Elastic Network Control

Hendrik Jaap Bos

Wolfson College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

August 1999

Summary

Connection-oriented network technologies such as Asynchronous Transfer Mode are capable, in principle, of supporting many different services. Control and management of these networks, however, are often rooted in the monolithic and inflexible design of a traditional telephone network. This is unfortunate, as the speed at which new services can be introduced depends on the flexibility of the control and management system.

Recent attempts at opening up network control and management have achieved promising results. Using non-proprietary interfaces and strict partitioning of network resources, multiple control systems are allowed to be active simultaneously in the same physical network. Each control system controls a virtual network, i.e. a subset of the network resources. Success of this approach has been limited, however, due to the inflexibility of its software components. The way in which resources are partitioned, or virtual networks are built, is determined once and for all at implementation time. Similarly, the control systems themselves are rigid. Building and running a specialised control system in a separate virtual network for each application area, although possible in principle, is too heavy-weight for many applications.

This dissertation presents a solution for these problems, the implementation of which is called the *Haboob*. It represents the next step in opening up the network, by permitting customisation of all aspects of network control, including the software components. For this purpose, an agent environment, called the Sandbox, was developed, which is both language and implementation independent, and general enough to be used for purposes other than network control as well. It includes a simple uniform way for agents on different nodes to interact. Various mechanisms enforce protection and access control.

Sandboxes have been successfully introduced to all components that make up the network control and management system. Code running in Sandboxes is able to extend or modify the functionality of the components. This is called elastic behaviour. The customisability of all aspects of network control and management eases the development of new services. It is shown how recursive repartitioning of resources allows for application-specific control at a very low level and even enables clients to differentiate the traffic policing associated with these partitions. Such low-level control by dynamically loadable code may lead to significant performance improvements. Elasticity has also been introduced to generic services, such as traders, and components on the datapath. Elastic behaviour allows network control and management to be completely open.

When multiple control systems are active, interoperability becomes extremely important. Existing solutions suffer from problems to do with translation of control messages from one domain into those of an incompatible neighbouring domain. These mappings are fixed and suffer from loss of information at the domain boundaries, leading to functionality degradation. This is solved by making the mapping between domains programmable and by establishing inter-domain signalling channels across control domains with only limited functionality. In other words, the interoperability between control domains has been made elastic as well.

It is concluded that elastic network control and management eases the introduction of new functionality into the network.

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Publications

Some of the work described in this dissertation has been published, and is described in references [**Bos97**, **Bos98a**, **Bos98c**, **Bos98b**, **Bos99a**, **Bos99b**] and [**Bos99c**].

Acknowledgements

I would like to thank my supervisor, Ian Leslie, for giving me the opportunity to come to Cambridge and start my PhD in the Computer Laboratory and for his support and encouragement during my time here.

I am indebted to the members and former members of the Systems Research Group, for all help, advice and stimulating discussions, without which this dissertation would never have been completed.

In particular, my gratitude goes to Simon Crosby, Rebecca Isaacs, Richard Mortier and Sean Rooney, for reading and commenting on earlier drafts of this dissertation.

I would also like to thank the people at Sun Microsystems Laboratories, for giving me a new perspective on research, and for making sure that the time that I spent there as an intern was extremely enjoyable. Furthermore, I am indebted to Sun Microsystems for their financial support.

Many thanks to my parents, for their support and encouragement. They have taught me to be curious about and interested in everything.

Finally, I would like to thank Marieke, who travelled all over the world, and gave up her job, house and life in Bangkok to live on student loans in Cambridge. This dissertation is dedicated to her.

Contents

List of Figures

List of Tables

Glossary

AAL	ATM Adaptation Layer
ABR	Available Bit Rate
AIN	Advanced Intelligent Networks
API	Application Programmer Interface
ATM	Asynchronous Transfer Mode
ATMF	ATM Forum
AVA	ATM Video Adapter
BIB	Binding Interface Base
B-ICI	Broadband Inter-Carrier Interface
B-ISUP	Broadband Integrated Service User Part
B-ISDN	Broadband Integrated Digital Services
CAC	Call Admission Control
CBR	Constant Bit Rate
CDV	Cell Delay Variation
CDVT	Cell Delay Variation Tolerance
CLR	Cell Loss Ratio
COD	Control-On-Demand
CORBA	Common Object Request Broker Architecture
CS-1	Capability Set 1
DLA	Dynamically Loadable Agent
DPE	Distributed Processing Environment
DLCA	Dynamically Loadable Control Architecture
DVS	Distributed Video Server
EBW	Effective Bandwidth
ERA	Eternally Roaming Agent
FEC	Forward Error Correction
FSR	First Segment Replication
GSMP	General Switch Management Protocol
IDL	Interface Definition Language
IFMP	Ipsilon Flow Management Protocol
IID	Independent and Identically Distributed
IIOP	Internet Inter-ORB Protocol
IISP	Interim Inter-Switch Protocol
ILMI	Integrated Local Management Interface

IN	Intelligent Networks
INA	Information Networking Architecture
IP	Internet Protocol
IPOA	IP over ATM
INTSERV	Integrated Services Internet
ISDN	Integrated Services Digital Network
ISSC	Inter-Sandman Signalling Channel
ITU	International Telecommunications Union
ITU-T	International Telecommunications Union, Telecommunications sector
JPEG	Joint Photographic Experts Group
JTAPI	Java Telephony API
LAN	Local Area Network
LDT	Large Deviation Theory
LPB	Latency Probability Bound
CTD	Maximum Cell Transfer Delay Max
CTD	Mean Cell Transfer Delay Mean
MCS	Multicast Server
MIB	Management Information Base
MID	Multiplexing Identification Field
MPLS	Multi-Protocol Label Switching
MSF	Multi-service Switching Forum
MTP	Message Transfer Part
NAFUR	Negotiation Approach with Future Reservation
NAT	Negotiating Agents for Telephony
NCL	Network Command Language
NNI	Network-Network Interface
NSAP	Network Service Access Point
OAM	Operation, Administration and Maintenance
OMG	Object Management Group
ORB	Object Request Broker
OSPF	Open Shortest Path First
OSSA	Open System Support Architecture
PCR	Peak Cell Rate
PDU	Protocol Data Unit
P-NNI	Private Network-Network Interface
POTS	Plain Old Telephone System
PSTN	Public Switched Telecommunication Networks
PTT	Post, Telegraph and Telephony
PVC	Permanent Virtual Circuit
PVP	Private Virtual Path
QoS	Quality of Service
REV	Remote Evaluation
RM	Resource Management
RMON	Remote Monitoring of Network Devices
RPC	Remote Procedure Call
RSVP	Resource Reservation Protocol
SAAL	Signalling ATM Adaptation Layer

SAP	Service Access Points
SCAPI	Switch Control Applications Programming Interface
SCGF	Scaled Cumulant Generating Function
SCP	Service Control Point
SFI	Software Fault Isolation
SNMP	Simple Network Management Protocol
SPANS	Simple Protocol for ATM Signalling
SS7	Signalling System No. 7
SSP	System Switching Point
STP	Signalling Transfer Point
SVC	Switched Virtual Circuit
SUFI	Simple Uniform Framework for Interaction
TCAP	Transaction Capability Part
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TINA	Telecommunications Information Networking Architecture
TINA-C	TINA Consortium
TMN	Telecommunciations Management Network
TTL	Time-To-Live
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
UNI	User-Network Interface
VBR	Variable Bit Rate
VC	Virtual Circuit
VCI	Virtual Circuit Identifier
VP	Virtual Path
VPI	Virtual Path Identifier
VPN	Virtual Private Network
VSI	Virtual Switch Interface
XRM	Extended Reference Model

Chapter 1

Introduction

Connection-oriented network technologies such as Asynchronous Transfer Mode (ATM) networks allow various types of traffic to be multiplexed on the same physical connection. The control and management of these networks, however, are often still rooted in the monolithic and inflexible design of traditional telephone networks. This dissertation proposes an open and extensible control and management solution that does not display the rigidness of existing solutions.

1.1 Motivation

The work discussed in this dissertation represents the next step in the evolution of opening up control and management of communication networks. The evolution started with proprietary, closed, rigid and monopolistic systems, often owned and operated by national PTTs (Post, Telegraph and Telephony organisations). Next, standardised approaches with only limited support for installing new functionality were introduced. These systems can be characterised as closed and monolithic. The physical network is controlled by a single, more or less fixed, system and by a single operator (for a detailed overview, see [**Garrahan93**]). In order to allow new services to be implemented and introduced into the network more rapidly, control and management were opened up to some extent, for example in the *xbind* project [**Lazar96a**]. In this approach, new functionality is relatively simple to build and to make available, but control and management are still monolithic, the assumption being that there is a single network operator in a physical network. This limitation was addressed by the *open system support architecture* (OSSA), which allowed for multiple control systems owned by different operators to be active in the same physical network [**van der Merwe98**]. In this approach, which is also advocated by [**Rooney98**], the interface to the network elements is made public and generic and the network resources are partitioned. In other words, the control of network elements is opened up, in the sense that any control software can be used to control a partition of the network, as long as it uses the generic interface. This dissertation represents the next step, in which the control software (including the interfaces) itself is opened up. The implementation of this solution is called *Haboob*. *Haboob* allows clients to customise their control and management solutions completely. This somewhat simplified representation of the evolution of telecommunications network control and management is shown in Figure 1.1.

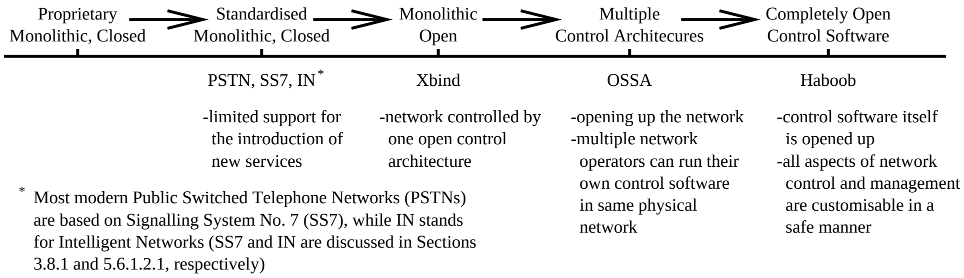


Figure 1.1: Evolution of telecommunications network control

Although the proposed solution is not ATM-specific, the case of ATM is highlighted as it is one of the most widely deployed connection-oriented technologies with support for quality of service. Furthermore, a prototype implementation of the proposed control and management solution was built for ATM.

ATM is a connection-oriented network technology that was designed for multi-service networks on which different types of traffic may be carried simultaneously. Prior to communication, a connection, known as a virtual circuit (VC), must be set up between the participating parties. A *connection type* can be defined as an interaction between parties on a network, in terms of endpoint participation and the roles played by the various parties. Conventional connection types are unidirectional and bidirectional point-to-point communication, and multicast (point-to-multipoint) communication.

Multi-service networks promise to integrate connection types with different Quality of Service (QoS) requirements in a single network. It is impossible to predict either what sort of connection types, or what sort of QoS requirements future applications may demand of a network. To be future proof, the multi-service network should prescribe neither the connection types to be used nor the QoS guarantees that may be associated with them. A particular connection type combined with a particular corresponding QoS requirement is termed a *connection* in ATM. Ideally, the multi-service network should support any possible combination of connection types and QoS requirements that an application may require, while still using the network resources as efficiently as possible.

An individual application, on the other hand, generally requires only a small number of connection types with very specific QoS requirements. The QoS requirements of an application can be loosely defined as its resource and performance demands. Different applications may have widely varying performance requirements, for example concerning the delay and loss in the network. Similarly, there must be support for different levels of resource requirements. Moreover, it should be recognised that resource requirements (and possibly also performance requirements) differ not only from application to application, but may vary over time for a single application as well. A well-known example of this is variable bit rate (VBR) video where scenes with very little change may be followed by resource-hungry action scenes. As many applications with potentially conflicting service requirements may be active at the same time, a true multi-service network should be able to handle the many different connection types, with many possible combinations of performance and resource guarantees, simultaneously.

From the outset ATM was designed with this purpose in mind. ATM allows applications to specify their QoS requirements in a connection request. The request is submitted to an admission control entity which decides whether or not these requirements can be met, taking into account previously accepted connections. Rather than compromising the QoS requirements, either of the new connection or of previously accepted connections, the network will reject connection requests for which the QoS requirements cannot be met. Alternatively, it can enter a state of renegotiation of QoS demands. In this way, it is able to guarantee a certain level of QoS to the applications. Depending on how well the control system does its job, it will be able to provide these guarantees while still making efficient use of the network resources.

This dissertation deals with network control. The ease with which new services can be introduced and used is determined largely by the network control system. Consequently, the success or failure of the multi-service network and in particular of ATM, hinges on the quality of the network control and management. However, although the nature of the underlying ATM data-transfer technology is probably a given, the way in which these networks are controlled is not. A variety of approaches toward the control and management of ATM networks have been proposed by many different organisations, ranging from proprietary solutions proposed by research institutes or vendors of ATM switches, to complex, all-encompassing solutions proposed by standards organisations such as the ATM Forum (ATMF) and the Telecommunications sector of the International Telecommunications Union (ITU-T).

In this dissertation the term *control* is used in a very general sense and often includes *management* tasks as well. The distinction between control and management is mostly one of time scales, where the latter is generally understood to pertain to longer time scales (up to minutes or even days) than the former (of the order of milliseconds). Another distinction is that management consists of those functions concerned with the general *well-being* of the network, while control consists of functions which try to do something *useful* with the network [**van der Merwe98**]. Henceforth, in this dissertation the distinction between control and management is not made, unless required for better understanding of specific issues.

The standardised solutions for network control tend to be complex, highly inflexible and uncooperative: there is a fixed set of control primitives that is considered to be sufficient for all applications, present and future, and it is implicitly assumed that there is only a single control system controlling a piece of equipment, e.g. a switch. However, which approach toward network control and management is the best depends not only on the technology, but also, and even more so, on the environment and the applications. In other words, there is not a single best solution for network control and management.

There are two main problems with the all-encompassing approach. The first is that an implementation of network control and management that is designed to cater to all applications, present and future, tends to be bulky and too heavyweight for certain application areas. For example, a control and management implementation that includes support for a large number of QoS parameters provides a huge amount of redundancy when applied in an environment where a best-effort connection is all that is ever needed. The second problem is the inverse of the first: since there is only a limited functionality that can be offered by any one implementation of network control, certain applications may require functionality that is not supported by the chosen control and management system.

This dissertation argues that no finite set of end-to-end primitives and resource partitioning can be assumed to be able to cater to all possible applications, present and future. As a first step towards solving this problem, research conducted in the Cambridge Computer Laboratory has provided for ways to partition the resources in the network, enabling multiple clients and network service providers each to run its own management and control system on the same physical network. This means that instead of there being one all-encompassing control system, there can now be multiple management and control architectures active simultaneously on the same physical equipment. This is the approach taken in the Open System Support Architecture [**van der Merwe98**] and in [**Rooney98**]. Another approach to allow client-specific solutions to network control is taken in active network technology [**Tennenhouse96**]. Both approaches and their shortcomings are discussed below.

1.1.1 The Open System Support Architecture

In the Open System Support Architecture (OSSA) a high-level distinction is made between several levels of network control and management. The first level considers the *switch control interface*. It consists of the interface between control software and physical switch. In traditional network control, this interface is closed. In other words, the interaction between control software and switch uses a proprietary solution that is specific to the switch and vendor. The OSSA, however, proposes an *open* and *generic* switch control interface called *Ariel*, which contains a small number of basic operations that are expected to be implemented by all switches.

The second level, called *divider* allows the partitioning of resources on switches. Each partition, or *switchlet*, can be controlled by its own control software. For each of the partitions the divider level exports an *Ariel* open switch control interface. This allows multiple control entities to execute the basic switch control operations on their own partitions by invoking operations over their respective *Ariel* interfaces. These requests are validated by the divider level to ensure that they only access resources belonging to the requesting control architecture's switchlet. The third level, called *netbuilder* allows users to build virtual networks by combining *switchlets* on multiple switches.

