the switching plane from the control plane. However, IP switching, unlike the Hollowman, is not a general purpose ATM signalling protocol; it is dedicated to the transporting of IP packets and signalling is not end-to-end.

3.10.8 UNITE

UNITE [**Hjálmtýsson97**] (the conjunction of UNI and LITE) is an ATM control architecture that reduces the overhead in standards-based ATM signalling. UNITE separates the allocation of one resource, VPI/VCI values, to a connection from the other resources. A connection is established using a lightweight signalling mechanism across a predefined signalling channel. Only after the connection is established does the application specify its exact resource needs. This is performed in-band along the connection using Resource Management (RM) cells. Services which only require best-effort connections avoid the overhead in processing complex signalling messages. UNITE affords low latency for the creation of best-effort connections.

This increase in efficiency is traded against both greater complexity in call admission control and the need for simplifying assumptions in resource constrained routing. VPI/VCIs may be assigned to connections whose resource demands subsequently cannot be satisfied, while connections with fewer requirements might be refused because of the shortage of VPI/VCI space. The two phase allocation means that the 'cranking back' of a signalling message when routing information about remote switches turns out to be out-of-date is also more difficult. Moreover, in order to perform resource constrained routing UNITE assumes that there is a finite number of well-known resource classes. Whether this is generally true is arguable[9]; one of the major reasons for the introduction of ATM is its ability to support a large set of unspecified resource classes, and if this is not needed then the usefulness of ATM is diminished.

Situations may exist — for instance when there is a small number of resource classes, a high proportion of calls do not require resource guarantees and call blocking is rare — in which the assumptions made by UNITE are valid. The Tempest allows innovative control architectures such as UNITE to be used when appropriate[10]. The existence of UNITE demonstrates some of the concerns that the research community has with standards-based ATM signalling.

3.11 Summary

This chapter has introduced the Hollowman exemplar Tempest-aware control architecture. The Hollowman is a simple, but practical, ATM control architecture, efficiently supporting both unicast and multicast connection operations. The development of the Hollowman has served to demonstrate the feasibility of open control, and the practicality of the Tempest framework. A range of related work in the domain of ATM control has been detailed and compared with the Hollowman.

By detailing the role of the core services within the Hollowman and explaining some general techniques related to naming, addressing and trading, this chapter

[9]Although such assumptions can become self fulfilling.

[10]UNITE would require an extended version of Ariel, allowing RM cells to be sent from the fabric to the control plane.

has shown how the clear separation of the control plane from the physical networks allows more flexibility, for example in the evolution of routing algorithms. It has also shown that by abandoning the need for an all encompassing universal control architecture simplifying assumptions can be made, for example in addressing, which greatly reduce complexity. When network operators decide it is appropriate to use these simpler control architectures, gains are achieved both in the efficiency of control operations and the effort required to implement and maintain the control architectures.

The next chapter considers more fully the issues related to performing open control operations by explaining the patterns of interaction between the components introduced in this chapter.

Chapter 4

The Hollowman: Patterns of Communication

This chapter explores how end-user applications initiate the creation and deletion of connections using the Hollowman and how the core Hollowman services interact to perform the required control operations. Having fully explained the structure of the switch-independent control architecture this chapter concludes by examining some problems general to all ATM control architectures, and explains how they are influenced by open control.

4.1 Introduction

Chapter 3 described the services offered by the core Hollowman entities. This chapter details how the host-manager, connection-manager and end-user applications collaborate to perform control operations. Besides basic control operations, the Hollowman supports advanced functions such as multicast group management and third-party connection establishment. After having explained the structure of the Hollowman, the rest of this chapter uses it as an example switch-independent control architecture through which issues general to all ATM control architecture — and their manifestation in those supporting open control — may be better understood. These issues relate to:

- complex call types;
- handling failure;

- synchronising the atomic allocation and liberation of resources.

4.2 Hollowman Initialisation

Before the Hollowman can serve end-users, it must itself be initialised. First the traders are started, then the connection-manager and finally the host-managers. These are now explained.

4.2.1 Trader Initialisation

When started, both federated-traders and host-traders reinitialise themselves from their database of active service offers stored during the last session. The trader determines which offers are no longer valid and removes them. The trader decides if a service offer is still valid by attempting to contact the application which advertised that service offer. The means of contacting an application is defined at the registration of that application; in the current implementation the application supplies an interface reference for a ping service that it supports.

The federated-trader checks that the host-traders are still running at periodic intervals and if not removes them from the federation. The host-traders also periodically check the status of the applications running on the host; a failed application's offers are removed from the trader and its resources are liberated. This is an extension of 'garbage collecting' techniques available in [**APM92**].

4.2.2 Connection-Manager Initialisation

Like any other application, the first thing the connection-manager does during initialisation is to register itself with its host-trader. It then reads from a configuration file a description of the required virtual network and negotiates with the Tempest's network builder to have it created. If this negotiation is successful, the network builder allocates the necessary network resources to the Hollowman instance and returns a set of Ariel interface references that can be used to perform control operations on the switches. The connection-manager uses these to initialise the appropriate soft switches.

4.2.3 Host-Manager Initialisation

A host-manager is started on each host of the Hollowman domain. Host-managers register themselves with their host-trader and then acquire the interface reference for the connection-manager from their trader. After this, the connection-manager gives each host-manager the set of network resources that have been allocated to it. In the current implementation, the host's network resources are a set of Service Access Points (SAP). Each SAP corresponds to a *potential* connection. The SAP may be thought of as a token that the host-manager redeems against a VPI/VCI when it wants to create a connection. A SAP has one of three states: `Free`, `Reserved` and `Active`.

It would be possible to give actual VPI/VCI values to the host-managers rather than tokens. However, letting the connection-manager rather than host-managers decide which VPI/VCI value to use in a connection greatly simplifies the creation of multicast connections.

Once the initialisation has finished, end-user applications may use the Hollowman.

4.3 Control Operations

Host-managers maintain information about the state and owners of the SAPs in the `Source_SAP_Table` and `Sink_SAP_Table` and about the current offers made by applications running on the host in the `Source_Offer_Table`, and `Sink_Offer_Table`. Host-managers support two distinct interfaces:

- the α-*interface* that end-user applications use to initiate control operations;
- the β-*interface* that connection-managers use to signal to host-managers.

The connection-manager orchestrates the activities of other Hollowman entities to perform an entire control operation. Control operations, such as connection creation, are normally initiated by host-managers on behalf of end-user applications. The connection-manager has two interfaces, namely:

- γ-*interface* that the host-manager uses to forward control requests;

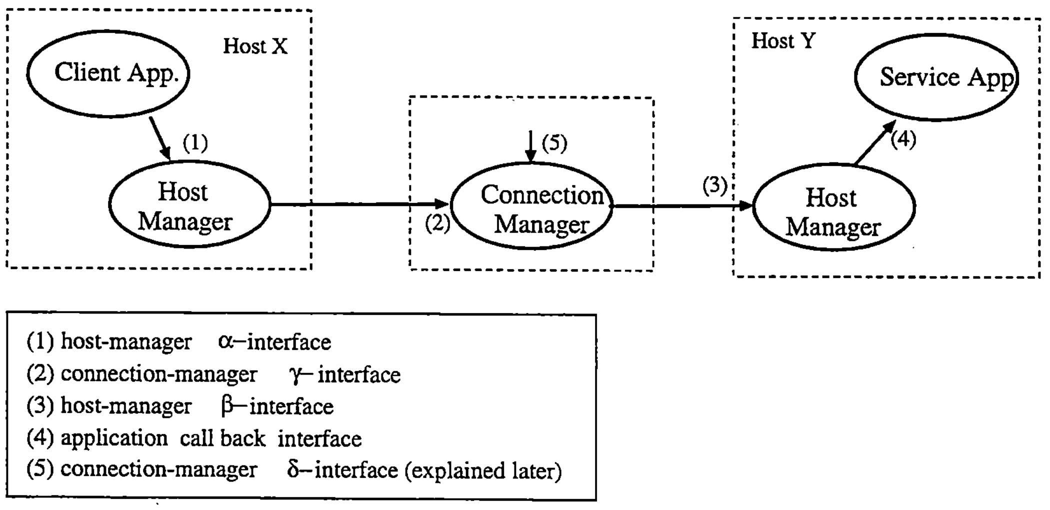

Host X
Client App.
(1)
Host Manager
(2)
(5)
Connection Manager
(3)
Host Y
Service App
(4)
Host Manager
(1) host-manager α–interface
(2) connection-manager γ– interface
(3) host-manager β–interface
(4) application call back interface
(5) connection-manager δ–interface (explained later)

Figure 4.1: The relationship between interfaces

- *δ-interface* used by connection closures. The discussion of the connection-manager's δ-interface is reserved until Chapter 6.

Figure 4.1 shows the relationship between the control interfaces and the clients of those interfaces. The rest of this section explains the purpose of the α, β and γ interfaces and shows how they interact to perform complete control operations.

4.3.1 Host-manager α-interface

The host-manager α-interface is the main Hollowman API. Table 4.1 shows the set of operations that are present in the host-manager α-interface. A Service Access Point (SAP) is an application's means of manipulating a connection. An application that reserves a SAP has that SAP accounted to it within the host-manager. The SAP remains the property of the application until it is freed. All control operations in the API are performed in relation to a SAP. During a control operation, the host-manager:

1. checks that the SAP has the correct state for the operation;
2. checks that the application has the right to perform this operation;
3. forwards the request to the connection-manager;

• getSourceSAP • getSinkSAP
• connectLocalSourceSAPToSinkOffer • connectLocalSinkSAPToSourceOffer • connectSourceSAPToSinkSAP • connectSourceOfferToSinkOffer
• getSourceEndPoint • getSinkEndPoint
• freeSourceSAP • freeSinkSAP

Table 4.1: The host-manager α-interface

4. updates its local state.

A service user typically does the following to create, use and delete a connection:

1. it gets a SAP from the host-manager, using `getSinkSAP` or `getSourceSAP`.
2. it calls `connectLocalSinkSAPToSourceOffer` to create a connection from the source service to itself or `connectLocalSourceSAPToSinkOffer` to create a connection from itself to a remote sink service.
3. it communicates with the service using the SAP. The current implementation is dependent on the Fore API for the actual transmission and reception of data. The primitives `atm_recv` and `atm_send` of the Fore API require VPI/VCI values as arguments. Although it would be possible to encapsulate these primitives in operations which took SAPs rather than VPI/VCI as values, this would require the modification of legacy applications. Instead, the actual VPI/VCI values used in the SAP are recovered by calling `getSourceEndPoint` or `getSinkEndPoint`. This is a layer violation, but in the proof-of-concept implementation it is not overly inhibiting.
4. finally it frees the connection by calling `freeSourceSAP` or `freeSinkSAP`.

Legacy applications may have their own means of learning of the existence of information producers and consumers, other than Hollowman trading. In addition, when the service provider's location is well known to the service user, trading may be inappropriate. The Hollowman allows for these situations by having an additional host-manager operation: `connectSourceSAPToSinkSAP`. This operation permits applications which know the host and SAP that a remote service provider is using to set up a connection to that service provider without the use of advanced features such as traded offers or callbacks. A legacy application that uses the Hollowman transparently is described in Section 5.4.2.

If, when `connectSourceSAPToSinkSAP` is called, both source SAP and sink SAP are on hosts other than that of the application, then the operation is a third-party connection. The equivalent operation for joining a foreign source and sink offer is `connectSourceOfferToSinkOffer`. An example is now given of a typical interaction between a service user and the host-manager.

Example: suppose an application wishes to receive data from an information producing service called `ProducerService` for which it has a service offer. The invocations between the entities are shown in Figure 4.2. The application calls the host-manager `getSinkSAP` operation passing its application identifier **(1)**; if there is a free sink SAP, then the host-manager sets the state of the SAP to `Reserved`, accounts it to the application, and returns the sink SAP identifier to the application.

The application invokes `connectLocalSinkSAPToSourceOffer` **(2)**, passing its application identifier, the SAP identifier and the source offer. The host-manager checks both that the SAP is `Reserved` and that it is accounted to the application. It then calls the connection-manager's `createConnectionToSAP_FromSource` operation **(3)** — explained in Section 4.3.4 — passing the sink SAP identifier and the source offer. If the connection-manager establishes the connection, then the state of the SAP is set to `Active`.

The application recovers the VCI/VPI value associated with the SAP by calling `getSinkEndPoint` **(4)**. It can now use the normal Fore API to start receiving data from the source **(5)**.

At some stage the application calls `freeSinkSAP` **(6)**. The host-manager checks that the SAP belongs to it. If the SAP is in the `Reserved` state, then the host-manager simply sets its state to `Free`, if the SAP is in the `Active` state, then the host-manager invokes the connection-manager's `freeSinkSAP` **(7)** operation to remove the end-to-end connection as well. In both cases the SAP is no longer

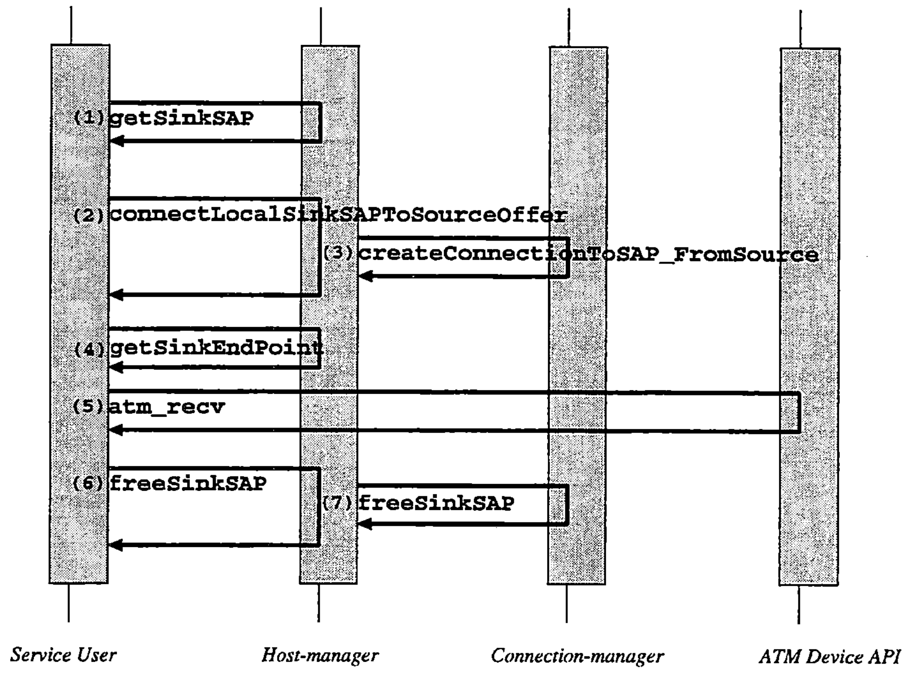


Figure 4.2: Service user interaction

accounted to the application.

4.3.2 Host-manager β-interface

Host-managers control the assignment of SAPs to the applications running on their host. Potentially each host-manager could use a different allocation policy. In the current implementation host-managers, when asked, hand out a SAP if one is available. Future work will explore more complex resource allocation policies covering both network and operating system resources, for example ensuring that an application on a shared server offering a video display service is allocated enough operating resources, e.g. Direct Memory Access (DMA), CPU, framestore etc., to render the video correctly.

The host-manager's β-interface allows a connection-manager to reserve a SAP — using `obtainSourceSAP` or `obtainSinkSAP` — and to inform the host-manager about changes in the state of the SAP — using `notifySourceSAP` or `notifySinkSAP`. Table 4.2 shows the operations that are present in the host-manager β-interface.

• `obtainSourceSAP` • `obtainSinkSAP`
• `notifySourceSAP` • `notifySinkSAP`

Table 4.2: The host-manager β-interface

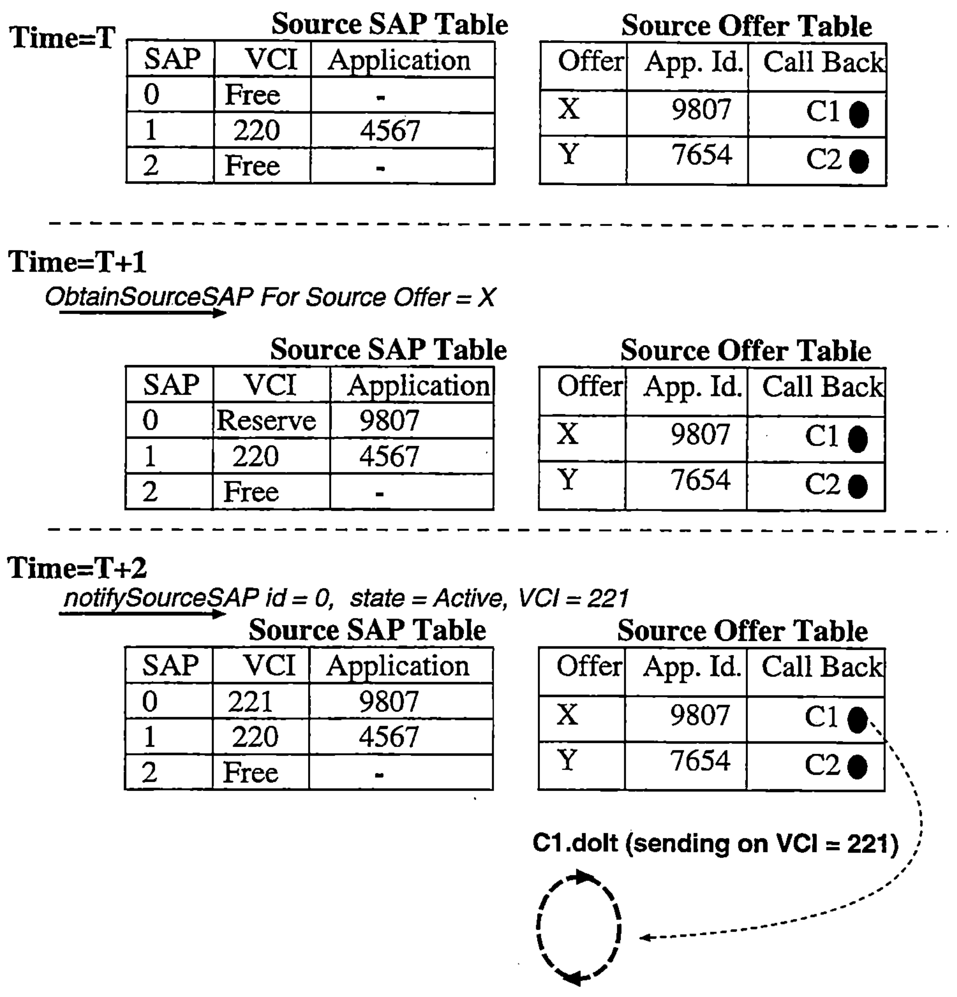
On receiving a SAP reservation request for a service offer, the host-manager:

1. verifies the offer's service provider is registered at that host;
2. finds an available source or sink SAP;
3. accounts the SAP to the service provider;
4. returns the SAP identifier to the connection-manager.

The notification operation can either notify that a SAP is now `Active` or notify that it is `Free`. A notification of the activation of the SAP is accompanied by the VPI/VCI values that are now associated with the SAP by the connection-manager. If the SAP has been reserved for a service offer, then the activation of the SAP provokes the invocation of the callback function associated with that SAP in the host-manager's offer table. If the SAP is not associated with a service offer then no callback is performed.

In the same way, during the notification of the liberation of the SAP, the host-manager checks if there is an associated service offer and if so invokes its callback to inform the service provider of the liberation of the connection. In either case the host-manager updates its view of the SAP state. An example illustrates the changes in state that operations in the β-interface induce on a host-manager.

Example: Figure 4.3 shows the evolution in the state of the host-manager's tables during this example. Suppose an application with identifier `9807` registers the producer service `X` with associated callback interface reference `C1`. Later, the connection-manager requests the reservation of a source SAP for `X`. The host-manager returns the source SAP identifier `0`, sets the state of the SAP to `Reserved` and accounts it to application `9807`.



Source SAP Table

SAP	VCI	Application
0	Free	-
1	220	4567
2	Free	-

Source Offer Table

Offer	App. Id.	Call Back
X	9807	C1 ●
Y	7654	C2 ●

Time=T+1
ObtainSourceSAP For Source Offer = X

Source SAP Table

SAP	VCI	Application
0	Reserve	9807
1	220	4567
2	Free	-

Source Offer Table

Offer	App. Id.	Call Back
X	9807	C1 ●
Y	7654	C2 ●

Time=T+2
notifySourceSAP id = 0, state = Active, VCI = 221

Source SAP Table

SAP	VCI	Application
0	221	9807
1	220	4567
2	Free	-

Source Offer Table

Offer	App. Id.	Call Back
X	9807	C1 ●
Y	7654	C2 ●

Figure 4.3: State changes in service provider's host-manager

After the connection-manager has established the connection, it notifies the host-manager that a connection is in place, passing the VPI/VCI values which define the connection source end-point. The host-manager invokes `C1`, passing these values, and the application may start sending data across the connection.

4.3.3 Connection-manager γ-interface

Table 4.3 shows the operations that are present in the connection-manager's γ-interface. The `getSourceSAPs` and `getSinkSAPs` operations are used by the host-manager at start-of-day to obtain its allocation of SAPs. The `createConnectionToSAP_FromSource`, `createConnectionToSAP_FromSink` and

• `getSourceSAPs` • `getSinkSAPs`
• `createConnectionToSAP_FromSource` • `createConnectionToSAP_FromSink` • `createConnectionBetweenSAP_FromThird`
• `createConnectionToOffer_FromSource` • `createConnectionToOffer_FromSink` • `createConnectionBetweenOffer_FromThird`
• `freeSourceSAP` • `freeSinkSAP`
• `createConnectionClosure`

Table 4.3: The connection-manager γ-interface

`createConnectionBetweenSAP_FromThird` operations which host-managers to signal to SAPs. For each of these creation operations there is an equivalent with a service offer rather than a SAP defining the signalling end-point. The `freeSinkSAP` and `freeSourceSAP` operations allow the deletion of connections. The explanation of the `createConnectionClosure` operation is reserved until Chapter 6.

The action of the connection-manager during the creation and deletion of ATM connections is now considered.

4.3.4 Connection Creation

A host-manager initiates the creation of an ATM connection by calling `createConnectionToOffer_FromSource`, if it is the source's host-manager, or `createConnectionToOffer_FromSink`, if it is the sink's. In either case it supplies a local SAP identifier and a remote service offer.

The connection-manager uses the service offer's address to acquire the interface reference of the service provider's host-manager. First, it examines a cache of host-manager invocation references, using the address as an index. If the interface reference is not in this cache, it calls its host-trader to get the service offer of the appropriate host-manager, establishes an invocation reference and adds it to the cache. In this way, the connection-manager learns dynamically about the existence of host-managers. A penalty is paid for the first signalling request to a given host-manager, but all subsequent requests are efficiently performed by use of the cache. Failure to communicate with a host-manager using a cached invocation reference leads to the removal of that reference from the cache and an attempt to find a valid service offer for that host-manager through trading. Other control entities behave similarly when using the DPE; this allows a given control entity, e.g. a host-manager, to be stopped and restarted without causing the whole control architecture to fail.

The connection-manager then reserves a source or sink SAP for the offer on the service provider's host-manager, using the `obtainSourceSAP` or `obtainSinkSAP` operation of the host-manager's β-interface. If the host-manager refuses this reservation request, e.g. because it has no more available SAPs, the operation fails. Otherwise, the host-manager returns a SAP identifier. The connection-manager then attempts to find a route from source to sink. The route is composed of a sequence of soft switches.

At this stage the connection-manager has received a SAP identifier from both the source and sink parties and determined a route between them. The connection-manager knows the totality of VPI/VCI values that have been assigned to the control architecture. The connection-manager associates the source SAP and sink SAP identifiers with a free VPI/VCI value on the switch ports attached to the source and sink hosts. It then reserves suitable VPI/VCI values on all the other ports in the determined route and requests each soft switch to establish a connection between the VPI/VCI value on an input port and the VPI/VCI value on the output port using the connection service described in Section 3.2.2.

The connection-manager notifies the service provider's host-manager of the connection establishment using either `notifySourceSAP` or `notifySinkSAP`. The reaction of the host-manager to this notification was explained in Section 3.3. If the host-manager accepts the connection creation, then a new network connection is added to the book-keeping records of the connection-manager and the creation request operation returns to the initiating host-manager. When the calling host-manager signals to a SAP rather than a service offer, then the reservation invocation between the connection-manager and the host-manager is not necessary.

In the simple signalling described here, when a consumer signals to an information producer, the producer cannot be sure when the consumer is ready to receive information. For example, if the source starts sending immediately after notification, the sink may still be waiting for the completion of the creation operation and cells will be lost[1]. Better control can be achieved by using the Hollowman to also establish a control channel from the consumer to a *producer manager service* which manages the actual data producer, the consumer can then start and stop the flow of information on the data channel by sending appropriate messages on the control channel.

4.3.5 Connection Removal

The connection-manager maintains and updates the set of active connections. A host-manager calls `freeSinkSAP` or `freeSourceSAP` to free a connection. Each time this is done, the connection-manager:

1. verifies that the SAP corresponds to a connection;
2. frees the resources of the connection within the connection-manager;
3. notifies the appropriate host-manager(s), i.e. if the removal was prompted by the source, it notifies the sink and vice-versa;
4. removes the connection from its records.

Since two host-managers in a given connection — one whose host is a source and the other a sink — may simultaneously attempt to remove that connection it is not an error to try to delete a non-existent connection.

4.3.6 Third-Party Control Operations

A connection may be established by a party other than the source and sink applications of that connection; this is called *third-party signalling*[2]. The third-party application may be on the same host as the source sink or on a completely different host.

[1]Note that as UNI/NNI `CONNECT ACK` messages are not transported end-to-end the same is true of standards-based signalling.

[2]A related but distinct idea is proxy signalling where an end-system such as an ATM camera delegates the responsibility for its signalling to another entity. Proxy signalling is achieved in the Hollowman by running an end-system's host-manager on a separate end-system.

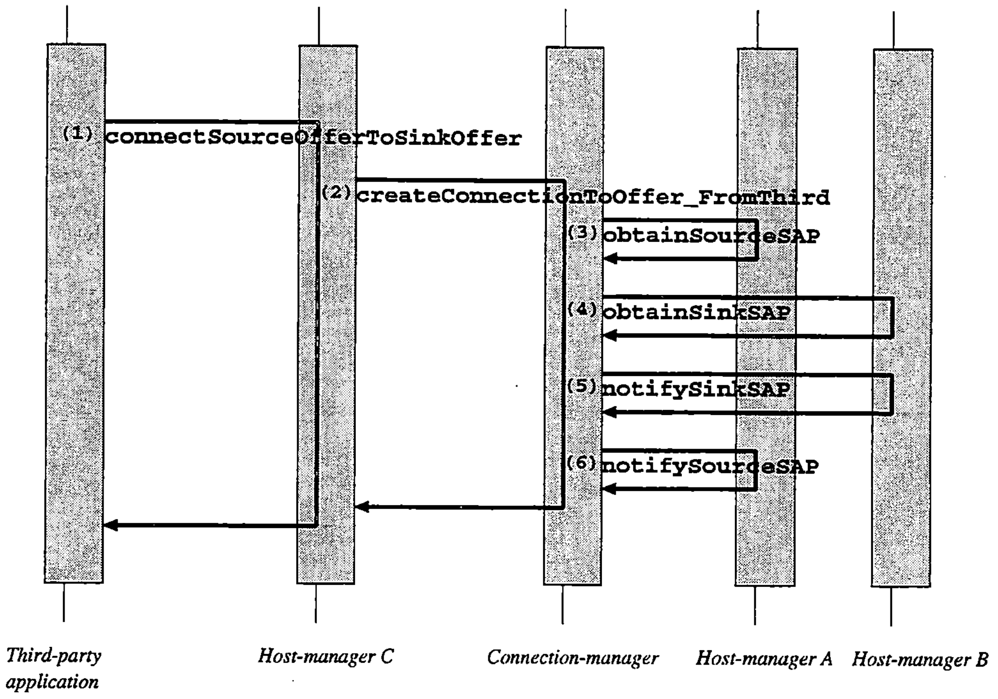


Figure 4.4: Third-party interaction

A third-party application calls the `connectSourceOfferToSinkOffer` operation of its host-manager, passing both a source and sink offer, and its own application identifier. The host-manager checks if either the source or sink offer is on the host-manager's host. If so, then the operation resembles the normal creation operations; otherwise, the host-manager calls the connection-manager's `createConnectionBetweenOffer_FromThird` operation. The effect of this is similar to the operations defined in Section 4.3.4 except that SAPs must be reserved and connection establishment notified on both source and sink host-managers. If the connection is created then the third-party application is returned the identifier of both the source and sink SAPs. An example demonstrates the interaction of the diverse entities during third-party connection establishment; Figure 4.4 shows these interactions diagrammatically.

Example: application `Third-party` running on host `C` wishes to establish a connection between `SourceService` offered by the `Source-party` application on host `A` and the `SinkService` offered by the `Sink-party` on host `B`. `Third-party` calls `connectSourceOfferToSinkOffer` on its host-manager (1), supplying the service offers. The host-manager checks that neither of the offers come from an application resident on `C`. It then calls the createConnectionBetweenOffer_FromThird on the connection-manager (2).

The connection-manager calls obtainSourceSAP (3) on the host-manager of A and obtainSinkSAP (4) on the host-manager of B. It determines a route, establishes the connection using the soft switches, then calls notifySinkSAP (5) and notifySourceSAP (6), each of these performs the appropriate callbacks. The source and sink SAP identifiers are then returned to Third-party.

4.3.7 Multicast Connections

All connections within the Hollowman are multicast connections, i.e. a source can be associated with many sinks; point-to-point connections are just a special case. The Hollowman follows the point-to-multipoint model of the ATMF's multicast [**ATMF95a**], in contrast to the multipoint-to-multipoint [**Deering89**] IP model[3].

From the application's viewpoint, joining and leaving a multicast group in the Hollowman is identical to creating and deleting a point-to-point connection. This contrasts with both IP multicast, where a distinction is made between packets sent to a multicast group and packets sent point-to-point, and standard ATM multicast, where special primitives are defined for joining and leaving a multicast connection.

Service providers, when registering source service offers, specify whether they are multicast offers or not. If an offer is not a multicast offer, then each time a reservation request is made by the connection-manager to the host-manager a distinct source SAP is returned. If, on the other hand, the offer is a multicast offer and there is an Active SAP associated with the offer in the host-manager tables, then that SAP is returned to the connection-manager.

The connection-manager, during connection creation, checks if the source SAP it has acquired from the host-manager is already associated with a network connection. If it is, then the creation operation adds a new branch to the existing multicast connection. When adding a new sink to a multicast connection, the connection-manager selects the location at which to add the new branch to the tree of switch connections. The normal routing algorithm is used to find an en-

[3]The ATMF is defining multipoint-to-point signalling [**Heinanen97**] to take advantage of virtual channel merging. Generalising the Hollowman model to allow multipoint-to-point connections is the subject of future work.

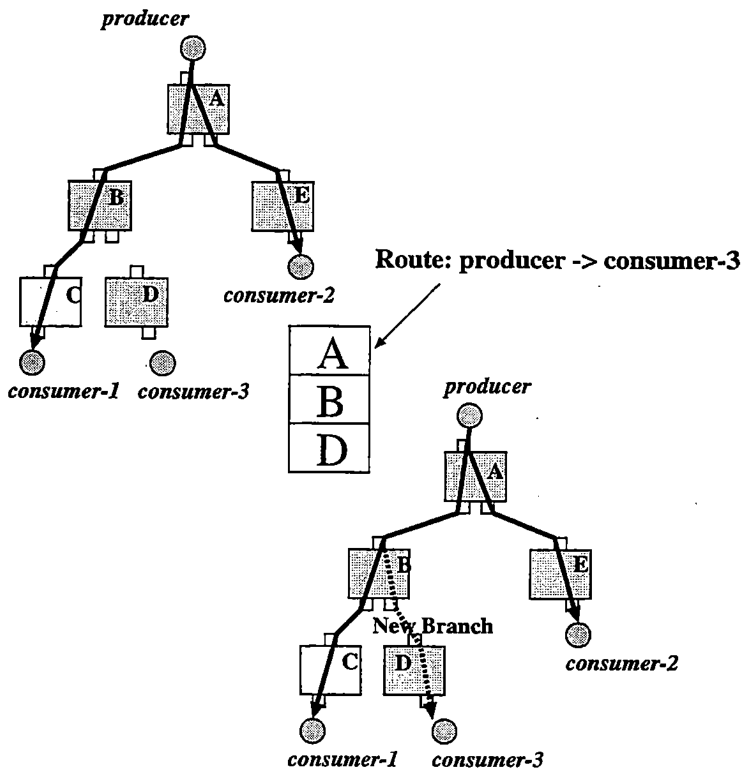


Figure 4.5: Example of multicast connection

tire path between the source and sink. The connection-manager, starting at the ingress soft switch, traverses this path matching elements against switch connections in the multicast tree. It does this until it can no longer find a match. It then creates the connection from the point of divergence to the sink. This simple algorithm allows the efficient creation of suboptimal multicast connections; better results could be attained by the complete reconfiguration of the multicast tree at each join, but this would greatly increase the complexity of the operation. It is arguable whether more sophisticated multicast algorithms are worth the extra complexity; [**Doar93**] demonstrates that such naïve multicast algorithms are only worse than optimal ones by a small factor. When there are multiple connection-managers in a Hollowman domain each one determines if a connection is multicast and where to add the appropriate branch in isolation. This again simplifies the handling of multicast at the cost of suboptimal multicast connection topologies

After adding a new branch to a multicast tree the connection-manager notifies the source host-manager of the addition of a new sink using the `notifySourceSAP` operation. The source host-manager notifies the source application using an operation of the callback interface. Figure 4.5 shows an example where a source, `producer`, is sending data to two sinks, `consumer-1` and `consumer-2`, when a third sink, `consumer-3`, requests a connection to `producer`'s service offer.

A host-manager that removes a sink from the multicast connection removes the branch of the connection that was dedicated to that sink; if the tree only has one branch then the entire connection is removed and the source notified. When the source of a multicast connection is freed, the entire connection is removed and all the sinks are notified.

In the Hollowman, as in UNI 4.0, multicast trees are formed by adding branches one at a time. The next section discusses how more complex operations, for example the simultaneous creation of a complete multicast tree, are handled within the Hollowman.

4.4 Call Types

Groups of related connections are often termed a *call*. A call type defines the connections and relationship between connections that constitute the call. The Hollowman does not directly support call types; the Hollowman implements a small set of functions which allow it to demonstrate the advantages of open control. The point-to-point, point-to-multipoint and third-party connection creation operations described in this chapter are adequate for this demonstration.

The Hollowman is flexible enough that it could easily be enhanced to support more complex connection types. As an example, within the ATM standards endpoints may register themselves as supporting a *groupcast* [**ATMF95a**] address; when an application signals to a groupcast address a connection is created to all the members of the group. By adding a flag to the Hollowman host-manager α-operation `connectLocalSourceSAPToSinkOffer` indicating that the connection-manager is to create a connection to all hosts that support a sink offer of the same type, the simultaneous creation of an entire multicast connection would be possible. In this solution, the connection-manager could gather the offers by asking all traders within some federation and would then use a multicast routing algorithm to determine the topology of the resulting connection tree.

More generally, defining a call type description language would allow an end-user application to specify arbitrarily complex calls. [**Minzer91**] describes such a call description language. Although some reflections was given to extending the Hollowman in this way, experience with implementing applications to demonstrate the power of open control suggested that simply giving end-users the ability to define complex meshes of connection was not in itself enough. The end-users should also be able to exercise a control policy over their calls. This led to the design of connection closures, described in Chapter 6, which allow users to introduce dynamically their own call types along with the control policy to manage those calls. Some end-user applications may require calls without needing the ability to define control policies for them. The implementation of the more general connection-closure concept shows that adding such calls to the Hollowman would pose no major problem.

4.5 Cleaning up Application Resources

End-user applications may fail while still holding network connections. The applications never free those connections and the Hollowman itself must do so.

As Section 4.2.1 explained, after an application registers with its trader, the trader periodically checks that the application is still there and if not removes its current offers. In line with the principle of simplicity, this same mechanism is used to clean up any network resources allocated to a failed application. If the application fails, the trader removes all service offers for the application and prompts the host-manager to remove all connections for which the application is either source or sink.

4.6 Atomic Control Operations

A connection is a set of distributed resources. Connection creation is the act of allocating the required resources on a set of network devices. There may be several possible acceptable outcomes to a creation request. For example, in a groupcast connection creation request it might be desirable that as many members of the group are connected as possible. However, if not all the resources can be assigned in such a way as to satisfy one of these acceptable configurations, then none should be assigned.

A location may refuse connection creation when resources are already assigned to that connection at other locations. For example, within the Hollowman, when creating a connection across a single switch between a source and sink application, SAPs may have been successfully allocated on the source and sink host and VPI/VCI values on the switch ports, only to have one of the hosts refuse the notification of the connection creation. The host-manager may have changed its priorities since the SAP was allocated or the application may refuse the connection for application-specific reasons.

When this happens all the resources associated with the connection must be liberated and the state of the control architecture returned to that which preceded the demand for the connection creation. This would be simple to achieve if only one connection creation or deletion request were active at a time, i.e. if the whole control architecture were *locked* during a control operation. However, this would mean that applications setting up connections concurrently would slow each other down significantly.

Reducing the time during which parts of the control architecture are locked increases control throughput, but increases the complexity of handling concurrent access. One solution is the use of distributed transactions [**ISO/IEC95b**] such that at the initiation of the create request at each host-manager, a transaction is started for that request. All state changing operations across the control architecture are made within the context of that transaction and it is only when all state changes have been successfully made that the transaction is committed. In the OMG standard for distributed transactions [**OMG95a**] an Object Transaction Server (OTS) coordinates the distributed entities in the modification of state. The distributed entities register the completion of their operation with the OTS. When the OTS has received a successful reply from all the participating parties it tells them to commit. If there are `N` entities in the transaction then 2 × `N` *extra* network invocations are required; this is a significant overhead.

[**Kramer92**, **Ranson95**] both recognise the efficiency problem in applying classical transaction systems to network control. Both suggest weakening the transactional model in order to make it more suitable for fast connection creation, e.g. optimistically use a single phase commit, rather than two phase, with corrective action if necessary. The consistency of persistent state is not as important within a control architecture as in a classic database model. Of more importance is rolling back from failure and ensuring consistency during concurrent operations.

In the Hollowman, each control entity, e.g. connection-manager, is responsible for returning itself to a consistent state after a control operation has failed and of informing all other concerned entities. An *ad hoc* transaction system is achieved by dividing all control operations into a set of sub-operations. Any sub-operation may fail during a request, resulting in the raising of an exception. The encapsulating operation catches the exception and undoes the effect that the operation has had up until that point. Consistency is ensured by associating a set of locks and a pre-condition with the sub-operation. The thread assigned to the control operation obtains access to the locks then checks the pre-condition. If the pre-condition is not true then the operation fails. For example requests may be active to both delete a multicast tree and add a branch to it; the sub-operation that joins the new branch to the multicast connection must check whether the connection still exists. At the end of the sub-operation, the thread releases all the locks it holds. Deadlock is avoided by ensuring that the order in which threads acquire locks within a given entity is fixed and that a thread cannot hold a lock while performing an RPC.

Determining the necessary set of pre-conditions for each sub-operation and the required actions needed to undo an operation is complex. The current implementation reduces the complexity by being conservative; better control throughput would be expected with a less conservative approach. Although the *ad hoc* methods adopted for ensuring atomic synchronisation within the Hollowman are sufficient for its purpose, they are not generally applicable. The following section discusses the problem in more detail.

4.6.1 Discussion of Atomic Control

The various user parts of SS7 — TUP, ISUP, B-ISUP — and UNI/NNI signalling all use essentially the same model for coordinating distributed entities in the creation of connections. Each controller reserves some resources, forwards the connection request upstream, and waits for an acknowledgement confirming the connection's creation or a release message requiring that the operation be rolled back. In regard to the PSTN, [**Kuhn97**] states that:

> *Paradoxically, this seemingly trivial task requires some of the most complex and sophisticated computing systems in existence. Software for a switch with even a relatively small set of features may comprise several million lines of code.*

The author of [**Kuhn97**] goes on to point out that despite superfluous hardware, extensive self-checking, recovery software, and extremely predictable call patterns, availability of the network is not total; approximately two thousand minutes of outage were experienced by some customers between April 1992 and March 1994 in the United States' PSTN and that almost half of that was caused by overloading.

[**Kant97**] suggests that the nature of multi-service network calls differs from that of calls in the PSTN in that:

- the call processing time at each node is likely to be longer, as the call types are more complex;
- call-blocking is likely to occur more often as traffic patterns are less predictable and end-points are more intelligent;
- the call topologies are more complex and more varied.

All of these factors mean that coordinating the management of distributed resources across the nodes of a multi-service network is more of an issue than in the current PSTN. This observation is born out by practical experience with implementing ATM control systems. For example, [**McMahon81**] mentions in the description of the implementation of the software for the Datakit Virtual Circuit Switching (VCS) switch, that: *the hard part of the [implementation of the switch] controller is to manage many connections in various states at once*; [**Newman97b**] states that the majority of code in an ATM signalling system is dedicated to recovery from failure. Experience with building a signalling system, even one as simple as the Hollowman, suggests that synchronising distributed resources is the key problem.

To an extent, the problem can be mitigated by reducing the amount of distribution. [**Veeraraghavan95**] affirms that the complexity of protocol engines supporting B-ISUP is as a result of their need to handle refusal from end-points and proposes the centralisation of the response from multiple end-points into one entity — *the connection server* — to simplify the atomic creation and deletion of connections. Experience with the Hollowman suggests that while this does reduce the complexity of the signalling protocol, the centralisation requires an efficient transaction system to prevent this central server becoming a bottleneck. Moreover, centralising the coordination of resource allocation reduces the complexity of the system, but introduces single points of failure and makes the network less robust. Network operators should be able to decide on the degree to which they are prepared to trade simplicity against robustness.

It might be tempting to identify the complexity of ATM control in this regard as a reflection of a fundamental weakness of the technology as a whole. Packet forwarding mechanisms such as IP which make packet forwarding decision at each node for each packet are inherently more adaptive than those such as ATM which require the creation of a connection before data can be sent. For example, after reception of an ICMP packet indicating router outage, all upstream routers send all further packets across an alternative route if one is available. This is implicit in IP, while in ATM it requires some separate error recover mechanism. Standard IP [**DARPA81**] cannot give resource guarantees to applications, excluding it from being a general solution to the transport of continuous media. Solutions to rectify this failing [**Zhang93**] have involved the addition of the concept of connections, called IP *flows*, and a signalling protocol ReSerVation Protocol (RSVP) to the IP model. RSVP was designed for small video-conferencing applications and there is some debate as to whether it is or should be the general IP signalling protocol [**Henning97**]. However, the 'ATMisation' of IP seems set to present the IP community with the same problems, in regard to the synchronisation of complex flow state in a highly distributed environment, as those discussed in this section.

In summary, the problem of implementing efficient atomic control actions in a large multi-service is still an open issue and worthy of further investigation. It seems likely that there will not be one 'right solution' but any number of different solutions appropriate in given contexts. This demonstrates again the need to customise control architectures and adds further motivation for the Tempest framework.

4.7 Summary

This chapter has explained in some detail how the Hollowman's control operations are performed. The Hollowman has no pretension to replace ATMF signalling; it was designed and implemented to demonstrate that open control is a practical technique. The Hollowman, while simple, is realistic enough for this purpose. The next chapter will describe some of the applications that the Hollowman supports, further motivating its realistic nature.

The description of the Hollowman's implementation has allowed some problems related to ATM control, such as atomic synchronisation and handling complex call types, to be better understood. These are general problems to all ATM control architecture, but their manifestation is affected by the use of open control. As is shown in the next chapter, the flexibility of the Hollowman makes it an ideal environment in which to carry out experiments in ATM control. Open control, by allowing parties other than switch vendors to implement control architectures, will increase the creativity that can be brought to bear upon these problem and expedite the finding of solutions.

In the long term, the Hollowman will become a component library with which network operators can implement simple proprietary (or small subsets of standard) control architectures. These control architectures can coexist with standard ones within the Tempest framework. This allows non-monolithic solutions to the problem of implementing control architectures.

Chapter 5

Experiments in ATM Control

This chapter demonstrates the flexibility of the Hollowman control architecture by detailing some of the control experiments that have been carried out using it. In addition, the performance of the Hollowman is compared against that of a range of other control architectures.

5.1 Caching Connections

A control architecture may decide not to remove a given connection between two hosts if there are frequent requests for connections of that type between those hosts; the connection is then said to be *cached.* A cached connection can be reused the next time a request is received for a connection of the same type, reducing the latency of the set up time.

The Hollowman was used to build an experimental implementation of connection caching. Although 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.

Figure 5.1 shows an example in which the host-manager on a host `A` requests a connection from host `B` to `A`. Previously a connection from `B` to `A` had been created and cached. From the viewpoint of the two host-managers the sequence of operations is normal. However, the connection-manager identifies the connection as cached and simply looks up the values to associate with the SAPs from the cache table.

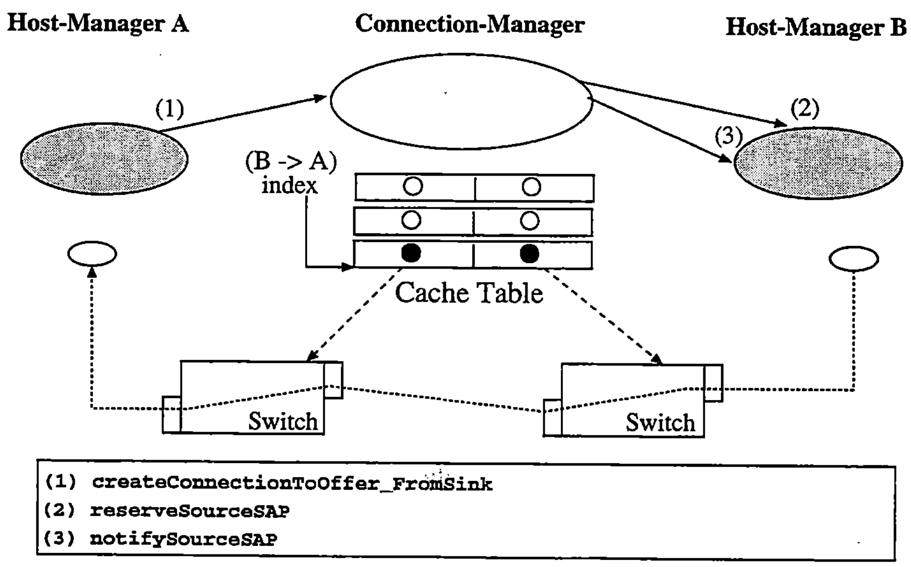


Figure 5.1: Cached connection

Connection caching allows extremely short signalling latencies as there is little or no need for any communication between the control architecture and the physical switch. Section 5.5 details the performance of the signalling using the Hollowman and describes the relative importance of the various factors which contribute to signalling latency.

Whether a connection should be cached or not depends upon the nature of the applications that are using the Hollowman. This makes the use of connection caching problematic within a generic control architecture, but promising for service-specific control architectures such as those described in Section 5.4.3. Preallocation of connections is also possible if the control architecture can make a reasonable guess about the patterns of communication of the applications using it. [**Lazar97**] has recently proposed a similar technique.

5.2 Persistent Connections

At termination, a Tempest control architecture loses its virtual network and information about all connections is lost. The network builder releases all the connections still in place when a virtual network is liberated.

A *persistent* connection is one that persists between executions of a control architecture. The Hollowman was used to experiment with persistent connections. In the implementation, the API was enhanced so that applications, during a creation request, could mark a connection as persistent. At the next execution of the Hollowman, all persistent connection are recreated. This is useful because it allows:

- applications to assume the existence of certain connections after the connection-manager has started executing; this is analogous to the use of PVCs in the standard ATM control architecture;
- the network operator to create complex connection topologies without doing this connection-by-connection;
- the Hollowman to be restarted, after failure, in the same state as it was when it failed. In the experimental environment this is helpful in error detection.

In the experimental implementation, information pertaining to a persistent connection is stored in a file in symbolic form. Each branch of a multicast connection is represented by a record containing, for both source and sink, the SAP identifiers and the host addresses. A complete multicast connection is described by a set of such records. The records are added to the permanent store in the order that the branches are created. The deletion of a persistent connection marks it as deleted in the file and at the next execution of the Hollowman it is removed from the file and not recreated.

During initialisation the host-manager acquires its network resources from the connection-manager. Some of the SAPs that the host-manager receives may already have been associated with an ATM end-point, i.e. they are end-points in permanent connections. The host-manager updates its local state accordingly.

5.3 Asynchronous Communication with Switch

In the Hollowman, during connection creation the connection-manager locks a part of its state, communicates with the switch, updates its local state and then releases the lock. For example, when adding a branch to a multicast connection, the connection-manager locks its representation of the network connection, asks the switch to create a switch connection, then if this is successful it adds the new branch to the network connection, before releasing the lock. The connection-manager blocks until the switch has replied or a time limit has been exceeded. When the connection-manager is physically removed from the switch this communication takes place over the network, implying some latency. If many connection requests arrive concurrently, all requiring access to the same piece of locked state, then the latency in communication with the switch will affect control throughput.

Since the control architecture has a complete view of the state of resources in the switch, once the control architecture has decided that, for instance, a create operation is valid, then the only reason why the switch would refuse the connection would be in the case of switch failure. This is likely to be rare. In general, there is no need to wait until the switch has confirmed what the control architecture has asked it to do. Moreover, if the control architecture does not wait for each switch to confirm the success of an operation, then the different switches involved in the connection can create it concurrently, further optimising the connection creation time.

In order to test asynchronous communication with the switch, an experimental implementation was carried out using the Hollowman. In this experiment, asynchronous communication was achieved by dispatching a separate thread in the connection-manager to perform the invocation. All the invocation threads shared an array of variables with three states: `Waiting`, `Confirmed` and `Failed`. Each thread modified one and only one of the elements in the array. All the variables of the array were initially set to `Waiting`. When a thread received a successful reply from its switch, it set its variable to `Confirmed`. If, on the other hand, the operation failed, or a waiting threshold was exceeded, then the thread set its variable to `Failed`. The associated connection could only be used when all the variables in the array were in the `Confirmed` state. If any of the threads set their variable to `Failed` the operation was rolled back.

The experimental implementation showed that parallelising the creation of a multiple switch connection permits the time spent by the control architecture in switch communication to be approximately the same as for a connection which involves only a single switch. However, the price of asynchronous communication is a significant increase in complexity and its utility must be judged with that in mind.

[**Veeraraghavan97**] presents an incremental improvement to the P-NNI protocol which allows the Parallel Creation of Connections (PCC) within a small P-NNI group; its purpose is similar to the that of the work explained here.

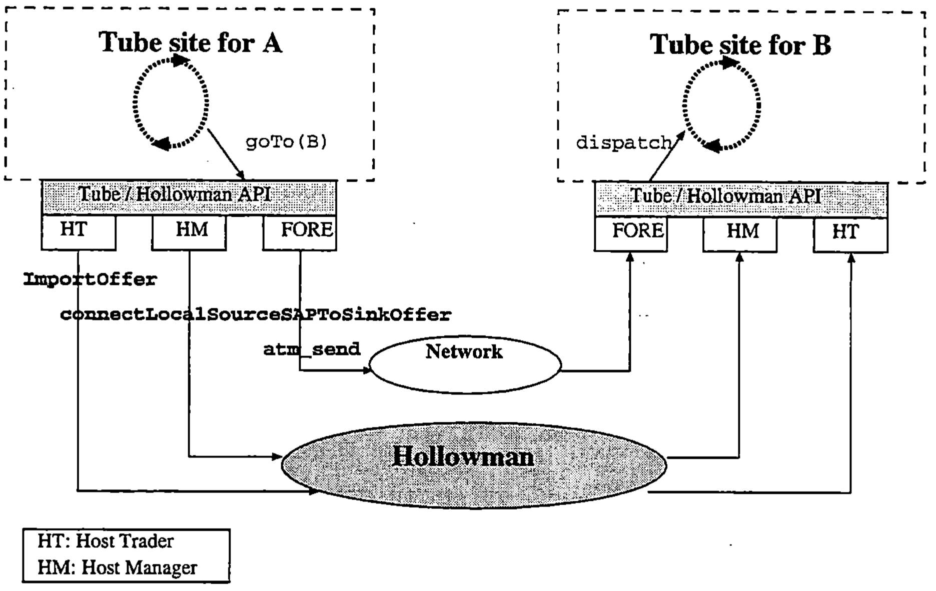


Figure 5.2: Transmitting a Tube mobile agent using the Hollowman

5.4 Applications

The Hollowman supports the communication needs of a number of applications at the University of Cambridge Computer Laboratory and those applications have, in turn, served to test the Hollowman. This support takes two forms; some applications link to the Hollowman libraries and simply use the Hollowman API, while others use the Hollowman as a source library of components for creating other Tempest control architectures. The first two applications described here are in the former group, while the third is in the latter.

5.4.1 The Tube

The *Tube* [**Halls97**] is a mobile agent system written by David Halls at the Computer Laboratory. It is able to send and receive marshalled expressions written in Scheme [**Clinger91**] to and from a network. Agents can create arbitrary functions and send them elsewhere along with their enclosing environments.

Tube mobile agents use the Hollowman to move between hosts. Figure 5.2 shows how a mobile agent moves from a Tube site on host A to another Tube site on host B. Each Tube execution site registers as both an agent receptor and an agent emitter service with a Hollowman host-trader. An agent moves by importing the offers of the site where it is and the site where it wants to move, requesting the creation of a connection between them and then sending itself over that connection. The receptor service reads the agent, dispatches it into the Tube site and deletes the connection.

Using this mechanism, dozens of mobile agents have been allowed to concurrently roam about a network for several hours at a time. Each agent makes random choices as to when and where to move[1]. The Hollowman's ability to support the unpredictable and concurrent activity of these mobile agents motivates belief in its stability. The Tube was used to investigate adaptive management within the Tempest and will be discussed again in Chapter 8.

5.4.2 Secure Video-Conferencing

Jacobus van der Merwe at the University of Cambridge Computer Laboratory wrote a secure video-conferencing application which allowed users to participate in a video-conferencing session whose integrity was ensured using a security protocol. The original version of this application used SPANS [**Fore95b**] — Fore's proprietary signalling protocol — and a set of libraries that are supplied with the Fore ATM cameras. Replacing this application's signalling mechanism with the Hollowman increased confidence that the Hollowman could serve as a general purpose signalling mechanism. It also indicated that advanced control architectures such as the Hollowman, might have a role to play in securing signalling. This is the subject of future work.

The video-conferencing application used a BSD-style socket API, with the typical `open`, `bind`, `listen`, `connect` and `close` operations. The Hollowman API was masked with a set of operations that resemble the classic socket operations. The primitives of this socket-like API are:

- `cam_open_sink`, which adds a new consumer offer to the host-manager, reserves a sink SAP for that offer and returns its identifier. The SAP identifier takes the place of the file descriptor in normal socket operations;
- `cam_open_source`, which reserves a source SAP and returns its identifier;

[1]This test was turned into a simple demonstration by giving each program a graphical display which it carries around from host to host.

- cam_connect, which clients use to connect their source SAP to a sink SAP at a stated Hollowman address. This maps directly to the host-manager connectSourceSAPToSinkSAP operation;

- cam_listen, which a server uses to block while waiting for a connection invocation. This is achieved by waiting on a mutex that is released when the associated offer is 'called back';

- cam_close_sink, which calls freeSinkSAP and removes the offer from the host-manager;

- cam_close_source, which calls freeSourceSAP.

This socket-like API allows legacy applications to use the Hollowman control architecture with only a small number of modifications.

5.4.3 The Sandman

Since many control architectures may run simultaneously within the Tempest, a small, specialised control architecture can be dedicated to a single service [**van der Merwe97**]. Although each of these control architectures is designed for a single specific service, e.g. video-conferencing, they share many common control functions. It should not be necessary within the Tempest environment to rebuild fundamental components, such as routing services, from scratch each time a new control architecture is developed. The components developed for the Hollowman serve as a library for other control architectures.

The Sandman [**Bos98**] is a service-specific control architecture being developed by Herbert Bos at the University of Cambridge Computer Laboratory. The Sandman controls the network resources of a set of video servers containing parts of commercial films. It is hoped that by being able to create a schedule of connections, the network resource usage can be optimised. The Sandman extends the Hollowman components, for example, by allowing network connections to have a temporal aspect.

5.5 Hollowman Performance Figures

The total time T taken to create or delete a connection with the Hollowman is governed by three factors:

- S, the total time for communication with the switches;
- D, the total time for communication between Hollowman entities;
- C, the time taken by the Hollowman for the control processing.

S is principally determined by the way in which Ariel is implemented and in particular the underlying protocol used. D is dependent upon the implementation of the DPE. Both S and D depend upon third-party software, and a testbed was written so that C could be measured independently from them. In the testbed, host-managers and connection-managers are linked into the same address space, so that communication between these services can be achieved without a DPE. Furthermore, in the testbed the Ariel implementation is replaced with a switch emulator.

Each of the graphs in this section was obtained by taking the mean of ten runs; each run is the mean of a thousand creation or deletion operations. The performance tests were carried out using a 167 MHz Sparc Ultra 1 under Solaris 2.5.1. Sun CC was used for the compilation with the `-fast` option set; threading was achieved with the Solaris thread packages.

5.5.1 Measuring the Control Processing Time — C

Figure 5.3 shows how the values for C in some representative scenarios vary as a function of the number of concurrent requests.

(1A) shows how the the latency of a source initiated creation operation across a single switch varies as a function of the number of simultaneous requests. **(1B)** plots the latency of a source initiated deletion operation under the same conditions. In **(1A)** and **(1B)** all the connection requests originate from the same source host-manager and are destined for the same sink host-manager, i.e. there is the maximum amount of contention possible between the competing control operations.

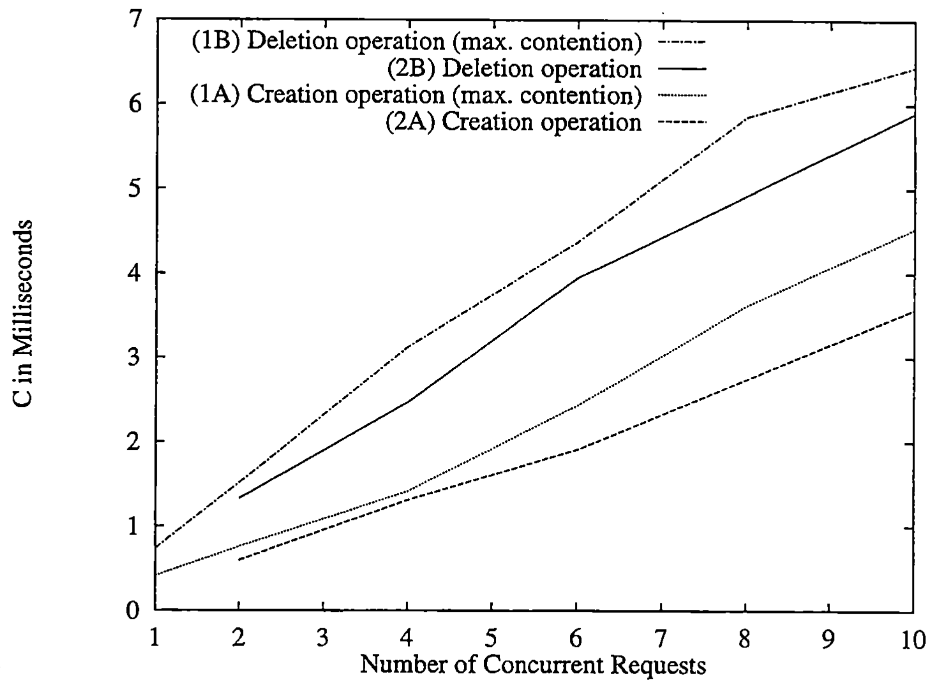


Figure 5.3: C as a function of the number of concurrent requests (source initiated)

(2A) and **(2B)** show how the values vary when there are two different sources and destinations and the requests are divided equally between them.

(2A) and **(2B)** are each respectively faster than **(1A)** and **(1B)** because there is less interference between the requests. **(1A)** and **(2A)** both have gradients less than the time for a single control operation until the number of requests exceeds a certain threshold; the gradient then rises to the equivalent of purely sequential access. The point at which this occurs is an indication of the granularity of the locking. Deletion is slower than creation because it is more expensive in the current implementation to remove an item from a hash-table than to add one.

Figure 5.4 shows how the value for C varies as a function of the number of requests in three different, sink initiated, control operations:

- **(3)** connection creation across three switches;
- **(4)** connection creation across a single switch;
- **(5)** joining a (single switch) branch to a multicast connection.