The fourth and final level is the control entity itself, i.e. the software responsible for such things as connection setup and teardown. It consists of the set of policies, algorithms and protocols that controls the partition of the network allocated to it and that is not part of any of the previous levels. Such a control entity is called a *control architecture*. Examples include P-NNI [**PNNI1.0:96**], IP Switching [**Newman97**], UNITE [**Hjalmtysson97b**], and Xbind [**Lazar96a**].

The OSSA can only be seen as a first step towards opening up ATM network control and management. In particular, the set of primitives in each of the OSSA levels is still fixed. At implementation time it is decided once and for all by the software developer what the functionality of the various components will be. This prevents applications and system operators from taking advantage of knowledge specific to the application or environment at hand. For example, because the open switch control interface only contains a small number of operations (since these operations have to be supported by all switches), more advanced vendor-specific operations, if available on a switch, can not be exploited.

Also, it is probably infeasible to write and run a control architecture for every single application that has specific network requirements. Using a small set of control architectures, each with a fixed set of primitives, is only marginally better than running a single control architecture which is (mistakenly) assumed to be all-encompassing. Developers of applications that have specific control requirements are forced either to write a complete control architecture, or to use a control architecture that is not an ideal match. For truly open network control it is desirable that control architectures allow applications to exploit application-specific knowledge as well. This requires an 'open' management and control architecture.

Furthermore, prescribing the way the network resources are partitioned and virtual networks are built limits their usefulness in situations where more dynamic behaviour is desired. For example, for scaling purposes clients may desire to aggregate switchlets and virtual networks following the private network-network interface (P-NNI) [**PNNI1.0:96**] model of aggregation. However, they may also prefer an altogether different model. For maximum flexibility, no specific model for resource partitioning/aggregation or network building should be prescribed which precludes any other model.

In the networking community, new mechanisms for specific network problems (e.g. admission control, routeing, connection setup, etc.) are proposed all the time. Often, these ideas do not find wide acceptance. Few network operators introduce the proposed solutions of others in their networks, even when the solution would be very useful in their environments. There exist at least four barriers that make it hard to pick up new ideas and mechanisms.

1. Operators often do not have access to particular elements of their control system. For example, it would be very difficult to replace the admission control module of the FORE SPANS signalling software[**FORE95b**]. In many cases, the source code of the control system is a well-kept secret of the vendor and changes cannot be made by the users. The only option may then be to replace the existing control system completely with the one that contains the new mechanism.

2. Even with access to the source code, the introduction of a new mechanism is a difficult task. It requires a thorough understanding of the control software and careful consideration of how and where the mechanism should be implemented

3. It is hard to test a solution's *quality assurance*, except by introducing the new mechanism (untested) to the system. If the mechanism is faulty, it will only be found out when real problems occur.

4. Any changes generally require the control system to be taken down for at least some time.

It can be concluded that there is need for a solution that allows changes to be made to the control system dynamically, using a small set of simple interfaces, without requiring access to the vendor's source code. This makes it easier to take up new ideas for specific control problems, introduced by the research community. The OSSA model can be seen as a useful albeit limited first step towards opening up network control and management.

1.1.2 Active Networks

Active network technology [**Tennenhouse96**], supports a certain amount of programmability of the network. Clients of a packet-switched active network are permitted to inject programs into the network in the form of *capsules* (program code together with embedded data) which get executed at the nodes they traverse. This allows network nodes to process data in an application-specific way. Programming the network is not separated from using it, as all capsules travel in-band. In-band control is problematic as the speed at which control code is executed is strongly related to the speed at which the packets can travel through the network. For example, control code to offload a congested router may be delayed a long time itself due to the congestion.

Even ignoring the long-standing issue of whether the separation of control path and data path is important, it can be observed that most active networks do not address the programmability of such processes as network resource partitioning, and virtual network building. The ability to partition resources in the network and combine them in completely separate virtual networks is an essential property of any open network technology. The reason for this is that it is desirable to run multiple control systems on the same physical network and misbehaviour or faults in one virtual network should have no effect on any other virtual network. The ability to customise the processes that carry out the resource partitioning and the network building is an important feature of open networks which needs to be addressed. Not doing so limits the usefulness of active networks.

This dissertation concerns itself both with opening up the *interfaces* used to control and manage networks, and with enabling applications to extend and modify existing components that exercise such control. These components need not necessarily reside on routeing/switching nodes, as is generally the case in active networks. Thus, while an important aspect is also 'activating the network' (and as such could be placed under the active networks umbrella), the location and role played by the active code is different from that in most active networks (see also Section 3.8.4). As such, the topic of this dissertation can be seen as an attempt to bridge the gap between active networks and open control. A key idea is that extensibility should be incorporated in *all* levels of network control and management. It will be demonstrated that providing support for extensibility and modification of the entire network control allows for a whole new class of applications and control architectures to be built.

A similar observation is a motivation for the NetScript project [**Yemini96**], where intermediate nodes in a network can be programmed with the NetScript programming language within the boundaries of what is also called a virtual network. NetScript's notion of a virtual network, however, is very different from that of the OSSA. It merely indicates a set of related execution environments that exist on some nodes in a network. It is possible to dynamically execute NetScript code in these environments which are linked via so-called virtual links that allow NetScript programs to communicate. In this way, it is possible to implement, for example, a specialised routeing protocol on a specific set of network nodes. It is much weaker than the OSSA's notion of virtual networks though, as it does not provide virtual networks with guarantees regarding resources: every virtual network can interfere with every other virtual network. Moreover, the NetScript project is flawed not only because of the integration of control and data, but also because it requires all data packets to carry a NetScript encapsulation (another header), to allow proper demultiplexing to the appropriate NetScript programs.

This dissertation describes an implementation of network control that is centered around the 'hard' virtual networks of the OSSA, which allow resource guarantees to be given to the control systems. Furthermore, it keeps a clean separation between control and data. Moreover, the data path is not touched at all, which implies that there is no need for undesirable extra headers or encapsulations.

1.1.3 Technology leads, standards follow slowly

A final motivation for the extensibility of network control concerns the time scales at which innovation can be introduced into the network. Currently, innovation only makes its way into commercial networks relatively slowly. For example, the gap between the development of protocols such as RSVP [**Zhang93**] and its large scale deployment in commercial networks exceeds half a decade already. Similarly, the technology of leaf-initiated joins of multicast trees in ATM networks existed a long time before the ATMF User-Network-Interface standard version 4.0 [**UNI4.0:94**] was fully developed, implemented and deployed by the switch vendors. In general, technological innovation moves fast, while standards are inevitably slow to follow.

Ideally, network applications developers would like to use the latest technology, but this objective is hampered, because they have to wait for the standards to catch up. If, on the other hand, an effort was made towards standardising the loading and execution of code in the network (or, as suggested in [**Wetherall96**], the computational model) in addition to, or even instead of, the standardisation of all-encompassing control protocols themselves, new services could be deployed on a much shorter time scale, far ahead of the standards. In fact, it would enable use and deployment of technological innovation at the pace of the innovation itself.

1.2 Contribution

It is the thesis of this dissertation that allowing users dynamically to combine, modify and extend a small set of basic building blocks providing well-defined control and management functions, eases the development and introduction of new and innovative network control.

[**Goldszmidt98**] defines the notion of an *elastic server* as a process, "whose program code and process state can be modified, extended and/or contracted during its execution." In this dissertation, this notion is adopted and generalised for network control. Specifically, distributed elastic network control is defined as:

> *Network control software that consists of multiple interacting components distributed across a network. The interaction takes place across well-defined interfaces, which are open and public (as opposed to closed and proprietary). The elasticity of the network con-*

> *trol means that the components can be dynamically modified and extended in a safe manner.*

Regarding programmable network technology, [**Campbell99a**] also distinguishes between the two schools of thought described in Sections 1.1.1 and 1.1.2 respectively. The first is based on *open signalling*, which abstracts communication hardware using a set of open programmable network interfaces and open access to switches and routers, thereby enabling third party service providers to enter the telecommunications market. The second is formed by the active networks community, which advocates the dynamic deployment of new services at runtime by the dispatch, execution and forwarding of *active packets*, "mainly within the confines of existing IP networks." This dissertation presents a third approach, which combines the useful elements of both open signalling and active networks in a single framework.

A prototype design and implementation of elastic network control called *Haboob* will be discussed in detail. *Haboob*, which spans all levels of network control, follows the distributed systems approach to network control. The service interfaces in *Haboob* will be typed to facilitate conformance checks at compile and bind time.

In the remainder of this chapter, the key contributions of this dissertation are described.

1.2.1 Elasticity

Elastic functionality is provided by a simple and implementation independent design of an execution environment for dynamically loadable code, as well as by a small yet extensible set of basic interfaces to leverage the use of such code. Although a very generic elastic runtime was designed and implemented as part of this work, it should be pointed out that such dynamically loadable code, or agent, technology *per se* is not the focus of this dissertation. Instead, it concerns itself predominantly with opening up network control all the way from the level of building virtual networks to the level of setting up connections across a switch. The dynamic agent environment has been developed merely as a proof of concept.

Elastic versions of all the network control components were built. It will be shown how elastic dividers and switch interfaces allow for very application-specific 'micro-control architectures' to be uploaded, possibly into the switch itself. As an application, a simple way of aggregating switchlets and virtual networks (P-NNI style) using the elastic nature of the various components was achieved.

Furthermore, this dissertation discusses a simple and uniform model for communication between instances of loadable code (and other elastic objects).

The communication model forms part of the proposed computational model for dynamically loadable code which ensures that regardless of which component a client is extending, the execution environment for the code will look the same. Care is taken that the computational model itself can be easily extended. The framework is general enough not to be limited to the field of network control. In fact, it will be shown that the same framework can be used to build extensible applications as well.

1.2.2 Sandman control architecture

As part of the *Haboob*, the implementation of two very different types of control architecture will be discussed. The first, known as *Sandman*, is an advanced control architecture, providing support for a number of new connection types, immediate reservation as well as reservation in advance, repartitioning of virtual networks, and finally, dynamic code loading. It will be shown how a measurements server can be added to the control architecture on the fly, upgrading the connection admission control algorithm from simple peak-based admission control to advanced measurement-based admission control.

The Sandman control architecture addresses a number of independent research issues that are problematic in many existing control architectures. For example, in many control architectures, bursty sources need to be characterised as accurately as possible (e.g. in terms of peak cell rate, sustainable cell rate, burst length, etc.), in order to gain from statistical multiplexing. It is not clear at all how this source modelling should be done. The Sandman therefore avoids extensive modelling and, following [**Crosby95b**], derives the required information about the traffic from observing the traffic itself. This information is then used to allocate resources to the connection. In this way, high resource-utilisation can be achieved without detailed *a priori* knowledge about the source behaviour. Furthermore, issues to do with reservations in advance will be discussed, as well as the interesting possibility of allowing differentiated levels of policing in virtual networks by extending the idea of switchlets into the control architecture. Finally, it will be shown how such a control architecture can be used to build a new type of distributed continuous media server.

1.2.3 Noman: dynamically loadable control architectures

The second type of control architecture comprises the *dynamically loadable control architectures* (DLCA). The DLCAs are built on top of a so-called 'empty' control architecture called *Noman*. Noman is not a control architecture in the normal sense of the word, as it does not provide any network control functionality whatsoever. Instead, it offers simple programming interfaces to dynamically loadable control architectures which enables them to build networks, partition resources and control switching equipment in any way they see fit. In other words, Noman provides an 'empty' runtime for network control on top of which DLCAs can be loaded to provide control functionality.

1.2.4 Interoperability

In an environment where multiple control architectures are expected to be active, interoperability between control architectures becomes an extremely important issue [**Decina97**]. Inter-domain signalling is difficult, because in general the control operations in one domain cannot be mapped perfectly on corresponding operations in the other domain. Instead, there is often no choice except to find the 'best possible match'. Something similar was observed in [**Cidon95**], where it was noted that inter-domain signalling causes information loss because of functionality degradation.

Another problem with current inter-domain signalling solutions is that the mapping between operations from one domain to the next is fixed, even though there may be multiple ways to do this mapping. The 'best match' mapping of signalling operations that is chosen may be optimal for certain applications, but not for others. Using the Sandman control architecture as an example, it will be shown how a general solution for both of these problems was developed.

1.3 Outline

It is important to understand the context in which the research described in this dissertation was conducted. A brief overview of this is given in Chapter 2. Chapter 3 describes the design of the Sandbox, which represents the elastic environment. The computational model will be discussed in general, implementation-independent terms, and it will be shown how it can be applied to the various components in the *Haboob* in particular. In Chapter 4, an implementation of the Sandbox is described in some detail. Next, in Chapter 5, control architectures are discussed. In this chapter many topics relevant to control architectures are touched upon. In particular, the focus will be on application-specific behaviour, advance reservation, measurement-based admission control and resource (re-)partitioning. For demonstration purposes an innovative video server was built, which will be discussed briefly in the same chapter. Chapter 6 is devoted to new ways of achieving interoperability between control architectures. Chapter 7 considers the switch interface, the switch divider process, as well as the netbuilder process. It will be shown how each of these could be extended using Sandboxes and examples of useful applications are given. Chapter 8 describes a number of experiments that were conducted to obtain a measure of the performance of the *Haboob*. Chapter 9, finally, contains conclusions and a discussion of future work.

Related work is discussed throughout the text.

Chapter 2

Research context

This chapter describes the research context, including the underlying approach towards network control on which the rest of this dissertation is founded. It explains the various levels of open control and management: the interaction with switches, the partitioning of resources, the building of virtual networks and finally, the control architectures associated with and controlling these virtual networks. In addition, issues to do with distributed processing environments and bootstrapping are discussed.

2.1 Multiple control architectures

For this work, the realisation that there is more than one valid approach to network control is important. Indeed, standards bodies such as the ATM Forum and ITU-T have tried for many years to standardise the control of ATM networks, with standards that develop in an evolutionary fashion[1], but radically new proposals for network control continue to spring up (for example: [**Bettati95**, **Crosby95a**, **Lazar96a**, **Newman97**, **Hjalmtysson97b**]).

Most of these alternative approaches are born out of dissatisfaction with the existing standards (or the way in which they evolve), or simply because there is a niche for a different type of control in a particular environment. Whatever the reason, it seems likely that the situation in which many different control architectures are used by different users is not going to change for quite some time. Moreover, a variety of solutions in other network technologies such as IP represent entirely different approaches to network control. Examples include the Resource Reservation Protocol (RSVP) [**Zhang93**], TAG switching [**Rekhter97**], Multi-Protocol Label Switching (MPLS) [**Callon99**], Differentiated and Integrated Services Internet [**Shenker97**, **White97b**].

[1] For example, the *User-Network Interface* standard [**UNI4.0:94**] was derived from [**UNI3.1:94**] and [**UNI3.0:93**], which in turn are based on even older protocols, such as Q.931, developed for telephone networks.

Another reason why it is desirable to have multiple control architectures in (separate partitions of) the same physical network is precisely the evolutionary development of control software. Whenever a new version of a standard or proprietary control architecture is developed, problems arise when the network is upgraded. For real applications it may well be required that all nodes in the system are upgraded to the new version, but if not all applications are able to use the new version of the control software yet, it is necessary to keep the old version running also, to ensure backward compatibility. Things get worse if the new version of the control software contains errors. If there is no clear separation between the resources for the old version and those of the new version of the control software, this affects not only applications using the new version, but potentially all other applications as well.

Thus, it is desirable to enable different control architectures to safely coexist in the same physical network. Safe coexistence in this context is understood to mean that faults and misbehaviour in the operation of one control architecture do not adversely affect the operation of any other control architecture.

2.2 Levels of network control

In the OSSA [**van der Merwe98**] a model is introduced in which network control and management are split up in a small number of levels. A prototype implementation of the OSSA, known as the *Tempest*, was developed by various people in the Computer Laboratory. The architecture is shown in Figure 2.1. At the lowest level of the architecture, we find *Ariel*, the non-proprietary interface to the ATM switch. *Ariel* represents an API rather than a protocol. The implementation is not specified. Method invocations on an implementation of *Ariel*, allow users to set up and delete switch connections, gather statistics, etc.

The next level up handles the partitioning of resources on the switch. Partitioning enables one to create what looks like a number of independent small switches, which are called *switchlets*. Each switchlet has its own non-overlapping ranges of virtual path identifiers (VPIs) and virtual channel identifiers (VCIs), as well as its own share of buffers and scheduling guarantees, etc. The partitioning process, called the *switch divider*, supports a separate *Ariel* interface for each of the switchlets (see also Figure 2.1). Since this interface is identical to that of the *Ariel* server on the switch, this makes it completely transparent to the control software that it is actually controlling only a portion of the switch: switchlets look just like switches.

The next level is the *virtual network builder*, or *netbuilder*. The netbuilder is responsible for creating switchlets on the various switches in its domain (by calls to their respective switch dividers) and combining these switchlets into virtual networks. In the same way that switchlets are simple *partitions* of the physical switch, virtual networks are simple partitions of the physical networks.

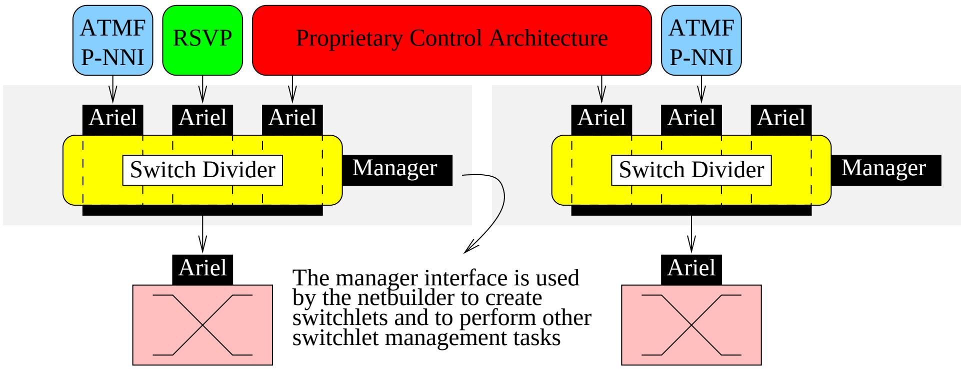


Figure 2.1: Partitioning networks

At the highest level we find the control software itself. The control software requests the netbuilder to create a virtual network on its behalf and as a result is given the references for the *Ariel* interfaces of the corresponding switchlets. It now controls these switchlets in exactly the same way it would control ordinary (unpartitioned) switches.

The next few sections will discuss the various components and their interactions and limitations in more detail. As a convention, throughout this dissertation the names of interfaces are capitalized and printed in italic, while the names of the processes implementing the interfaces are in lower case.

2.2.1 The open switch control interface

As explained above, the *Ariel* switch control interface is used for communication with the switch. The motivation for defining a generic switch control interface is that it gains independence from switch (vendor) specific implementations of control and management. As far as functionality is concerned, there is little to distinguish *Ariel* from these vendor-specific interfaces, except that the former is well-defined, public and as concise as possible to make it generally applicable, whereas the latter generally support very vendor-specific features. The procedures defined in *Ariel* should easily map onto the vendor-specific operations of any switch.

For this purpose, *Ariel* contains a small set of basic operations that can be expected to be implemented by any switch. The control software and the *Ariel* server on the switch communicate according to a client-server relationship. *Ariel* defines an interface only. The actual implementation of the switch control interface (as well as the communication between client and server) is left unspecified. Different implementations are possible for different switches.

It should be realised that *Ariel* is not the only attempt at specifying an open switch control interface. A less general approach is defined by the General Switch Management Protocol (GSMP) [**Newman96**] and yet another, the

Switch Control Applications Programming Interface (SCAPI), can be found in the *xbind* project [**Lazar96a**, **Lazar96b**]. A current standardisation effort, IEEE P1520 [**Biswas98**], aims to define standard interfaces for various levels of network control, including the interaction with the switch. Another standard that is similar to *Ariel* is the virtual switch interface (VSI), originally proposed by the Multi-service Switching Forum (MSF) [**Buckley98**]. Recently however, the MSF decided to settle on GSMP version 2 for the switch control interface [**Newman98**]. Unlike *Ariel*'s specification, this means that the protocol is also dictated. In principle, it makes no difference which switch control interface is chosen, as long as it is general enough for all tasks and basic enough to allow implementation for all switches. In the work presented in this dissertation implementations of *Ariel* over both SNMP (Simple Network Management Protocol, [**Case96**, **Stallings98**]) and GSMP were used[2]. There are many tradeoffs here, e.g. regarding efficiency, reliability, etc. For example, a fast implementation of *Ariel* over a home-grown protocol was developed which outperforms the GSMP implementation by a fair margin, but is less robust. These issues are not discussed any further in this dissertation.

Ariel consists of simple interfaces for configuration, port management, connection setup, QoS, statistics and alarms [**van der Merwe98**]. The *configuration* interface allows *Ariel* clients to discover the switch configuration. *Port management* allows one to check the state of a switch port, or set it to *active*, *inactive* or *loopback*. The *connections* interface enables users to setup or teardown connections across the switch without worrying about QoS. QoS in the form of specific resource requirements can be associated with connections via the *QoS* interface, if and when needed (this means that best-effort connection can by-pass the QoS interface completely. The *statistics* interface allows one to obtain various switch statistics, such as number of cells received, number of cells dropped, etc. The *alarms* interface, finally, enables clients to request notification in the case of a change in the state of a switch port.

It should be noted that the operations in the *Ariel* interface as described here are very general and basic. Their nature, unfortunately, is more or less fixed. This prevents clients of the *Ariel* switch control interface from really exploiting efficient switch-specific optimisations. For example, the *Connections* interface allows one to set up or delete a single switch connection per operation invocation. Even if the switch supported batch connection setup and teardown, this functionality could not be used with *Ariel*. Just as serious is the way in which *Ariel* (admittedly for understandable reasons) has tied itself to the ATMF services categories for QoS control. In other words, a more configurable switch control interface is desired.

[2]The *Ariel* implementations used in this dissertation were carried out by Kobus van der Merwe, Sean Rooney (both of the Computer Laboratory) and the author

2.2.2 Partitioning the resources

A generic switch control interface offers developers of control architectures a certain amount of independence from specific switch implementations. A control architecture that uses *Ariel* operations is capable of controlling any switch for which an *Ariel* server has been implemented. The next step is to partition the resources in the network. The process responsible for this task is called the *switch divider*, or *divider* for short.

2.2.2.1 Switchlets

On request, the divider allocates a subset of the switch resources to a switchlet. It also exports an *Ariel* interface for each switchlet that was created which allows control software to control the switchlet's resources (see also Figure 2.1). The control software (the control architecture) does not control its switch resources directly. All control requests go through the divider. The divider checks whether the request is valid, i.e. whether it does not attempt to access resources that are not allocated to that switchlet. If invalid, the request is rejected with an error message. Otherwise, it is forwarded to the *Ariel* server on the switch. Whether the divider process runs off-switch or on-switch is not important for the model. For prototyping it could just as easily run on a general purpose workstation connected to the switch. In commercial implementations, however, it probably makes sense (for performance purposes) to run the divider on the switch.

Switchlets are created and destroyed by calls over the divider's *Manager* interface. Typically this is done by the virtual network builder, as will be shown in Section 2.2.3. The call to create a switchlet takes as argument a specification of the switchlet to be created in terms of what ports are required, and for each of these ports the VPI/VCI space required, as well the bandwidth and buffer space. Since the latter may be difficult to obtain, switch internals could be hidden behind ATMF service categories. In that case, the client specifies per port what service categories are required as well as what capacity is required for each service category.

2.2.2.2 Shortcomings of current switchlets

The switch divider is an essential component of the *Tempest*. It allows multiple control architectures to coexist safely in the same physical network. This in itself sets the *Tempest* apart from most other advanced control solutions for ATM that have been proposed. However, the divider's tasks are fixed. Creating a switchlet results in some predetermined actions, which may limit its usefulness.

As a very trivial example, suppose that for testing purposes of a new control architecture it is considered desirable to create switchlets in 'emulation mode'.

In other words, switchlets are created on real and active dividers, but without actual communication to the physical switch. This allows network operators to test experimental control software without setting up datapaths. Emulated switchlets also allow operators to test the scalability of control architectures by extending a network with emulated switchlets, i.e. without requiring a large number of physical switches. Such emulated switchlet creation is not normally supported by the divider functionality and can currently not be instantiated on the fly in the *Tempest*. Emulated switchlets are quite simple. A more complex example concerns *aggregate switchlets*, where one logical switchlet really represents a group of switchlets. Both emulated and aggregate switchlets will be discussed in Chapter 7.

2.2.3 Building virtual networks

The switch divider together with the *Ariel* interface allows multiple controllers to be simultaneously active on the same switch. Thus safe coexistence of control architectures is achieved for a single switch. This section describes how a number of switchlets in the network can be combined to form a virtual network. The process responsible for this task is called the *virtual network builder*, or *netbuilder*. Figure 2.2 illustrates the steps involved. Clients of the netbuilder invoke operations in the netbuilder, requesting it to create virtual networks on their behalf (step 1 in Figure 2.2). This is known as *networks on demand*.

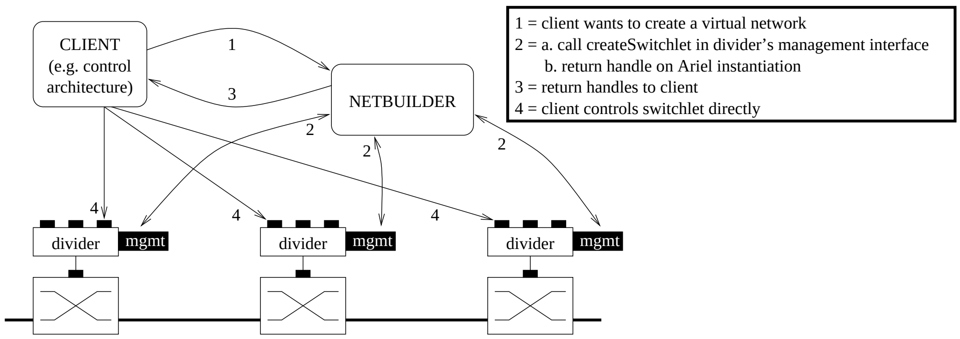


Figure 2.2: Creating virtual networks: the netbuilder

The netbuilder has access to the *Manager* interfaces of the switch dividers in its domain. Via calls over these interfaces, it attempts to create switchlets on the relevant switches and if successful, returns references for the corresponding *Ariel* instantiations to the client (generally a control architecture). This is shown in steps 2 and 3 in Figure 2.2. As shown in step 4, the client can now control these switchlets directly. The netbuilder ensures that the network is 'sensible', so that for example, the VPI/VCI spaces of switchlets for adjacent switch ports overlap. A further responsibility of the netbuilder is to handle incremental growth of virtual networks, e.g. so that new switchlets can be added to an existing virtual network.

Again, the nature of the netbuilding functionality is fixed. It is impossible for the virtual network provider, for example, to keep logs of what sort of networks are created when and by which clients, if this logging functionality is not explicitly supported by the netbuilder to begin with. Similarly, it is impossible to support interfaces to the netbuilder, other than the ones provided at compile time. For example, it would be useful if the netbuilder could support control architecture specific requests for resources, that go beyond what is offered in the "normal" API.

2.2.4 Controlling virtual networks

As shown above, the netbuilder returns to a client (e.g. a control architecture) a handle on what is essentially a subset of the physical network. These virtual networks consist of switchlets for each of which an *Ariel* interface has been exported for control. The entity that actually *controls* the networks is called the control architecture. In [**Rooney98**] the relationship between control architectures and the rest of the *Tempest* infrastructure is likened to that between applications and the operating system. Throughout this dissertation more similarities between operating systems and network control will be pointed out. Indeed, new developments in operating systems have been a source of inspiration for the work presented here.

In essence, control architectures are allocated a subset of the network's resources, which they are free to use as they please. Switchlets allow any control architecture to coexist with any other control architecture. Thus, each of the advanced, complex control systems such as PNNI, IP Switching, and RSVP, as well as any small proprietary solution can be seen as 'just another control architecture' that runs on top of switchlets. Allowing many incompatible control architectures to be active simultaneously in the same physical network, makes interoperability an important issue. The problem of control architecture interoperability will be dealt with in Chapter 6.

On the other hand, it seems unlikely that every application with particular network control requirements will be running its own control architecture. Instead, the number of active control architectures is probably relatively small[3]. For example, application programmers may not want to spend their time writing the one control architecture that fits their needs exactly, opting instead for the closest match that exists already.

This alone is sufficient motivation to open up the control architecture itself, allowing applications to specify their own policies to extend or override existing operations in the control architecture. Another motivation is that it makes it easier to modify control architectures. For example, if a control architecture is shared by a number of applications and one of the applications is upgraded so that its network control requirements are different, it can continue using the

[3] However, a large number of control architectures is not precluded by the switchlets model.

same control architecture. All that is required is a slight customisation of the control architecture for this application alone. Chapter 5 will mention other reasons for wanting to open up the control architecture. The main reasons, however, remain those mentioned in Section 1.1:

1. Control architectures that attempt to cater to all uses in all situations tend to be too bulky for certain uses in certain situations[4].

2. It is impossible to implement a control architecture which implements every conceivable operation (including those desired by future applications).

2.3 Distributed network control

Both the *Tempest* and the *Haboob* have an object-oriented and distributed character. Objects offer encapsulation and clear, well-defined semantics, while distribution follows the distributed nature of components in a network. Instead of using the over-loaded term 'object' for the various entities in the distributed system, the more neutral 'component' will be used.

The components in both systems operate and interact in a distributed processing environment (DPE), where they communicate using remote procedure calls (RPCs) and notifications. The implementation of the various components discussed here has been over CORBA [**OMG91**, **OMG95**] compliant DPEs, notably DIMMA [**Li95**] and OmniOrb [**ORL97**]. CORBA uses the notion of an object request broker (ORB), which allows components to find each other, obtain a reference to each other's interfaces and communicate in a platform independent way by enabling one component to invoke operations in another. Access to components is restricted to calls over well-defined interfaces (specified in an interface definition language, or IDL). The precise location of a remote component (or indeed, its implementation), is generally not known by the calling component.

A component A communicates with a component B by first obtaining a handle on the service offered by B. The handle, known as *interface reference*, contains enough information to locate B, find out what transport protocols to use for communication and establish communication with it. Interface references to which resources have been allocated are called *invocation references*. Part of the CORBA specification is the Internet Inter-ORB Protocol (IIOP) which runs over TCP/IP (Transmission Control Protocol / Internet Protocol). All communication between components in the *Tempest* is over IIOP.

[4] Concerning the complex ATMF signalling and routeing standards, it was first observed in [**Kalmanek97**] that such complex protocols are not needed everywhere. Some examples are given of environments where a lightweight signalling protocol might be preferred.

2.4 The bootstrap virtual network

The previous sections showed that netbuilder and divider communicate with each other and with clients over a network, while it takes the cooperative effort of both of them to create networks. The obvious question is then: who creates the first network over which the netbuilder and divider communicate? This is called the bootstrap network problem.

It is 'solved' by requiring a *bootstrap virtual network* for the *Tempest* environment. The way that the bootstrap network is set up is not specified. It could be as simple as a number of permanent virtual circuits (PVCs) that were set up by the network administrator, together with some simple communication primitives[5]. The bootstrap virtual network is controlled by the *bootstrap control architecture*. The bootstrap network used in this work is classic IP over ATM [**Laubach94**]. In the long term, the heavy-weight classic IP-over-ATM is an unsatisfactory solution, relying as it does on ATMF signalling. Current work in the Computer Laboratory aims to replace it with a light-weight implementation of IP over ATM.