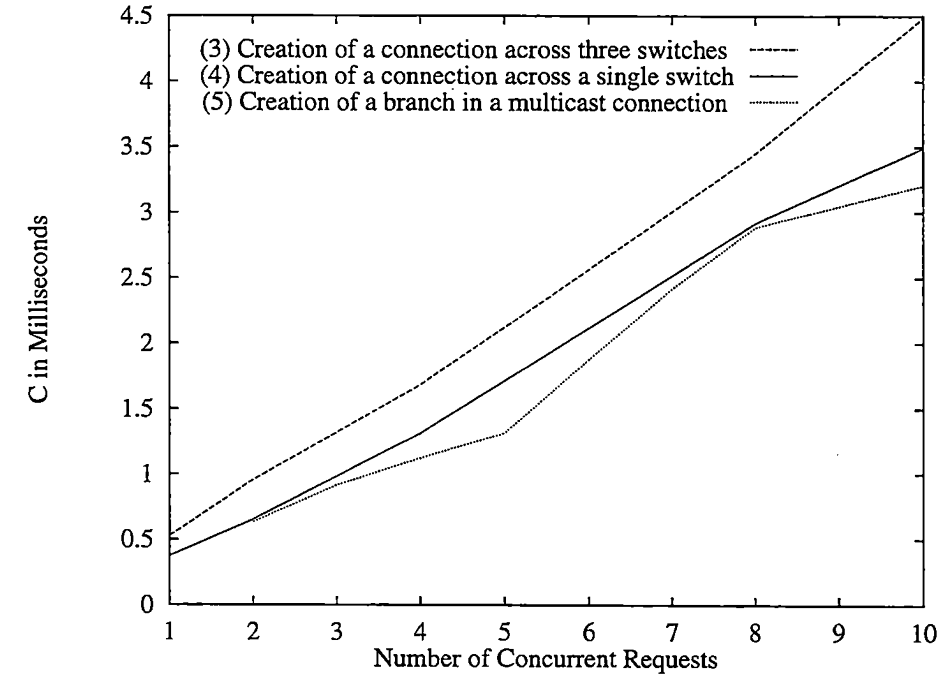


Figure 5.4: C as a function of the number of concurrent requests (sink initiated)

(**3**) shows that a switch connection across three switches is only slightly longer than that for one switch (**4**); the work that the host-manager does is independent of the number of switches the operation involves; the connection-manager must do more work, e.g. invoking the CAC interface on three switches rather than one, but this is not very costly.

In (**5**) host-managers connected to a given switch are joining to a multicast connection which crosses three switches. An anchor host-manager sets up the original connection and never releases it, ensuring that all subsequent connect operations to the offer are joins. All the joins take place across the switch that the host-managers are connected to — the connection across the first two switches is never touched during the test — and the results for (**5**) are very similar to those for (**4**).

5.5.2 Effect of D and S on Overall Latency

The previous section measured C in isolation from D and S. For a single control operation, not competing with other requests, it is enough to know the time for

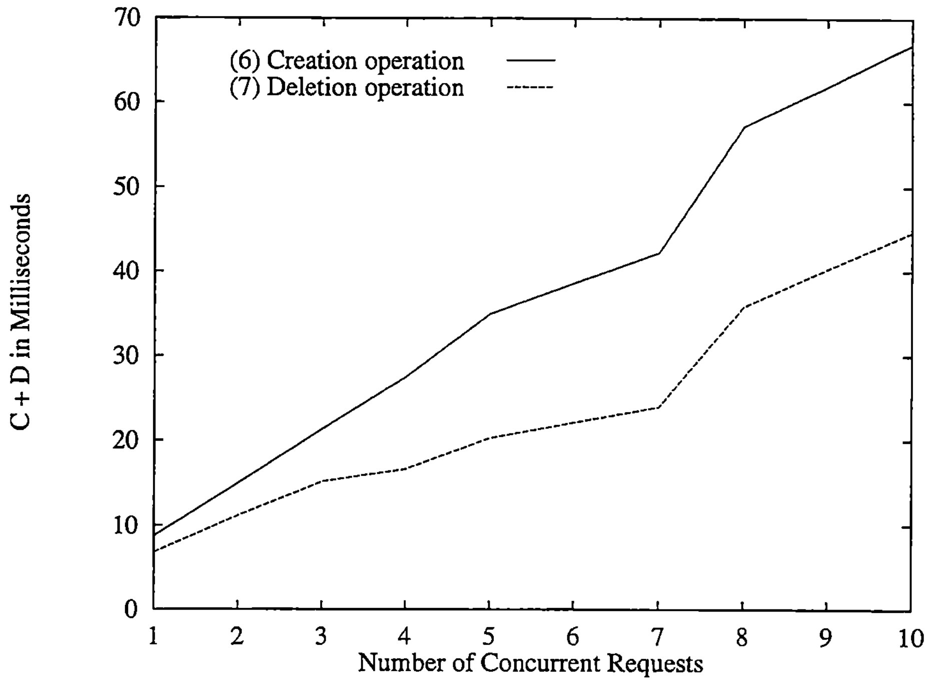


Figure 5.5: $C + D$ as a function of the number of concurrent requests (sink initiated)

an intra-machine RPC (T1_RPC) and an inter-machine RPC (T2_RPC). D is then approximately equal to: n $\times$ T1_RPC + m $\times$ T2_RPC, where n and m are respectively the number of intra-machine RPCs and inter-machine RPCs required for the control operation.

D has greater influence on the latency than C. In the DPE [**Li95**] used an inter-machine RPC takes approximately 1.5 ms; intra-machine RPC has not been optimised for the DPE and is only slightly quicker. Deleting a connection is faster than creating one, because it requires one less RPC. In the Hollowman, n = 2 and m = 3 for an application to signal to a service offer while n = 2 and m = 2 for an application to tear down a created connection.

When there are many concurrent requests the result depends on the implementation of the RPC mechanism and to what extent other operations can proceed while a given operation is performing an RPC. Experiments were performed on the normal Hollowman, but with the switches replaced by switch emulators local to the connection-manager. In this way, the combined latency of $C + D$ was measured in isolation from S. Figure 5.5 shows how the values of $C + D$ vary as a function of the number of concurrent control requests during creation (**6**) and deletion (7) operations.

In the testbed, the Tempest only controls part of the VPI/VCI space of the Computer Laboratory's ATM network; the rest of the network is controlled by non Tempest-aware control architectures. The Tempest prevents the Computer Laboratory's normal control architectures from obtaining VPI/VCI values in its range by setting up place holding connections over all its VPI/VCI space. Setting up a connection requires the costly reconfiguration of the place holding connections. Place holders are only an artifact arising from the need not to interfere with the normal functioning of the network; they would not be required if all control architectures were Tempest-aware. In consequence, communication with the switch was not included in the performance figures; a lightweight Ariel implementation based on message passing has been measured at 2 ms for creation and less than 2 ms for deletion operations [**van der Merwe97**]. Adding this value to the performance figures gives the entire application-to-application connection set up time ($C + D + S$) for an independent operation over a single switch as approximately 11 ms and deletion as 9 ms.

Better values have been published for inter-machine and intra-machine RPC, for example [**ORL97**] gives 0.54 ms for intra-machine RPC/IIOP and 0.71 ms for inter-machine RPC/IIOP/Ethernet for the same platform used in this experiment. A variety of commercial CORBA implementations have similar — though slightly longer — latencies [**Schmidt97**]. If these faster RPCs could be repeated within the context of the Hollowman, the total end-to-end connection set up time in the Hollowman would be approximately 5 ms.

5.5.3 Comparison with other Results

According to its release notes [**Bellcore97**] the Q.Port portable implementation of UNI 4.0 requires 6.5 ms of processor work to establish a connection across a single switch. The performance measurements were performed on a Sparc Station 5 running Solaris 2.3; the code was written in C++ and was compiled with the Sun CC compiler with `-O2` option set. The figures do not include the communication time with an actual switch, S, nor the communication time between the host controllers and the switch controller, D.

[**Niehaus97**] describes a benchmarking framework for conducting tests on the latency of UNI signalling, for point-to-point and point-to-multi-point connections, in both ATM LANs and WANs. It motivates these tests by giving results for:

- Q.Port on Linux, communicating with a DEC AN2 experimental switch using GSMP;
- two different implementations of UNI signalling on two different versions of the Fore ASX-200 switch — ASX-200WG and ASX-200BX;
- an implementation of UNI signalling on an unnamed commercial ATM LAN switch.

These tests show that a point-to-point connection across a single ATM LAN switch takes about 20 ms for both Q.Port and ASX-200WG, 10 ms for ASX-200BX and more than 50 ms on the unnamed LAN switch. Connection set up time increases linearly with the number of hops in the same LAN. Concurrent connection requests emitted from two different sources affect the switches differently; the latency for connection set up for the ASX-200BX is not increased while that for Q.Port and the unnamed switch almost double. Figures are not given for higher numbers of concurrent requests. Adding branches to a multi-point connection has extremely long latency rising from around 100 ms for a single hop to 3000 ms when the new branch traverses five hops.

[**Battou96**] describes a set of experiments to test UNI signalling for the ATM Forum in both a LAN and a WAN context. The type of the switches which make up this network is not named. 53 ms is recorded for point-to-point connection creation across a single switch in a LAN with much higher results given for multipoint connections. Results for connection deletion and concurrent connection requests are not given.

[**Shumate96**] compares the performance of Q.Port running off-switch in a variety of environments, against native implementations of UNI signalling. The conclusion is that there is no performance liability in off-board control and that in fact the fastest tested control architecture was Q.Port running over Linux and using GSMP[2].

[**Veeraraghavan95**] gives figures for the creation and release of a point-to-point call/connection using B-ISUP. The performance figures are calculated analytically based on available figures for diverse functions rather than from measurement. The call processing time for the creation of a call containing a single

[2]The figures given for Q.Port using Linux/GSMP are similar to those given in [**Niehaus97**]. However, when Q.Port is running using Solaris/GSMP the latency values are three times as high although the same processor is used in both cases. The author speculates that this is due to better memory management in Linux.

Name	Latency (ms)	Environment
UNI, Q.Port/GSMP	20	Off-switch on Linux PC
UNI, Fore ASX-200WG	10	On Switch
UNI, Fore ASX-200BX	20	On Switch
UNI, ATM Forum testbed	53	Environment not explained
Xbind/GSMP	16	Off-Switch on Sparc workstation
Hollowman/GSMP	11	Off-Switch on Sparc workstation

Table 5.1: Summary of results for the creation of pt-to-pt connection

bi-directional point-to-point connection is stated as 50 ms and its release as 20 ms. The B-ISUP processing deals with calls, rather than connections, but in the scenarios described the calls only contain single connections. [**Veeraraghavan95**] obtained the 50 ms figure from another analytical model given in [**Willmann90**] which calculated the mean processing time using a VAXstation 3200 under the VMS operating system, — typical in 1990 of the processing environment for an Intelligent Network Signalling Point. The time for a switch to process the B-ISUP connection creation message- is given at 4.5 ms, while that for acknowledgement is given at 1.5 ms. The communication latency between signalling entities is given at 5 ms, based on the B-ISUP message being carried over a 64 k/bs circuit. [**ITU-T93b**] defines the maximum permitted delay for SS7 signalling within an 'average sized' country as 2240 milliseconds for 95% of calls.

[**Lazar97**] reports 16 ms for connection creation in the Xbind open signalling control architecture, measured on a Sun Sparc 10. The greatest overhead in the Xbind control architecture is the communication between signalling entities; techniques similar to those described in Sections 5.1 and 5.3 are used to reduce the amount of communication. The results given for the Hollowman in this chapter do not use this type of optimisation.

Table 5.1 summarises some of the relevant results for point-to-point connection creation across a single switch. Although the Hollowman compares favourably with a variety of other signalling system, it is important not to overstate the importance of signalling latency. If a difference of some small number of milliseconds in signalling latency has adverse effects on an application, then it is probably inappropriate for that application to be carrying out signalling in its critical path. It is enough to demonstrate that there is no radical difference between open signalling systems and those more tightly bound to the switch.

5.6 Summary

This chapter has detailed a number of experiments that were carried out using the Hollowman control architecture. These experiments were related to:

- caching connections;
- making connections persistent between control architecture executions;
- making communication with the switch asynchronous;
- ensuring the correct support of a number of different applications;
- performance.

Overall these experiments provide evidence both of the flexibility and efficiency of the Hollowman.

Chapter 6

Connection Closures: Application-Specific Control

This chapter identifies a limitation in the use of high-level control APIs, namely that their generic nature prevents applications from taking advantage of application-specific knowledge. Allowing users to load application code into the Hollowman for the fine grained control of their own resources addresses this problem. The combination of the application code and network resources is called a *connection closure.*

6.1 Introduction

Chapter 4 described the host-manager interface that applications use to perform control operations. This high-level operational interface is similar to those offered in more conventional signalling systems.

For most applications this will probably be sufficient. However, this dissertation contends that certain applications need to pass their own *application-specific* control policy for connections into the network and have that policy interact at a very fine level of granularity with the resources allocated to the connections during their lifetime. This allows users to take advantage of their high-level knowledge of the function of connections within an application, for example to optimise their use of network resources.

The combined control policy and connection is called a *connection closure* following the usage in programming of the term *closure* to mean a behaviour combined with a context over which that behaviour executes.

A proof-of-concept implementation of closures has been carried out using the Hollowman. Closures allow applications to extend the function of the Hollowman. For example, loading a closure for mobile communication enables the Hollowman to behave like a mobile control architecture for a period; when the mobile closure stops running the Hollowman can 'forget' about mobility.

Allowing foreign code to execute on a shared server raises security issues. The resolution of these issues is the subject of future work. The security problems are reduced if the environment in which the foreign code is introduced can strictly partition all available resources. The recent interest in the transmission of small pieces of code within the Internet means that the securing of shared servers is an active research area [**Adl-Tabatabai96**, **DARPA97**, **Cardelli97**].

6.2 Motivation for Connection Closures

Within existing ATM control architectures, after signalling its requirements for a connection, an application 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. This can be accomplished by offering applications a lower level API than that which is commonly offered by existing control architectures, with which to manipulate network resources. Lowering the level of the API increases the number of invocations that the application is required to execute to achieve some complete control function. If all these invocations need to be transported across the network from application to control architecture, then the latency of the operation will increase. This is a major disadvantage in itself, and also means that certain types of resource management are excluded purely due to the latency in communication between network and application.

At connection creation, an application should 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 suitable programming language which the control architecture can read, load and execute.

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 through the handles on the resources it is allocated. Connection closures are free to manipulate these resources in whatever way they see fit; the role of the control architecture is simply to:

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

An application-defined entity can have knowledge about the use that end-systems are making of connections that a generic signalling system cannot. Executing a connection closure within the control architecture allows the closure to take advantage of this high-level knowledge while interacting with the network resources at a very low level. This technique will not replace the need for distinct control architectures but allows applications greater flexibility in the use of their connection resources. For certain services this is very useful. Some examples of the uses of connection closures are given in the following subsections.

6.2.1 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. The designers of this service may know that the guard will only ever observe one camera at a time and therefore that at any given moment one of the connections is superfluous. Figure 6.1 shows this system diagrammatically.

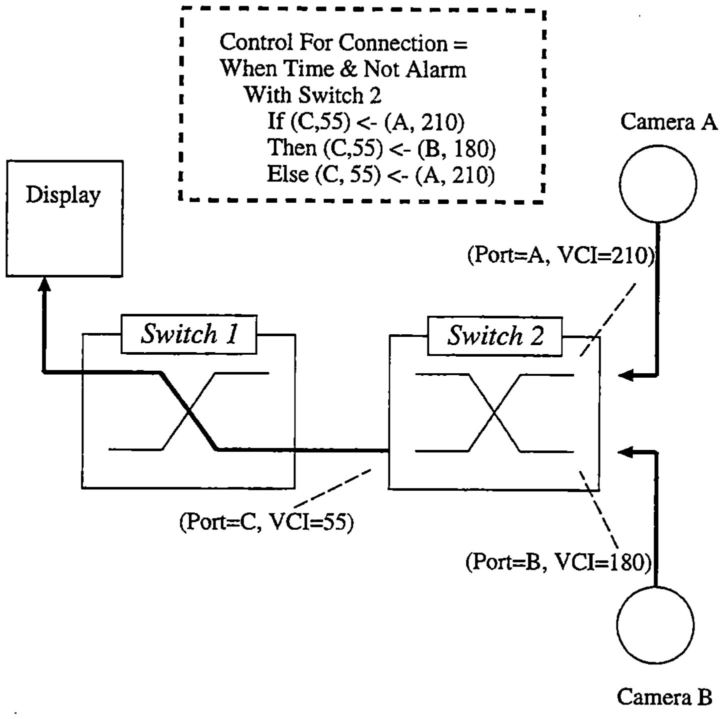


Figure 6.1: Security guard example

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. 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.

The result is a reduction both in the amount of bandwidth that needs to be reserved for the application, and in the total number of VCIs that are used.

The proposed IP signalling system, RSVP [**Zhang93**], achieves a similar effect to connection closures through the use of filters. Filters allow a small fixed set of sharing and merging policies to be defined for network resources. For example, RSVP filtering allows a video-conferencing application to specify that only one of the potential sources will be active at a given moment and that resources between these sources can be shared. RSVP demonstrates that some multimedia applications require more than simple point-to-point and multipoint connections. However, RSVP tries to allow for this within the signalling protocol, rather than accepting that it is best left to the applications. Within a video-conferencing system an application layer floor controller is still needed to turn sources on and off. The activity of the floor controller is tightly bound to that of the network resource controllers and would be better merged with it. Connection closures allow this to be done.

IP switching [**Newman97c**] offers another example of how end-users might take advantage of their application-specific knowledge to optimise resource usage. IP switching (already mentioned in Section 3.10.7) allows more efficient transport of IP packets by mapping IP flows onto ATM connections when appropriate. The decision as to which flows should have their own dedicated ATM connection is made heuristically by the switch controller. IP switching uses GSMP for communication between the switch controller and the ATM switch. Allowing applications to load closures into an IP switch controller would allow applications rather than switch controllers to determine when the 'cut-through' should take place. The closures could communicate with the switch using GSMP directly[1].

6.2.2 Reacting to Changes in Network State

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

Within traditional IP, routing decisions are made for each packet at each router [**DARPA81**]. Routers periodically exchange information with their neighbours about the state of the network, e.g. whether a given link is up or down. Routers make decisions about the best next hop per packet. Two packets in the same message may take different routes to traverse the network if, for example, a link fails or an administrator adjusts the weights assigned to the router links. The network can transparently — from the application point-of-view — 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's topology cannot be altered without the application being aware of the change. For instance, if an Available Bit Rate (ABR) connec-

[1]Moreover, as RSVP does not specify the format of the flow specification that it carries, it would be possible to transport the code using RSVP. Multi Protocol Label Switching (MPLS) [**Callon97**] — which generalises IP switching and other similar techniques, e.g. tag switching [**Rekhter96**] — profits from the lack of structure of RSVP's flow specifications to use RSVP for label distribution.

tion is established across a given set of switches and consequently those switches, because of heavy congestion, start to drop packets, 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. The closure might potentially, after receiving an alarm about dropped packets from the control architecture, decide that the connection's quality is no longer adequate. It might then ask the control architecture for an alternative route and replace all or part of the connection. The closure could, if necessary, contact the parent application to tell it to back-off during this operation.

Self-healing is the ability of a network to automatically adapt to network failure, without the need for human intervention. After a switch failure has been identified by the control architecture, connection closures having connections crossing that switch may adapt to the failure by finding, for example, an alternative route. This is different to other methods of achieving self-healing [**Kawamura95, Veitch96**] because applications decide the recovery policy, e.g. drop connection or reroute, rather than some generic policy being used.

6.2.3 Mobility

[**Agrawal96**] describes the SWAN mobile ATM control architecture and identifies one of the challenges of mobile ATM as the need to distinguish within the mobile control architecture different types of applications. Certain applications are keen to learn about changes in network conditions in order to adapt their behaviours while others prefer to treat the mobile network as if it were fixed. [**Agrawal96**] 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 medium access level.

A connection closure can communicate with its parent application 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 6.6 shows how the loading of a connection closure into the Hollowman allows it to function as a flexible mobile ATM control architecture without the Hollowman having to be modified.

6.3 Architecture

The connection-manager has two interfaces: the γ-interface for requesting the atomic creation and deletion of connections, as described in Section 4.3, and the δ-interface for more fundamental management of network resources. The former is used by the host-manager and is typically implemented using the transport mechanism of the underlying DPE. The latter can only be used by entities in the same address space and can therefore be implemented very efficiently.

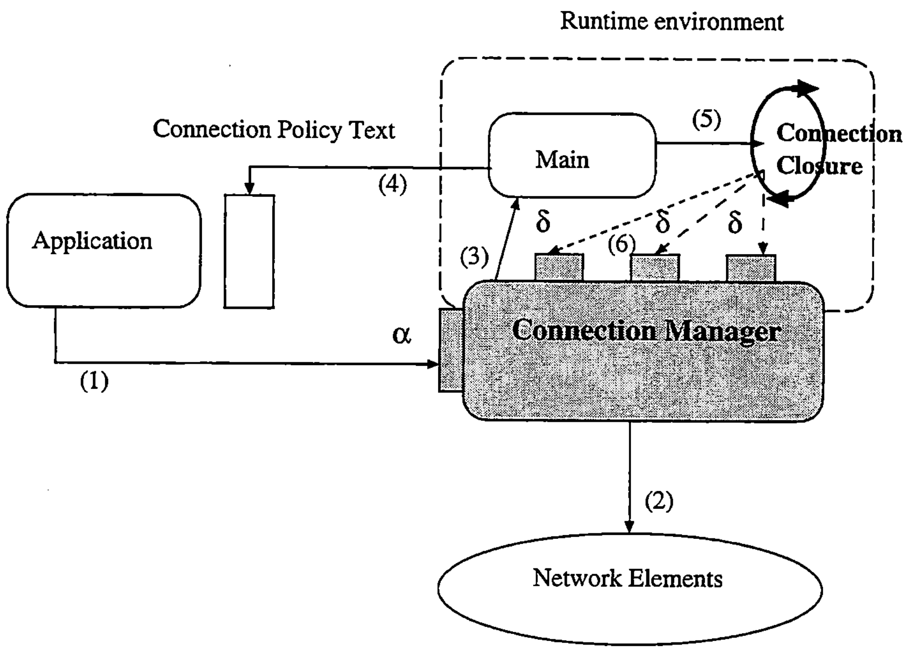
One of the operations in the γ-interface, `createConnectionClosure`, is used to create a closure. The location of the connection closure's behaviour is passed to this operation as a parameter. The connection-manager loads the code from this location and starts executing it. The newly formed closure uses the δ-interface for its interactions with the rest of the control architecture.

Figure 6.2 shows the interactions in the the reading, loading and instantiation of connection closures. The closure is shown being executed within an interpreted runtime environment; more information about the nature of this system is given in Section 6.5. The application's host-manager is not shown for reasons of simplicity.

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

- registration;
- resource management;
- virtual channel management;
- routing;
- notifying host-managers;
- communicating with applications.

These are now explained.



(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 connection closure
(6) Read context and link to the δ-interface

Figure 6.2: Closure creation

6.3.1 Registration

Closures must register themselves before doing anything else. Unregistered closures cannot acquire any resources and therefore cannot do anything useful. As Section 3.6 described, all Hollowman applications are assigned a unique application identifier. Closures use the identifier of their parent application to register. Registration, as well as allowing the closure to obtain resources, results in the creation of a *mailbox*, which is used for communication both between closures and between a closure and its parent application. This is explained in more detail in Section 6.3.6.

6.3.2 Resource Management

A closure is a combination of state and behaviour. To be well-formed a closure must obtain some network resources. The required resources are described by a string in a simple resource specification language. In the current implementation two types of resource exist, VCIs on switch ports and Service Access Points (SAPs) on hosts. 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 assigned to an application, the application must already have registered itself with the host-manager at that host.

Handles on the connection's resources are returned to the closure which controls it. The closure uses these handles to manipulate the underlying resources. Closures are restricted to using only their own resources, thus enabling at least some protection from interference between closures. A distinction is made between source and sink VCIs because of the asymmetric nature of ATM connections that results from the point-to-multipoint model of ATM multicast. A similar distinction, for similar reasons, is made between source and sink SAPs.

The environment in which a closure runs must be capable of ensuring that it only modifies network resources that belong to it, and that it does not inadvertently or deliberately impede the progress of other closures, for example by using a significant amount of processing time. The Nemesis operating system [**Leslie96**], already mentioned in Section 3.3, would allow limits to be placed on the amount of operating resources, e.g. CPU time, that each closure could use. Operating system resources would be specified in the same manner as other resources.

6.3.3 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[2].

[2]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.

6.3.4 Routing

When a closure knows the switch ports on which it needs to obtain VCIs to create its network connections, it does not need to route. If it does not 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: a *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 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 switch traversed and for the routing function the default is the weighted spanning tree. A closure could however define any cost function and any routing function. Whether in a commercial system applications should be allowed to define their own cost functions is questionable, but it is useful in an experimental environment.

6.3.5 Notifying Host-Managers

The ability to create virtual channels across switches is not in itself enough to create an application-to-application connection. In order for an application to use a connection, the connection's end-point must be associated with a Service Access Point (SAP).

Section 4.2.3 described three SAP states: `Free`, `Reserved` and `Active`. An `Active` SAP is one which is associated with a VCI and involved in an end-to-end connection. The finer granularity of connection management enabled by closures means that a new SAP state is required — `Suspended`. A suspended SAP is one which is associated with a VCI, but which is not currently involved in an end-to-end connection. 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 SAP state. Besides doing its own book-keeping the host-manager executes an application specified callback to inform the application about the change in state. Application responses to changes in SAP state are application-specific.

6.3.6 Communicating with Applications

As a side-effect of both host-managers being informed of changes in SAP state and applications being called back when the SAP changes state, closures can prompt applications to start, stop and suspend receiving and sending information. For many applications this is enough, but to take full advantage of the power of closures requires 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.

To do this a *mailbox* is created for each closure when it registers. A mailbox is simply a pair of FIFO buffers 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.

Connection closures can communicate with *each other* using the same mechanism. It would be possible for closures to trade resources amongst themselves with little 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 has to be modified 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 get its required resources. Other situations can be imagined in which closures sell or exchange resources. [**Huberman93**] gives examples of computational economic models and ecosystems and [**Grover97**] proposes the use of economic models for the allocation of resources between ATM virtual paths. This type of 'bartering' between connection closures is the subject of future work.

6.4 Distributed Closures

If there are many connection-managers in the Hollowman domain or the connection crosses several different domains then closures execute on many connection-managers simultaneously. A closure will always be sent to the closest (or ingress) connection-manager. If its request for network resources cannot be completely satisfied by that connection-manager then it can ask the connection-manager to send it to the connection-managers downstream. This process is repeated until the *set* of closure instances has all the resources that are required. The distributed closures can communicate amongst themselves using the mailboxes described in Section 6.3.6. The implementation of distributed closures is the subject of future work.

6.5 Implementation

This section describes the implementation of the architecture defined in the previous section. The implementation described in Section 3.9 was extended so that the connection-manager executable could be created and run in two ways:

- by linking the relevant Hollowman's libraries with a C language `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 former is identical to that described in Chapter 3 and 4. The latter gives an extra degree of flexibility since the runtime environment allows the dynamic integration of new modules of interpreted code. Simple control operations, not involving closures, behave exactly as if the code were being executed independently of this runtime environment. There is no penalty for the fact that they are embedded inside an interpreted runtime system and the performance figures given in Section 5.5 remain valid. Other entities in the control architecture make no distinction between the virtual machine and stand-alone versions of the connection-manager.

Currently a closure's control policy is written in Java bytecode and the dynamically incremental runtime environment is a Java virtual machine[3]. It would

[3]Version 1.1 of the Java Developer's Kit (JDK).

be possible to use a dedicated control language with only control related primitives. This approach would make the security issues easier to solve albeit at the cost of restricting what the users could do. Since the main concern was not primarily one of security it was 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 because of its ubiquity. In addition, the Java Native Interface (JNI) permits complete interaction between Java and C, allowing Java entities to be called from C/C++ and C/C++ entities to be passed into the Java virtual machine[4].

When executed in a virtual machine, the connection-manager's Java `Main` loads the relevant native code libraries using the Java `Runtime` class. The connection-manager service is initialised, exports a service offer to its host-trader, and waits on a socket for incoming invocations. When a simple connection request is received by the connection-manager service, it is processed in the normal manner without any interaction with Java. A request for a connection closure creation is up-called into Java. The 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; after reading in the behaviour it is instantiated and the `Runnable` method `start` is called. The closure uses the interface described in the previous section to create, maintain and delete connections.

The concept of connection closures requires that the control architecture — or parts of it — be run in an environment to which user code can be dynamically added; it is a requirement that the environment be able to ensure that different closures do not interfere with each other. The exact nature of the dynamically incrementable environment is an implementation detail; [**Alexander97**] suggests that Java, due to its lack of a mathematical definition, is flawed, resulting in many security weaknesses and making it an inappropriate environment for running third-party software on a shared server. The authors propose the use of a strongly typed functional programming language called Caml, while the same group more recently have implemented a dedicated language called Programming Language for Active Networks (PLAN) [**Hicks97**].

[4]The fact that the closure's policy is written in Java bytecode does not mean that application developers are forced to write their high-level code in Java as well. Java bytecode compilers now exist for many languages [**Tolksdorf97**], so connection closures are effectively language independent.