The advantages of using IP in experimental, evolving network control are manifold. There are many tools and resources for IP; naming and routeing problems are automatically solved; CORBA-compliant ORBs can communicate over IP using IIOP, and off-the-shelf implementations of many other useful protocols can be used as well: SNMP, TCP, NFS, etc. The advantages are particularly great, because the bootstrap virtual network can be used to leverage the communication of control architectures as well. In other words, the bootstrap network offers a number of basic communication services which can be used by the *Tempest* network building services and by those control architectures that do not require their own specific protocol for their control messages. In many other research projects IP is proposed as the basis for communication between network control components as well [**Kalmanek97**, **Biswas97**, **Pavon98**].

2.5 Related projects

Many alternatives to the ATMF and ITU-T standards for network control have been proposed. These alternative approaches to network control, as well as ATMF and ITU-T standards, will be discussed in some detail in Chapter 5.6. Several research projects in the Computer Laboratory have influenced the work presented in this dissertation. Although their focus is not on network control *per se*, either their design, underlying principles or implementation, were used in the implementation of the *Haboob*.

[5]This bootstrap problem occurs not only within the *Tempest*. All ATM network control needs some initial means for communication.

2.5.1 Measure

In the *Tempest*, as described in Section 2.2, the issue of resource management and admission control is important at at least two levels: (1) the divider, and (2) the control architecture. At the divider level, it must be decided whether, given the resource guarantees currently given to existing switchlets, a new request for the creation of a switchlet can be accepted or not. Similarly, at the control architecture level, one must determine whether a new connection can be accepted, given the resources currently reserved for existing QoS connections.

Resource management and call admission control in ATM networks are based on the specification of QoS requirements and traffic descriptions in connection setup requests. There are several ways to make such decisions. Signalling protocols defined by the standards bodies (e.g. UNI4.0) contain very elaborate specifications of QoS-requirements as well as detailed traffic characterisation. Traffic descriptions may include such things as peak cell rate, sustainable cell rate, burst tolerance, etc. The problem with such detailed specification is that it is impossible to accurately characterise most sources as their behaviour is not known *a priori*.

The *ESPRIT MEASURE* project investigates a different approach whereby source characterisation is minimal and interesting properties of the traffic are derived from online measurements of the actual flows [**Crosby95b**]. Instead of specifying the sustainable cell rate say, an effective bandwidth for the connection is estimated based on periodic measurement of the traffic. The effective bandwidth can be used for resource management and call admission control. In [**van der Merwe98**], the same approach is used to perform admission control at the divider level. For this purpose, the divider collects statistics from the switch and computes the aggregate effective bandwidth estimates for an entire switchlet. Based on these aggregate effective bandwidth estimates it is determined whether enough resources are available to create a new switchlet.

Thus, only limited use is made of a huge amount of statistics that is collected from a switch. It may well be that other applications would like to use this information as well. However, no application-specific operations can be performed on the data, which is 'owned' by the divider. Furthermore, it is impossible to predict exactly what use applications might want to make of it anyway. For this reason, and because it is impractical to send all the statistics to all applications that are interested in them (due to the volume), such applications would benefit greatly from an approach that provided them with a *elastic traffic server*: an independent and extensible process to deal with traffic measurements. This would allow clients to implement their own ways of processing this data. The application of measurement-based admission control with an independent traffic server is described in Section 5.2. The way in which the traffic server is made extensible is explained in Section 5.4.2.3

2.5.2 The Nemesis operating system

If multiple control architectures are active simultaneously on the same switch, it must be ensured that each control architecture gets its due share of the switch's resources (such as CPU time). Especially if control architectures are allowed to extend the functionality of the components that make up the control and management system, there should be limits on the amount of resources these extensions are allowed to use.

The Nemesis operating system [**Leslie96**] is a 'vertically structured' operating system. Instead of implementing a host of services in the kernel, Nemesis gives individual applications low-level control over the resources they use. This enables it to eliminate QoS cross-talk, which is impossible in traditional operating systems such as UNIX. One of the goals of the operating system is to provide QoS guarantees to applications. Nemesis is termed a *soft real-time* operating system, as it is able to provide applications with guarantees as to the availability of resources. For example, it guarantees to periodically give schedulable domains a specific slice of the CPU (Central Processing Unit) time.

One of the lessons learnt from the Nemesis operating system, is that to be able to provide guarantees, multiplexing of resource use should happen only once and at the lowest possible level. The partitioning of resources (e.g. the CPU) is comparable to the partitioning of switch resources by the divider. It will be shown in Section 3.3.3.2 that the ability to give resource guarantees is also very relevant to elastic network control.

2.6 Summary

In this chapter a framework for ATM network control and management was presented which forms the context for the work in this dissertation. Network control and management is split up in four relatively independent levels consisting of: *Ariel*, the open switch control interface; the divider, which partitions switch resources into switchlets; the netbuilder, responsible for combining switchlets into virtual networks; and finally, control architectures, where the actual control of the virtual networks takes place. It was suggested that although the decomposition of network control and management into the various layers is very useful, the fixed nature of the interfaces and the lack of flexibility in the components limits their general applicability.

Chapter 3

Elastic Control

In Chapter 2, a brief description of the research context was given, in particular of an open approach to network control. It was demonstrated how the fixed nature of the functionality of the components seriously limits their usefulness. For example, although customised control architectures could be built, it is impossible to extend these control architectures on the fly. In Chapter 6 it will be shown that this presents great difficulties when interoperability between incompatible control architectures is required. Also, the way in which the network resources are partitioned and virtual networks are built is determined *a priori*, at the time of implementation of the divider and netbuilder processes. This chapter presents the design of a solution that opens up network control completely—to the extent that functionality can be loaded into all aspects of network control, on a dynamic basis. Such network control is termed *elastic*. The implementation of the design will be called the *Haboob*[1]. The *Haboob* greatly improves the flexibility and 'openness' of network control.

The design was developed using a network architecture that extends the model of Section 2.2. This is discussed in Section 3.1. Next, the computational model is explained in Sections 3.2 and 3.3, which show that the elastic functionality in *Haboob* is based on a generic computational model, expressed in uniform interfaces. A small set of basic operations is defined to support dynamically loadable code. It includes a simple way of extending the basic functionality, allowing, for example, application-specific support. In fact, the computational model is both implementation and language independent and general enough to be applied to many application areas, not just network control. Well-defined interfaces at the component boundaries ensure clean interaction. The remainder of this chapter is devoted to a discussion of elastic versions of the various components of the network model: control architectures in Section 3.4, switch dividers in Section 3.5, netbuilders in Section 3.6, and remaining components in Section 3.7. Finally, related work is discussed at the end of this chapter.

[1] The *Haboob* is a violent sandstorm in the south of the Sahara.

3.1 Decomposition of network control

Before discussing elasticity in detail, this section splits up the network control and management tasks into six relatively independent components. Such a modular approach allows independent development of the components. It is useful also, because it facilitates the differentiation of policy implementation among the components, e.g. with regards to access control (components that cannot cause much harm when broken into may be less well-protected than components that can). In subsequent sections, each of the components will be associated with elastic behaviour. As shown below, the core of the network control model is formed by the four-level decomposition of Section 2.2. However, two new levels are added to complete the model. The complete model contains:

1. a generic switch control interface, known as *Ariel*;
2. *switch dividers*, that are able to partition the resources on switches into *switchlets*;
3. *netbuilders* that combine switchlets into virtual networks;
4. *control architectures* that perform the actual control over the virtual networks;
5. *generic services*, which are independent processes that can be used by any component;
6. *datapath components*, which differ from all other components in that they reside on the data path.

The two additional levels are now briefly discussed. The first new level is formed by the *generic services*. This comprises services that are implemented as independent processes which may be used by any component. An example of a generic service is the *traffic server* (discussed in Section 5.2.2.3), which gathers traffic statistics from an ATM switch, processes these statistics and interacts with other components in the control system that need such information. Another example of a generic service is the trading component that was implemented as part of the *Haboob*. Although similar in functionality, the *Haboob* trader has certain features that distinguish it from more traditional trading mechanisms, such as [**Bearman91**]. The trader will be discussed in Section 4.7.

The second additional level concerns the *datapath components*. Datapath components include applications and endpoints that take part in control, for example a video source that is notified via signalling that it should start sending JPEG (joint photographic experts group) video data of a certain quality on a specific connection. They also include components such as filters, encoders, decoders and other data processing engines that are not endpoints *per se*. In the

Haboob, an attempt is made to separate control and data as much as possible. This is not always desirable, however, and a small amount of *in-band control* is sometimes useful. An example datapath component of this type is the FEC (forward error correction) encoder discussed in Section 7.6.

The decomposition of network control and management tasks is quite unlike approaches such as ANTS [**Wetherall98**], which considers only a single component for programmibility: the router. Such models either (1) do not permit policies like resource partitioning and virtual network building, if implemented at all, to be programmable, or (2) combine them in a monolithic system (often including the data as well), that is hard to understand and maintain. In the *Haboob* the tasks in each of the component levels are relatively independent. As mentioned before, this allows independent development of each of the component levels, which in turn is beneficial to the rapid introduction of innovation. Other models of decomposition are possible. Whichever model is used, however, the question if and to what extent components should allow programmability, modification and extensibility and by whom, deserves careful consideration.

3.2 A design for elastic network control

Programmability in networks and the ability to load code in network nodes in itself is nothing new. Switch manufacturers for example, have always been able to program their switch equipment. Also, whenever switch vendors or network operators update the software in a switch (e.g. in order to support a new version of the signalling protocol), it makes no difference whether this is done by installing a new circuit board, or by using protocols such as TFTP (trivial file transfer protocol) to transfer a new boot image to the switch: what matters is that code is loaded into the switch and the equipment is reprogrammed.

What makes elastic network control different from such reprogramming techniques can be summed up as follows:

- *Time scales.* Upgrading hardware and software on a switch by vendors is a rare event, generally in the order of once every few months or even years; elastic code loading, in contrast, may happen as frequently as many times a second. This allows for very short-lived extensions and modifications, e.g. that provide support for only a single application.
- *Uninterrupted operation.* The process of upgrading the switch controller traditionally takes the switch control out of service for some time; elastic code loading allows changes to be made on a per-application basis without affecting other applications at all.
- *Access restrictions.* Changing network control and management software is typically a privileged task that only one or a few people are permitted to perform (e.g. the system administrator). Elastic control allows access

restrictions at a much finer granularity. For example, it may grant access to the internals of a specific control architecture to *everybody*, while allowing a smaller group to obtain access to certain operations within a switch divider and restricting only the most sensitive network operations (such as granting access rights to certain operations) to the system administrator.

Conceptually, this can be thought of as introducing safe execution environments to the component levels where foreign code can be loaded and run. This is illustrated in Figure 3.1 for the first four component levels. In the figure, dynamic execution environments have been introduced in the netbuilder, switch divider and control architecture components (and even in an application). The figure also shows that the dynamic code running in these environments is able to extend the normal interfaces. Furthermore, there are two types of execution environments in the switch divider: one for the management part and one for the switchlet/*Ariel* part. Figure 3.1 is only intended to give a flavour of what elastic control is about. The details will be explained in the remaining sections of this chapter. Observe that in the figure only the *Ariel* server in the divider has been considered, not the one on the switch. There is no *technical* barrier, however, to having the switch support such an execution environment as well, except in the case of very dumb switching equipment that may not want to be burdened with such functionality. For now, a *safe* environment is taken to mean a runtime in which foreign-code is 'sand-boxed'. In other words, where the code cannot perform operations that compromise the integrity or security of the component. Security issues will be addressed in Section 3.3.3. In the next few paragraphs the nature of the execution environments of Figure 3.1 will be discussed in more detail. Informally, the elastic execution environment will be called the *Sandbox*.

3.3 Elastic components

Elastic components can be defined as components that provide basic building blocks and allow dynamically loadable code to combine, modify (override) and extend these building blocks to provide new functionality. Code that is dynamically loaded onto some runtime has been around for a long time in the field of agent technology [**Pham98**]. In fact, the elastic environment presented here is similar to existing agent technologies in that it could be used as a generic mobile agent platform. The techniques and implementations are applicable to many problem areas, not just network control. However, this dissertation concerns itself with network control. The focus is on the application of dynamically loadable code to the network control problem, where the code-loading functionality is part of a more generic mobile agent platform. Therefore, such code will from now on be referred to as *dynamically loadable agents* or *DLAs*.

Following [**Franklin96**], an agent is loosely defined as "a system situated within and a part of an environment that senses that environment and acts on

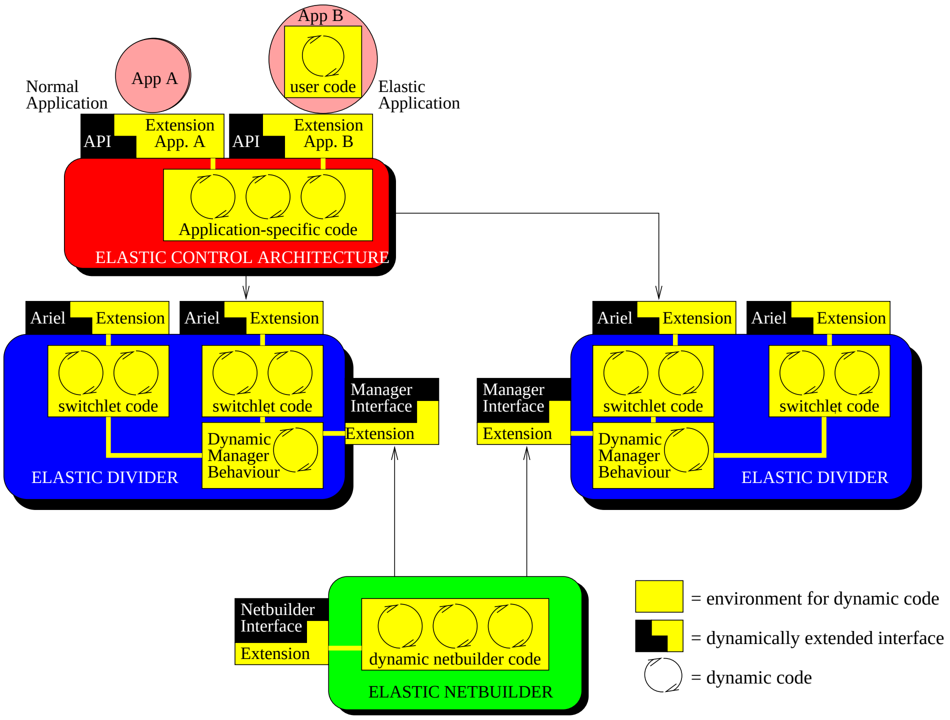


Figure 3.1: Elastic network control

it, over time, in pursuit of its own agenda and so as to effect what it senses in the future". DLAs in the *Haboob* are composed of (potentially many) relatively independent sub-components known as *granules*. A granule is the smallest unit of autonomous program behaviour that can be identified. A simple administration keeps track of which granules correspond to which DLAs, which is used, for example, to ensure that all granules corresponding to a DLA are removed from the system when the DLA is destroyed. Apart from this administration, granules are logical units only: there exist operations to register and deregister granules with a DLA, but there is no explicit operation to create or destroy granules. Granules help programmers think about DLAs. Examples of granules may include such things as individual services, background tasks and all other active components in a DLA that are more or less autonomous (in an extreme case, a granule could even be an individual statement). Code that is not dynamically loadable will be called *static*. Figure 3.2 illustrates this decomposition of code. In the figure, a DLA consisting of a 'main' part and several background tasks and services is running in Sandbox. Here, "background tasks" indicate functions that are executed periodically, while "services" identify server operations that can be called by clients. Both the services and the background tasks are set up as autonomous entities and are registered as granules.

The rest of this section will consider in some detail a concrete design for DLA support. There are many ways in which support for elasticity can be provided. This dissertation argues that a good design should take at least the following issues into account:

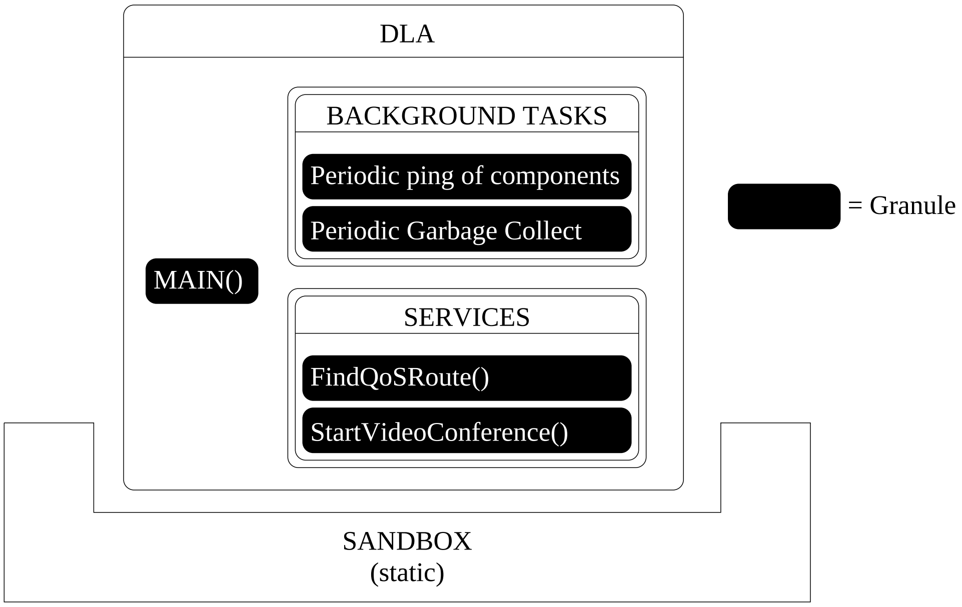


Figure 3.2: The decomposition of an example DLA in granules

1. **Support for existing code**
   It should be easy to provide elastic functionality to applications, even if these applications were not written with elastic behaviour in mind.

2. **Uniform interfaces**
   Regardless of application area or location, the core functionality that provides the elastic environment should always look the same.

3. **Extensibility**
   It should be simple to support elasticity for specific applications which goes beyond the basic elastic functionality.

4. **Interoperability**
   In order to allow DLAs to cooperate, a uniform framework for interaction is needed. Interacting DLAs could both reside on the same host (and indeed the same address space), but may also be distributed over a network. Support for location transparency is therefore desirable.

5. **Implementation independence**
   Interfaces should be used at component boundaries, while the implementation should not be prescribed.

6. **Language independence**
   Similarly, the programming language in which to express the DLA should not be prescribed *a priori*.

7. **Security**
   Issues such as trust, access control (who is allowed to do what) and resource management should be given careful consideration.

3.3.1 Making code elastic

The first goal is achieved by making the execution environment for DLAs (the Sandbox) a single object, that only needs to be instantiated to make any existing program elastic. Henceforth it will be assumed that all elastic components have an amount of static code with static interfaces, both of which can be extended or overridden by DLAs. This is illustrated in Figure 3.3. The original static code is at the bottom of the layered computational model. This static code, or *base layer*, exports a static interface. Built partly on top of this interface and partly directly over the base layer is the execution environment or *elastic runtime*.

The core of the runtime consists of a basic execution environment for DLAs in the form of a *virtual machine* (i.e. any entity that accepts dynamic code and is able to execute this code in a safe manner). The DLAs may consist of source code, partially compiled bytecode, or even fully compiled platform-specific code (possibly with multiple versions of the code allowing it to run on various platforms). The elastic model does not prescribe any one of these alternatives, although the former two are probably the easiest to check for safety/security.

The Sandbox implements both the hooks and mechanisms to load and run DLAs, and the extensions to the original code's functionality. The runtime is essentially static as well: it is compiled and linked with the base layer. The extensions to the basic functionality provided by the runtime are organised in *modules*, each of which may export one or more interfaces.

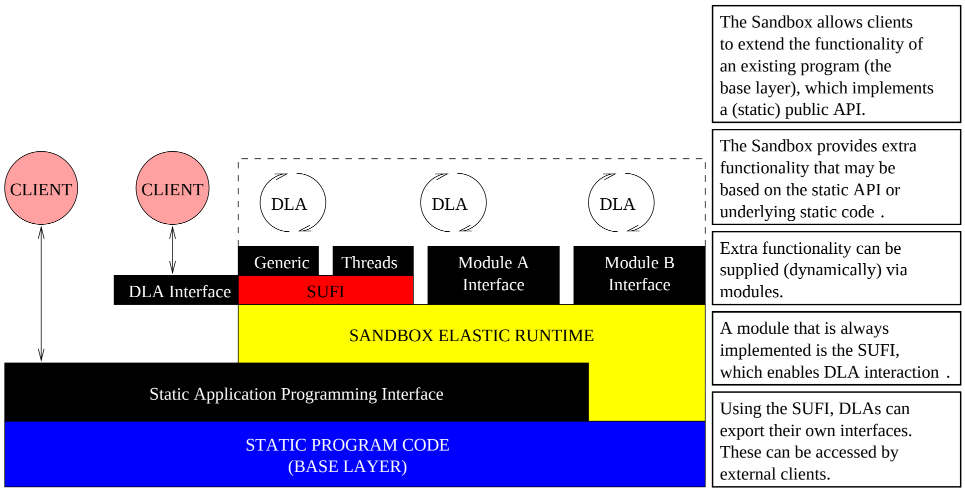


Figure 3.3: The Sandbox elastic execution environment

These interfaces allow DLAs to access the internals of static code and state in a controlled fashion. Beyond these methods, there is no way DLAs can tamper with either code or state. One module that is always implemented is called the *simple uniform framework for interaction* (SUFI). As illustrated in

Figure 3.3, the SUFI consists of two sub-modules, called *Generic* and *Threads*. The SUFI provides very simple types of interaction both between DLAs and between threads within a single DLA. It is also responsible for such things as exporting interfaces on behalf of DLAs and access control on individual operations. There is a one-to-one mapping between a Sandbox and a SUFI interface. The implementation of the SUFI operations, however, may be shared by any number of runtimes. The SUFI will be discussed in Section 3.3.2.

3.3.1.1 Elastic runtimes

In the previous section, it was shown how DLAs run on top of an elastic environment known as the Sandbox. They are allowed to execute any operation that was made available by the SUFI or some other Sandbox module, as long as they have the appropriate access rights. Access rights are provided in the form of capabilities. DLAs are also allowed to export their own interfaces. Again, this is done via the SUFI. External clients wishing to call a specific DLA's operations obtain an interface reference (for example, CORBA style interface references as described in Section 2.3, or something similar) which allows DLA services to be automatically located and bound to. The SUFI will be described in detail in Section 3.3.2.

The Sandbox is a single object. Upon creation, the runtime is initialised, the corresponding modules are added and either an initial DLA is run, or a handle is exported which allows external clients to load DLAs on top of it, or both. Destruction of Sandboxes is a rather more delicate matter, as it may be the case that one or more threads of a DLA are still active when the runtime is destroyed under its feet. Rather than requiring the DLAs to give up control completely before their runtimes can be destroyed, the elastic model allows a more relaxed mode of destruction. Active runtimes can be destroyed provided that this does not compromise the safety of the underlying process.

The integrity of other runtimes and DLAs communicating with the destroyed runtime is a higher-level responsibility, beyond the scope of the elastic model. As explained below, however, it is possible for DLAs to register a *cleanup* procedure, which will be called just prior to destruction of the runtime. The cleanup procedure allows DLAs to take the necessary action to restore or rollback to a consistent state.

3.3.2 Simple Uniform Framework for Interaction (SUFI)

The SUFI module is found in every runtime in all elastic components. The SUFI can be thought of as the 'standard' interface between the Sandbox and the underlying implementation. In [**Calvert98**] a similar sort of interface is called the *execution environment* $\leftrightarrow$ *node operating system* interface. The SUFI only defines the interface; the language and implementation are left unspecified. The way the SUFI and other interfaces are presented to users (or DLAs) depends on the environment, e.g. the programming language that is used. This is defined by an environment-specific interface which is called the *user* $\leftrightarrow$ *execution environment* interface.

The SUFI consists of a *inter-DLA* interaction sub-module called *Generic* and an *intra-DLA* interaction sub-module called *Threads*. Of these only the *Generic* interface is mandatory, i.e *must* be made available to DLAs. The *Threads* interface need not be exported, because there may be good reasons not to allow multi-threading in a simple DLA environment.

The SUFI provides in the *Generic* sub-module a simple and general means for interaction that is both access controlled and location transparent. A threading and mutex-locking mechanism is provided in the *Threads* sub-module. The SUFI can be extended in the future simply by defining more SUFI sub-modules. Other responsibilities of the SUFI include the registration of new modules (which includes making the modules' operations available to DLAs) and the offering of an interface which allows external clients to call individual DLAs (or their operations). As such, SUFI can be considered the root of the elastic functionality.

3.3.2.1 Remote evaluation

Interaction in the SUFI goes beyond traditional remote procedure calls (RPC) communication [**Birrell84**], although it retains the RPC flavour. Indeed, in its simplest form, there is nothing to distinguish a SUFI interaction from a normal remote procedure call: well-known operations on remote interfaces are called with a number of parameters, while the results of these operations are marshalled back to the caller. In addition to this, the SUFI allows what is called *remote evaluation*, whereby the caller (with the appropriate access rights) also supplies the code (a granule) to run on the remote host. The results of the operation that is executed remotely, if any, are still marshalled back to the caller. Related work on remote evaluation will be discussed in Section 4.8.1.

The granule that is supplied by the caller can be *active*, e.g. an iteration over an information base maintained at the remote side, or *passive*, whereby no code is really executed, but a new granule of functionality is installed at the remote side. An example of the latter is the passive installation of a new procedure that contains an iteration over the same information base. This code is only activated when it is actually called. A combination of both active and passive code in the same interaction is also possible. Granules may or may not contain state.

One advantage of remote evaluation is that it allows new functionality to be installed at a remote component dynamically. Another advantage is that for certain applications it may lead to improved performance, as discussed in Section 4.6.

3.3.2.2 Issues in remote evaluation

The focus of the design of the Sandbox and the SUFI was primarily on simplicity. It should be intuitive to use the SUFI's remote evaluation facility and likewise, it should be easy to build Sandboxes for different languages and different environments. For this reason, the Sandbox currently does not prescribe such things as rigid call semantics or static parameter checking for remote evaluations.

In a heterogeneous multi-language environment, these are both difficult issues. The former is complicated by the fact that the precise semantics of a remote evaluation (like the semantics of RPC) depends on the semantics of the communication/invocation primitives of the underlying runtime. For example, a runtime which communicates according to at-most-once invocation semantics will lead to slightly different semantics for remote evaluation than a runtime with zero-or-more invocation semantics. It is expected that different environments will use different semantics. In a particular environment, however, the semantics should be fixed. This is no different for RPC implementations.

Strong parameter or type checking at compile time is a useful feature that is included in many languages and DPEs. Because of its complexity in a multi-language remote-evaluation context where code is dynamically loaded onto and unloaded from runtimes, it is currently not included in the Sandbox specifications. Compile-time checking is difficult, for example, because certain languages are interpreted dynamically and also because type information may not be known at compile time (but only discovered at load or run time). However, in [**Stamos90**] techniques for static checking are outlined which could be adapted for more rigorous type checking in SUFI-based interactions. The SUFI remote evaluation semantics currently are extremely simple:

1. A DLA requests the Sandbox to execute some granule at a remote side (this may also be a simple RPC, in which case the 'granule' consists of the procedure name and its parameters).
2. The local runtime transmits the granule to the corresponding remote Sandbox.
3. The remote Sandbox attempts to evaluate the granule.
4. The remote Sandbox returns the results or an exception to the calling DLA.

These semantics preclude neither strong typing with type checking prior to execution nor weak typing without *a priori* type checking. It is up to the implementers to decide which model is preferable. Observe that the semantics can be specialised without changing the SUFI interface. It should therefore be concluded that neither of the two issues discussed in this section are limitations on the Sandbox SUFI. If rigid semantics and static type checking are required, it is perfectly acceptable in an environment to choose one particular implementation for remote evaluation which does supply these features. The SUFI only prescribes the interface.

3.3.2.3 Byte sequences and adaptors

A simple alternative to static type checking which is easy to implement in Sandbox systems does not impose any structure on the data at all. Instead, the parameters in a remote evaluation call are simply passed around as a single byte sequence (or string). With (or prior to) the call an application-specific *adaptor* granule is sent which parses the parameter string at the remote site, extracts the parameters and structures the parameters in the appropriate way and returns the structured data. This is illustrated in Figure 3.4. The name "adaptor" is taken from a similar idea in the TUBE system [**Halls97**].

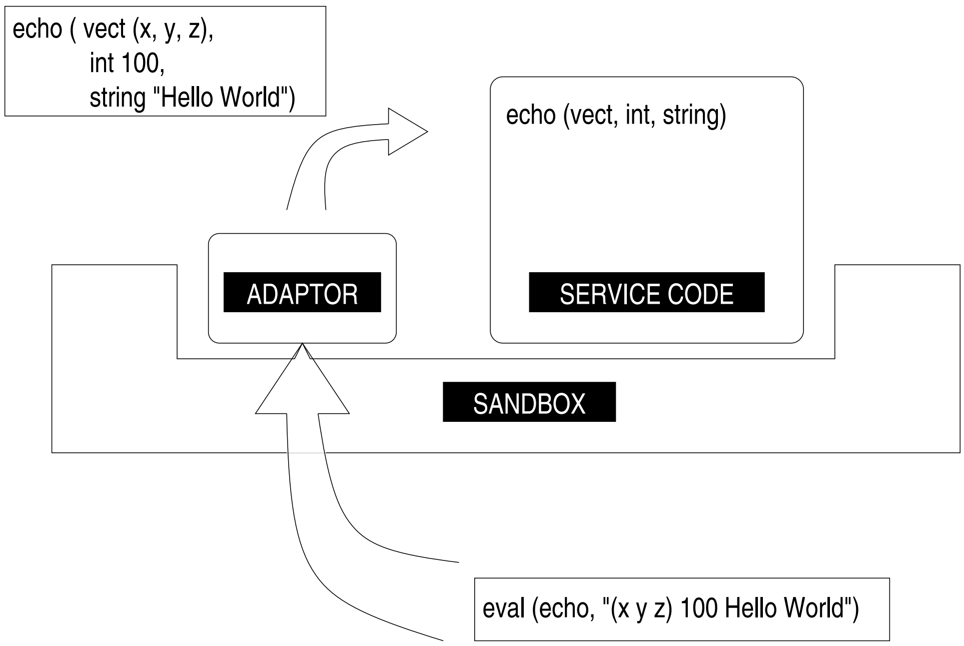


Figure 3.4: Adaptors structure the data on arrival

In the figure, a remote evaluation call is made with an unstructured string of arguments. The argument list is checked by the adaptor and the appropriate structure is restored. Finally, the corresponding call is made. This is essentially an example of dynamically loading marshalling and unmarshalling code.

3.3.2.4 The *Generic* interface

Of all module interfaces, the *Generic* interface offers the most basic functionality to DLAs. Without it DLAs would not be able to communicate with each other or with external (static) code. Its availability to DLAs is mandatory. It is simple enough to allow implementation in many different environments, supporting many different programming languages. The *Generic* interface offers the following functions to DLAs:

1. *Remote evaluation*
   It allows one DLA to call an operation in another DLA or even to perform a remote evaluation of its own code.

2. *Access control*
   It enables DLAs to restrict access to operations (or to the ability to perform remote evaluation of foreign code) via capabilities.

3. *Reflection*
   It allows DLAs to find out which operations are supported by the current elastic runtime and what arguments these operations take.

4. *Exporting interfaces.*
   It provides DLAs with the possibility to export their interfaces, so that external clients can find out about them.

5. *Describe method*
   The describe method (which can be called from remote DLAs) allows DLAs to find out information about the runtime environment or other (user-defined) items.

6. *Finally method*
   The *finally* method, which can be defined by DLAs, is called just prior to the destruction of the elastic runtime. It enables DLAs to perform a cleanup operation (for example, to tear down all connections that were set up by a DLA).