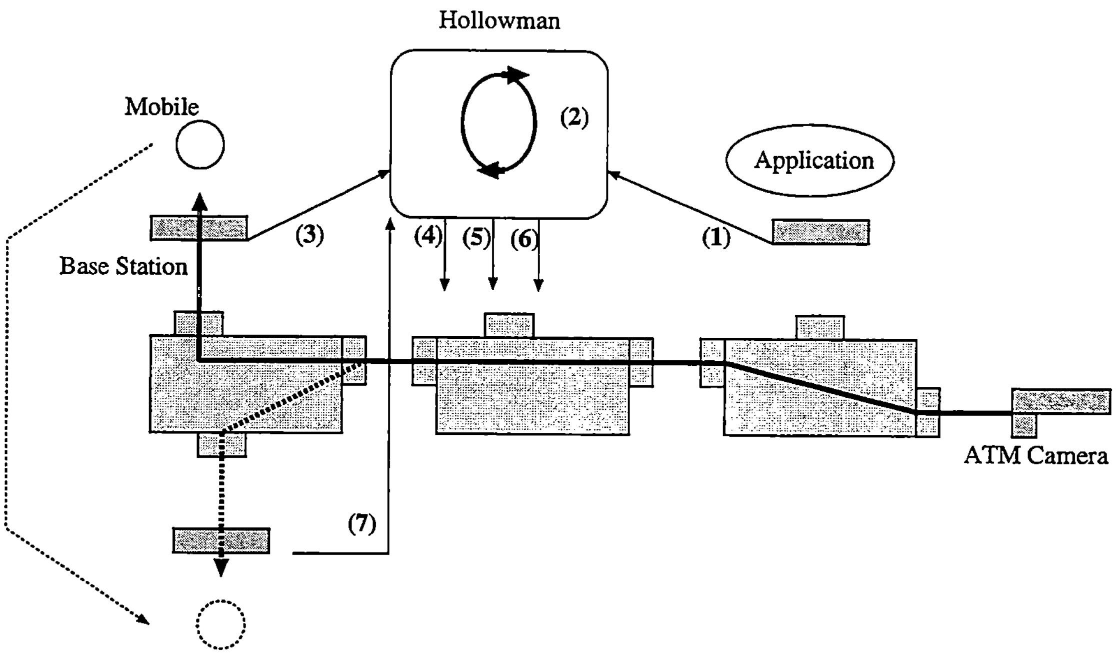


Figure 6.3: Mobile connection closure

6.6 Proof-of-Concept

Section 6.2.3 mentioned that the need for fast connection creation in mobile ATM signalling systems means that the connection topologies are not necessarily optimal. The control architecture needs to periodically reconfigure connections which have become too distorted, for example those which have loops. When and how to modify a mobile connection is application-specific: some applications can support frequent alterations if they are short, while for other applications the inverse may be true.

This section describes an experiment which uses closures to define application-specific strategies for managing mobile ATM connections. In the experiment, applications running on workstations take the place of mobile base stations and a human wandering around the laboratory the role of the mobile. Figure 6.3 shows a typical sequence of control messages resulting from the use of a mobile closure.

The user creates a closure in the Hollowman **(1)**. The closure registers itself then waits **(2)** for notification of the user's location. The user types on the keyboard of a workstation and the base-station reacts by telling the closure where the user is **(3)**. The closure requests resources for a source SAP on the host on which an ATM camera is running, a sink SAP on the host on which the user has typed and resources on all the switches that constitute a path between them (4). It then creates the connection (5) and notifies the video and user service (6). When the user moves to another workstation and types, the base station communicates to the closure the fact that the user has moved (7). The closure then decides how best to modify the connection, e.g. the closure adds a new branch to the connection, before deleting the old one. Many different mobile closures can run simultaneously allowing each to have its own policy.

The mobile connection closure is about 200 lines of Java source and compiles to 4 kilobytes of Java bytecode. After the closure is loaded and resources are assigned, the time to establish and modify connections is mainly determined by the time taken to communicate between the mobile and the closure. 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. This experiment demonstrated the feasibility and utility of connection closures.

6.7 Related Work

Many attempts have been made to make network nodes programmable by adding interpreters or dynamic linkers to the node software. Programs are sent along the data path and tagged for interpretation. The network node, e.g. the switch, detects these programs and executes them. It is possible to distinguish two different motives for doing this:

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

The first happens infrequently, stopping or inhibiting the normal function of the node until the transfer and loading are complete, and is normally initiated by the network operator. An example is the modification of the access control policy of an Internet firewall. Networks which have the second aim are often called *Active Networks*; [**Tennenhouse96**, **Wetherall96**, **Smith96**, **Alexander97**] are examples. [**Alexander97**] states that:

"Active Networks" are packet-switched networks in which network infrastructure is programmable and extensible and where network be-

haviour can be controlled on a per packet, per user or other basis. For example, a packet might carry executable code.

The execution of these active packets — or *capsules* as [**Tennenhouse96**] calls them — happens on a time-scale which is at, or close to, the speed with which they are transferred across the network node. The IP model of *store and forward* has a major influence on this work. In fact, the intention of the Active Network could be summarised as modifying this model to *store, execute and forward.* While it might be tempting to equate an ATM cell with an IP packet[5] this ignores the important distinction in ATM between the control and data path and would mean that functions that were placed by the designers of ATM in the control path are pushed back into the data path. ATM was designed to allow fast, predictable switching and the control/data path distinction is a necessary feature of that.

An Active Network enabled ATM switch would have to distinguish active cells from normal data, assemble them into a program and execute the program. In a sense this is what Operation and Maintenance (OAM) cells within the ITU-T forum standards [**ITU-T93a**] 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. The use of OAM cells is restricted to ensuring the health of the entire network rather than optimising the resource usage for specific applications. The problem is *significantly* more complex when the programs, as in Active Networks, have the following general properties:

- they are written in a general programming language, e.g. Java;
- they are sequenced across an arbitrary number of protocol data units, which must be identified as special and assembled into a program;
- they can manipulate the flows on which these protocol data units arrive;
- they execute on network nodes with different capabilities;
- they must avoid interfering with programs written by other users.

More experimental evidence is required to demonstrate that these active programs can be executed both quickly and predictably enough to warrant their

[5]Some of the Active Network research seems to suggest just this, for example [**Smith96**] proposes the application of active networks to ATM switches.

presence in the data path of a multi-service network. The approach described in this chapter restricts itself to the control path.

Netscript [**Yemini96**] is a language for programming a Netscript Virtual Network, consisting of a set of Netscript-aware nodes. Each Netscript-aware node, e.g. a router, runs a Netscript interpreter. The language contains primitives for controlling the allocation, scheduling and transmission of packets over virtual links, allowing users to define, for example, their own routing algorithms. Netscript programs operate on streams of packets allowing them to be multiplexed, demultiplexed, parsed and filtered. Netscript is similar to Active networks in that it allows application code to directly manipulate packets in the data stream. Netscript could be confined to just the control path in which case it would offer a similar function to connection closures.

[**Biswas95**] proposes the use of a roaming software agent, known as a *representative*, for the distribution of mobility management load within the fixed backbone network. In essence, the agent is a proxy for the mobile itself which moves less frequently than the mobile. It can convert normal ATM signalling into mobile signalling and hides some mobile specific features e.g. hand-overs, from the fixed ATM signalling network. The representative treats a mobile connection as a set of segments that it joins into a path to form a complete connection, allowing it, for example, to pre-reserve segments thereby optimising hand-overs. The approach differs from that presented in this chapter in that the representative is supplied by the control architecture rather than the user. Moreover, a distinction is made between signalling to mobiles and signalling to fixed points rather than treating this simply as another type of fine grained resource management.

Intelligent Networks (IN) [**ITU-T92a**] allows non-basic call functions to be associated with signalling end-points. An IN checks in a database to see if a special script, e.g. call forwarding, is associated with a dialled telephone number, and if so triggers that script. The scripts are network operator defined; since they add knowledge to the basic call model about special call types, they require extensive modification to the signalling infrastructure and can only be introduced on a long time scale. AIN (Advanced Intelligent Networks) [**Garrahan93**] has lessened the introduction time by making a clearer separation between the switching and controlling planes, but the nature of the interface between the two means that while adaptations of existing services, e.g. calling forwarding, can easily be introduced, completely new services still require modifications to the switching plane. [**Veeraraghavan97**] addresses this problem by separating basic signalling from service signalling and defines the second in terms of the first.

[**Rizzo97**] identifies one of the problems in the introduction of IN services as the *coarse grain service interface*. The authors remark that most research work in IN is dedicated to solving the problem of feature interaction whereby two separate predefined IN services when combined have unexpected side-effects. They propose allowing telephony users to define service-specific policies for managing their calls. The primitives of these scripts are still fairly coarse grained, e.g. *forward-the-call-for-approval* rather than directly manipulating network resources as the work described in this chapter does.

JTAPI [**JavaSoft97**] is an API which allows Java applications to implement advanced telephony call models. The nature of the API is similar to the Hollowman connection-manager's δ-interface. However, the JTAPI is dependent on the interfaces supported by telephony servers, which means that it is at a higher level than the δ-interface and is telephony specific. It is interesting to note that at the same time the telephony community is actively researching ways of making telephony services more flexible and telephony calls more programmable, the ATM standards are adopting the telephony model of signalling in order to deal with the more complicated problem of establishing and managing multimedia services.

[**van der Merwe97**] describes another solution to the limitations of trying to define a single generic control API for all present and future services. The authors propose the creation of *service-specific* control architectures within the Tempest environment, each dedicated to a single service, e.g. video-on-demand. Section 5.4.3 has already mentioned one service-specific control architecture, the *Sandman*. [**van der Merwe97**] describes one for video-conferencing, the *Videoman*. Connection closures and service-specific control architectures differ in the granularity of control and the speed with which they can be introduced. The two techniques are complimentary. For example, connection closures could be used as a prototyping technique for service-specific control architectures and some service-specific control architectures, e.g. for mobility, might offer end-users the ability to introduce closures as part of their control function.

6.8 Summary

This chapter has described how allowing applications to take advantage of their application-specific knowledge permits more flexible and efficient control. Within the Hollowman this is achieved by the loading and execution of application defined code within the control architecture itself. The combination of this policy and the network resources that it manipulates is called a *connection closure*. Examples have been given to motivate this technique and a proof-of-concept implementation has been described which demonstrates its feasibility.

The closures are restricted to manipulating only their own resources, permitting at least some degree of security. Future work will explore in more depth the security issues, in particular the application of mechanisms for the partitioning of operating system resources.

Chapter 7

Inter-Control Architecture Signalling

It is likely that the techniques described in this dissertation for the simultaneous execution of multiple control architectures will not be generally deployed in commercial networks in the near future. This means that for at least some time open signalling control architectures will have to interoperate with more conventional ones. This chapter explains how instances of advanced control architecture can interoperate with other ATM control architectures.

7.1 Introduction

Advanced control architectures such as the Hollowman allow network operators and end-users a great deal of flexibility in the control over their network resources. This makes them convenient experimental platforms. To be more generally useful they need to be able to interoperate with other, more widely deployed, control architectures. This interoperation requires that the control operations of the advanced control architecture can be translated to and from the format of other control architectures, that routes to addresses outside the advanced control architecture's domain can be determined, and that routing information can be made available to other control architectures. In short an interoperation protocol is required.

This chapter considers the form of such an interoperation protocol and how an open control architecture could make use of it. The Hollowman has been used for experimenting with inter-control architecture signalling. The rest of this chapter is structured as follows:

- firstly, the research described in this chapter is motivated;
- secondly, the issues involved are explained by describing the parts of the Hollowman dedicated to inter-control architecture signalling;
- thirdly, some experiments are described which demonstrate the patterns of interaction between the Hollowman and other control architectures;
- finally, P-NNI [**ATMF96**] is considered and the extensions required to allow the Hollowman to use it are discussed.

7.2 Motivation

A single standard ATM signalling mechanism — assuming it was accepted and widely implemented — would guarantee universal interoperation. Preceding chapters have argued that this *one size fits all* approach is too restrictive and is unlikely to be successful. Alternatively, if every switch could support the switch divider described in Section 2.2.6, then potentially any control architecture could be deployed universally. The following reasons argue against this as a solution:

- in all probability users will be restricted in their use of network elements outside their management domain;
- the number of virtual networks that can be supported by a network element at a given moment is bounded;
- not all switches will have a switch divider.

So, while the Tempest environment allows a service provider to operate a global virtual network, in practice the extent of the virtual network is likely to be closely related to the equipment owned by the body that authorised that virtual network. There is still a need for interoperation between control islands belonging to different management domains.

The ATM Forum terms the group of signalling interfaces between two switches the Network-to-Network Interface (NNI). The Private-NNI or P-NNI [**ATMF96**], is intended for use within private networks. P-NNI contains two distinct interfaces; one for the exchange of routing information between switches and a second for the management of connections. P-NNI is designed to scale to planetary scope. It is discussed in more detail in Section 7.5.

While this dissertation argues that standard UNI/NNI signalling will not be the only ATM control architecture it is likely to be the one which is most widely deployed. If an advanced control architecture could interoperate with P-NNI, then applications using that control architecture would have the same scope in their signalling as those directly using the UNI. An advanced control architecture which implemented P-NNI would in fact be a partial implementation of the standard ATMF one; it would behaves as a normal P-NNI instance to other control architecture instances, however within its own domain it could avail of non-standard signalling techniques, e.g. connection closures. This is analogous to CORBA/OMG [**OMG95b**] where, within the domain of a given ORB, any proprietary protocol can be used, but all ORBs must use the GIOP protocol when communicating with other ORBs.

Due to its complexity, there is as yet no readily available version of P-NNI. Other means had to be found to experiment with interoperation,

Chapters 3 and 4 detailed the Hollowman connection management functions. In order to simplify the description, they assumed that there was only one connection-manager per Hollowman domain. It was mentioned that the connection management functions could be distributed, but the explanation of how this is achieved has been reserved until now. The distribution is achieved using two interfaces:

- the Simple Inter-Control Interface (SICI), which allows two connection-managers to collaborate in the creation of a connection;
- the Simple Inter-Routing Interface (SIRI) which allows two connection-managers to exchange routing related information.

The SICI and SIRI are explained in Section 7.3. Section 7.4 gives an example of how they are used to distribute connection management. While the SICI and SIRI are implemented as services of the DPE and intended for communication between Hollowman control entities, their *functions* are similar to those of the two interfaces of P-NNI. The SICI and SIRI were used to experiment with the interoperation of control architectures. The exact nature of the control architecture being communicated with was hidden behind these interfaces, facilitating the experiments.

The experiments described in this chapter show that a Hollowman instance can interoperate with both another Hollowman instance and a standard control architecture instance — ATMF UNI 4.0 — using the SICI and SIRI. The main distinguishing feature between the ATMF UNI and the P-NNI is the exchange of routing information that takes places across the NNI; the connection control primitives are very similar. The fact that the Hollowman can interoperate with a UNI implementation gives credence to the claim that the Hollowman will be able to interoperate with P-NNI when implementations of that control architecture are more widely available.

7.3 Interoperation with the Hollowman

A control architecture instance manages resources located on some set of network elements, called its *domain*. Two control architecture instances are said to be neighbours if one or more of their network elements are directly connected. To interoperate with another, a control architecture needs to be able to:

- recognise that a signalled address is outside the scope of its domain;
- identify which, if any, of its neighbours permits that address to be reached;
- map signalling requests to and from a format recognised by its neighbour.

In the standard control architecture the first and second are achieved by having a common hierarchical addressing scheme and through the continual exchange of routing information. The third is enabled by a signalling interface across which network elements signal to each other.

The elements of the Hollowman required for control architecture interoperation are now discussed. The intention is not to implement an interoperation protocol to compete with P-NNI, but only to show how the Hollowman control architecture, described in Chapters 3 and 4, can easily be extended to allow interoperation. The rest of this section explains the Hollowman control gateway service, the Simple Inter-Control Interface (SICI) and the Simple Inter-Routing Interface (SIRI). The gateway completes the set of core Hollowman services introduced in Chapter 3.

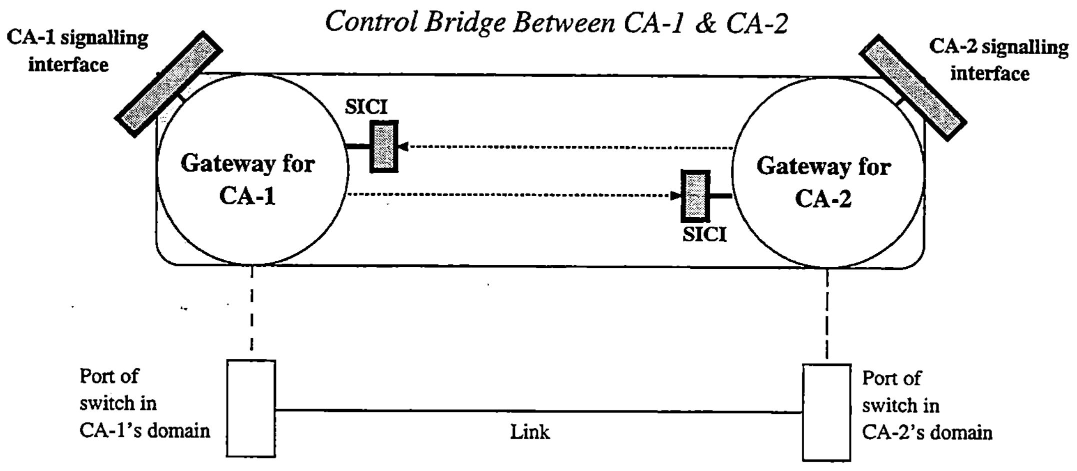

Control Bridge Between CA-1 & CA-2
CA-1 signalling interface
CA-2 signalling interface
SICI
SICI
Gateway for CA-1
Gateway for CA-2
Port of switch in CA-1's domain
Link
Port of switch in CA-2's domain

Figure 7.1: Overview of a control bridge

7.3.1 Control Gateway

This section examines the features — other than those described in Chapter 3 — which allow the Hollowman to signal to an address outside its domain. This is achieved through the generalisation of entities used for the inter-connecting of Hollowman connection-managers and the distribution of the connection management functions.

A *control bridge* is an entity that can map a control request emitted from one domain into a format suitable for another. The two domains may be managed by different instances of the same control architecture or by completely different types of control architecture. A bridge is addressable in both domains and is capable of interpreting and generating control operations for the control architectures that manage those domains. The end-points of a bridge are called *control gateways.*

The Hollowman gateway manages some subset of the resources of a switch's port. By the definition of gateway, this port is attached to a switch outside the Hollowman's domain. Two gateways are coupled if the ports that they control are connected across the same link. Each gateway couple constitutes a control bridge. A local gateway communicates with the foreign gateway using the Simple Inter-Control Interface, described in Section 7.3.2. Figure 7.1 shows a control bridge between two control architectures, `CA-1` and `CA-2`.

A gateway can be the start or end point of a connection within a given Hollowman domain. In this respect a gateway resembles a host-manager. The gateway implements the host-manager's β-interface described in Section 4.3.2; this allows the Hollowman connection-manager, after routing, to make no further distinction between signalling to hosts and signalling to gateways. The gateway also supports a gateway interface — containing two operations `addForwardingRecord` and `removeForwardingRecord` — which the connection-manager uses to update the routing information in the gateway.

A gateway maintains an `Offer_Table`, containing information about the current service offers available through that gateway and the SAPs assigned to them, and a `Forwarding_Table`, containing a set of forwarding records which enables it to determine through which foreign gateway it may reach a foreign address. Forwarding records are added to the gateway by calling the `addForwardingRecord` operation of the gateway interface. The forwarding record contains information related to foreign addresses, such as the foreign gateway to use in order to reach them. When the connection-manager calls the `obtainSinkSAP` or `obtainSourceSAP` operations on the gateway's β-interface to reserve a SAP, the service offer, passed as argument, is automatically added to the offer table, if it is not present already.

The connection-manager activates a SAP by calling the `notifySinkSAP` or `notifySourceSAP` operation on the gateway. The activation of a SAP on a gateway involves:

1. identifying with which forwarding record the SAP is associated;
2. forwarding the operation to the foreign gateway;
3. updating the state of the SAP;
4. telling the connection-manager whether the operation succeeded or not.

Failure of the gateway to establish a distant connection is not perceived by the initiating connection-manager as any different to refusal by a host-manager and is handled in the same way, i.e. rolling back the signalling operation.

In this chapter for the purposes of clarity the connection-manager and the gateways it communicates with are shown as distinct applications, but in practice the connection-manager and gateway instances are combined in the same process and all communication between them takes place using efficient local function calls.

The Simple Inter-Control Interface

• `connectSourceChannel`
• `connectSinkChannel`
• `unconnectSourceChannel`
• `unconnectSinkChannel`

The Simple Inter-Routing Interface

• `isAccessible`
• `exportForeignOffer`

Table 7.1: The SICI and SIRI

7.3.2 The Interoperation Interfaces — SICI & SIRI

This section describes the interfaces — the SICI and the SIRI — that are used by the Hollowman to distribute the connection management functions. The SICI and SIRI were designed for communication between control entities in the same Hollowman domain and are intentionally very simple. They are not intended to have the same scalability as a general purpose interoperation protocol such as P-NNI. Table 7.1 shows the operations that make up the SICI and SIRI.

The *Simple Inter-Control Interface* (SICI) allows a gateway to ask its homologue gateway in the control bridge to create or delete a connection. The `connectSourceChannel` and `connectSinkChannel` operations create a connection between a given source or sink SAP and a service offer. The `unconnectSourceChannel` and `unconnectSinkChannel` operations delete connections. Each control architecture maps these operations onto its own control primitives.

The `connectSourceChannel` and `connectSinkChannel` operations of the SICI both take a service offer as argument. The Hollowman service offer contains information about the nature of the service and the location of the service provider; it is general enough to allow the transport of service descriptions in a range of control architectures. For example, in the case of inter-operation with the ATMF standard control architecture, an NSAP format address can be used for the service location and the service properties can be used to transport the description of the network resources, in the form of ATMF Information Elements, required to use that service, as well as additional information, e.g. higher level identifiers.

A given instance of the Hollowman finds a route to a foreign host's location by determining with which local gateway it should be associated and then calculating a route to that gateway. The Hollowman instance determines which gateway to use by asking its neighbouring control architectures using the *Simple Inter-Routing Interface* (SIRI).

The SIRI allows routing information to be exchanged between neighbouring control architecture instances. The SIRI `isAccessible` operation returns whether an address is reachable within or through the domain of a control architecture and if so the means to reach it, i.e. the interface reference of the service that supports the SICI. It takes the foreign address and a list of already visited control architectures as arguments. The action of `isAccessible` is first to check if the address is present in the domain control architecture and if not to check all the yet to be visited neighbours.

The Hollowman's lack of structured addresses means that any address format can be used within the Hollowman. So, for example, an application can signal to a standard NSAP format ATM address using the API described in Chapter 4. If the address is not present in the Hollowman's domain, the connection-manager calls `isAccessible` on the domain's neighbours passing the address as argument. The address can be passed from the application, across the Hollowman, to a control architecture which is capable of interpreting its structure. This allows Hollowman applications to signal to locations outside the Hollowman's scope.

For other control architectures to signal to Hollowman services requires that the Hollowman service offer be converted into a suitable format for that control architecture. The SIRI `exportForeignOffer` operation allows the availability of a service offer in one domain to be advertised in another. The Hollowman supplies information about the nature of the offer in a form suitable for the foreign domain[1], and passes the interface reference of the the appropriate SICI service to use to reach it. The receiving entity within the foreign control architecture identifies the gateway to use to reach the offer, updates the gateway, completes the offer with the address of the gateway and makes the offer known within its domain. To entities within the foreign control architecture the gateway will

[1]Note that in order for the Hollowman to export an offer to a foreign control architecture it is required to know the format of foreign service offers; in practice offers which are to be made *globally* available should follow the ATMF standards on addressing and service description.

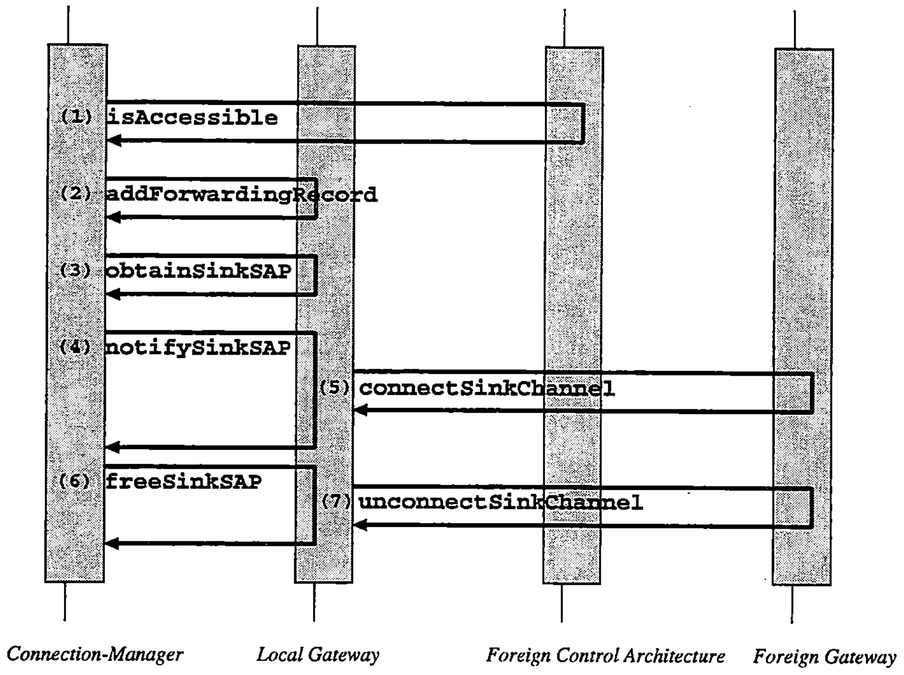


Figure 7.2: Pattern of inter-control architecture signalling

appear as the offer supplier.

In order for the Hollowman to interoperate with a foreign control architecture, the operations of the SIRI and SICI must be mapped onto control primitives in that control architecture. Section 7.4 outlines how this was achieved for a commercial implementation of UNI signalling. An example demonstrates the use of the SICI and SIRI.

Example: Figure 7.2 shows the typical interaction that occurs when the Hollowman connection-manager, during a control operation, recognises that a signalled address is foreign. The connection-manager calls the `isAccessible` of its neighbour passing the foreign address (**1**). The foreign control architecture confirms the address is accessible and returns the interface reference of the foreign gateway through which it can be reached. The connection-manager then uses the interface reference to update the forwarding table in the local gateway using `addForwardingRecord` (**2**). The connection-manager calls the `obtainSinkSAP` (**3**) operation of the gateway to obtain a SAP identifier, and after the connection has been established to the gateway calls `notifySinkSAP` (**4**). This results in the local gateway calling `connectSinkChannel` to create the connection to the appropriate sink in the foreign domain (**5**). When the connection-manager calls `freeSinkSAP` on the local gateway **(6)**, the gateway prompts the foreign gateway to delete the rest of the connection by calling `unconnectSinkChannel` **(7)**[2].

After a neighbour of the Hollowman instance has confirmed that an address is reachable, the Hollowman updates the forwarding table in the appropriate gateway and modifies its topology tables such that the foreign address is associated with the gateway. The next time that the address is used in a control operation within the domain, the information about how to signal to that location will already be present; so with reference to Figure 7.2, operations **(1)** and **(2)** would not need to be repeated.

The Hollowman has no knowledge of the topology outside its domain, yet the set of gateways that the connection creation request traverses after leaving the Hollowman is decided before the request is made. The SIRI permits a type of routing which lies somewhere in between source-based and hop-by-hop routing. Inter-control routing establishes the appropriate entry and exit points for the domain of each control architecture. During connection creation each control architecture uses its own routing mechanism to calculate the best route between these end-points. The complete path of gateways to use in reaching a location is not returned to the initiating control architecture but is scattered across the forwarding records in the gateways of all the concerned control architectures. It would be possible to do without the SIRI and just optimistically attempt to signal to a neighbour whenever an address was not found in a given domain. This would involve the complexity of cranking-back the create operation if the address turned out to be erroneous or if there are multiple neighbours and the correct one was not chosen first. Having a distinct routing phase ensures that the address is reachable (or likely to be reachable) before any network resources are assigned to the connection, removing or reducing the need for crank-back.

The SICI and SIRI are not appropriate as a general inter-operation protocol; they are neither complete nor scalable enough. For example, in the current implementation no attempt is made to find a best route, the search stops when any route is found and the Hollowman learns about its neighbours during start up simply from a configuration file. However, they are sufficient to demonstrate the patterns of interactions which are required to allow the Hollowman to inter-operate with foreign control architectures; these are explained in the following section.

[2]Note that when the connection-manager and gateway instances are resident in the same process, the example shown in Figure 7.2 requires only three RPCs (`isAccessible`, `connectSinkChannel`, `unconnectSinkChannel`). The rest are all local function calls.

7.4 Experiments in Interoperation

This section motivates the description of the structure of inter-control architecture signalling by detailing the patterns of communication for a number of different inter-control architecture exchanges. The types of inter-control architecture signalling examined are those between:

- two instances of the Hollowman[3];
- the Hollowman and an implementation of UNI 4.0;
- two instances of the Hollowman separated by a UNI 4.0 domain.

All of the examples described here were implemented within the testbed network in order to experiment with control architecture interoperation.

7.4.1 Hollowman/Hollowman Signalling

In the testbed network, there are three switches, `cogan`, `aynho` and `aller`. Attached to `cogan` is a host called `clyde`, and attached to `aller` is a host called `magnus`. Figure 7.3 shows the network. Three different instances of the Hollowman are started such that:

- `Hollowman-1` is allocated a virtual network on `cogan`. `Hollowman-1` has a gateway, `G-A1`, which manages the `cogan` port attached to `aynho`.
- `Hollowman-2` is allocated a virtual network on `aynho`. `Hollowman-2` has two gateways, `G-A2` and `G-B2`, such that `G-A2` can form a control bridge with `G-A1` and `G-B2` can form a control bridge with `G-B3`.
- `Hollowman-3` is allocated a virtual network on `aller`. `Hollowman-3` has a gateway, `G-B3`, which manages the `aller` port attached to `aynho`.

Experiment: in this experiment service provider `SP` on `magnus` — within the domain of `Hollowman-3` — is offering a service of type `VideoService`. The

[3]Note that normally within the testbed network there is no need for two instances of the Hollowman to interoperate, since any given instance can acquire some subset of resources on the entire physical network.

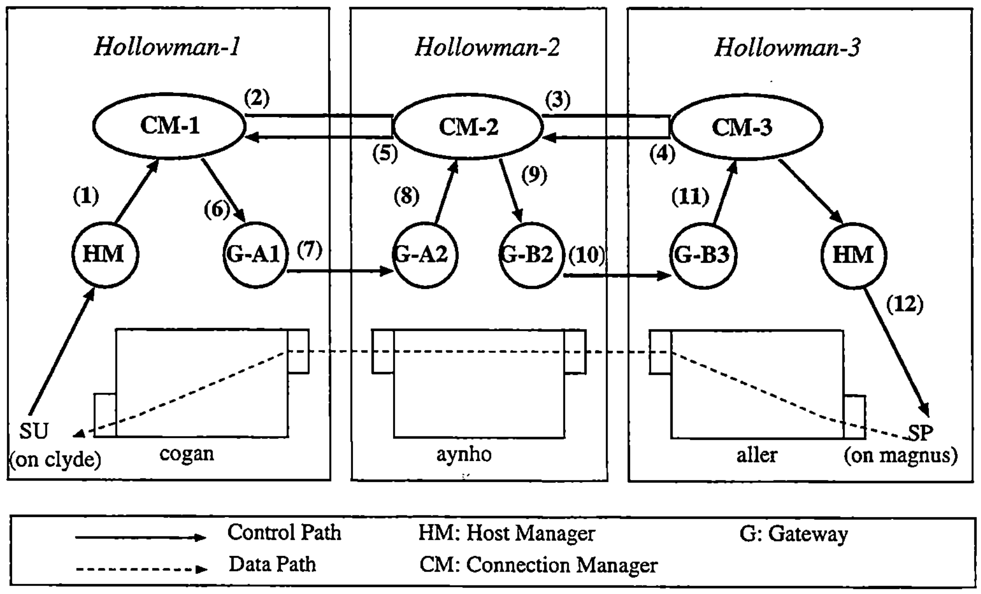


Figure 7.3: Example of Hollowman/Hollowman signalling

sequence of operations required for the application SU on clyde — within the domain of Hollowman-1 — to sink a video stream from SP is now explained; they are also shown diagrammatically in Figure 7.3.