7. *Trust-establishment method*
   Interestingly, this is not so much related to the risk of running untrusted DLAs in a Sandbox, as to running DLAs in untrusted Sandboxes. More will be said about this in Section 3.3.3.

3.3.2.4.1 Protection Protection and access control in the elastic model rely heavily on the use of capabilities [**Wilkes79**]. The SUFI contains operations that allow DLAs to restrict access to a specific operation to a group of capabilities. Any entity presenting any one of these capabilities is granted access. Alternatively, it is possible to make operations 'publicly accessible', which means that the operation will match any capability. To manage access control, primitives are provided to incrementally add or remove capabilities from an operation's capability group. Finally, it is possible to restrict access to all operations to a group of capabilities in one go. The capability based access control also guards remote evaluations: only clients with the appropriate capability are allowed to run foreign code on the local runtime.

The problem of the creation, safe distribution and destruction of capabilities is beyond the scope of the SUFI. By default, the Sandbox treats them as sequences of octets. Upon creation of a Sandbox, however, programmers can specify their own capability evaluation code. Whether or not encryption and signatures are used, is also a problem that by default is not dealt with at the SUFI level. In the simplest case, the capabilities are treated as per-application passwords which allow applications access to an operation. Revoking capabilities is made easier by the possibility of revoking a capability for all operations provided by a runtime.

[**Delgrossi99**] splits up the interaction of applications in (active) networks in two types: *datapath interference* and *communication*. For both types three levels are distinguished: (1) none (no interference or communication is desired), (2) intra-application (all interference or communication takes place within a single application), and (3) inter-application (other network entities are also allowed to interfere or communicate with the application). It is suggested that each of these levels be supported. The SUFI with its access control explicitly provides this support in a single mechanism[2].

3.3.2.4.2 Roaming agents The support for remote evaluation creates another desirable possibility: DLAs that submit *themselves* (i.e. all of their granules and possibly their state) for remote evaluation. This, combined with the possibility of removing existing DLA state at the original node, allows for a whole new type of DLA: the roaming agent. In addition, it is possible, by submitting only a selection of a DLA's granules for remote evaluation, to decompose one DLA into multiple smaller (possibly roaming) agents. This in turn allows for dynamic partitioning and (re-)combining of granules in almost arbitrary ways. In this sense, DLAs can be thought of as temporary and possibly changing collections of granules.

There are multiple roaming-DLA models that can be implemented in this way. For example, it is possible to generate roaming DLAs according to the model proposed in [**Appleby94**]. In this model, DLAs may carry state around, but they will never carry any critical portion of the control and management information, where the loss of such information might jeopardise the integrity of the system. Instead, the DLAs access and modify the state that is left behind in the runtime by other agents. These DLAs should never communicate directly with other roaming DLAs—instead, all communication takes place via

[2] Datapath interference requires access to the data, i.e. the implementation of a datapath component. An example of such a component will be given in Section 7.6.

shared state in the elastic runtime in the nodes. This means that the roaming DLAs can be grouped together in teams with specific tasks (e.g. coordinators, information gatherers, load managers, etc.), where each DLA can take over the role of any other DLA of the same family. If the DLAs are present in reasonably large numbers (and new DLAs of a specific type can be generated by coordinating DLAs on the fly), this provides a high level of robustness: any individual DLA can be lost in the network without serious consequences.

Alternatively, one could opt for more heavy-weight roaming DLAs, where the DLAs function rather like traditional processes, containing most of their state themselves and probably roaming at a slower time-scale. In this model, the loss of a DLA means the loss of its functionality in the network. Such a loss may be acceptable, however, if its occurrence is rare and affects few applications. Recovery procedures may be implemented to reincarnate lost DLAs. Both models are supported by the elastic environment discussed in this dissertation.

The Sandbox SUFI does not require support for automatic freezing of execution state, which enables one to halt code at any time and continue its execution somewhere else. Such functionality (as provided for example by the TUBE [**Halls97**]) can be used if available, but if not, explicit state saving and evaluation can be used as well.

3.3.2.4.3 Reflection Calling the reflection operation without arguments will list all operations (and their arguments) that are accessible to at least some capabilities. This allows a roaming DLA to find out about supported operations in its current runtime. Knowing which operations are supported does not guarantee that the DLA will be able to invoke them, as it may not have the appropriate capability. Alternatively, the reflection operation can take a specific capability as argument, in which case only the operations accessible to that capability are listed.

3.3.2.4.4 Describe method In addition to the supported procedures, DLAs are able to find out information about the execution environment via the *describe* method. Without arguments, this method returns information about the Sandbox and SUFI. This includes such things as major and minor version number, vendor, language(s) supported, etc. DLAs can add descriptions for other items by supplying an item identifier together with a description (i.e. a null-terminated string), e.g. they may supply name-to-semantics mappings for functions they support.

3.3.2.4.5 Exporting interfaces The SUFI offers an operation to allow DLAs to export (references to) themselves to a trading mechanism. The SUFI does not specify the nature of this trading mechanism. All it requires is a name for the service being exported. This name, in turn, is used whenever a DLA wants to call an operation in a remote DLA. In that case the service name is resolved, which means that something akin to a CORBA interface reference corresponding to the name is found, after which the appropriate binding can take place.

3.3.2.5 The *Threads* interface

The *Threads* submodule offers a simple threading mechanism, which allows DLAs to spawn threads, block on mutices, release mutices, etc. The scheduling policy that is offered by the *Threads* module offers non-preemptive threads. Other scheduling policies and threads models can be implemented by defining new sub-modules. The goal of the *Threads* module is only to offer the most minimal threading functionality possible.

SUFI threading is not always necessary or even desirable. It is pointless to implement simple threads if the language's runtime environment offers more advanced threading already. It is undesirable if one wants the DLAs to remain simple, single-threaded programs. In such cases, there is no need to implement the *Threads* interface, or to make it available to DLAs.

3.3.3 Security

Running foreign code in the heart of components like the control architecture introduces risks that range from the risk that the code affects other applications or steals sensitive information, to the risk that the code starts running computationally intensive programs that take up a lot of processor time or other resources. Both problems require the ability to restrict the amount of resources a DLA is allowed to use. Also, sometimes a distinction is made between *safety* and *security*, where the former is commonly assumed to provide protection against errors of trusted users and the latter against those of untrusted users. No such distinction is made here. The terms *safe* and *secure* are interchangeable and concern both kinds of errors. It should be stressed that the security mechanisms discussed in this dissertation only cover a small portion of all possible threats. A good overview of security issues in the context of mobile agents is given in [**Greenberg98**].

3.3.3.1 Encryption and trust

When sensitive code or data is exchanged in an unprotected manner, e.g. when DLAs migrate in source text encoding, it is recommended that encryption is used for security. To shield DLAs from each other, different Sandboxes should be used. Provided these Sandboxes' access to resources is bound, a DLA in one Sandbox cannot corrupt data in either the underlying system or in another Sandbox. More interesting problems arise when the Sandbox itself cannot be trusted. This may be a common phenomenon if many nodes in the network provide elastic runtimes and DLAs migrate over these nodes. A full implementation of the Sandbox includes a method that DLAs or remote processes can call to make the Sandbox prove its trustworthiness.

3.3.3.2 Restricted access to resources

Security is founded on the enforcement of controlled access to resources. A security *policy* determines which operations can be performed on the resources. This is similar to the restricted access that programs in user space commonly have to the data structures in the kernel. A useful classification of mechanisms enforcing controlled access to resources is given in [**Adl-Tabatabai96**]. Most systems are either based on what is called *abstract machine interpretation* or on *language semantics*. In the former case, an interpreter manages the mapping between virtual resources as used by the abstract machine (and the DLAs) and the physical resources of the host. This allows the interpreter to prevent unauthorised access. The latter mechanism uses language semantics to make sure that a program can't use resources that it cannot name. Of course, this ties the mechanism to a specific programming language.

In the *OmniWare* system [**Adl-Tabatabai96**] a third approach is proposed, whereby source code is compiled to very low-level (RISC-like) operations that map easily on the underlying processors. OmniWare's virtual machine, the OmniVM, loads this code and uses software fault isolation (SFI) to ensure safe execution. SFI checks every unsafe memory access or indirect branch access to make sure that it is safe. The OmniVM inlines these checks at DLA load time. An additional advantage of using a low-level intermediate language is that the compiler can do most of the optimisations. As a demonstration, both the GNU C Compiler *gcc* and a linker were retargeted to produce OmniVM instructions (with optimisation turned on). The performance of the safe mobile code was virtually the same as the performance of native code generated by *gcc*.

The operating system should not only limit a Sandbox's access to resources, but also provide it with guarantees as to the availability of resources. For example, a Sandbox should be allocated a share of the CPU, memory, communication bandwidth, etc. This way, Sandboxes are not able to interfere with each other and are at the same time given QoS guarantees allowing them to make guaranteed progress. Fortunately, new developments in operating systems permit such guarantees to be given [**Leslie96**].

3.3.3.3 Time to live

When DLAs start to roam (or replicate) there is a risk that they will roam (and replicate) forever, e.g. traversing a loop in the network. This may be the case, even if the Sandbox and the application where they were created originally have long since died. This will be called the eternally roaming agent (ERA) problem. A possible solution is to annotate DLAs with a time-to-live (TTL) attribute similar to the TTL field in the IP protocol. Apart from this limit on the duration of a DLA, it has been proposed to restrict the number of duplicates an agent is allowed to make of itself or its constituent parts [**Greenberg98**].

3.3.3.4 Node safety versus network safety

Security in distributed systems is a hard problem and no attempt is made here to fully solve it. Even when individual nodes are made safe, it doesn't automatically follow that the whole distributed system is safe. For this purpose, the work presented in [**Alexander98b**] makes a distinction between *node-safe* software and *network-safe* software. An example similar to the ERA problem is given: a program that simply multicasts all incoming packets on a port to two output ports. This program satisfies node safety. If this program is installed on many nodes in the network, it may cause a traffic explosion that brings down the network. In other words, the node-safe program is not network safe. This dissertation will not pursue these issues any further.

3.3.4 Language independence

The elastic model prescribes neither the implementation of the runtime, nor the programming language to use. For example, a good candidate for implementing the Sandbox SUFI would be the TUBE environment [**Halls97**]. Other implementations will be discussed in following chapters. The only conditions that the code in a DLA should satisfy are that:

1. the code is dynamically loadable;
2. the execution of the code takes place in a *safe* manner;
3. the code supports the remote evaluation paradigm for both passive and active code as described in Section 3.3.2.1.

There are of course limits to the extent to which this last condition can be met in a multi-language environment. For example, language independence requires that a DLA that is implemented in language L_1 is able to call a procedure in a remote DLA implemented in L_2. However, it is absurd to require that a runtime that supports L_1 to be able to evaluate L_2 code. So, the requirement of language independence is relaxed to include only RPC-style interactions. The more complicated remote evaluation need only be supported between compatible runtimes. Note however, that an L_1 DLA can still easily provide an L_2 Sandbox with a fragment of L_2 code to perform the remote evaluation, which in that case should be supported[3]. Moreover, it is possible to facilitate the communication between DLAs of different language environments, by employing meta-communication languages such as KIF and/or agent communication languages such as KQML [**Finin98**].

Although in theory the above requirements do not preclude any language, it is realised that these conditions are more easily met by certain programming languages than by others. Recent years have seen a flurry of programming languages and execution environments that allow code to be dynamically loaded. A major impulse for this development has been the need to support active code on a client's local machine in the world wide web. In this dissertation, the choice of the programming language is considered an implementation detail.

3.4 Elastic control architectures

Elastic control architectures allow clients to override or extend the default functionality of the control architectures. For simplicity, this dissertation adopts a very coarse model that is general enough to match most control architectures. Conceptually, control architectures are subdivided into components as illustrated in Figure 3.5. The next section briefly describes each of the components and their functions.

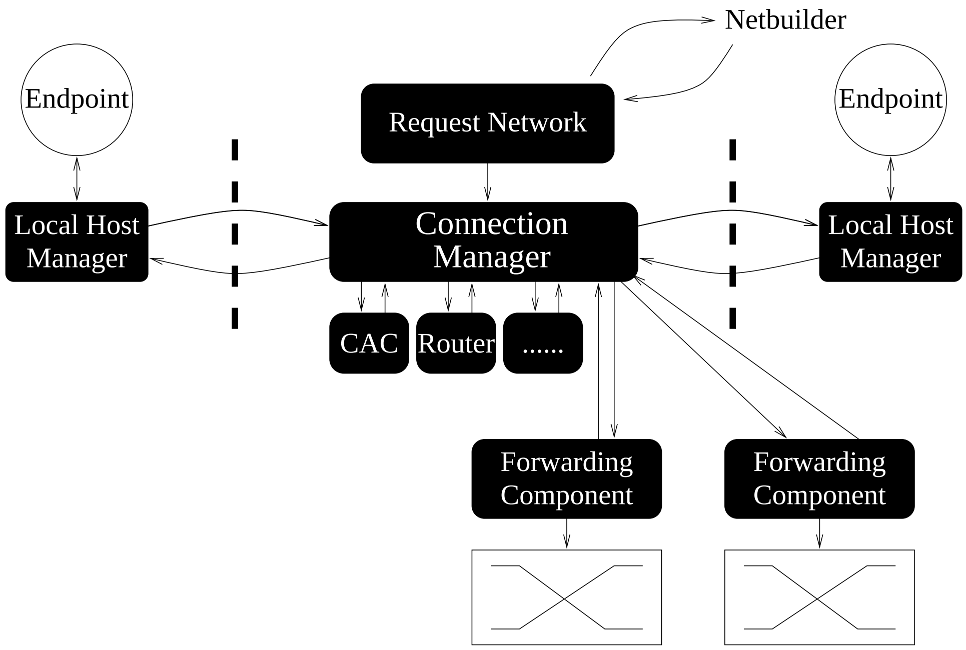


Figure 3.5: Components of a control architecture

[3] In the most extreme case, DLAs could use the *describe* methods to find out the sort of encoding that is expected in the remote Sandbox and explicitly compile the source for this target before submitting it for evaluation.

3.4.1 A model for control architectures

At start of day, control architectures will get a number of network resources to control. This is shown in Figure 3.5 as the *request network* component. In a traditional system, this component is almost empty: the control architecture controls the entire physical network (e.g. a single switch). In the *Tempest*, however, this involves building a virtual network for the control architecture. Either way, the result of this step is a number of handles on resources (or partitions thereof) which the control architecture uses to exercise control.

The central component is the *connection manager*. It controls one or more switches. Larger networks are controlled by multiple cooperating connection managers. The connection manager relies on the services of a small set of components, such as the *call admission control* (CAC) component, the *routeing* component and possibly others. The roles these entities play and the way in which they interact will be explained presently. Part of the control architecture interacts with clients and either resides on the client's host or communicates with it extensively. This component will therefore be called the *local host manager*. It manages resources local to the client's host and provides an interface to the rest of the control architecture. The separation between connection manager and local host manager is indicated by the thick dashed lines.

The way control typically takes place is as follows. A client (either one of the endpoints or a third party) submits a request to the local host manager. Taking the example of a connection setup request, the local host manager checks whether the appropriate resources to satisfy the request are available locally and if so, it sends the setup request to the connection manager. For example, many control architectures have a notion of service access points or SAPs, e.g. UNI4.0, which uses Network Service Access Point (NSAP) addressing for identifying endpoints [**UNI4.0:94**]. In order for a node to be able to be an endpoint of communication there has to be a free SAP. The local host manager checks that this is the case and only then submits the request to the connection manager.

The connection manager consults the routeing component as to which resources (e.g. which switch ports) should be involved in this connection. It presents this information to the CAC component, which determines whether each of these resources has enough available capacity to satisfy the request[4]. If not, the connection setup is aborted and a failure message is returned. Otherwise, if the connection request is also accepted by the endpoint(s), the appropriate resources are reserved without actually allocating them to any physical connection. If the other endpoint lies within the connection manager's domain, the reservation can be made definite (otherwise it is forwarded to the next hop connection manager). In case of an *immediate connection* setup, the resources are also allocated to the connection, while in the case of an *advance reservation* for a connection, only an acknowledgement is sent back to the requesting client:

[4]Routeing and CAC are often closely coupled or even integrated, whereby the availability of resource capacity determines the route and *vice versa*.

the allocation of resources to the connection only happens at the specified start time.

Either way, resources are eventually allocated to the connection by the *forwarding* component at the request of the connection manager. This component also makes sure that the appropriate label mapping and resource allocation is performed by the switch[5]. After all forwarding components have allocated the resources, the endpoints are notified of the existence of the connection. This notification also carries the identifiers of the connection (in ATM, the VPI/VCI pair). Connection teardown is analogous. The endpoints are notified and the resources are freed: physically by the forwarding components and logically by the connection manager and CAC component.

In principle, components other than CAC and routeing may be consulted by the connection manager. In Figure 3.5 this is illustrated by the empty box. The architecture of Figure 3.5 can be applied to connection oriented network technologies other than ATM as well. Frame Relay [**Cisco96**] or IPv6 flows [**IETF98**], for example, can easily be described in terms of the model's components.

3.4.2 Instantiating elastic runtimes

In an extreme case, each individual control architecture subcomponent could be made elastic by instantiating a Sandbox in its scope. Such a view is illustrated in Figure 3.6. This preserves at the DLA level the original modular structure depicted in Figure 3.5. The disadvantage, however, is that a lot of complexity is added to the system, while the benefits seem rather limited.

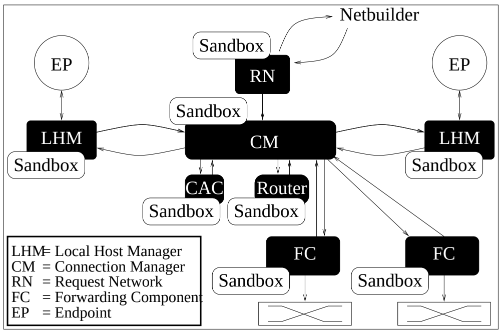


Figure 3.6: Multiple Sandboxes

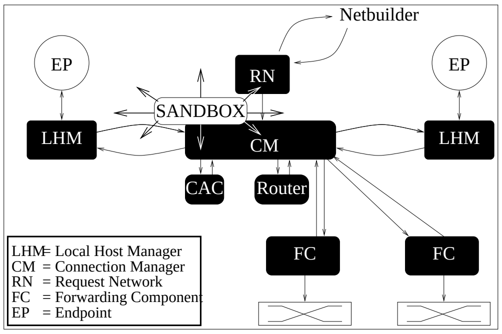


Figure 3.7: Central Sandbox

The other extreme is to add a single Sandbox which acts as the elastic runtime for the entire control architecture. The logical place for such a central Sandbox would be in the connection manager's scope as this is the entity that controls all other components anyway. Such a model is shown in Figure 3.7.

[5]Observe, however, that the *logical* resource allocation in the control architecture and the *physical* resource allocation in the switches need not be the same.

A central approach would be as powerful as the distributed one, as long as the appropriate interfaces to control the other components are provided to the granules in the central Sandbox.

Besides determining *where* to instantiate the runtimes, it must also be decided *how many* runtimes will be instantiated in these locations. The two extremes here are (1) one Sandbox per application (or indeed connection), and (2) a single Sandbox to be shared by all DLAs. For both location and number of runtimes to instantiate holds that the optimal choice varies from application area to application area. The decision is a tradeoff between functionality, complexity, safety and efficient use of resources.

The scope for elastic code functionality in control architectures is very wide. DLAs could potentially take on the roles of all of the components depicted in Figure 3.5. Implementations of such control architectures will be introduced in Chapter 5, when the Noman control architecture family is discussed. A Noman control architecture consists solely of DLA code running on top of a Sandbox containing a network-control module. Alternatively, the DLAs could be used to extend or customise an existing control architecture. This is the approach taken in the Sandman control architecture, discussed in the same chapter.

3.5 Elastic switch dividers

It is the switch divider's task to partition resources on a switch. This means that each partition (switchlet) obtains its own VPI/VCI space, buffer space, scheduling guarantees, etc. The divider further ensures that operations on switch resources by a control architecture are valid, i.e. pertain only to resources owned by the control architecture. It interacts with the outside world via two interfaces:

1. *Management*: this interface provides operations for the creation or destruction of switchlets.
2. *Ariel*: this interface allows a control architecture to control the resources in its switchlet.

3.5.1 Creating and controlling switchlets

The interfaces are independent. A managing entity, generally the netbuilder, requests the divider to create a new switchlet with certain properties on behalf of a control architecture. The divider allocates the requested resources to the new switchlet. This includes instantiating an implementation of the *Ariel* interface for these resources. The resources are only controlled via calls over this interface. After the instantiation, the divider returns a handle on these resources by way of an interface reference corresponding to the *Ariel* instantiation.

Instantiating *Ariel* for a switchlet, implements the methods defined in the interface. What this implementation means concretely at the divider level is that whenever a call is made over the interface, all parameters are checked to ensure that they only refer to resources owned by this switchlet. For example, a setup request is at least checked to see whether the specified port/VPI/VCI values belong to the switchlet's address space. If so, the request is forwarded to the switch (where the connection is made across the physical switch). If not, an exception is raised.

3.5.2 Divider elasticity

It can be concluded from the discussion in the previous paragraph that there are two places where elastic runtimes would be useful. The first is in the module where new switchlets are created. The second is in the *Ariel* instantiation itself. This is illustrated in Figure 3.8 which shows a divider with three switchlets and a management component. The possible locations for the two types of elastic runtime are marked by X and Y respectively. Where confusion is possible, DLAs running on locations marked by X will be called *Ariel* DLAs, while DLAs running on the Y location will be called *Management* DLAs.

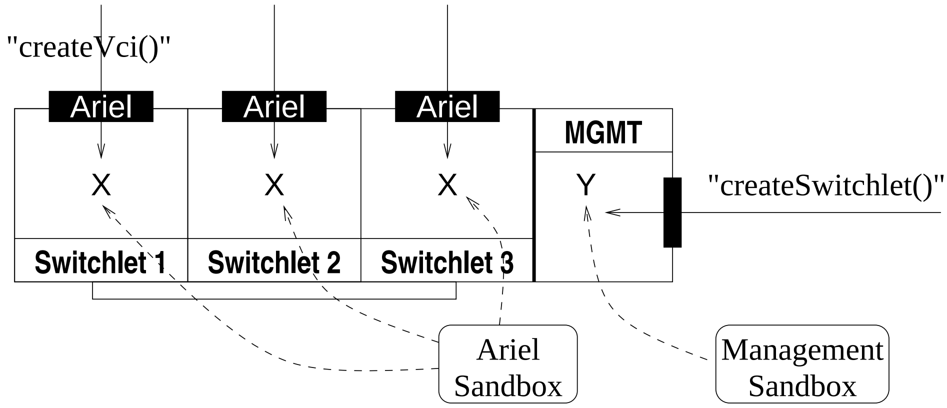


Figure 3.8: Places to instantiate elastic runtimes

3.5.2.1 DLAs in an *Ariel* Sandbox

Placing a Sandbox in a switchlet itself allows one to load application-specific code as close as possible to the switch (this could be on the switch itself). This code is able to call any switch control operation available to a traditional control architecture, the only difference being that the call need not be transmitted over the network. If the divider runs on the switch, this could even be a simple function call in the same address space. The DLA may implement control architecture functionality for a single switch, in which case it is called a micro control achitecture. Micro control achitectures are either controlled from external applications, or in a more extreme case, contain their own application code. Micro control achitectures that are controlled from external applications can easily extend *Ariel*'s functionality by implementing for example such things as batch connection setup or teardown. Examples of micro control achitectures will be discussed in Chapter 7.

Furthermore, it is desirable to make the *Ariel* Sandbox support an *override* operation, which allows it to safely replace, or modify the standard *Ariel* operations. Such functionality enables one to change the behaviour of a switchlet. A trivial example would be to override the connection setup and teardown operations with code that not only sets up or tears down connections on the physical switch but also sends notifications of the events to a remote monitor process, e.g. that graphically displays the current connections on the switch. This and more interesting examples will be discussed in Chapter 7.

Because of the nature of the divider, Sandboxes should not be shared between switchlets. A switchlet need not instantiate a Sandbox if it does not require elastic functionality, but if it does, the Sandbox should be unique to the switchlet and *vice versa*.

3.5.2.2 DLAs for switchlet creation

The need for elastic runtimes in the process of creating switchlets itself may be less obvious. One simple use is that it allows network administrators to customise switchlet creation by automatically instantiating an appropriate *Ariel* DLA in an *Ariel* Sandbox at switchlet construction time. This *Ariel* DLA can then extend or override the *Ariel* operations. For example, a *Management* DLA installed by the system administrator may be triggered each time a create-switchlet operation is called (i.e. the DLA has overridden the method to create switchlets). It can then examine the request and depending on certain conditions (e.g. the identity of the requester), create a new switchlet with an *Ariel* DLA which extends the operations to create and delete connections with event notifications to a remote monitor, as discussed in the previous paragraph. No further actions are needed by the system administrator. Chapter 7 will show more interesting examples of switchlet management DLAs.

3.6 Elastic virtual-network builders

Netbuilders are responsible for combining switchlets into virtual networks. In other words, virtual networks are essentially a set of associated switchlets. Because they are shared servers with very little functionality that is specific to any particular control architecture, netbuilders are interesting when it comes to adding elasticity to them. The question arises: who decides how virtual networks should be created? In the *Tempest*, this is decided once and for all by the implementers of the netbuilder. This dissertation argues that this is unnecessarily restrictive.

For example, there is no reason why control architectures should not be able to push code into the netbuilder when this code is restricted to using only those operations that were exported by the netbuilder anyway. In other words, it should be possible to load active or passive DLAs into a Sandbox which, besides the SUFI, offers the exact same API as offered to control architectures. This optimises performance (since netbuilders may run very close to, or even on, a switch as well). The DLAs in turn are able to build new services by combining existing netbuilder services and exporting interface references to them.

So far, however, only the ability to allow DLAs to (re-)combine existing netbuilder functionality has been discussed. A more interesting question is: who is allowed to add to, remove from or modify the netbuilder's default functionality? It seems reasonable to give at least the system administrators such powers. After all, they may want to override certain operations to make them include extensive logging, or to disable certain functions, or even to preclude the use of certain switches for certain operations. The system administrators then, should also be allowed to grant these privileges to other network users (in the same way as a system administrator is allowed to grant the root password of machines to certain trusted users). The new functionality that is thus created by the system administrators can subsequently be used by non-privileged DLAs with the appropriate capability. In this way, the divider's functionality can be changed and extended on the fly.

3.7 Generic services, datapath and other components

In many cases, it makes sense to make individual applications and other components in the system elastic as well. In particular, extending the functionality of generic services or datapath components might be very useful. It makes these components much more generally applicable. For example, a component that was originally built for a particular purpose may be extended and generalised to include other purposes as well. An example of such a generic service is the traffic server that is first introduced in Section 5.2.2.3 and extended in Section 5.4.2.3. Its functionality that was originally limited to that of a simple effective bandwidth estimator is generalised to include policing functions as well. A second example of generic services is the *Haboob*'s trading mechanism, which can be used to find the *best* (e.g. the nearest or most lightly-loaded) servers for a particular request. In Section 7.6, a datapath component is introduced that is able to add forward error correction (FEC) to an existing data stream.

It will also be shown, that it becomes trivial to add little "plug-in" components to a system on the fly, where the location of these components is unimportant. This will allow us to extend the functionality of a distributed system, while the system is in operation.

3.8 Related work

Programming the network by loading control code on the fly has been proposed in a number of research projects, in particular in the Intelligent Networks and Active Networks communities. This section discusses these and other approaches. Related work in dynamically loadable code and remote evaluation is discussed in Section 4.8, while open signalling is discussed in Section 5.6.

3.8.1 Intelligent networks

Intelligent Networks (IN) [**ITU-T92**] allow the introduction of new services by associating them with signalling endpoints. Basic calls are separated from IN-based calls. For example, dialling an 0-800 number triggers a temporary suspension of call-processing and initiates a series of transactions between the local switching point (in IN terminology: the Service Switching Point or SSP) and the so-called Service Control Point (SCP), which is essentially a real-time database. A lookup in this database tries to find the corresponding application-specific *service logic*, i.e. the code which is then executed. The code sends back instructions to the SSP on how to process the call. Additional information about the IN model is given in Section 5.6.1.2.1, on signalling system no. 7. The bulk of current IN transactions consist of translating the number dialled by the caller into another number depending on the needs of the service. This functionality is part of IN's capability set 1 (CS-1). Services based on the new *IN capability set 2* (CS-2) are now becoming increasingly common. CS-2 extends the scope of IN to the influence of stable narrowband calls with two or more parties and is also considered as the foundation for future broadband multiparty services [**Roche98**]. Building services with the capability sets introduces the problem of *feature interaction* [**Cameron93**], where the effects of one service interfere with those of another. Advanced Intelligent Networks (AIN) is Bellcore's IN effort which aims to speed up IN introduction and address IN evolution in Bellcore client networks in the USA [**Garrahan93**].

IN is very limited in the amount of network programmability that it offers to clients. The capability sets offer a small number of end-to-end services such as call forward, call hold, malicious call identification, call tracking, narrowband conference calling, etc. It is not possible to gain low-level access to the physical switches, e.g. to set up a single switch connection, which would enable clients to build their own end-to-end services. Instead, IN does not allow optimal use to be made of its network. Even if, in principle, the resources could be used to provide certain functionality that is desired by the client, often these services cannot be provided. [**Rizzo97**] terms this the *service interface bottleneck* and shows how *feature interaction* exacerbates the problem. The service interface bottleneck is a serious flaw, because it means that it is determined *a priori* which services are needed by the clients. Chapter 5 will show that the *Haboob* does not have this problem.

3.8.2 Negotiating agents for telephony (NAT)

[**Rizzo97**] also observes that in IN, offering little expressive power to subscribers, the extent to which IN clients may customise the services is rather limited. *Negotiating agents for telephony* (NAT) is an alternative approach to provide telephony services, which allows users to express policies which describe how calls should be handled. The policies are used to guide agents which negotiate and operate on behalf of the client. An important observation is that the service interface bottleneck is primarily caused by the coarse granularity of the service interface offered to the clients. NAT proposes to offer a lower-level service interface to subscribers, which will give them more expressive power to specify call management requirements. Unfortunately, NAT seems to limit itself to end-to-end connectivity, and falls short of allowing clients to manipulate individual switches. Chapter 5 shows control architectures that go significantly further in providing low-level access and offer more flexibility.

3.8.3 Java Telephony API (JTAPI)

The Java Telephony API (JTAPI) is an API for Java-based telephony applications that consists of a number of packages providing support for specific telephony functionality [**JavaSoft97**]. At the center of JTAPI is the *core* package, which provides the basic framework to model calls, as well as simple telephony features. Layered around the core is a set of extension packages that bring additional functionality. JTAPI allows users to program to a certain extent their own telephony services, but is very telephony based and like NAT, it is based on endpoint connectivity and does not seem to provide very low-level access to network resources.

3.8.4 Active networks

Active Networks are packet-switched networks where each packet may carry executable code. In [**Tennenhouse96**] these packets are called *capsules*, i.e. little programs with embedded data that are evaluated in a transient execution environment, allowing network nodes to process the data in an application-specific way. In this extreme model, there is no distinction between control and datapath[6]. It is suggested in [**Tennenhouse96**] that:

> instead of standardizing the computation performed on every packet,
> we standardize the computational model.

[6] The Operation, Administration and Maintenance (OAM) cells in ATM offer a similar, albeit much more restricted functionality (the OAM cell effectively carries one of a finite number of pre-defined programs).