SU calls the host-manager connectLocalSinkSAPToSourceOffer operation, passing the source offer for the VideoService. The host-manager relays the invocation to the Hollowman-1 connection-manager (1). The Hollowman-1 connection-manager recognises that magnus is not an address in its domain. It calls the isAccessible operation of its only neighbour's SIRI passing the address magnus as well as adding the label Hollowman-1 to the list of already visited control architectures (2). The address is not in Hollowman-2's domain either, so it in turn calls the isAccessible operation of Hollowman-3's SIRI (3). As magnus is in Hollowman-3's domain, it tells Hollowman-2 that the address is reachable (4). Hollowman-2 alters its topology information such that magnus is associated with G-B2 and updates G-B2's forwarding table. Hollowman-2 then confirms the reachability of magnus to Hollowman-1 (5). Hollowman-1 updates its topology information and forwarding tables in a similar way.

Hollowman-1's connection-manager reserves a source SAP on G-A1, sets up the connection from G-A1's port to SU and finally notifies G-A1 of the activation of the source SAP (6). This notification causes G-A1 to read its forwarding tables and call the connectSourceChannel operation on the SICI associated with the foreign address `magnus`. The arguments in this operation are the activated source SAP and the service offer for `VideoService` (**7**). `G-A2` creates a sink SAP with the same VPI/VCI values as that in the passed source SAP. `G-A2` then calls `connectLocalSinkSAPToSourceOffer` on `Hollowman-2`'s connection-manager passing this sink SAP (**8**).

`Hollowman-2`'s connection-manager sets up a connection from `G-B2`'s port to `G-A2`'s port and notifies `G-B2` that the reserved source SAP is now active (**9**). This notification causes `G-B2` to call `connectSourceChannel` using the SICI associated with `magnus`, i.e. that implemented by `G-B3` (**10**). `G-B3` behaves like `G-A2` (**11**). A connection is established between `magnus` and `G-B3`'s port. Notifying `magnus`' host-manager of the SAP activation causes the callback function of `SP` to be invoked (**12**). The control operation initiated by `SU` has successfully completed; `SP` starts producing a video stream on the source SAP passed to it and `SU` starts receiving the stream on its sink SAP.

If another application in `Hollowman-1` wished to use the video service then steps (**2**) ... (**5**) would not need to be repeated[4]. When the connection-manager and the gateways in its domain are present in the same process 9 RPCs are required to establish the first connection from `clyde` to `magnus` and 7 thereafter. This is not significantly greater than the 5 RPCs required to establish a connection within a single Hollowman domain.

The scenario described here is an extreme form of distributed connection management in which each switch is controlled by a distinct and independent instance of the Hollowman. A single instance of the Hollowman, controlling one unified virtual network and with a connection-manager per switch is similar to, but simpler than, the above, e.g. traders can be shared.

7.4.2 Hollowman/Q.Port Signalling

This experiment shows how a Hollowman application can signal to a service controlled by an implementation of UNI 4.0.

Q.Port [**Bellcore97**] is a portable version of the UNI 3.0, 3.1 and 4.0 signalling protocol written by Bellcore. In Q.Port, the signalling engine on the host is called

[4]In the current implementation failure to create a connection to an offer resident on a foreign location causes the routing information about that location to be freed as potentially out-of-date. More sophisticated means of maintaining consistency can be imagined.

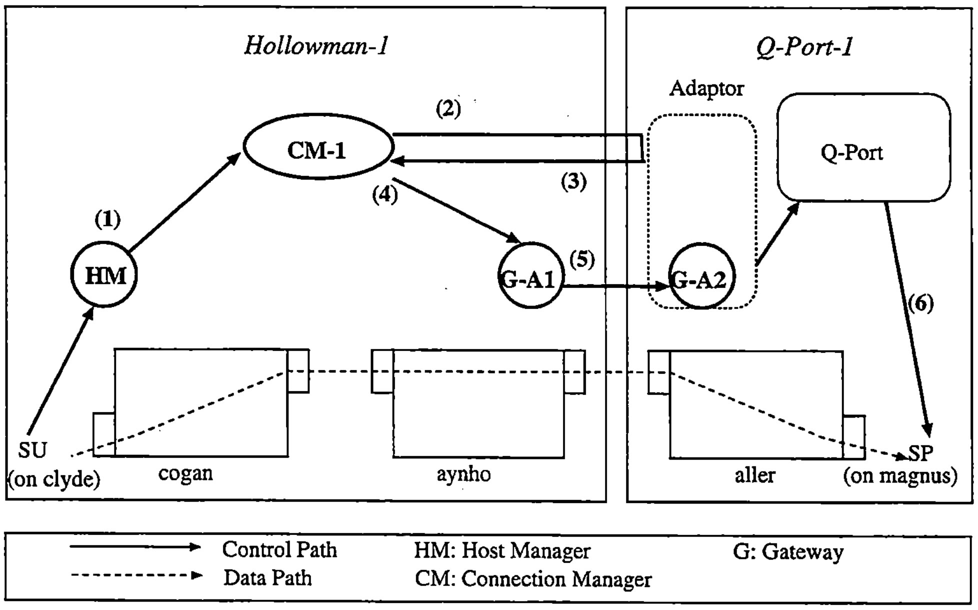


Figure 7.4: Example of Hollowman/Q.Port signalling

the *host controller*. An adaptor allows the commands defined in Section 7.3.2 to be mapped onto Q.Port commands. This adaptor is added to the Q.Port host controller, making it a Hollowman gateway. The signalling PVC and Q.Port configuration files are altered so that signalling messages to an external port are sent to the appropriate gateway. No other modification to Q.Port is made. The test configuration is as shown in Figure 7.4.

Experiment: in this experiment a Q.Port service provider `SP` on `magnus` — within the domain of `Q.Port-1` — is offering a service whose Q.Port type is `4031`, i.e. it is a UNI 4.0 service that sinks an AAL5, point-to-point, ABR connection. Q.Port offers this shorthand description of service offers to applications and does the conversion to the corresponding ATMF Information Elements carried in the signalling request. Alternatively the service could be identified within the Hollowman directly by some combination of ATMF Information Elements. The Q.Port NSAP address of `magnus` is `nsap-magnus`. The sequence of operations required for a service user `SU` on `clyde` — within the domain of `Hollowman-1` — to use that service is now explained; they are also shown diagrammatically in Figure 7.4.

`SU` creates a suitable Hollowman service offer — one whose type is `4031` and address is `nsap-magnus` — and calls its host-manager connectLocalSourceSAPToSinkOffer operation. The host-manager relays the request to the Hollowman-1 connection-manager (1). Hollowman-1 realises that the address is not in its domain and calls its only neighbour's isAccessible operation (2).

The Q.Port-1 adaptor does not have complete knowledge of all actual addresses within Q.Port-1's domain. It does know how to check if an address conforms to the NSAP standard and is therefore *potentially* a valid address. The adaptor verifies this, and confirms to Hollowman-1 that nsap-magnus is potentially reachable (3). If the Hollowman has multiple instances of Q.Port as neighbours then the implementation would have to be modified so that failure to connect with one gateway would result in trying other routes. However, in practice a control island is likely to be connected to the rest of the world through only one port.

Hollowman-1 modifies its topology information, such that the foreign address, nsap-magnus, is associated with G-A1 and updates G-A1's forwarding table. The Hollowman-1 connection-manager establishes a connection from the port managed by G-A1 to SU and notifies G-A1 of the connection creation (4).

G-A1 calls connectSinkChannel (5) on the gateway, G-A2, implemented by the Q.Port adaptor. The adaptor gets the type and address of the service from the offer, and the VPI/VCI to use from the SAP. It then creates a connection from G-A2 to SP using normal UNI 4.0 signalling messages (6). The Hollowman application SU can now send information to the Q.Port service SP.

7.4.3 Hollowman/Q.Port/Hollowman Signalling

The final experiments show how one instance of the Hollowman can signal across a domain controlled by Q.Port, to communicate with another instance of the Hollowman.

Experiment: in this experiment a service provider SP on host magnus — in the domain of Hollowman-2 — is offering a service of type VideoService. The sequence of operations required for an application SU on host clyde — within Hollowman-1 — to receive a video stream from SP is now explained; they are also shown diagrammatically in Figure 7.5.

SP, to make the source offer VideoService generally available, tags it as externally exportable when registering it with its trader. The trader calls

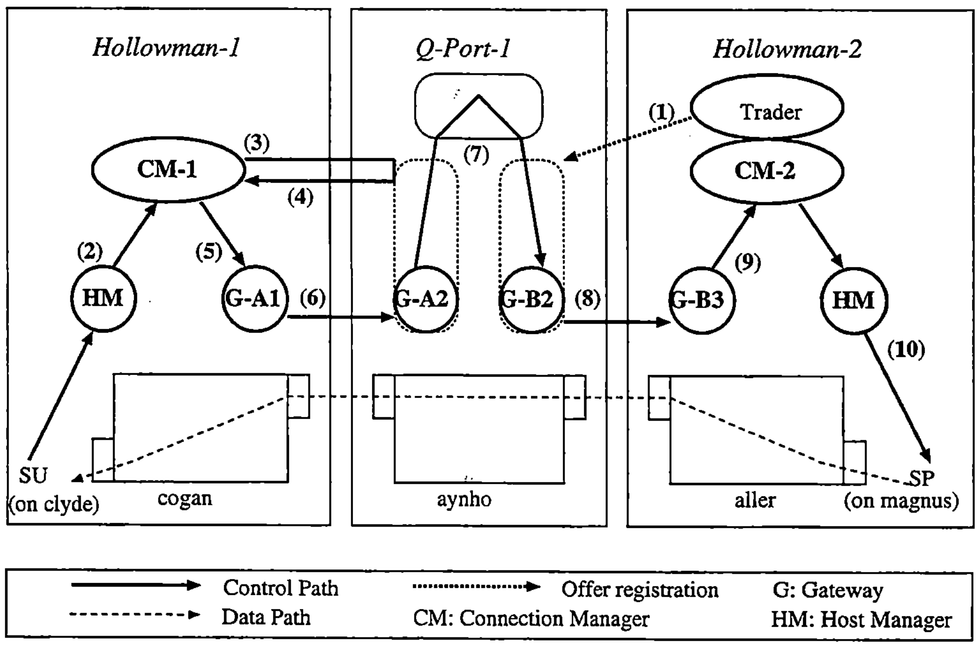


Figure 7.5: Example of Hollowman/Q.Port/Hollowman signalling

the `exportForeignOffer` (1) operation of its neighbouring Q.Port adaptor's SIRI. `G-B2` records the `VideoService` with a Q.Port alias as service name, and with the NSAP address of the port that `G-B2`'s controls. This alias and address are the form in which the offer is known within both `Hollowman-1`'s and Q.Port-1's domains. When Q.Port notifies `G-B2` of a connection request, the `connectSourceChannel` operation of `Hollowman-2`'s gateway is invoked with appropriate values; this is step (8) in Figure 7.5. The other interactions are similar to those described in Section 7.4.2.

Connections can also be established across the external network for the exchange of control information between Hollowman domains. There is nothing special about these connections as far as the external network is concerned, but the Hollowman instances can use them to perform control architecture specific signalling. For example, if `Hollowman-2` had two video cameras then an application within the domain of `Hollowman-1`, after creating a signalling connection, can pass the security closure described in Section 6.2.1 to `Hollowman-2` to ensure that a video stream from only one of these cameras is sent across the external network at any given moment[5].

[5]This is similar to Permanent Virtual Path (PVP) tunnelling [**ATMF96**], except PVP tunnelling requires manual configuration.

This section has demonstrated how a Hollowman instance can inter-operate with other control architectures. Although the simple techniques discussed here are inappropriate for general inter-operation, they allow a better understanding of how advanced control architectures could make use of a scalable solution such as P-NNI. The required extensions are considered in the following section.

7.5 Related Work

This section explains P-NNI in more detail and describes how the Hollowman could interoperate with it. In the interest of completeness some other Network-Network Interfaces are also mentioned.

7.5.1 P-NNI

The Private NNI [**ATMF96**] is designed to scale to planetary scope. P-NNI routing is based on IP routing techniques, such as Open Shortest Path First (OSPF) [**Moy97**]. Switches exchange information with their peers about their current state. The ingress switch of a connection request uses this information to determine the entire route from source to destination. A generic set of routing information is constantly exchanged between peer entities.

P-NNI defines a generic CAC algorithm that switches can use to determine the likelihood of a foreign switch accepting a given connection request. The number of switches is potentially very large, so the switches are organised into hierarchical peer groups. All elements of such a group have complete knowledge of the topology of the group and exchange routing information directly with each other. A peer group leader is elected by the nodes of a group. The peer group leader exchanges routing information with other peer group leaders in this higher group. The entire group appears like one logical node at the higher level. This federating process can be repeated an arbitrary number of times. The route determined by the ingress switch consists of both logical and physical nodes; during connection creation the logical nodes must be resolved as a path across the group. This in-band routing is performed by the group leader. If a node refuses a connection because, for instance, the ingress switch's routing information is incomplete or out-of-date, then the group leaders of the group in which this occurred have the responsibility of attempting to find an alternative route.

If the Hollowman can interoperate with P-NNI, then Hollowman applications will have the same scope as applications directly using UNI signalling. In P-NNI routing information and signalling requests are sent on VCI=15 and VCI=5 of VPI=0 respectively. A Hollowman instance which could send, receive and respond appropriately to P-NNI messages on these channels would appear to P-NNI as a normal P-NNI group. One entity within the Hollowman domain must play the role of the P-NNI group leader, exchanging information with P-NNI peers. The connection-manager controlling the switch connected to the external network is the natural termination and emission point of P-NNI routing information.

All external connection management requests can be received or initiated by Hollowman gateways. The work described in this chapter has shown how this can be done for the SETUP, CONNECT and RELEASE UNI primitives; these have almost identical homonyms in P-NNI. The operations related to multicast have not been tested, but should pose no significant problem. NSAP format addressing can be accommodated within the less structured Hollowman addressing, while additional information required in the ATMF Signalling Information Elements, e.g. the resources required for the connection, can be contained within the Hollowman notion of a service offer.

In the experiments detailed in Section 7.4 a given Hollowman instance was unaware if it was signalling to another Hollowman instance or to Q.Port. While this facilitated the experiments, it is not generally appropriate as the SICI and SIRI are too simple to completely encapsulate P-NNI. Hollowman control entities must make a clear distinction between communicating with other Hollowman entities and communicating with P-NNI. However, only the connection-manager and gateways attached to foreign control architectures have to implement P-NNI; two Hollowman connection-managers can still exchange routing information using the SIRI and non P-NNI aware connection-managers and gateways can be kept simple. Requests to signal to services offered by the standard ATMF control architecture can be transported across the Hollowman, using the SICI and SIRI in the way described by this chapter, until a P-NNI aware connection-manager recognises their significance.

To signal across P-NNI to a service under the control of the Hollowman requires that the service be identifiable within P-NNI. The service must be exported with the appropriate format and the Hollowman must do the mapping when a signalling request is received for that service. For example, if a Hollowman service is capable of supporting a P-NNI anycast service, then the connection-manager must export the willingness of the gateway attached to P-NNI to support that service, using the normal P-NNI routing information exchange, and when the gateway receives the connection request, the Hollowman must be capable of doing the additional signalling to the correct Hollowman service.

Running Hollowman as a P-NNI group is the subject of future work; the integration with Q.Port motivates its feasibility.

7.5.2 IISP

P-NNI is complex, so another protocol, the Interim Inter-Switch Signalling Protocol IISP [**ATMF94b**] — sometimes called P-NNI phase zero — has been defined which uses UNI signalling for implementing the NNI between switch controllers. In each IISP exchange one switch controller plays the role of a host controller. All switch controllers know the locations of all the other switch controllers that they communicate with; this information is manually added to the routing tables of each switch controller. IISP does not have the same scalability as P-NNI and is simply an interim measure while P-NNI is deployed.

7.5.3 B-ICI

P-NNI is used for the interconnection of two private nodes; the Broadband Inter-Carrier Interface (B-ICI) [**ATMF95b, ITU-T96**] serves to connect two public networks. B-ICI specifies a wider range of physical layers over which the ATM layer can run than UNI/P-NNI and also particular adaptation layers for running common inter-carrier services, e.g. Frame Relay Service (FRS). B-ICI signalling is based on the ITU-T Signalling System No.7 [**ITU-T93b**]. The upper layer signalling protocol (B-ISUP) is similar to that of Q.2931, with some modifications to allow it to interoperate with the Public Switched Telecommunications Network, e.g. addresses use E.164 format. It is not clear how the public network will interoperate with private networks, particularly in the exchange of potentially sensitive routing information.

7.6 Summary

This chapter has shown how an instance of an open control architecture can interoperate with other open control instances and with instances of Q.Port — a commercial implementation of UNI 4.0. In the longer term, P-NNI is proposed as the universal interoperation protocol; in this scheme P-NNI would serve as a 'glue' to interconnect different control architectures. Experience with interconnecting with Q.Port has motivated the feasability of achieving this. Control architectures can implement the ATMF NNI without implementing the UNI, and without requiring that all control requests within the control architecture's domain be ATMF compliant. This will allow network operators more flexibility over the nature of the control architectures used within their own domain while still permitting signalling which scales to planetary scope.

Chapter 8

Adaptive Fault Management

Chapter 1 described how switch-independent control is characterised by a loosening of the association between the out-of-band switch control functions and the switch fabric. This has implications for the way such a control architecture is administered and in particular how control architectures can detect and recover from network failure. The ability of a fault management system to adapt itself to the current network state is a precondition for it being robust. This chapter considers how the Tempest can facilitate the implementation of adaptive fault management

8.1 Introduction

A Tempest control architecture must ensure that its view of the network state remains in phase with that of the actual state. This is not prohibitive, since every permitted state changing operation is by definition sanctioned by the control architecture. The control architecture, for example, decides whether a connection request should be granted by examining its local view of the network. Once the control architecture decides to admit the request, it instructs the switches to create the connection and updates its local view. Normally, the switch should never refuse an operation that the control architecture has authorised, i.e. the control architecture has all the intelligence while the switch is relatively dumb.

If an unauthorised change of network state occurs, e.g. port failure or routing table corruption, then that change must be propagated to the control architectures concerned so that they can respond to it. All unauthorised changes in network state are termed *network failures.* In the period between a failure occurring and the control architecture taking it into account, the control architecture is said to be *unstable.* The process of returning the control architecture to a stable state is called *failure recovery.*

The reaction of a control architecture to network failure is control architecture specific. For example, some control architectures might simply release their network resources and finish when confronted by any unforeseen change in network state; other, more resilient control architectures, might adapt to the failed state. Failure recovery may or may not require the participation of a human operator.

Besides detecting and recovering from failure, fault management also entails determining its cause. Some cases, for example outage of a port, are trivial to determine; others are more subtle, requiring the correlation of information from many different sources. [**Yemini93**] gives an example whereby a long burst of noise on a link carrying a large number of different connections from a given host leads to packet loss. The link layer protocol invokes automatic retransmission which results in a burst of retransmission tasks running on the interface processor queue, leading to thrashing. The transport layer protocols time out and the host CPU responds with a burst of corrective action leading to more thrashing and the emission of alarms from the host. To identify the reason for the alarms the operator needs to have information about both the error rate on the link and the queue length on the interface processor. However, neither of these two pieces of information by themselves is enough. What the operator must determine is that a sharp increase in error rates was followed by a sharp increase in queue length and that even after the error rate dropped off the queue length remained long, pointing to lots of failed retransmissions. Correlation between two different types of data from two different sources is needed to identify the cause of the problem.

The human operator's decision about probable cause is guided by the available information. Often the problem is too much information rather than too little. A failure may cause other failures as side-effects which in turn provoke the emission of more alarms, making it difficult for the operator to distinguish between primary and auxiliary alarms. The problem becomes worse the further across the unstable network the alarms have to be transported to the operator. If the management system is centralised then all the alarms will be heading towards the same place. When alarms start being emitted, because of other alarms, then exponential growth in emissions may put the network in jeopardy [**Dupy91, Miller97**][1].

[1][**Miller97**] reports an episode in a SONET network where two Add and Drop Multiplexors (ADMs), constantly exchanging alarms, brought the network management system to within 3%

This chapter starts by examining the problems involved in implementing one aspect of fault management — recovery from port failure. The constraints on a fault management system are elucidated through this examination. This solution is then generalised so as to be applicable to a wide range of fault management functions.

8.2 Recovering from Port Failure

This section considers the implementation of recovery from port failure, i.e. the unexpected non-functioning of an ATM switch port, such that it is no longer able to send and/or receive cells. It starts by looking at a naïve strategy for achieving this, using the Hollowman as an example control architecture. A discussion identifies the weakness in this strategy and leads to the definition of a more robust one for the Hollowman. Finally, the sharing of the fundamental infrastructure between many control architectures in the Tempest is motivated.

8.2.1 Naïve Strategy for the Hollowman

After port failure, the Hollowman attempts to regain a stable state by:

- updating its topology information so that no further connections will be routed through that port;
- removing all connections that run across that port;
- notifying the source and sink applications about the connection's deletion.

The Ariel server, as described in Section 2.2.4, is capable of notifying registered parties about changes in port state. For the purposes of this discussion suppose that the port has only two states of interest: *up* and *down*. Within the Hollowman, each soft switch registers an interest with the Ariel server about the state of the ports of its switch fabric using the Ariel server. If a port goes into the *down* state, the corresponding soft switch within the Hollowman is notified. The reaction of the soft switch to reception of a port failure notification is to mark that port as unreachable and to propagate the failure to the connection-manager.

of its virtual memory limits. The author comments that if these limits had been exceeded the entire system would have crashed.

The Hollowman connection-manager updates its network topology information so that no further connections will be routed through that port. The connection-manager searches through its information about the current connections to identify which of them crosses the port. It removes all those connections from the switch using the Ariel server and notifies the source and sink applications about the connection's liberation in the way described in Chapter 4. If an affected connection has been marked as persistent then, assuming an alternative route exists, it will be automatically rerouted, i.e. it is treated as a Soft PVC [**ATMF96**].

At the end of the failure recovery process, the Hollowman is once again in a stable state, albeit with a reduced domain.

8.2.2 Discussion of the Naïve Strategy

The strategy described in Section 8.2.1 is naïve since it assumes that normal communication between Hollowman entities will not be affected by the port failure. Many of the solutions proposed in the literature, [**Shrivastava97**, **Frey97**, **Hong97**, **TINA-C97**] also ignore this fact[2]. For example, in [**TINA-C97**], after a network element has raised an alarm, four TINA computational objects are involved in detecting and recovering from the failure: notification server, alarm manager, fault coordinator and testing/diagnostic server. Implicitly these computational objects must communicate with others, e.g. TINA connection manager, to carry out their task, yet it is not clear what happens if communication between these entities themselves is affected by the network fault.

The implementation of failure recovery is complicated by the fact that there is no guarantee that any Hollowman service can still communicate with any other service executing on a different host.

In existing ATM systems, Operation and Maintenance (OAM) cells [**ITU-T93a**] are exchanged between neighbouring connected ATM switches to ensure the health of the physical link and the logical layers built over it. The OAM cells are inserted into the data stream of normal connections. For example, if an OAM cell for a virtual path is not received after some time from the downstream switch, the OAM functions on the upstream switch may assume that the

[2]Many papers seem to consider the integration of CORBA into a alarm management system as an end in itself, without considering whether the advantages of CORBA, e.g. location transparency, are desirable in detecting the cause and nature of faults.

virtual path has been removed and take suitable action. This allows all reachable parts of the network to be informed about failure. A network which supports open control can still use the same mechanisms at the switch level. However, since Ariel separates the switch controller from the switch fabric, the control architecture cannot examine the data stream of normal connections. The controlling software is not necessarily even running on the network elements through which the OAM cells are transported. The higher level control architecture also needs to be prompted to update its view.

Control architectures are distributed entities that in general use the network itself for communication between their constituent parts. The Ariel sever — assuming it monitors the OAM cells — may send an appropriate message to a part of the control architecture, but the control architecture as a whole still has to synchronise itself. If port failure caused an alarm to be emitted, there is no guarantee either that the Ariel server can communicate with the control architecture or that the control architecture can contact all its constituents. The recovery is needed because the network has failed. As small a number of assumptions as possible should be made about the network state during the recovery process. Doing so also reduces the possibility of the recovery process itself inflicting harm on the network.

Recovery from port failure entails each part of the control architecture trying to determine with which other parts it can still communicate. During failure detection and recovery, knowing the location of the entities that are being communicated with is of primary importance.

In summary, during failure recovery few expectations should be placed on the network, i.e. the resources used for detecting and recovering from failure should be minimal and bounded. The next section takes these constraints into account to obtain a more robust strategy for failure recovery.

8.2.3 Robust Strategy for Hollowman

The discussion in the previous section concluded that the transport of messages between control architecture entities while the control architecture is unstable should be minimal and bounded. Therefore, during recovery, the distributed processing environment is not used for communication between hosts, instead the ATM adaptation layer is used directly. Signalling and failure recovery use separate mechanisms; this is analogous to the situation in the current standards

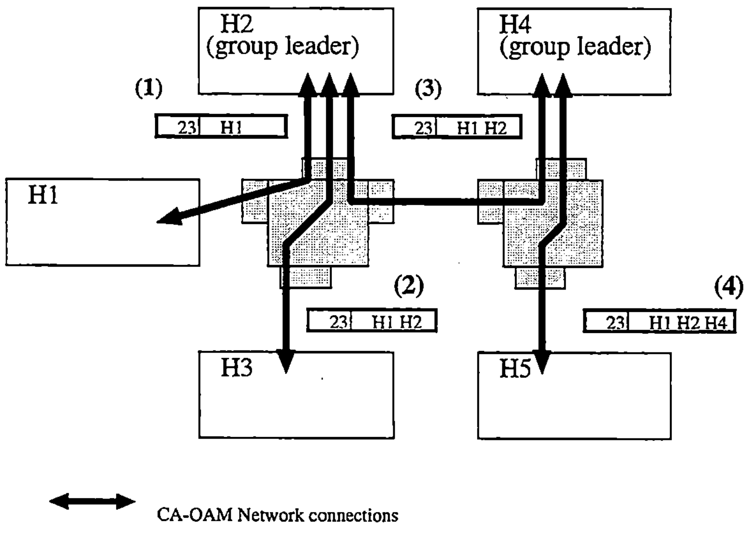


Figure 8.1: Example of the CA-OAM network

where reliable message passing — SSCOP — is used for signalling but single cells are used for operation and maintenance. The messages exchanged between Hollowman entities to return the Hollowman to a stable state are termed Hollowman Operation and Maintenance (H-OAM) messages; in a sense the H-OAM messages are control architecture level OAM messages. All the H-OAM messages described in this section fit into a small number of ATM cells[3].

A set of connections is dedicated to the transport of H-OAM messages. These connections are collectively called the control architecture OAM network (or CA-OAM network). This network allows all hosts to be interconnected, while using a small number of connections. From amongst the set of hosts attached to a switch one host is nominated group leader. Each group leader has a bi-directional connection with all members of its group. In addition each group leader has a bi-directional connection with each of its neighbouring group leaders.

The number of VCIs dedicated to the CA-OAM network on a host which is not a group leader is one source and one sink. On a group leader it is *number of neighbouring groups + number of hosts in groups* sources and sinks, i.e. it is upper bounded by the *number of ports - 1* of the switch.

[3]In fact, in all the experiments they fit into one ATM cell, but since the addresses of the locations the messages pass through are added, to avoid looping around forever, *potentially* the messages can be many cells long.

In the Hollowman implementation, the host-manager is the application end-point of all these connections. The CA-OAM network is established during the Hollowman's initialisation. All host-managers communicate with other host-managers by sending messages to their group leader. The group leader decides if and to where the message should be forwarded. Each host-manager, before forwarding or sending a message, adds its address to the message. This allows the group leader to decide to where it should forward a message and avoids messages constantly being looped around.

Figure 8.1 shows the CA-OAM network between five hosts labelled with addresses `H1` ... `H5`. **(1)** The host-manager at `H1` sends a H-OAM message with identifier 23 to its group leader. The host-manager at the group leader then forwards it to its group members **(2)** and its neighbouring group leader **(3)**. The host-manager at the neighbour sends the message to all its group members **(4)**.

Host-managers, which are running on hosts that have been nominated as group leaders, periodically send out *still-alive* message to all their neighbours and group members and listen on all their sink end-points for arriving *still-alive* messages. Failure to receive such a message from a host — group member or neighbour — leads to the assumption that either one of the ports connecting the two hosts has failed, or that the remote host-manager has failed. The sequence of events that occurs after failure is best explained with an example.

Example: suppose the network is as shown in Figure 8.2. The group leaders for the three switches, `cogan`, `aynho` and `aller`, are respectively: `thistle`, `irishsea` and `heather`. The diagram shows the Tempest applications running on each host. Suppose that the link between `cogan` and `aynho` is broken.

After `thistle`'s host-manager has not received a *still-alive* message from its neighbour for a given time it will start the recovery process. Since `thistle` is still receiving *still-alive* messages from `clyde`, it deduces that the switch it is connected to is still functioning and the port by which it is attached to that switch is working. Therefore, the cause of the non-reception of the message is one of the following:

- its neighbour's host-manager has failed;
- the port or link by which its neighbour is attached to its switch has failed;
- the port or link through which its switch is attached to its neighbouring switch has failed.

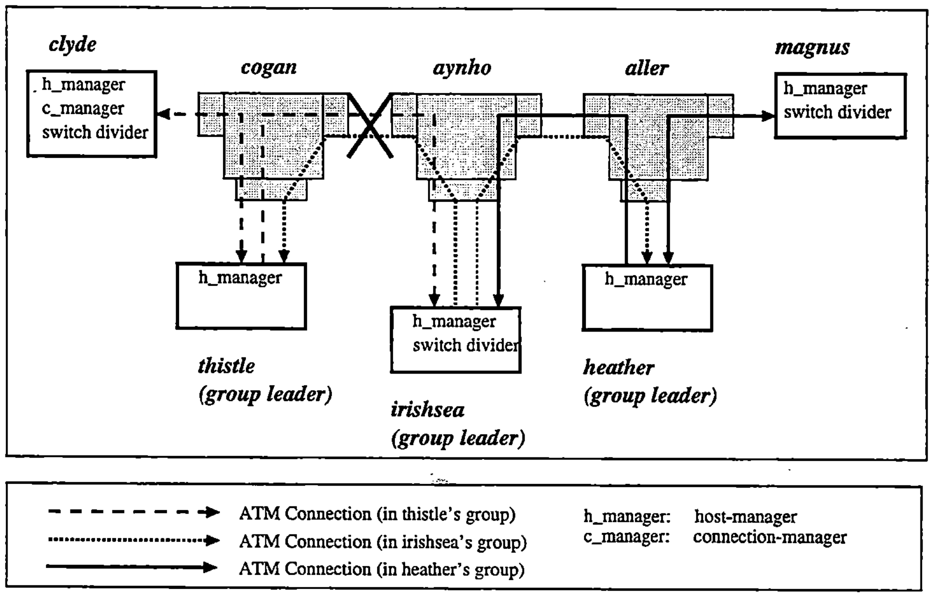


Figure 8.2: Robust error recovery

The first two cases have the same effect, i.e. it is no longer possible to signal to `irishsea`. They are, for the recovery process, therefore equivalent. The latter case can be distinguished from the first two by asking the Ariel server implemented by `cogan`'s switch divider about the state of the port. In the example shown, the switch dividers are run on a workstation, so communication with them is carried out using the CA-OAM network. This will verify that the port is down. The host-manager then initiates the recovery process for its part of the network.

This entails sending an *are-you-running-a-Connection-Manager* message to all the members of its groups and then waiting for them all to reply. There is only one member in `thistle`'s group — `clyde` — and as it is running the connection-manager it returns `true`. The host-manager on `thistle` then sends the message *port-failed*, with the appropriate parameters, to all its group members. The connection-manager on `clyde` behaves in the same way as that described in Section 8.2.1. It uses the Ariel server to remove the parts of the connections that use the port, and that cross `cogan`. It changes its topology so that `aynho` and `aller` are no longer part of its domain and removes all its local state about connections that cross them.

The host-manager on `irishsea` behaves similarly. Normally it sends an *are-you-running-a-Connection-Manager* message to all the members of its group and waits for a reply. Since in the example it has no group members, it immediately sends the *are-you-running-a-Connection-Manager* message to its neighbours. Its neighbour, `heather`, checks if it is running the connection-manager by consulting the host-trader, and when it finds that it is not, it forwards the message to all its group. Since all return `false` and it has no other neighbours it returns `false` to `irishsea`. The host-manager on `irishsea` deduces that no connection-manager exists in its subnetwork.

The host-manager then starts a new connection-manager on `irishsea`. This connection-manager determines the topology of the subnetwork by asking each reachable switch divider for the allocated partition on that switch. The connection-manager running on irishsea then requests each of the reachable switch dividers to reinitialise the VCI space in the Hollowman's virtual network and each of the reachable host-managers to release all their connections, i.e. all existing connections are lost[4]. The result of the recovery process is to split the original Hollowman domain into two separate and distinct ones.

8.2.4 Sharing the CA-OAM Network

Section 8.2.3 described how a single instance of the Hollowman responds to a port failure. Within the Tempest, many instances of different control architectures may be running simultaneously over the same physical network. They will all need to detect and respond to network failure.

Each of them could define its own individual mechanisms for doing this. This is wasteful, as each of them would be doing similar things in response to failure. Moreover, the message exchanges described in Section 8.2.3 are general enough that many different control architectures could multiplex OAM messages over the same CA-OAM network[5].

To share the CA-OAM network, some entity must be capable of multiplexing and dispatching messages to and from the appropriate control architectures. An entity, termed the *Tempest site*, was created to experiment with sharing the CA-OAM network. A Tempest site runs on each host within the Tempest and is the application end-point of the connections. In this configuration, the CA-OAM network is a distinct virtual network within the Tempest, created at start-of-day.

[4]It would be possible to save these connections by correlating the state of the switch dividers and the host-managers, but this has not yet been attempted in the current implementation.

[5]When control architectures have their own well-defined patterns of interaction to recover from failure, e.g. P-NNI, they use their own separate mechanisms.

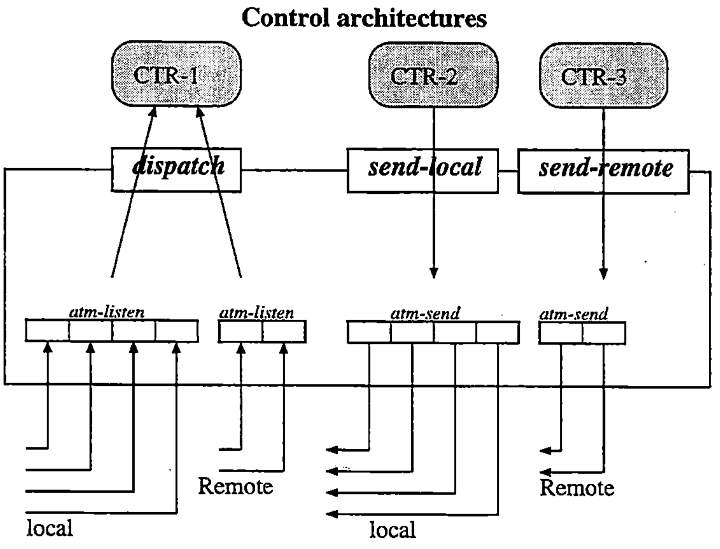


Figure 8.3: The Tempest site

Since CA-OAM messages may only be passed on this virtual network, the amount of resources assigned to operation and maintenance is bounded.

At network creation, the Tempest network builder assigns each control architecture a unique identifier encoded in one integer. Adding this identifier to the start of a CA-OAM message allows messages from different control architectures to be distinguished. The total length of the message is also sent. No other constraints are placed on the structure of OAM messages by the Tempest site. Control architectures are free to define their own patterns of communication over this fundamental infrastructure, e.g. reliable communication. The Tempest site could implement policies for preventing a given control architecture using up all the resources of the CA-OAM, e.g. setting a bound on the number of CA-OAM messages a control architecture can send per second. This is the subject of future work.

The Tempest site allows control architectures to send messages to other group members by calling *send-local.* When invoked on a non group leader host, it sends the message to the group leader; when invoked on a group leader host it sends the message to all the members of the group. The Tempest site *send-remote* operation enables a control architecture to send a message to all the group's neighbours. It is only defined for group leader Tempest sites. Figure 8.3 shows three control architecture entities communicating with the CA-OAM using the

Tempest site.

A control architecture entity wishing to receive OAM messages from the Tempest site on the same host as itself must register with that Tempest site. When a control architecture entity registers, it is informed about whether the host is a group leader or not. The control architecture passes its identifier and the interface references to the operations to call back when a message is received for that control architecture. A Tempest site communicates with the control architectures using the normal DPE, but all such communications are local as they reside on the same machine. In the current implementation, two call backs are defined: one, *port-failure*, is the means by which a given Tempest site informs the local control architecture entities about a network failure. The other, *received-message*, is the means by which the Tempest site dispatches a message to a control architecture entity after receiving it from another Tempest site.

The Tempest sites exchange *still-alive* messages amongst themselves. The *still-alive* messages have a special identifier and are not dispatched to control architectures. If the Tempest site fails to receive a *still-alive* message from its peers, it initiates the failure recovery process by calling all the control architecture entities registered at that host with the *port-failure* callback. In this way, different control architectures share the CA-OAM network to recover from failure.

8.3 Generalisation

The scheme defined in Section 8.2.4 allows different control architectures to share the CA-OAM network, allowing them to implement their own, control architecture specific, recovery processes in response to port failure. To adapt this arrangement to be generally usable for fault management, some additional issues need to be addressed:

- Each message of every control architecture goes to every host in the physical network regardless of the topology of the control architecture's virtual network. This is inefficient.
- For two entities in a control architecture to exchange a CA-OAM message, requires them both to agree on the meaning of that message. It is not possible for a new message to be introduced without updating all control architecture entities about its meaning.

- The previous sections have only described how to detect and recover from one simple type of failure. Other types of failure require more correlation of information.

These points are now considered in more depth.

8.3.1 Directed Message

In the scheme defined in Section 8.2.4, the *send-local* and *send-remote* operations send a control architecture message to a set of hosts. A control architecture's domain consists of only some subset of the physical network. Therefore, messages relevant to one control architecture are sent to hosts which are not part of its domain. For example, suppose that in Figure 8.2 there were another host attached to cogan, but that it was not part of the Hollowman control architecture's virtual network. The *are-you-running-a-Connection-Manager* message would be sent to the Tempest site at that host. This does not cause any problem, as the Tempest site checks if an incoming message's identifier corresponds to any registered control architecture entity and if not simply drops it. The number of dropped messages increases as a function of the number of control architectures and the size of the physical network. If the solution defined in Section 8.2.4 is to scale, then there is a need for more discrimination about where CA-OAM messages are sent.

A Tempest site does not have any knowledge about the topology of the control architecture's virtual network. Giving it this knowledge would greatly complicate the scheme as the Tempest sites would have to be informed about the topology of all the virtual networks and this information would have to be constantly updated as these changed. So the Tempest sites cannot make the decision as to where a CA-OAM message should be sent.

Alternatively, the *send* operations could be modified so that a control architecture entity using the Tempest site could specify to which locations the message should be sent. This would require that control architecture entities on each site maintain a complete view of their topology, leading again to more complexity.

8.3.2 Dynamic Message Type

The meaning of each message exchanged between control architectures must be fixed in advance. For example, the payload of the Hollowman OAM message starts with a message identifier which defines the type of that message. The host-manager, on receipt of the message, knows the operation that must be performed, for example, *are-you-running-a-Connection-Manager* means that it should return whether its host-trader contains a connection-manager offer or not. The introduction of a a new CA-OAM message, e.g. *how-many-connections-have-you-got*, requires the updating of all of the host-managers.

8.3.3 Correlating Information

The network failure described in Section 8.2 is simple. In general, fault management is more complex involving the gathering of many different types of information from many diverse entities. If, for example, the port in Figure 8.2 was oscillating between being *up* and *down*, then the different parts of the control architecture might not be able to agree on whether a failure had occurred or not. The control architecture entities in cogan's groups might believe that they had become separated from aynho and aller, while the entities in aynho's and aller's groups might believe they were still connected; [**Dupy91**] gives an example of this type of problem in a commercial network. The different views must be correlated to come to a conclusion about the actual state. For information to be compared it must first be gathered; the more centralised the correlation of information the less likely it is to be accurate or understandable because the further information must travel the more probable it is to be affected by the instability of the network.

The next section explains how these problems can be alleviated by sending code as well as state in the OAM messages.

8.4 Fault Management with Mobile Code

Mobile code is data that can be executed as a program; an autonomous mobile code system is one in which an executing program can stop executing on one location, and ask to be moved to another location on which its execution will be resumed. The combination of mobile code and the environment it executes in is called a mobile agent. [**Nwana96**] gives an overview of the research currently being carried out on various types of mobile agent systems.

Experimentation in passing autonomous mobile agents in CA-OAM messages has allowed some of the issues raised in Section 8.3 to be addressed. In this system, each control architecture has its own set of mobile agents, supplied to it by the Tempest infrastructure. The mobile agents of a control architecture are informed by the Tempest infrastructure about the topology of the virtual network of their control architecture. CA-OAM messages are tagged as either executable or non-executable. The Tempest site treats non-executable messages as defined in Section 8.2.3, while executable messages are loaded and executed. The control architecture specific mobile agent can communicate with the control architecture entities at that host using the normal interfaces of those entities. The sequence of operations that a mobile agent uses to perform its function is determined by the mobile agent itself. Since new mobile agents can be introduced without affecting a control architecture, message types can be added dynamically.

The mobile agent uses its knowledge about the topology of its control architecture's virtual network to determine where it should be sent next. The mobile agent may also take additional factors into account, for example where it has already been, when taking this decision. This allows the CA-OAM messages to discriminate about where they are sent without adding complexity to the Tempest sites or the control architecture. Moreover, the normal operation of the control architectures is kept completely distinct from their exceptional behaviour when managing failure.

Mobile agents allow processing to be brought to state, rather than state to processing. The mobile agent can examine the state at one location and then correlate this with the state which it finds at the next execution location. Since the mobile agent can process the state locally, it can reduce the amount of information to be transported. If this saving is larger than the size of the agent, then the total amount of data that needs to be transported is reduced. As the agent is control architecture specific, it need only examine parts of the state that concern its control architecture.

The use of mobile agents reduces the need for doing the correlation centrally, allowing a more scalable solution. It increases the probability of correct fault detection as the distance between the location where an alarm is emitted and the location where it is interpreted is shortened, thereby reducing the amount of 'noise' that has to be filtered out. It allows more adaption, since the mobile agents can change their behaviour as a function of the state of the network. For example, if they detect that alarm emission is causing the network to overload they start suppressing the emission of alarms. Such a system is inherently decentralised, allowing information to be gathered and management decisions to be made closer to the concerned network elements.

The next section describes a strategy for recovery from a complex failure requiring a large amount of correlation.

8.5 Experiments with Tempest Mobile Agents

This section describes some practical experiments that have been carried out to test the use of mobile agents for fault management within the Tempest environment.

8.5.1 Overview

Research into network control at the University of Cambridge Computer Laboratory has resulted in the implementation of many experimental control architectures. An incorrectly programmed control architecture which only partly deletes or creates a connection pollutes the VCI space. Sometimes failure can occur in subtle ways. For example, programming error in handling concurrent connection creation requests can lead to erroneous connections being created. The control architecture's view of the network becomes out of phase with the real state. This is an example of the problem of desynchronization that was described in Section 8.1, although in this case it is the control architecture rather than the network that fails.

The existence of erroneous connections may not become manifest until some time after their creation. Deciding whether the entire set of connections that a given control architecture possesses is valid or not requires correlating information from the virtual channel tables of all the switches that the control architecture uses. This task is laborious and better automated. It would have been possible to write a central entity which constantly polled the switches across the network to determine whether the VCI space for all control architectures was healthy or not. This would require the transfer of the complete virtual channel tables from all switches, and is therefore not a scalable solution. Consider a switch that has 16 ports with a single virtual path used for connections and 256 virtual channel identifiers associated with each path. The number of distinct entries in the virtual channel tables is 4096. Assuming that each entry is a sextuple of integers, and all integers are encoded in 2 bytes, then the entire state of the switch's virtual channel tables takes up 48 kilobytes. On the other hand, as is demonstrated in Section 8.5.3, the amount of code required to do the correlation is quite modest. It makes more sense to move the program to the switch state rather than the other way around. Mobile agents offer a natural way of achieving this.

The experiments that this section describes address a practical problem; this problem is particular to the experimental environment but is an example of a class of problem in which distributed data must be correlated.

8.5.2 Automatic Resource Freeing

When a control architecture starts, the Tempest infrastructure can initiate one or more monitoring agents dedicated to it. These agents know:

- the resources assigned to the control architecture;
- the location of the Ariel servers in the control architecture's virtual network;
- the location of a control entity capable of determining if a connection is still in use.

The mobile agents move between these locations using the CA-OAM network, supplied by the Tempest infrastructure and shared between all control architectures.

Each agent constantly moves between the Ariel servers and examines the state of the parts of the virtual channel tables that have been allocated to its control architecture. The agent follows connections upstream, determining if sinks on one switch have corresponding sources on the upstream switch. It only carries relevant information between the Ariel servers and does all correlation locally, reducing the amount of information that needs to be carried around. At the end of its tour of the switch dividers, the agent moves to the location of a known control architecture entity to report on any potentially erroneous connections it has found.

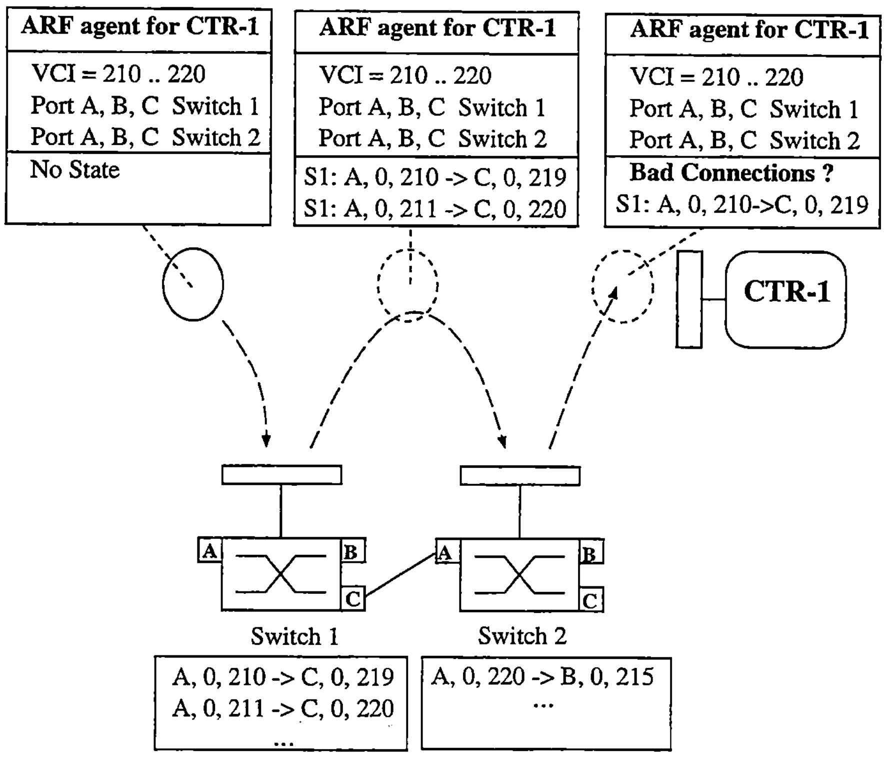


Figure 8.4: Robust error recovery

Control architectures may have their own reasons for maintaining connections which appear invalid to the agent, so the agent cannot make the decision independently of the control architecture. For instance, as explained in Section 5.1, the control architecture may leave redundant connections in place if pending new ones can make use of them. Another reason that the agent should liaise with the control architecture is that it may have read switch state which was the result of an incomplete control architecture operation. The control architecture can recognise such cases and tell the agent that the state it found to be potentially erroneous is in fact valid.

The simplest version of the agent flags an alarm to a human operator if the control architecture confirms that the connection should not exist. The human operator then takes appropriate action, for example to clean up the VCI space manually. To make the recovery process more dynamic, the agent's function was enhanced so that it could automatically free erroneous connections as soon as it identified them. Such agents are called Automatic Resource Freeing (ARF) agents. They are useful during testing since they avoid the need for constantly stopping and restarting control architectures.

Figure 8.4 shows an example use of an ARF agent. The agent is associated with a control architecture CTR-1, which has the VCI space 210 ... 220 allocated on the ports A, B and C of both Switch-1 and Switch-2. All connections are made with a virtual path identifier of zero.

The agent goes to the location of the Ariel server for Switch-1 and collects all the state associated with connections, using the Ariel interface. It then goes to the location of the Ariel server of Switch-2 and correlates the information it has brought from Switch-1 with that of Switch-2, to determine if any of the connections is potentially erroneous. In the example, the connection on Switch-1 from port A, VCI=210, to port C, VCI=219, (A,0,210)→(C,0,219), is suspicious, because (A,0,219) is not in use on Switch-2. The ARF agent then goes to the location of the interface to its control architecture and asks whether this connection is known. If not, it is removed.

If 'ARFing' is done too quickly then resources are wasted, as the ARF agents must be frequently transmitted and executed; if it is done too slowly then the inconsistencies between switch and control architecture may cause connections to be inexplicably refused. 'ARFing' is made adaptive by varying the number of ARF agents executing and the frequency with which they move as a function of the number of inconsistencies found in the immediate past. When an agent returns to the control architecture, the control architecture can change the agents behaviour by modifying certain parameters, e.g. the amount of time it should wait at each Ariel interface. The control architecture can also launch more agents when it is proper to do so.

The ability of ARF agents to move themselves means that the processing moves to the state, rather than the state to the processing (i.e. to the control architecture). Since ARF agents are smaller than the entire contents of the network's virtual channel tables, the amount of network communication required to clean up resources is reduced.

8.5.3 Implementation

The implementation was carried out by David Halls using the Tube [**Halls97**] mobile agent system, mentioned in Section 5.4.1, in order to demonstrate its utility. The Tube allows the sending and receiving of marshalled expressions written in Scheme to and from a network. The Tempest site is capable of identifying, unmarshalling and executing the Tube mobile agents. The size of an ARF agent is around 1 kilobyte. A full account of the actual implementation of ARF agents is given in [**Halls97**, **Halls98**] and is not repeated here.

8.5.4 Discussion of the Utility of Mobile Agents

Executing CA-OAM messages places a lower bound at the latency at which they can be processed. An executable CA-OAM message may be many cells long, it must be assembled into a program and executed within an appropriate environment. However, the efficiency with which Operation and Maintenance functions are executed is of much less importance than their need for robustness and making minimal demand on network resources. Typically, failure recovery is initiated some small number of seconds after the failure's manifestation; the overhead in executing interpreted code is therefore not critical[6]. Of more concern is that the use of software mobile agents in the way suggested requires every location in the network that they visit to be capable of executing them. To some extent, the usefulness of the approach defined here depends on the ubiquity of environments to run those agents and the security guarantees which those environments can enforce.

8.6 Related Work

The work of the ATMF and ITU-T on operation and maintenance [**ITU-T93a**] has already been mentioned in Section 8.2.2. OAM cells are organised into a five layer hierarchy; *F1* cells carry information about the physical layer and higher layer OAM cells advance up the protocol stack. OAM cells are used for fault detection, notification and performance testing. The work described here extends the OAM cell technique to an open signalling environment.

The *MAGNA* project [**Magedanz96**] has been exploring how the TINA model can be extended to address the drawbacks of traditional client-server interactions. The proposed framework covers both control and management. The

[6][**Halls97**] gives results for the latency with which the elements in an array of integers on a remote server can be added together by a client using: an RPC call to obtain each element and performing the sum at the client, or sending a Tube agent from the client to the server and performing the sum at the server. For a configuration with the client and server both running on Sun Ultra Sparcs, communicating using TCP/IP and an array containing one hundred thousand integers, the former method requires more than 5 seconds, while the latter less than 400 ms.

work presented in Chapters 6 and 9 and in this chapter, all looks at how certain network functions can be dynamically extended; the issues related to signalling and fault management are radically different and there is no reason to believe that the patterns of interaction that are valid for one should also be valid for the other.

British Telecom (BT) [**Appleby94**] uses mobile agents for updating routing information in a large source routed circuit switched network. The work uses two types of agents: a *load* agent which uses Dijkstra's algorithm for updating routing tables and a *parent* agent which determines at what frequency these updates are performed from observing the overall load of the network. Simulations of BT's U.K. PSTN have been used to motivate the assertion that the agents allow more adaptation than more conventional load balancing techniques. The work presented in this chapter is similar to that in [**Appleby94**] and has shown how such agents can be implemented in a real network.

Agent Tcl [**Gray97**] is a software agent system used for the distributed gathering of information. Early experiments suggest that mobile agents allow better adaptation in fast changing networks and secondly that the parallelisation of the task permits greater efficiency[7].

The *WAVE* project [**Sapaty94**] defines a general purpose low-level language interpreter which can be run on a variety of network nodes and which allows the propagation and execution of a user defined WAVE string across the network. WAVE has been used for a number of very diverse projects including:

- integration of distributed databases;
- distributed coordination of multiple objects in space;
- management of collective behaviour of robots;
- management of open computer networks.

Some of the related work detailed in Section 6.7 ([**Tennenhouse96, Yemini96**]) and 9.7 ([**Meyer95, Waldbusser95**]) is also relevant to this survey but is not repeated here.

[7]Interestingly the authors state that smart routing is problematic, because: *Unfortunately, this [routing] information is usually hidden inside the routers and is not propagated to user workstations.* In an open signalling environment such as the Tempest, more information about the physical state of the network can be made available to clients.

8.7 Summary

This chapter has looked at the issues related to implementing fault management functions within the Tempest. Some of these issues are general to all control architectures; for example the avoidance of the failure recovery process causing further failure. Others are particular to open control systems; open control systems may be physically separated from the network devices they manage, and therefore OAM messages sent in-band will not reach all control entities. The fact that the Tempest is designed to support multiple control architecture, each of which has its own control architecture specific reaction to failure, requires the failure recovery mechanism to be flexible enough to accommodate all of them, while still allowing common features to be shared.

This chapter concludes that general fault management within the Tempest is best achieved by having a virtual network, dedicated to OAM, over which control architectures can send their own OAM messages and define their own patterns of communication for failure recovery. Little constraint is placed on the structure of these messages, so they can contain executable as well as non-executable data. The experimental Automatic Resource Freeing function described in this chapter supports the assertion that executable messages permit less centralised and more adaptive fault management.

Chapter 9

Caliban: A Switch Interface for Health Functions

Health functions are those dedicated to ensuring the correct functioning of the network. Tempest clients will wish to manage their virtual network in the same way as they currently manage their physical one. This chapter describes a switch interface — *Caliban* — which allows network administrators to verify and ensure the overall health of their virtual networks within the Tempest environment.

9.1 Introduction

Traditionally within ATM there has been a distinction made between network management, e.g. TMN [**ITU-T92b**], and network control, e.g. Q.2931 [**ITU-T94b**]. This separation of network functions into two groups is based as much on who performs the function as what the function does: in general a human network operator (or 'intelligent' proxy, e.g. an expert system) is the initiator or recipient of a management function, while control takes place without operator intervention. This has led to some confusion about what is management and what is control. Some authors [**Crosby95**, **Crutcher93**] have decided it is best to distinguish them only by the time-scales on which they operate. In this view there is no sharp division but only a continuum from functions such as in-band traffic management at the scale of microseconds right up to functions such as network provisioning at minutes and even longer time-scales.

Although this gives a single simple parameter on which to differentiate between functions, it ignores completely the reason for performing the function. This dissertation distinguishes between network functions based on their purpose. Network operators require two different groups of functions to exploit their networks:

- *control functions* that allow a network to be used;
- *health functions* that permit the health of the network to be maintained.

Control is analogous to the useful functions offered by a program, while health is akin to debugging, profiling, garbage collecting, etc. If the network was perfectly reliable, possessed resources that were more abundant than the maximum demand and had an invariant topology, then health functions would not be required. Health functions are used by the network operator to enable the detection of problems and to help in the determination of their cause, for example by correlating the manifestation of faults against the values of counters held in switch memory. A health function may be as simple as some convenient presentation of the memory of switch, or as complex as an expert system examining and correlating the historic information drawn from many distinct sources. [**McConnell97**] gives a description of some case studies of actual problems experienced in commercial networks. These include:

- frayed cables causing high-level messages to be lost leading to many retransmissions and the slowing down of the network;
- faults in the routing tables of a LAN connected to a large meshed backbone causing all the backbone's traffic to be routed through the LAN;
- ARP cache reflush values erroneously being set for milliseconds rather than minutes, leading to periods of slow response.

[**Cisco97**], which describes known problems in a commercial ATM switch, is also revealing about how faults manifest themselves in commercial networks. For example:

- setting the sustained cell rate to zero for a VBR connection caused the switch to crash;

- failure to flush packets under certain conditions within LAN emulation caused packets to be continuously exchanged between the LAN client and bus resulting in a *broadcast storm*;
- connection cells at the ingress switch could be policed at a higher rate than configured due to a wraparound in the expected arrival time counter.

What is striking about all of these situations is that the cause of the problem is not easy to determine from its manifestation. In general, extensive monitoring is required to determine the cause of failure and this involves an overhead in network, computer and human resources. The sophistication of the health functions that a network operator implements is a trade-off between the cost of carrying out this monitoring and the cost of network failure. This in turn is likely to be strongly influenced by the nature of the services that the network supports. [**Feldkhun97**, **Yamamura97**] state that companies running *private customer networks* over a public network often wish to manage their slice of that network in the same way that they would their own physical one. It is likely within the Tempest environment that certain network operators will require similar flexibility over their virtual networks. The Tempest must offer infrastructure which allows different network operators to implement their own health functions without them interfering with each other.

The Ariel switch interface, discussed in Section 2.2.4, allows the full range of ATM control functions to be implemented within a logically distinct control layer. The Ariel interface is a set of fundamental ATM switch operations that allows a higher level control architecture to implement its required control functions. Ariel is, in effect, a switch Abstract Data Type and the control functions built using Ariel need not know any details about the implementation of a particular switch.

Health functions require more information than control functions about the switch implementation to achieve their goal. For example, it may be necessary to read any piece of information contained in the switch memory in order to determine why it is not functioning as expected. For this reason, management interfaces have tended simply to be abstractions of memory with some restricted means of retrieving and modifying memory locations rather than higher level operations. The next section explores the existing management protocols.

9.2 Existing Solutions

The Simple Network Management Protocol (SNMP) [**Case90**] and the Common Management Interface Protocol (CMIP) [**ISO/IEC91**] are the two most commonly cited management protocols in the literature[1].

SNMP and CMIP both define the state of a network element as a Management Information Base (MIB). A MIB is made up of typed MIB elements — ASN.1 typing [**ISO/IEC87**] is used in both SNMP and CMIP. MIB elements have a position within the MIB and are ordered with respect to each other. Each MIB element has a unique identifier which permits it to be distinguished from other MIB elements. The MIB is only a convenient presentation for state that the network element reveals to a manager and has no influence on the real location at which that state resides. The MIB integrity is preserved by some software process running on or 'close' to the network element. Managers communicate with this process to perform management operations; in both SNMP and CMIP this process is called an agent. In this chapter, the term *agent* covers both SNMP and CMIP agents and any software processes in other management schemes which have an equivalent task.

Each protocol defines a wire representation for the transfer of data, e.g. the ASN.1 Basic Encoding Rules (BER), and a set of generic primitives for retrieving and modifying network element state. In order for a given manager to be able to manage two different pieces of network equipment requires each of them to support a significant subset of the same MIB.

The generic nature of management operations means that many manager-agent exchanges may be required to execute a given management operation. For example, [**Rose91**] states that one of the criteria for including a particular element in SNMP's standard MIBs was that it should not be derivable from other MIB elements. So, as a result, in order to determine the total number of ICMP packets received at an IP interface it is necessary to read twelve different MIB elements and add them together. CMIP solves this problem with an *action* primitive which invokes a function defined in the MIB that can read and modify many pieces of state and perform calculations at the server. However, it is generally not possible to define *a priori* a complete set of actions for all possible health

[1]SNMPv2 [**Case96**] is a newer version of SNMP addressing perceived problems in SNMP relating to security, bulk transfer and manager-to-manager communication. It has not been as successful as its antecedent, so this dissertation concentrates on the first version of SNMP.

functions.

9.3 Problems with Existing Solutions

SNMP is a very successful, widely deployed management protocol. Currently all commercial LAN ATM switches support SNMP. The reasons which militate against simply using SNMP alone for management of Tempest virtual networks are:

- **Access Control**: within the Tempest, one control architecture should not be able to read or modify data which belongs to another. It is possible to modify the virtual channel tables of the Fore ASX-200 switch using SNMP operations on the Fore MIB. A control architecture could subvert the strict partitioning imposed by the Tempest if no restrictions are placed on the SNMP operations it can perform. Per-user access control in SNMP is accomplished by checking the SNMP *community* to which the user belongs and deciding how that should influence the predefined access rights of a MIB element. While in principle this could allow very fine grained control, in practice it is often only binary, i.e. if the user is in the appropriate community he has write access to all read-write elements otherwise he has only read access to them. Often there is only a fixed set of these communities. While communities could be created dynamically, this is not possible with many currently deployed SNMP agents. Moreover, the access rights are associated with single MIB elements, while the partition imposed by the Tempest cuts across tables. SNMP communities cannot be used for restricting Tempest control architectures without modifying existing SNMP MIBs and agents. SNMPv2 allows better access control — defining MIB views to which a client is restricted — but these still would not limit retrieval and modification to only certain elements in a table.

- **Micro-management**: [**Goldszmidt95**] identifies the difficulty in achieving generic, efficient management and refers to it as the *micro-management* problem: the nature of management operations means that they are low-level, e.g. stylised reads and writes; in consequence, many operations — and therefore network exchanges — are required to do anything useful. [**Goldszmidt95**] proposes a dynamic incrementable server, termed an elastic server, to tackle this problem. In this system, the client adds code to the management server, for example the SNMP agent, running on the switch.

The client code becomes an intrinsic part of the switch management interface for the client. This is a promising technique and has met with some approval within the SNMP community [**Wellens96**], but is not implemented in any widely available version of SNMP and probably will not be in the near to medium future. This loading of code is particularly attractive within the Tempest, since it allows different Tempest users to configure their management policy for their own needs. For example, users can:

  - correlate and filter data at the MIB;
  - trigger the emission and reception of arbitrary patterns of OAM cells to test performance;
  - add control architecture specific alarms.

  An important restriction is that any code addition must preserve access control.

- **Heterogeneous Management Protocols**: small ATM switches for the home, such as those developed by the Warren project [**Greaves98**], may not even have a processor and so cannot be managed directly by SNMP. Moreover, it might be preferable to use GSMP [**Newman96**] rather than SNMP, for example, when the management operations are restricted to the gathering of simple statistics. So while SNMP is the most used management protocol, there are alternatives. An ideal solution would allow health functions to be as independent of the protocol used for communicating with the switch as possible.

  Clients that are capable of implementing health functions are by definition privileged and have a good knowledge of the nature of the switch. Completely hiding the software that the switch is running is impossible. However, since all management protocols are stylised forms of retrieving and modifying values from switch memory, it is possible to define a small set of operations that all management protocols support, i.e. get value at location, modify value at location and move to next location. Management protocols differ only in what constitutes a location (e.g. a Managed Object in SNMP; a physical memory location in a Warren switch), how they are addressed, and how the values at the locations are encoded.

SNMP is not adequate for building efficient, control architecture specific health functions usable across a range of switches with different processing capabilities in an environment — like the Tempest — which requires fine grained

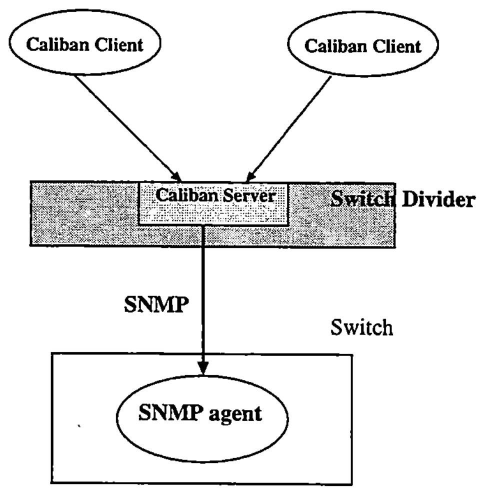


Figure 9.1: Caliban server implemented with SNMP

access control. That said, it would be quixotic to try to define from scratch a completely new management protocol as this would require a large amount of effort and would be unlikely to achieve anything approaching the current deployment of SNMP. The preferred solution is one which lessens the identified problems while remaining backwardly compatible with existing management protocols. This is achieved by defining an interface containing a small set of primitives which can be mapped onto a variety of underlying protocols. This interface is called *Caliban* and is described in the following section.

9.4 Caliban Interface

Caliban is a simple ATM switch management interface adapted for use within the Tempest environment. Network managers communicate with a Caliban server using this interface and the server translates the request into the appropriate format for communicating with the switch. Network managers do not address the switch directly and therefore fine grained access control can be implemented within the Caliban server if the switch itself cannot support it. Within the Tempest, the Prospero switch divider is already required to know the resource allocation of each control architecture so it is natural to run the Caliban server as a part of the switch divider.

Figure 9.1 shows the relationship between Caliban clients, the switch divider and the switch within the context of the Tempest. Caliban may be thought of as an indirection which allows some independence from the precise switch communication protocol and permits the required fine grained access control to be enforced. The cost of this is that the client is restricted to using only those primitives in the Caliban interface, rather than the potentially richer management protocol, as well as some overhead in communication time.

Caliban also addresses the problem of micro-management. Caliban clients can change the behaviour of the Caliban server by dynamically loading and executing code as close to the agent as possible. Dynamically loaded code has the same access control restrictions as its emitting client regardless of where it is executed.

The operations of the Caliban interface are divided into four groups:

- **initialisation operations**: each control architecture has a unique network identifier which allows the switch divider to determine the switch resources allocated to it. This network identifier is obtained from an authorising entity, e.g. the network builder, at start-of-day. Caliban clients must create a Caliban session using their network identifier before any other Caliban operation; all operations in the rest of the Caliban interface are made in the context of a given session.

- **location operations**: Caliban clients view the switch state as a single linked list of locations. Each location has an ASN.1 type. Clients create references to locations using the Caliban interface and all retrieval and modification operations are performed relative to a given location. Clients can only create references to locations which are valid for them. Location reference creation takes a character string as argument. The exact meaning attached to this string is implementation-specific. To some extent, the means of identifying a location hints at the underlying protocol. For example, identifying locations by stringified forms of ASN.1 Object identifiers suggests that SNMP is being used; while stringified forms of physical memory addresses would suggest that the Caliban server can directly read from and write to switch memory.

- **retrieval and modification operations**: Caliban has three operations related to the manipulation of the contents of the switch locations: `get`, `set` and `getnext`. The `get` operation takes a location as argument and returns its value[2]. The value of a location is an instance of an ASN.1

[2]The Caliban `get`, `set` and `getnext` operations only take a single argument, rather than

type in the subset of ASN.1 allowed by SNMP. The `set` operation takes a location reference and an ASN.1 value pair as arguments and, if possible, modifies the value at the location. The `getnext` operation is similar to the `get` operation, but in addition it modifies the location reference so that it points to the next valid location for the client. Although this is very similar to SNMP, note that by adding an indirection Caliban allows access control not present in the underlying agent to be implemented and moreover, these SNMP-like operations are not necessarily implemented using SNMP.

Caliban does not have an equivalent of an SNMP trap or a CMIP notification. Notifications reduce the amount of polling that a manager has to do at the cost of increasing the complexity of both the agent and the manager. Polling requires both the manager and agent to be constantly active and many network exchanges. The next section describes an alternative to both polling across the network and notifications.

- **code loading operations**: clients can load code into Caliban for management purposes; this code is executed as close to the actual switch memory as possible. Three possibilities exist:

  - foreign code can be executed on the switch;
  - foreign code cannot be executed on the switch either for security or technical reasons, but can be executed on the Caliban server;
  - foreign code cannot be executed on the switch or the Caliban server.

  Regardless of where the loaded code accesses the switch locations from, it is subject to the same restrictions as the client that emitted it.

  Figure 9.2 shows a situation in which a client has loaded code into a Caliban implementation using SNMP. The code has been passed by the Caliban server to the switch and is communicating with the SNMP agent across a local version of the Caliban interface. Since it is difficult to elaborate these points without reference to an implementation a fuller discussion of this is reserved for Section 9.5.

  The loaded code communicates with its client using appropriate means, e.g. one or more virtual channels. The messages sent over these channels are sent and received by code with the same provenance, so only the transport level and below needs to be standardised by the infrastructure. For

a list as in SNMP. This was done simply to ease the implementation of the proof-of-concept Caliban server.

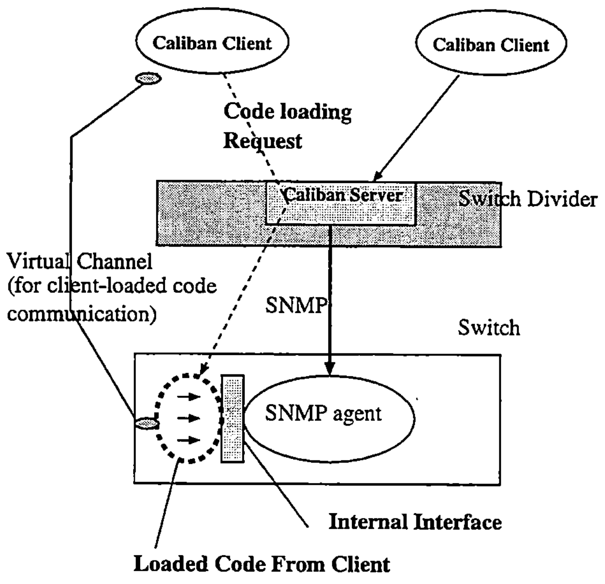


Figure 9.2: Caliban server with loaded code

example, the method of encoding data structures can be decided by each client for each connection. This, as demonstrated in Section 9.6, allows greater efficiency as well as more flexibility. Clients can define arbitrary communication patterns for the exchanges over these connections; alarm, performance data and reconfiguration operations can all be fitted into the same scheme. For example, the loaded code can locally poll a part of the switch state on behalf of a client and notify the client if it is appropriate; the client does not have to poll across the network nor does it have to be capable of receiving and interpreting generic alarms.

9.5 Caliban SNMP Implementation

This section explains how the Caliban interface summarised in the previous section is implemented with SNMP. The SNMP implementation used — UCD SNMP [**Hardaker97**] — is freely available.

9.5.1 Initialisation

Each Caliban client is associated with a session in the Caliban server. A Caliban session is a structure containing the client's access rights, an SNMP community name and a UDP socket. The access rights and community name are obtained from the switch divider using the network identifier. The socket is connected to the SNMP socket on the switch and is used for sending client requests to the switch.

9.5.2 Location

The UCD SNMP agent offers little in the way of assistance for handling fine grained access control. There is a set number of predefined communities which determine whether or not the user has write access or not. So, in this implementation, nearly all access control is performed by the Caliban server.

The string passed to the *create-location* operation is a stringified form of an ASN.1 Object Identifier (OID). The Caliban server verifies that this OID is valid for the user by checking it against the access rights of the user's session. The SNMP access rights are defined by two lists, one which gives the valid prefixes to the OID and the other which gives the valid postfixes. For an OID to be valid it must match at least one of the items in both the prefix and postfix list.

Since OIDs are organised as trees, each prefix defines a subtree and each postfix defines an branch terminating in a leaf. The prefix allows the inclusion of large parts of the MIB without having to list them exhaustively; the postfix permits a finer grain of access control[3]. An example illustrates the point: The *interface* table in SNMP MIB-I contains information about the various network interfaces that a host possesses. The OID of this table is `1.3.6.1.2.1.2.2`. On the workstation on which this dissertation was written, the OID for all the columns in the the ATM interface table terminate in 2 while all those which refer to the Ethernet interface terminate in 3. Including prefix `1.3.6.1.2.1.2.2` and postfix 2 allows a user access to the information about the ATM interface but not the Ethernet.

Similarly, within the Fore ATM Switch, MIB information about virtual channels is contained within many different tables. Each of these tables is indexed

[3]This is similar to the view mechanism defined in SNMPv2, except that views are defined only by a prefix.

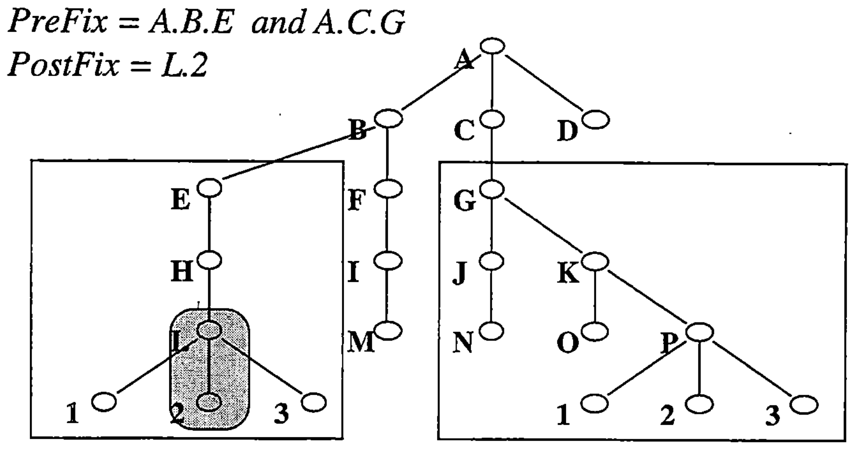


Figure 9.3: Example of access control

by a port, VPI and VCI identifier tuple. Typically, the client's access rights will include postfixes corresponding to the values of these items which have been allocated by the network builder to the client. This means that a client can examine and change only the values in the SNMP MIB which it has been assigned. Figure 9.3 shows an example of how OID prefixes and postfixes are used to limit the set of valid locations for a user within the OID tree. In the example the user only has access to the MIB element whose OID is `A.B.E.H.L.2`.

The list of prefixes and postfixes for a client is generated from a static profile description of the MIB and the network description returned from the network builder. The profile description typically gives the prefixes for the client, while the postfixes are generated from the network description. For example a client that has been assigned on port 9 and VPI 0 the VCI range 200 ... 210, would have postfixes (9,0,200) ... (9,0,210) and prefixes defining which tables the client had the right to examine and modify.

9.5.3 Retrieval and Modification

The Caliban `get` and `set` operations map directly onto their SNMP homonyms. The `getnext` operator is an extremely powerful aspect of SNMP as it is a simple way of dynamically learning about a MIB's contents. The Caliban `getnext` may call the SNMP `getnext` operation one or more times until SNMP returns an OID that Caliban considers to be valid for the user in question.

9.5.4 Code Loading

A user specifies the code to be loaded into Caliban by defining the location of the file in which the code resides. In the current implementation the location is defined as the absolute name of a file in an NFS file system. The code is bytecode runnable on a Java virtual machine. The Caliban server knows how close it can get the code to the switch, and informs the client. The client decides on the mechanism for communicating with the code, for example, by creating a virtual channel between itself and the location at which the code is executing. In order to do this it must possess the necessary network resources. The parent informs the code, before sending it, of the means to receive and send information, e.g. the appropriate VPI/VCI values. If the switch cannot run foreign code, then the code is loaded into the Caliban server itself. If the Caliban server cannot accept code, i.e. if it is not running on a virtual machine, the code is rejected. The loaded code performs its management operations using the Caliban interface regardless of where it is loaded.

In the proof-of-concept implementation, the SNMP agent was run inside a Java virtual machine. This was achieved by creating a shared library from the SNMP agent code; when the virtual machine is started the SNMP code is loaded into the virtual machine and is prompted to begin listening for messages on the relevant SNMP port.

All UCD SNMP access operations are implemented using a function called `snmp_agent_parse`. This function expects a byte sequence as input which it parses as an ASN.1 request. It executes this request and gives a byte sequence as output which is an encoded form of an ASN.1 reply. The internal `snmp_agent_parse` function of the UCD implementation is exposed to the virtual machine using the Java Native Interface (JNI). This allows Java code to call into the SNMP agent directly. Although the `snmp_agent_parse` is implementation-dependent, it is likely that all SNMP agents have a similar function; standardising SNMP agents so that this function is exposed on all implementations would pose no *technical* problem.

The Caliban server passes the user's code and access rights — as a set of OID prefixes and postfixes — to the virtual machine running on the switch, via a well known port. The user's code is then added to this virtual machine where it executes its function using an implementation of an interface built directly on top of `snmp_agent_parse`. This interface has its own `get`, `set` and `getnext` operations, but these call functions in the same address space as themselves and

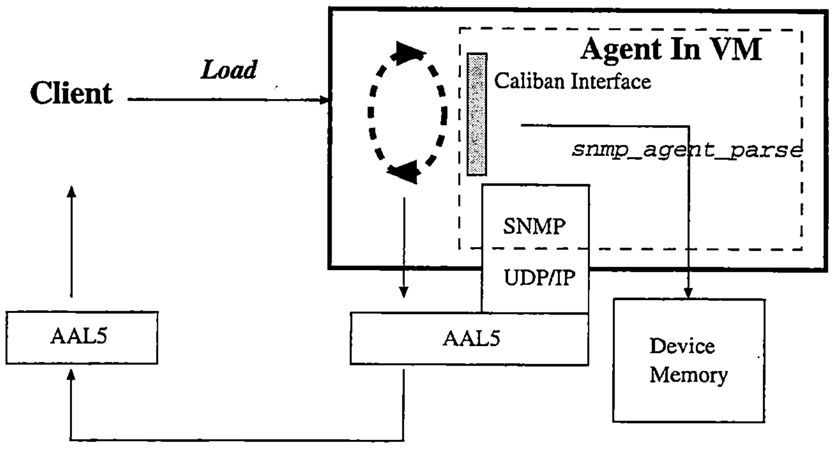


Figure 9.4: Code loading into SNMP agent

hence are very efficient. The access control on these local calls uses the client's access rights supplied by the Caliban server and the calls are as secure as any other form of interaction using Caliban. Although the code is constrained to access only the MIB variables that belong to it, it may inadvertently or deliberately cause problems by consuming too large a share of the resources of the shared server. The environment in which such code runs needs to specify precise bounds for its resource usage. Operating systems for soft real-time services [**Leslie96**] already provide such guarantees; their application to constraining foreign code is the subject of future work.

In the current implementation, the client uses its own control architecture to create virtual channels for communication between itself and the location of the code. The implementation of code loading has been achieved almost without modifying the code for the SNMP agent. The only modification made is the replacement of the *main* function in the code with the mechanism needed for starting it from Java. Existing applications can continue to use normal SNMP operations to communicate with the agent. Figure 9.4 shows the interactions between a piece of loaded code and the SNMP agent within the current implementation.

9.6 Experimental Results

Section 9.3 illustrated by giving examples some of the advantages of loading code closer to the switch. Loading management code closer to the device to be managed has been the subject of a lot of research. However, the literature, see Section 9.7, has largely concentrated on exploring the increased flexibility that code loading allows; this is necessarily qualitative, rather than quantitative. The intention of the experiment described here is to demonstrate how loading code can permit more *efficient* management operations by removing unnecessary overhead in the communication between manager and the agent. Increasing the speed of communication permits the manager to sample more data within a given time frame; this is useful for both performance and error monitoring[4].

The SNMP MIB-I has a variable `ifInUcastPkts` in the `ifTable` which is a counter of the number of IP packets received from a given network interface. The experiment involved observing the evolution of the `ifInUcastPkts` counter over time and comparing the efficiency with which the value of this counter could be retrieved for a workstation's ATM interface using:

- a Normal SNMP client (`N-client`) with a normal SNMP agent (`N-agent`);
- a Caliban SNMP agent (`C-agent`) with dynamically loaded code controlled by a Caliban client (`C-client`).

In both cases the client and agent ran on different machines, so all data exchanges between manager and agent took place across the network. There were three different versions of the dynamically loaded code, which varied in the method by which they returned the value of the counter:

- the first returned each value one at a time (`C-1`);
- the second polled the agent ten times in quick succession and got ten consecutive values of the results before returning them (`C-10`);
- the third did the same but called the agent a hundred times and returned the results in groups of one hundred (`C-100`).

[4]For convenience, the SNMP agent for these experiments was run on a workstation rather than an ATM switch. This should not have any consequence on the conclusions. The Fore ASX-200 LAN ATM switches used in the experiment contain Sparc processors and run SUN OS; they have the same processing capacity as a workstation.

Load	N	C-1	C-10	C-100
L/L	2.2	2.0	1.8	1.7
H/L	4.0	2.0	1.8	1.7
L/H	4.9	4.7	4.5	4.4

Table 9.1: Average time in milliseconds to obtain `ifInUcastPkts` counter

In all three cases the results were returned across an ATM virtual channel. They were read and written using the Fore ATM API with AAL5 as the adaptation layer.

In each run of the test the counter was accessed one thousand times. Twenty runs were carried out for each test case, interspersing runs to reduce the possibility that unintentional variations in CPU usage would have an influence on the comparison between the results. The results for the C test cases do not include the time to load the code since this is a start-of-day operation that the client need only perform once. Moreover, the time to read the code into the virtual machine is constant — some tens of milliseconds — so as the total number of counters exchanged increases, the overhead per-packet decreases until eventually it becomes negligible. The experiment was carried out with different balances of load between client and server. These were:

- lightly loaded client and server workstations (L/L);
- heavily loaded client and lightly loaded server workstation (H/L);
- lightly loaded client and heavily loaded server workstation (L/H).

Table 9.1 shows the average time in milliseconds to obtain a single value for the counter. When both the client and server are lightly loaded (L/L) C-1, C-10 and C-100 are all quicker than N, with the results improving as the number of counters returned in a given PDU is increased. The C case does all that the N case does, but it also performs fine grained access control[5]. The reason that the C cases are quicker is that although the exact same amount of useful information is being sent as in the N case there is no need to encode, send and decode superfluous data,

[5]Consideration of these results should also take into account that both the loaded code and the extension to the SNMP agent are Java bytecode and that interpreted Java is twenty times slower than the C programming language [**Flanagan96**].

e.g. the ASN.1 PDUs; only useful information is sent. In effect, the dynamically loaded code communicates with its client using a client-specific protocol stack.

As the load is increased on the client the `C` test cases do not change, while the time for the `N` test case more than doubles. This is unsurprising, as more work is done by the `N-client` which has to encode, send, receive and decode ASN.1 PDUs while the `C-client` simply receives AAL5 packets. Increasing load on the server causes all test cases to slow down about equally. In summary:

- the `C-agent` is an `N-agent` run in an environment which can be dynamically incremented; the actual SNMP agent does not need to be modified;
- the `C-agent` behaves like an `N-agent` for `N-clients`;
- the same amount of *useful* information is sent in the `N` and `C` cases;
- the `C`-case is quicker than the `N`-case in the tested situations.

Dynamically loading code into a server permits the communication patterns and structure of protocol data units to be defined for each client. This is worthwhile when the reduction in the overhead of communication is greater than that added by the loading and processing of foreign code, i.e. long-lived, information-intensive communication exchanges.

9.7 Related Work

[**Yemini93**] provides a good overview of the issues in network management stating that the problem of network management can be summarised by three fundamental questions: What should be monitored? How should it be interpreted? How should this analysis be used to control the network? [**Meyer95**, **Goldszmidt95**] propose the dynamic addition of code to the SNMP agent as a means to deal with the problem of micro-management mentioned in Section 9.2. [**Crutcher93**] also identifies the problem of micro-management. [**Meyer95**, **Goldszmidt95**] introduce a style of management, called *management by delegation*, that allows management functions to be divided into a set of distributed processes that can be dispatched to the most suitable location for their execution. The claimed advantages of management by delegation are flexibility, scalability and robustness. Many others, for example

[**Busse97**, **Grimes97**, **Keshav97**, **Susilo98**], have proposed variations on the same theme. These are broadly similar to the code loading aspect of Caliban. The distinction comes both from Caliban's emphasis on access control and the evidence offered in this chapter for the hypothesis that code loading can increase efficiency.

[**Vassila97**] describes an implementation of mobile code for Telecommunication Management Networks (TMN) [**ITU-T92b**]. The client program is an *Attribute* of a special type of GDMO Managed Object Class called an *Active Managed Object* (AMO). The client can modify and perform actions on the AMO using normal CMIP operations. The AMO interacts with the other managed objects using local communication.

[**Rose93**] argues that the introduction of network management should make minimal demands on the managed network elements. The work outlined in this paper does not require the network elements to be able to dynamically increment code, only that if they do have this potential then advantage may be taken of it. As [**Wellens96**] points out, current network devices often contain processors and memory exceeding that of only slightly older management platforms. That in itself is not an excuse to be profligate with the resources of the network device, but if resources are available and under used then management tasks should be able to take advantage of them.

RMON [**Waldbusser95**] is a standard SNMP MIB for remote network monitoring of an Ethernet link. Modification of certain tables has as a side-effect the starting or stopping of a probe function for obtaining statistics about network performance. For example, a client can execute a function which will order a user-defined number of hosts by a user-defined statistic. The RMON functions are all predefined — although parametrised — and the client cannot modify them dynamically. RMON's success demonstrates the need for greater computation on the network element itself.

9.8 Summary

This chapter has described Caliban, a switch interface for the use of health functions within the Tempest environment. Caliban makes use of existing management protocols, while attempting to solve problems related to:

- their diversity;
- their lack of fine grained access control;
- the need for most computation to be performed within the manager even when this is unsuitable.

The Caliban interface consists of a simple set of operations which can be mapped onto a variety of different underlying protocols by a Caliban server. In addition, the Caliban server can perform the required access control if the agent is unable to. Clients can load client-specific code as close as possible to the switch agent using Caliban. This allows greater flexibility and efficiency in the communication between client and server.

Although Caliban's design has been influenced by the fact that SNMP is the most commonly deployed management protocol, the solutions proposed here are independent of SNMP. A Caliban implementation using a freely available version of SNMP has been explained in some detail, with an emphasis on the implementation of fine grained access control and dynamic code loading into an SNMP agent. The Java environment has been used as the means for creating an agent which can be dynamically incremented with client code, but the technique outlined is not specific to any particular programming language. It would be possible, for example, to do the same with native code if a dynamic linker were available.

The hypothesis that dynamically loading code into a server can increase efficiency by reducing overhead has been supported with experimental evidence. Another advantage of this technique is that clients may specify client-specific modes of interaction with management servers allowing greater managament flexibility, for example through the use of client-specific alarms.

Chapter 10

Summary, Future Work and Conclusions

This chapter summarises the dissertation. It indicates some potential areas for future work and draws the overall conclusions from the research described here.

10.1 Summary

This dissertation has described the *Tempest* open signalling environment, in which many distinct control architectures may coexist. The Tempest environment depends on being able to make a clear distinction between the control and switching layers, and being able to strictly partition the resources of the switch between distinct controllers. The Tempest consists of:

- a switch-independent control interface, called *Ariel*;
- a switch management interface, called *Caliban*;
- a partitioning mechanism for switch resources, called *Prospero*;
- a network builder for creating virtual networks.

Open signalling within the Tempest framework has been investigated through the implementation of a Tempest-aware control architecture called the *Hollowman.* The Hollowman implements a set of functions which include all required and most optional capabilities of UNI 4.0. The flexibility of this control architecture has been demonstrated through a description of the experiments carried out using it. The control architecture's performance is more efficient than the published results for a variety of implementations of UNI signalling.

Traditional high-level signalling APIs have been identified as being too restrictive for certain types of application, as their generic nature prevents applications from making use of their application-specific knowledge. The basic control architecture has been modified so that users can dynamically extend it with their own application-specific code. This allows them control over their resources at a fine level of granularity, and permits them to take advantage of their application-specific knowledge, for example, to optimise their resource usage. Experimental results confirm the practicality and utility of this technique.

The need for ubiquitous signalling has been addressed by describing how a switch-independent control architecture can interoperate with standards-based systems. The implementation of a simple interoperation protocol has been described and evidence has been offered to support the assertion that this protocol could be replaced with P-NNI when it is more widely available.

The special issues concerning the management of a virtual network in the Tempest environment have been explored. Operation and Maintenance techniques from the ATM standards have been extended so that distributed control architecture entities can detect and recover from failure. Experimental evidence has suggested that allowing messages to be executable permits less centralised and more adaptive fault management to be achieved.

Existing management protocols do not permit the strict partitioning required by the Tempest environment. Moreover, they have a well-known weakness relating to the number of network exchanges that are required to achieve a complete management task. These problems have been addressed by a switch management interface which allows the virtual network's integrity to be preserved during management operations, while also solving the micro-management issue by the integration of code into the management agent. Performance results have shown that the latter — by allowing the removal of unnecessary overhead in communication — permits more efficient management operations to be performed.

10.2 Future Work

Directions for future work have been identified throughout this dissertation. This section examines only some of the more critical points. The issues that remain to be resolved can be divided into three groups, namely those that:

- **are common to all ATM control architectures**, but which may manifest themselves slightly differently in regard to switch-independent control. Some issues in this group include call admission control, atomic synchronisation of resources, accounting and bootstrapping. The work presented in this dissertation suggests that switch-independent control does not make this group of problems harder to solve, while providing the means to experiment and test solutions even in large networks. Moreover, the Tempest framework allows the choice of solution to be varied as a function of the nature of the services the control architecture is supporting.

- **arise from allowing third-party code to gain access to network resources**. The major issue in this group is security. Many of the security problems arising from executing foreign code can be solved if the code is restricted to using only those resources allocated to it. The work presented in this dissertation has shown how this can be achieved for network resources at several different levels of granularity. However, operating system resources must also be considered and this requires operating systems which have been designed with the fine grained partition of resources in mind. Work already accomplished in the domain of soft real-time operating systems may address at least some of the security problems.

- **result from the separation of the switch from the control plane.** The major open issue in this group is network management and in particular fault management. The handling of failure and the management of faults is a problem for any network; switch-independent control might be said to exacerbate this problem as the control plane is less aware of how the switching plane is implemented, and is therefore less able to identify the reasons why it is not functioning as expected. This dissertation has explored some of the issues related to fault management and proposed solutions. The effectiveness of network management techniques can only be really tested 'in the field'; better understanding of the problems of managing a large multi-service network supporting open signalling will result from the

Learnet project [**Crosby97**], which will use the Tempest framework as the basis for controlling a wide area ATM network.

10.3 Conclusions

It is the thesis of this dissertation that:

- **a well-defined low-level interface between the control plane and the switch enables both to evolve independently**: switch vendors can concentrate on building cheap and efficient switches, while network operators can quickly extend their control architectures in reaction to the need for new services. There is no reason why such a control interface should not be standardised, allowing the standard ATMF control architecture and other newer control systems to take advantage of this, and thereby increasing the rate at which innovation can be introduced.

- **an open switch control interface, coupled with a switch resource partitioning mechanism, allows the simultaneous execution of many control architectures over the same physical network.** This elegantly solves the problem of change migration as a new version of a particular control system, can be run on one virtual network, while at the same time allowing existing applications to use another older version. Network operators may customise the implementation of their selected control architecture in order to suit the services their network supports. For example they may only implement those parts of the standards which are actually used by their services, thereby reducing the complexity of the implementation and reducing the overheads in its execution. The network operator may even implement completely proprietary control architectures for implementing advanced control functions or new service features which permit the operator to differentiate their services.

- **open control, by allowing the coexistence of many control architectures, frees the telecommunication industry from having to define one single monolithic control architecture.** As anyone who can obtain a virtual network can effectively become a network operator, an increase in the creativity that can be brought to bear upon the problem of ATM control is to be expected. The complexity of controlling a multi-service network means that this is much needed.

- **open control is a practical technique.** This dissertation has demonstrated, through a description of the structure of a fully functional switch-independent control architecture, the advantages of open control. It has explained how practical concerns, such as robustness and scalability, which are important in determining whether the technique is adopted or not can be addressed. The switch-independent control architecture has served both to motivate the explanation of the problems and as a platform for experimenting with solutions.

In conclusion, this dissertation has supported the thesis that open control is both feasible and desirable.

Bibliography

[Acharya97] Arup Acharya, Jun Li, Bala Rajagopalan, and Dipankar Raychaudhuri. *Mobility Management in Wireless ATM Networks.* IEEE Communications, 35(11):100–109, November 1997. (p. 3)

[Adam97a] C. Adam, M. Chan, J.F. Huard, A. Lazar, and K. Lim. *Binding Interface Base Specification Revision 2.0*, April 1997. University of Columbia Technical Report 475-97-09. (p. 42)

[Adam97b] C. Adam, A. Lazar, and M. Nandikesan. *QoS Extension to GSMP*, April 1997. University of Columbia Technical Report 471-97-09. (p. 38)

[Adl-Tabatabai96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. *Efficient and Language-Independent Mobile Programs.* In Proceedings of ACM SIGPLAN '96 Symp. on Programming Language Design and Implementation (PLDI), pages 127–136, May 1996. (p. 86)

[Agrawal96] P. Agrawal, P. Mishra, and M. Srivastava. *Network Architecture for Mobile and Wireless ATM.* In IEEE - The 16th International Conference on Distributed Computing Systems (ICDCS '96), pages 299–310, May 1996. (p. 90)

[Alexander97] D. Alexander, M. Shaw, S. Nettles, and J. Smith. *Active Bridging.* In Proceedings of ACM SIGCOMM'97, Cannes, France, September 1997. (pp. 97, 99)

[Alles95] Antony Alles. *ATM Internetworking.* Cisco System Inc. white paper, May 1995. Also presented at Engineering InterOp, Las Vegas, March 1995. (p. 10)

[APM92] APM. *ANSAware 4.1. Systems programming in ANSAware Manual.* Poseidon House, Castle Park, Cambridge, UK, 1992. ANSA project. (pp. 34, 43, 49)

[APM95] APM. *ANSAware/RT 1.0 Manual.* Poseidon House, Castle Park, Cambridge, UK, March 1995. ANSA project. (p. 30)

[Appleby94] S. Appleby and S. Steward. *Mobile Software Agents for Control in Telecommunications Networks.* BT Technology Journal, 12(2):104–113, April 1994. (p. 143)

[Arango93] M. Arango and *et al.* *The Touring Machine System.* Communications of the ACM, 36(1):69–77, January 1993. (p. 43)

[ATMF93] ATMF. *ATM User-Network Interface Specification - Version 3.0.* The ATM Forum: Approved Technical Specification, 1993. (p. 10)

[ATMF94a] ATMF. *ATM User-Network Interface Specification - Version 3.1.* The ATM Forum: Approved Technical Specification, 1994. (p. 10)

[ATMF94b] ATMF. *Interim Inter-Switch Signalling Protocol Specification (PNNI 0).* The ATM Forum: Approved Technical Specification, December 1994. af-pnni-0026.000. (p. 122)

[ATMF95a] ATMF. *ATM User-Network Interface Specification - Version 4.0, (UNI 4.0).* The ATM-Forum: Approved Technical Specification, July 1995. af-sig-0061.000. (pp. 3, 10, 13, 33, 35, 40, 61, 63)

[ATMF95b] ATMF. *B-ISDN Inter Carrier Interface Specification - Version 2.0 (B-ICI 2.0).* The ATM-Forum: Approved Technical Specification, December 1995. af-bici-0013.003. (pp. 40, 122)

[ATMF95c] ATMF. *LAN Emulation over ATM Specification, Version 1.0.* The ATM-Forum: Approved Technical Specification, 1995. (p. 19)

[ATMF96] ATMF. *Private Network-Network Interface Specification - Version 1.0 (P-NNI 1.0).* The ATM Forum: Approved Technical Specification, March 1996. af-pnni-0055.000. (pp. 2, 3, 27, 40, 105, 119, 120, 127)

[Barr93] W. J. Barr, T. Boyd, and Y. Inoue. *The TINA initiative.* IEEE Commununications, 31(3):70–76, March 1993. (pp. 5, 41)

[Battou96] Abdella Battou. *Connections Establishment Latency: Measured Results.* ATM-Forum T1A1.3/96-071, October 1996. (p. 82)

[Bellcore97] Bellcore. *Q.Port Portable ATM Signalling Software, Product Information,* 1997. Available at: http://www.bellcore.com/QPORT/qport-ov.html. (pp. 14, 19, 81, 116)

[Biswas95] Subir Biswas and Andy Hopper. *A Representative Based Architecture for Handling Mobility in Connection Oriented Radio Networks.* In Proceedings of ICUPC'95 International Conference on Universal Personal Communications, Tokyo, Japan, November 1995. (p. 101)

[Bloem95] J. Bloem, J. Pavón, H. Oshigiri, and M. Schenk. *TINA-C Connection Management Components.* In Proceedings of TINA'95, Integrating Telecommunications and Distributed Computing - from Concept to Reality, pages 485–493, February 1995. (p. 41)

[Bos98] Herbert Bos. *ATM Admission Control based on Measurements and Reservations.* In Proccedings of IEEE International Performance, Computing and Communications Conference, February 1998. (p. 76)

[Busse97] I. Busse and S. Covaci. *Customer facing components for network management systems.* In Integrated Network Management V, pages 31–43. IFIP & IEEE, Chapman & Hall, May 1997. (p. 162)

[Callon97] R. Callon, P. Doolan, N. Feldman, A Fredette, G. Swallow, and V. Viswanathan. *A Framework for Multiprotocol Label Switching.* Internet Draft, November 1997. (p. 89)

[Cardelli97] Luca Cardelli and Andrew Gordon. *Abstractions for Mobile Computation,* 1997. To be published, Available at: http://www.cl.cam.ac.uk/~adg/Research/Ambit/. (p. 86)

[Case90] J. Case. *A Simple Network Management Protocol.* Internet RFC 1157, May 1990. (p. 148)

[Case96] J. Case. *Version 2 of the Simple Network Management Protocol.* Internet RFC 1905, January 1996. (p. 148)

[Chen94] Thomas Chen and Stephen Liu. *Management and Control Functions in ATM Switching Systems.* IEEE Network, 8(4):27–39, July/August 1994. (p. 10)

[Cisco97] Cisco. *Release Notes for LightStream 1010 ATM Switch Software (Release 11.1).* Cisco System Inc. product information reference manual, March 1997. Available at: http://www-europe.cisco.com/univ-src/ccden/data/doc/hardware/wbu/ls1010. (p. 146)

[Clinger91] William Clinger and Jonathan Rees (editors). *Revised(4) Report on the Algorithmic Language Scheme.* ACM LISP Pointers IV, July–September 1991. (p. 74)

[Crosby95] Simon Crosby. *Performance Management in ATM Networks.* Cambridge University PhD dissertation, May 1995. Available as Technical Report 393. (p. 145)

[Crosby96] S. Crosby, I. Leslie, M. Huggard, J. Lewis, B. McGurk, and R. Russell. *Predicting Bandwidth Requirements of ATM and Ethernet Traffic.* In Proceedings of IEE 13th UK Teletraffic Symposium, Strathclyde University, Glasgow, March 1996. (p. 28)

[Crosby97] Simon Crosby, Jon Crowcroft, Ian Leslie, Lionel Sacks, and Chris Todd. *Proposal for Experimental Academic Research using LEARNET.* BT project description, September 1997. PEARL 1 Document 2-25/4/97-UCL/CAM-(CJT-UCL EE). (p. 167)

[Crutcher93] Laurence Crutcher and Aurel Lazar. *Management and Control for Giant Gigabit Networks.* IEEE Network, 7(6):62–71, November 1993. (pp. 145, 161)

[DARPA81] DARPA. *Internet Protocol - DARPA Internet Program, Protocol Specification.* Internet RFC 791, September 1981. (pp. 68, 89)

[DARPA97] DARPA. *Workshop on Foundations for Secure Mobile Code.* Monterey, California, US, March 1997. Available at: http://www.cs.nps.navy.mil/research/languages. (p. 86)

[**Deering89**] Steve Deering. *Host Extensions for IP MultiCasting.* Internet RFC 1112, August 1989. (p. 61)

[**Doar93**] Matthew Doar and Ian Leslie. *How Bad is Naive Multicast Routing?* In Proceedings of IEEE INFOCOM, San Francisco, California, volume 1, pages 82–89, March/April 1993. (p. 62)

[**Duffield95**] N. Duffield, J. Lewis, N.Connell, R. Russell, and F. Toomey. *Entropy of ATM Traffic Streams: A Tool for Estimating QoS Parameters.* IEEE Journal on Selected Areas In Communications, 13(6):981–990, August 1995. (p. 28)

[**Dupy91**] Alexander Dupy, Soumitra Sengupta, Ouri Wolfson, and Yechiam Yemini. *NETMATE: A Network Management Environment.* IEEE Network, 5(2):35–43, March 1991. (pp. 125, 136)

[**Feldkhun97**] L. Feldkhun, M. Marini, and S Borioni. *Integrated Customer-Focused Network Management: Architectural Perspectives.* In Proceedings of Integrated Network Management V, pages 17–30. IFIP & IEEE, Chapman & Hall, May 1997. (p. 147)

[**Flanagan96**] David Flanagan. *Java in a Nutshell 1st Edition.* O'Reilly and Associates Inc., May 1996. ISBN: 1-56592-183-6. (p. 160)

[**Fore95a**] Fore. *ForeRunner ASX-200, ATM Switch User's Manual.* Fore Systems Inc, 1000 Fore Drive, Warrendale, PA 15086-7502, US, June 1995. MANU0013 - Rev. E. (p. 13)

[**Fore95b**] Fore. *SPANS UNI: Simple Protocol for ATM Signalling.* Fore Systems Inc, 1000 Fore Drive, Warrendale, PA 15086-7502, US, 1995. Release 3.0. (pp. 13, 75)

[**Frey97**] J. Frey and L. Lewis. *Multi-level Reasoning for Managing Distributed Enterprises and their Networks.* In Integrated Network Management V, pages 5–16. IFIP & IEEE, Chapman & Hall, 1997. (p. 127)

[**Garrahan93**] James Garrahan, Peter Russo, and *et al. Intelligent Networks Overview.* IEEE Communications, 31(3):30–36, March 1993. (pp. 39, 101)

[Goldszmidt95] Germán Goldszmidt and Yechiam Yemini. *Distributed Management by Delegation.* In Proceedings of the 15th International Conference on Distributed Computing Systems. IEEE Computer Society, June 1995. (pp. 149, 161)

[Gray97] R. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko. *Mobile Agents: The Next Generation in Distributed Computing.* In Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis, pages 8–24. IEEE Computer Society Press, March 1997. (p. 143)

[Greaves98] David Greaves and Richard Bradbury. *Warren: A Low Cost ATM Home Area Network.* To appear in IEEE Network, 12(1), January/Febuary 1998. (p. 150)

[Grimes97] G. Grimes and B. Adley. *Intelligent Agents for Network Fault Diagnosis and Testing.* In Integrated Network Management V, pages 232–244. IFIP & IEEE, Chapman & Hall, May 1997. (p. 162)

[Grover97] Wayne Grover. *Self-Organizing Broad-Band Transport Networks.* Proceedings of the IEEE, 85(10):1582–1610, October 1997. (p. 95)

[Halls97] David Halls. *Applying Mobile Code to Distributed Systems.* Cambridge University PhD dissertation, September 1997. Available as Technical Report 439. (pp. 74, 141, 142)

[Halls98] David Halls and Sean Rooney. *Controlling the Tempest: Adaptive Management in Advanced ATM Control Architectures.* Accepted for Publication in IEEE JSAC, 1998. (pp. i, 142)

[Hardaker97] Wes Hardaker. *SNMP implementation: UCD SNMP Version 3.1.* University of California at Davis, Davis CA 95616, US, 1997. Available at: ftp.ece.ucdavis.edu:/pub/snmp/ucd-snmp.tar.gz. (pp. 37, 154)

[Heinanen97] J. Heinanen. *Multipoint-to-point Virtual Circuits.* ATM-Forum contribution, ATMF/97-0261, April 1997. (p. 61)

[Henning97] Ian Henning, Steve Sim, Chris Gibbings, Mick Russell, and Peter Cochrane. *A Testbed for the Twenty-First Century.* Proceedings of the IEEE, 85(10):1572–1581, October 1997. (p. 68)

[Hicks97] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles. *PLAN: A Progamming Language for Active Networks.* Submitted to PLDI'98, November 1997. Available at: http://www.cis.upenn.edu/~switchware/PLAN. (p. 97)

[Hjálmtýsson97] G. Hjálmtýsson and K.K. Ramakrishnan. *UNITE - An Architecture for Lightweight Signalling in ATM Networks.* To be published, April 1997. (Also presented at OpenSig Spring'97, Cambridge UK). (pp. 43, 45)

[Hong97] James Won-Ki Hong and et al. *Web-Based Intranet Services and Network Management.* IEEE Communications, 35(10):100–109, October 1997. (p. 127)

[Huberman93] B. Huberman and T. Hogg. *The Emergence of Computational Ecologies.* SFI Studies in the Sciences of Complexity. Addison-Wesley, Reading, MA, 1993. Editors: L. Nadel and D. Stein. (p. 95)

[ISO/IEC87] ISO/IEC. *Open Systems Interconnection, Specification of Abstract Syntax Notation One (ASN.1).* ISO Publication, December 1987. IS 8824. (p. 148)

[ISO/IEC91] ISO/IEC. *Open Systems Interconnection, Common Management Information Protocol Specification.* ISO Publication, 1991. IS 10165-I. (p. 148)

[ISO/IEC95a] ISO/IEC. *Information Processing Systems – Data Communications – Network Service Definition.* ISO Publication, 1995. IS 8348. (p. 33)

[ISO/IEC95b] ISO/IEC. *Information technology – Open Systems Interconnection – International Standardized Profiles: OSI Distributed Transaction Processing – Part 1: Introduction to the Transaction Processing Profiles.* ISO Publication, 1995. IS 12061-1. (p. 65)

[ITU-T91] ITU-T. *Recommendation E.164/I.331. Numbering Plan for the ISDN Era.* ITU publication, 1991. (p. 33)

[ITU-T92a] ITU-T. *Recommendation I.312/Q.1201. Principles of Intelligent Network Architectures.* ITU publication, 1992. (pp. 39, 41, 101)

[ITU-T92b] ITU-T. *Recommendation M.3010. Principles for a Telecommunications Management Network.* ITU publication, 1992. (pp. 41, 145, 162)

[ITU-T93a] ITU-T. *B-ISDN Operation and Maintenance Principles and Functions.* ITU-T Recommendation I.610, ITU publication, 1993. (pp. 100, 127, 142)

[ITU-T93b] ITU-T. *Introduction to CCITT Signaling System No. 7.* ITU Recommendation Q.700, ITU publication, 1993. (pp. 2, 38, 41, 83, 122)

[ITU-T94a] ITU-T. *B-ISDN SAAL Service Specific Comnnection Oriented Protocol (SSCOP).* ITU Recommendation Q.2110, ITU publication, 1994. (p. 14)

[ITU-T94b] ITU-T. *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-T Recommendation Q.2931, ITU publication, 1994. (pp. 40, 145)

[ITU-T96] ITU-T. *Functional Description of the Broadband ISDN user part of Signalling System No. 7.* ITU Recommendation Q.2761, ITU publication, 1996. (pp. 40, 122)

[ITU/ISO97] ITU/ISO. *ODP Trading Function - Part 1 ; Specification.* ISO/IEC IS 13235-1, 1997. ITU-T Draft Recommendation X950 - 1. (p. 34)

[JavaSoft97] JavaSoft. *The Java Telephony API, An Overview 1.1.* JavaSoft white paper, January 1997. Available at http://java.sun.com/products/jtapi/jtapi-1.1. (p. 102)

[Kalmanek97] Charles Kalmanek, Srinivasan Keshav, William Marshall, Samuel Morgan, and Robert Restrick. *Xunet 2: Lessons from an Early Wide-Area ATM Testbed.* IEEE/ACM Transactions on Networking, 5(1):40–55, February 1997. (pp. 27, 43, 44)

[Kant97] Krishna Kant and Lyndon Ong. *Signalling in Emerging Telecommunications and Data Networks.* Proceedings of the IEEE, 85(10):1612–1621, October 1997. (pp. 10, 67)

[Kawamura95] R. Kawamura and I. Tokizawa. *Self-healing Virtual Path Architectures in ATM Networks.* IEEE Communications, 33(9):72–79, September 1995. (p. 90)

[Keshav97] Srinivasan Keshav. *Open Signaling with Active SNMP.* To be published, October 1997. (Also presented at OpenSig Fall'97, Columbia University). (p. 162)

[Kramer92] Michael Kramer and Seshadri Mohan. *Applications of Transaction Processing for Session Management in Multi-Media Information Networks.* In Proceedings of Globecom'92, volume 2, pages 764–769, 1992. (p. 65)

[Kuhn97] D. Kuhn. *Sources of Failure in the Public Switched Telephone Network.* IEEE Computer, 4(30), April 1997. (pp. 66, 67)

[Laubach94] Mark Laubach. *Classic IP and ARP over ATM.* Internet RFC 1577, January 1994. (p. 13)

[Lazar96] A. Lazar, K.S. Lim, and F. Marconcini. *Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture.* IEEE JSAC, 14(7):1214–1227, September 1996. (pp. 5, 10, 27, 42)

[Lazar97] A. Lazar. *Programming Telcommunication Networks.* IEEE Networks, 11(5):8–18, September/October 1997. (pp. 71, 83)

[Lee95] Whay Lee, Michael Hluchyj, and Pierre Humblet. *Routing Subject to Quality of Service Constraints in Integrated Communication Networks.* IEEE Network, 9(4):46–55, July/August 1995. (p. 32)

[Leslie96] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE JSAC, 14(7):1280–1297, September 1996. (pp. 30, 93, 158)

[Li95] Guangxing Li. *Dimma Nucleus Design.* Poseidon House, Castle Park, Cambridge, UK, October 1995. APM Technical Report 1553.00.05. (pp. 14, 38, 80)

[**Magedanz96**] T. Magedanz, K. Rothermel, and S. Krause. *Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?* In Procceding of IEEE INFOCOM, San Francisco, US, March 1996. (p. 142)

[**McConnell97**] McConnell. *RMON Methodology: Towards Successful Deployment for Distributed Enterprise Management.* 8324 Westfork Road, Boulder, CO 80302, May 1997. Available at: http://www.3com.com/nsc/500251.html. (p. 146)

[**McMahon81**] L. E. McMahon. *An Experimental Software Organization for a Laboratory Data Switch.* In Proceedings of the IEEE International Conference on Communications, 1981. (p. 67)

[**Meyer95**] K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt, and Y. Yemini. *Decentralizing Control and Intelligence in Network Management.* In Integrated Network Management IV, pages 4–16. Chapman & Hall, 1995. (pp. 143, 161)

[**Miller97**] David Miller. *Weathering Sonet Alarm Storms.* America's Network, April 1997. Available at: http://www.americasnetwork.com/issues. (p. 125)

[**Minzer91**] Steve Minzer. *A Signaling Protocol for Complex Multimedia Services.* IEEE Journal on Selected Areas in Communication, Vol 9, 9(9):1383–1394, December 1991. (p. 63)

[**Moy97**] J. Moy. *OSPF Version 2.* Draft Standarad, Internet RFC 1247, May 1997. (p. 120)

[**Murthy97**] Shyam Murthy. *UNI 3.1 QoS additions to GSMP.* University of Kansas project report, July 1997. Available at: http://hegel.ittc.ukans.edu/projects/. (p. 38)

[**Newman96**] Peter Newman and *et al. Ipsilon's General Switch Management Protocol Specification Version 1.1.* Internet RFC 1987, August 1996. (pp. 14, 19, 28, 37, 150)

[**Newman97a**] Peter Newman and Greg Minshall. *Quality of Service Enhancements to the General Switch Management Protocol.* To be published, October 1997. (Also presented at OpenSig Fall'97, Columbia University, US.). (pp. 25, 28, 38)

[Newman97b] Peter Newman, Greg Minshall, and Tom Lyon. *IP Switching: ATM Under IP*. Submitted to IEEE/ACM Transactions on Networking, 1997. (p. 67)

[Newman97c] Peter Newman, Greg Minshall, Tom Lyon, and Larry Huston. *IP Switching and Gigabit Routers*. IEEE Communications, 35(1):64–69, January 1997. (pp. 2, 10, 11, 43, 45, 89)

[Ngoh97] Lek-Heng Ngoh, Hongyi Li, and Weiguo Wang. *An Integrated Multicast Connection Management Solution for Wired and Wireless ATM Networks*. IEEE Communications, 35(11):52–59, November 1997. (p. 3)

[Niehaus97] Douglas Niehaus, Abdella Battou, Andrew McFarland, Basil Decina, Henry Dardy, Vinai Sirkay, and Bill Edwards. *Performance Benchmarking of ATM Networks*. IEEE Communications, 35(8):134–143, August 1997. (pp. 81, 82)

[Nilsson95] Gunnar Nilsson, Fabrice Dupuy, and Martin Chapman. *An Overview of the Telecommunication Information Networking Architecture*. In Proceedings of TINA'95, Integrating Telecommunications and Distributed Computing - from Concept to Reality, pages 1–12, February 1995. (p. 41)

[Nwana96] H. Nwana and D. Ndumu. *An Introduction to Agent Technology*. BT Technology Journal, 14(3):55–67, October 1996. (p. 137)

[OMG95a] OMG. *Object Service Architecture*. Technical Report, The Object Management Group (OMG), January 1995. Revision 8.1. (pp. 14, 65)

[OMG95b] OMG. *The Common Object Request Broker: Architecture and Specification Version 2.0 (CORBA 2.0)*. Technical Report, The Object Management Group (OMG), 1995. (pp. 14, 37, 38, 106)

[ORL97] ORL. *OmniORB version 2*. Olivetti/Oracle Research Centre, 24a Trumpington St, Cambridge, UK, 1997. Available at: http://www.orl.co.uk/omniORB/omniORB.html. (p. 81)

[Pratt94] I. Pratt and P. Barham. *The ATM Camera V2 (AVA-200)*. University of Cambridge Computer Laboratory - ATM Document Collection 3, March 1994. (pp. 13, 30)

[Ranson95] R. Ranson. *Less-Than-Transactional Semantics for TINA.* In Proceedings of TINA'95, Integrating Telecommunications and Distributed Computing - from Concept to Reality, pages 243–247, February 1995. (p. 65)

[Rekhter96] Y. Rekhter, B. Davie, D. Katz, E. Rosen, and G. Swallow. *Tag Switching Architecture Overview.* Internet Draft, September 1996. (pp. 45, 89)

[Rizzo97] M. Rizzo and I. Utting. *A Negotiating Agents Model for the Provision of Flexible Telephony Service.* In Proceedings of ISADS 97, 3rd International Symposium on Autonomous Decentralized Systems, pages 351–358. IEEE Computer Society Press, April 1997. (p. 101)

[Rooney97a] Sean Rooney. *An Innovative Control Architecture for ATM Networks.* Integrated Network Management V, pages 369–380, May 1997. (pp. i, 43)

[Rooney97b] Sean Rooney. *Connection Closures: Adding Application-Defined Behaviour to Network Connections.* ACM Computer Communication Review, 27(2):74–88, April 1997. (p. i)

[Rose91] Marshall Rose. *The Simple Book.* Prentice-Hall, 1991. ISBN 0-13-812611-9. (p. 148)

[Rose93] Marshall Rose. *Challenges in Network Management.* IEEE Network, 7(6):16–19, November 1993. (p. 162)

[Rublin94] H. Rublin and N. Natarajan. *A Distributed Software Architecture for Telecommunication Networks.* IEEE Network, 8(1):8–17, January/February 1994. (p. 41)

[Sapaty94] P. Sapaty and P. Borst. *An Overview of the WAVE Language and System for Distributed Processing in Open Networks.* Technical Report, Dept. of Electronic and Electrical Eng., Univ. of Surrey, UK, June 1994. (p. 143)

[Sathaye95] Shirish S. Sathaye. *ATM Forum Traffic Management Specification Version 4.0.* In ATM Forum Technical Committee - Contribution 95-0013, 1995. (p. 16)

[Schmidt97] Douglas Schmidt, Aniruddha Gokhale, Timothy Harrison, and Guru Parulkar. *A High-Performance End System Architecture for Real-Time CORBA.* IEEE Communications, 14(2):72–77, January/February 1997. (p. 81)

[Schoenwelder96] J. Schoenwelder. *Scotty - Tcl Extensions for Network Management*, 1996. Available at: http://wwwsnmp.cs.utwente.nl. (p. 37)

[Sfikas97] Georgios Sfikas, Costas Apostolas, and Rahim Tafazolli. *The UK LINK-PCP Approach to the Wireless ATM System.* IEEE Communications, 35(11):60–70, November 1997. (p. 3)

[Shrivastava97] Santosh Shrivastava. *A Transactional Workflow System for Network Services.* Presented at: OpenSig Fall'97, Columbia University, US, October 1997. (p. 127)

[Shumate96] Scott Shumate. *A Performance Analysis of an Off-Board Signalling Architecture for ATM Networks.* University of Kansas, Graduate School MSc Dissertation, August 1996. (p. 82)

[Smith96] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, D. C. Feldmeier, and W. D. Sincoskie. *SwitchWare: Accelerating Network Evolution.* Technical Report, CIS Department, University of Pennsylvania and Bell Communications Research, June 1996. White Paper, Available at: http://www.cis.upenn.edu/~jms/SoftSwitch.html. (pp. 99, 100)

[Susilo98] G. Susilo, A. Bieszczad, and B. Pagurek. *Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?* In NOMs'98, February 1998. To be presented at the IEEE/IFIP Network Operations and Management Symposium. (p. 162)

[Tennenhouse96] D. Tennenhouse and D. Wetheral. *Towards an Active Network Architecture.* ACM Computer Communications Review, 26(2):5–18, April 1996. (pp. 99, 100, 143)

[TINA-C97] TINA-C. *Network Resource Architecture Version: 3.0.* TINA-C Publication, February 1997. Available at: http://www.tinac.com. (pp. 43, 127)

[**Tolksdorf97**] Robert Tolksdorf. *Programming Languages for the Java Virtual Machine.* Available at: http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html, 1997. (p. 97)

[**van der Merwe97**] Jacobus van der Merwe. *Open Service Support For ATM.* Cambridge University PhD dissertation. To be available as Technical Report, September 1997. (pp. 9, 11, 18, 29, 37, 76, 81, 102)

[**van der Merwe98**] Jacobus van der Merwe, Sean Rooney, Ian Leslie, and Simon Crosby. *The Tempest - A Practical Framework for Network Programmability.* To be published, 1998. (p. i)

[**Vassila97**] A. Vassila, G. Pavlou, and G. Knight. *Active Objects in TMN.* In Integrated Network Management, pages 139–150. Chapman & Hall, May 1997. (p. 162)

[**Veeraraghavan95**] M. Veeraraghavan, T. La Porta, and W. S. Lai. *An Alternative Approach to Call/Connection Control in Broadband Switching Systems.* IEEE Communications, 33(11):90–96, November 1995. (pp. 10, 67, 82, 83)

[**Veeraraghavan97**] Malathi Veeraraghavan. *Connection Control in ATM Networks.* Bell Technical Journal, 2(1):48–64, Winter 1997. (pp. 73, 101)

[**Veitch96**] Paul Veitch, Ian Hawker, and Geoffrey Smith. *Administration of Restorable Virtual Path Mesh Networks.* IEEE Communications, 34(12):96–101, December 1996. (p. 90)

[**Waldbusser95**] S. Waldbusser. *Remote Network Management Information Base.* Draft Standard, Internet RFC 1757, October 1995. (pp. 143, 162)

[**Wellens96**] Chris Wellens and Karl Auerbach. *Towards Useful Management.* The Simple Times, 4(3), June 1996. (pp. 150, 162)

[**Wetherall96**] Dave Wetherall and David Tennenhouse. *The ACTIVE IP Option.* In Proceedings of 7th ACM SIGOPS European Workshop, Connemarra, Republic of Ireland, September 1996. (p. 99)

[**Willmann90**] Gert Willmann and Paul Kühn. *Performance Modeling of Signaling System No. 7.* IEEE Communications, 28(7):44–56, July 1990. (p. 83)

[Yamamura97] Tetsuya Yamamura, Tsutomu Tanahashi, Miyoshi Hanaki, and Nobuo Fujii. *TMN-Based Customer Networks Management for ATM Networks.* IEEE Communications, 35(10):46–52, October 1997. (p. 147)

[Yemini93] Yechiam Yemini. *The OSI Network Management Model.* IEEE Communications, 30(5):20–29, May 1993. (pp. 125, 161)

[Yemini96] Y. Yemini and S. da Silva. *Towards Programmable Networks.* In IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy, October 1996. (pp. 101, 143)

[Zhang93] Lixia Zhang, Stephen Deering, Deborah Estrin, Scott Shenker, and David Zappala. *RSVP: a new resource ReSerVation protocol.* IEEE Network, 7(5):8–18, September 1993. (pp. 68, 88)