Under the active networks umbrella we find a number of projects that differ for example in the adherence to the pure capsules model (every packet is a program). The *ANTS* project at MIT stays very close to the model [**Wetherall98**]. The *SwitchWare* project, on the other hand, allows both traditional and active packets [**Alexander98a**, **Smith96**]. SwitchWare is designed as a three-level architecture, where the first level comprises the active packets, the second level concerns so-called active extensions (programs that can be dynamically loaded over the network and offer functions that can be used by active packets) and the third layer is the infrastructure that enforces the rules for dynamic code loading and takes care of resource allocation. In other words, SwitchWare allows users to install new functionality on a switch in the form of programs using out-of-band-signalling. These programs offer an API that is then used by subsequent packets. Formal methodologies are used to prove security features of the dynamically loadable programs.

At Columbia University the NetScript project is particularly well-suited for protocol building [**Yemini96**]. NetScript specifies both the programming language and the execution environment. The developers of NetScript also propose a novel solution to network management using *delegated* programs, where the programs consist of dynamically loadable code that can be dispatched using a so-called delegation protocol to an executing elastic (extensible) server [**Goldszmidt98**]. This helps prevent the explosion of management traffic from all over the network to a central site, which results from using management models that were designed when management was still a relative simple task and the traffic generated by it was minimal. Delegating management also makes the control loop (from managing code to managed device) smaller, decreasing the probability of failure at times when there are problems in the network (and management is needed the most) [**Goldszmidt95b**]. Delegated management is now proposed for standardisation as well [**Schönwälder99**].

Active networks have been used for such things as bridging between incompatible network segments [**Alexander97**], enhancing protocols in order to make them perform better when and where needed [**Marcus98**], reliable multicast [**Lehman98**] and many others. RCANE, the resource controlled active network environment, developed at the Cambridge Computer Laboratory over the Nemesis operating system, provides robust control and accounting of system resources, including CPU, I/O scheduling and memory [**Menage99b**]. The RCANE architecture resembles that of SwitchWare and supports a similar type of active network. The Node Operating System [**Peterson99**] is an attempt to define an underlying system to support multiple executing environments for active networks, each with its own set of resources (e.g. CPU, threads, bandwidth, memory, etc.). At the University of Lancaster, the Lancaster active router architecture (LARA) also allows multiple different execution environments for active code to be instantiated [**Cardoe99**]. Using the LARA/PAL (platform abstraction layer), it is possible to simultaneously support environments, e.g. for Switchware, ANTS, Lancaster's own runtime (LARA/RT), and others. Packet filters are used to ensure that data packets are demultiplexed to the appropriate environment. Other projects, such as ANON [**Tschudin99**], concentrate on providing an overlay network on top of underlying network technologies (e.g. IP), which can be used to support whichever active networks implementation users may desire. Network nodes that have to be made active start a daemon process `anond` to handle active packets. A similar approach is found in the ANETD project [**Ricciuli98**]

An important problem with pure active networks (capsules) is the coupling of control and data. This integration makes it more difficult to use fast label-based switching techniques (e.g. [**Newman97**, **Rekhter97**]). Each packet has to be inspected for program code, which subsequently must be evaluated[7]. Moreover, control packets can easily get stuck behind data packets. For example, assume that part of the network becomes congested and active network code is used to deal with the congestion [**Bhattarcharjee97**]. Because of the congestion and the fact that there is no distinction between control and data packets, it takes a long time before the control code reaches the congested nodes (if they are reached at all), by which time the congestion may already have spread. This is an example of control packets being slowed down by a network problem that they are meant to solve. Other serious shortcomings of current active network technology were mentioned in Section 1.1.2.

The *Haboob* keeps control and data separate, so that these problems will not occur. Indeed, the datapath is not touched at all. In this way, the *Haboob* also adheres strictly to a guiding principle for the placement of functionality in networks [**Bhattarcharjee97**]:

> If not all services will make use of a service, it should be implemented in such a way that only those applications using it have to pay the price of supporting it in the network

According to this dissertation, this gives the *Haboob* an edge over existing active networks projects, especially as it will be shown in Section 7.6 that it can provide the same functionality as active networks, if desired, while the inverse is not true. The fact that the current *Haboob* was implemented for ATM means that the separation of control and data comes essentially "for free". Implementation for IP will require more work, e.g. the establishment of different queues for data and signalling traffic. Fortunately, many proposals these days are directed at providing different traffic classes with different QoS within IP, e.g. [**Shenker97**]. Combining such differentiation with fast switching techniques based on flow labels, provides sufficient functionality to build an IP version of the *Haboob* that also adheres to the above principle.

Moreover, by now the problem of making networks programmable by dynamically loadable code (or 'active packets') is a well-established research area

[7] This concerns primarily the extreme *capsule* approach. Projects like [**Smith96**] are less extreme, allowing packets that are not active and which can be switched on the 'fast path'.

for which many solutions have been proposed. Although certain issues still need to be resolved, this dissertation goes a step further and addresses questions such as: how best to exploit such programmability, which aspects of control and management can be distinguished and how can they benefit from programmability by dynamically loadable code? For this purpose, an approach must be taken that also incorporates higher levels than the one where active code is classified, demultiplexed, run by the appropriate execution environment, etc.

3.8.5 Control on demand

Control-on-demand (COD) is a design for providing a control architecture that allows customised control in routers and switches [**Hjalmtysson97a**]. It allows applications to upload their own flow-specific controllers in the switch. The controllers can customise both control and data path. Flows that do not need in-band processing are not penalised by data-path operations of flows that do. COD assumes per-flow queueing and proposes to give controllers low-level access to the flow's buffers. In particular, it proposes to allow *frame peeking* which allows controllers to see part of an AAL frame currently in the buffers, without necessarily reassembling the frame completely, or removing it from the data path. As a key idea it proposes the notion of *enhancement control*, where the customisation implements a service or performance enhancement, but is not essential for correctness. Specifically, it is possible to introduce what is called *best-effort* control, which is not guaranteed to work on all frames of the flow. As an example consider a filter in a multi-layer encoding scheme, installed to block higher-quality frames towards a client on a low-bandwidth connection. In a best-effort version of the filter high-quality frames may occasionally trickle through. COD is an interesting approach, but it relies on assumptions, such as per-flow queueing and frame-peeking, that are currently not true for many switches. It is also not clear how useful best-effort control is in practice.

3.9 Summary

This chapter discussed in detail how dynamically loadable code can be safely introduced in advanced network control. It started with an overview of elastic network control and a discussion of both the computational model and the network control decomposition model. By only specifying well-defined interfaces, the computational model gains a large degree of implementation and language independence. Instances of code running in the Sandbox execution environment are called dynamically loadable agents (DLAs). The API offered to DLAs is organised in modules. The default module is called the simple uniform framework for interaction (SUFI), while other modules can be added on the fly. The SUFI is decomposed in a submodule for inter-Sandbox interaction and a submodule for intra-Sandbox interaction and threading. Interaction between DLAs is based on remote evaluation. Access control is provided by capabilities. For each of the components in the network control decomposition model, specific issues related to adding elasticity were addressed.

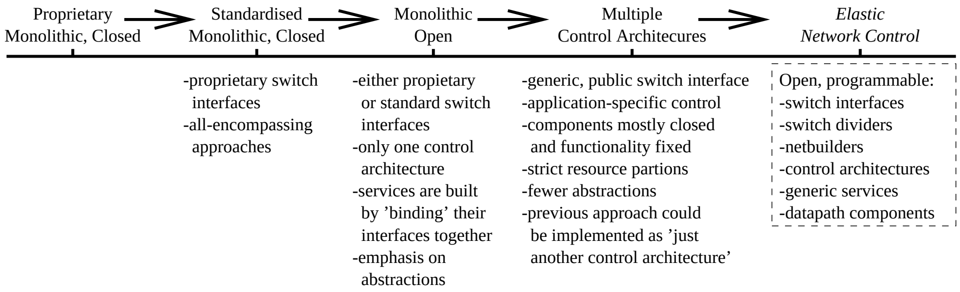


Figure 3.9: Evolution of network control

For reference, Figure 3.9 again illustrates the evolution of telecommunications network control, whereby each of the levels of the network model is incorporated. In the next few chapters, the implementation of various components in all levels of the network control model will be discussed in detail.

Chapter 4

Implementing the Sandbox

In the previous chapter, the design of the Sandbox (an execution environment for dynamically loadable code) and the SUFI (an interface allowing dynamically loadable programs to interact) was described. This chapter discusses implementation issues of both of these elements. The guiding principle of the implementation is that the Sandbox should be intuitive to use. The chapter starts with a discussion of the underlying communication system which enables DLAs to communicate. Next, it is shown how an initial Sandbox is instantiated and how it can be dynamically extended with new modules. This is followed by an explanation of security and access control mechanisms implemented in the Sandbox. The current version of the Sandbox is described and evaluated next. The chapter concludes with a summary and a discussion of related work.

4.1 Introduction

From the outset, the key requirement for the Sandbox has been usability. The elastic functionality should be a natural extension to a network programmer's normal work environment. For this purpose, use is made of programming languages that are commonly used by network programmers: *C/C++*, *Java* and *Tcl/Tk*. However, implementations in other languages and environments should be straightforward. In particular, *Agent Tcl* [**Gray96**] and the TUBE [**Halls97**] centered around *Scheme* are good candidates for building a Sandbox[1]. Agent Tcl and the TUBE are discussed in Section 4.8.1.

[1] The reason why standard Tcl was favoured over Agent Tcl is that the latter modified the core of the Tcl interpreter to implement state capturing. This resulted in a very significant performance penalty [**Gray96**]—while the functionality, although useful, is not essential. Also, Agent Tcl introduces a host of other commands to support *agents* explicitly, which in a minimalistic approach such as the Sandbox are not needed (and can always be added as extensions)

4.2 Communication

For the implementation of the SUFI's interaction primitives, much use was made of ideas borrowed from DPE technology. In particular, Sandboxes were implemented on two CORBA compliant DPEs, namely DIMMA [**Li95**] and OmniOrb [**ORL97**]. These Sandboxes communicate using the Internet Inter-ORB Protocol (IIOP) as the means of communication. A complete Sandbox was implemented for Tcl and a partial implementation was made for Java.

The motivation for using CORBA-style communication to implement the SUFI methods is that it also provides the necessary functionality for interface references and location-transparency. Moreover, since the network control components are built on top of such DPEs anyway (see Section 2.3), the necessary functionality comes essentially for free. The disadvantage is that CORBA compliant DPEs are rather heavy-weight. It is straightforward to implement Sandboxes on a more light-weight communication system[2].

4.3 Instantiating a Sandbox

The Sandbox is implemented as a single *C++* class. Its constructor automatically creates a *Tcl* interpreter or *Java* virtual machine (depending on the compile option), initialises the internal data structures (e.g. the capability lists for access control) and exports the SUFI methods to the runtime. When the construction process is complete a basic Sandbox has been created which provides exactly one module: the SUFI. New modules can be added dynamically to the Sandbox by registering them with the SUFI. Registration automatically exports the module's methods to the Sandbox. By default, external processes have neither access to the operations in the various modules, nor to the procedures and state implemented by the DLA. Explicit access enabling is required to expose procedures and state to the outside world.

Like NodeOS, the interface used to create a Sandbox allows a specification of the resource requirements of the Sandbox. This currently includes Nemesis-style specifications of period and slice of CPU time, maximum number of threads, heap size, stack size, bandwidth for inter-Sandbox communication, etc. In the implementation on Solaris, however, these resource requirements are not actually allocated to the Sandbox by the operating system. A future implementation on an operating system like Nemesis will allow the resource requirements to be strictly enforced. Unlike [**Kulkarni98**], the work presented here needs no *a priori* assumption that DLAs typically have short execution times.

[2] For example, as a proof of concept, Sandboxes were built over simple TCP/IP sockets.

4.4 Security

The introduction of foreign code in the heart of a component introduces security risks which require, amongst others, careful shielding between the DLA and the rest of the component, and control over the resource consumption of DLAs. The latter problem can be dealt with if an operating system such as Nemesis is used. Nemesis allows one to control the amount of resources (such as processor time) that a specific application is able to use. One could even allow fully compiled code to be executed in a *Xenoserver* [**Reed99**]. Shielding between two DLAs can be achieved by using different Sandboxes. Provided these Sandboxes' access to resources is bound, a DLA in one Sandbox cannot corrupt data in either the underlying system or in another Sandbox.

Security is an important issue in a system that allows dynamic code loading and largely beyond the scope of this dissertation. An interesting architecture for secure interworking services which addresses more wide-ranging security issues is the locally developed *OASIS* system [**Hayton96**]. A good example of an active networks implementation that has taken security issues into account at all levels (from the hardware up) is described in [**Alexander99**].

4.4.1 Trust establishment

A special method was added to the SUFI to deal with the challenging problem that emerges when the Sandbox itself cannot be trusted. The solution consists of enabling clients that want to load security sensitive code in a possibly untrustworthy Sandbox, to request the Sandbox to prove its trustworthiness (by sending it a *trust request*). The current implementation of this trust method uses a trusted third party to check whether the proof that the Sandbox generates is valid. The trust algorithm is extremely simple and influenced by the public key version of the Needham and Schroeder authentication mechanism [**Needham78**]. A client can either request a Sandbox to prove its trustworthiness directly, or it can generate a *trust finder DLA* to assist it in finding trustworthy Sandboxes (see Figure 4.1). The trust finder roams around the network (either at random, or via a predetermined route) and requests any untrusted Sandbox that it encounters to prove its trustworthiness to the trust finder's owner. Authentication of clients that want to upload DLAs in a Sandbox is based on essentially the same trust algorithm. The trust algorithm's encryption uses a home-grown version of RSA encoding [**Rivest78**], implemented by the author. The details of the trust algorithm are beyond the scope of this dissertation and are described in [**Bos99a**]. The RSA cryptography can also be made available to DLAs in the form of a Sandbox module, allowing them to exchange information securely.

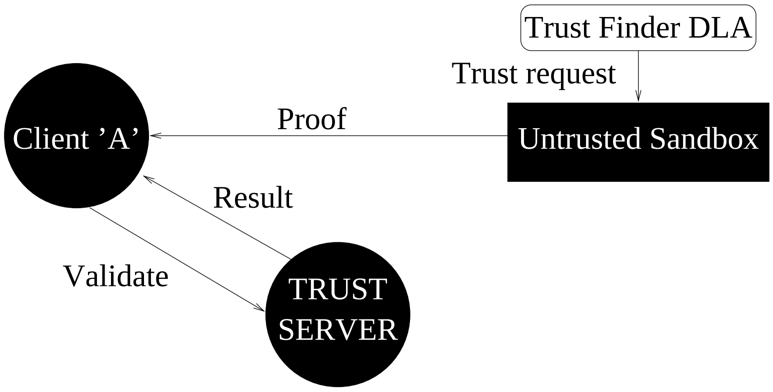


Figure 4.1: Trust establishment using a trusted third party

4.4.2 Safe runtimes

It is important that DLAs do not compromise the safety of the system. This requires restricting a DLA's access to resources. For example, DLAs should not be allowed arbitrary access to memory or disk space. For the Java implementation, use is made of the built-in Java protection. Java provides its compile-once-run-anywhere functionality by using an intermediate instruction set, or bytecode, which can be interpreted by a Java virtual machine [**Gosling96**]. Like most languages, Java uses both static analysis to ensure that programs conform with its safety specifications and runtime checking to perform checks which could not be carried out at compile time. Java bytecode is designed to reduce operand validation per executed instruction [**Tennenhouse97**].

Normal Tcl does not provide the safe execution environment required for sandboxed code. To remedy this, a variant of Tcl called *Safe-Tcl* is used [**Borenstein94**]. Safe-Tcl is a restricted version of Tcl, which in its basic form prevents Tcl code from such things as reading from or writing to file, executing external commands, etc. This basic functionality can be extended in a controlled manner.

4.4.3 Capabilities and interface references

As described in Section 3.3.2.4 remote access to operations and state is controlled by simple capabilities. Only invocations that present the appropriate capability are allowed access. In the current implementation, capabilities are simple unstructured and unprotected strings. By default, a string compare decides whether the capability matches or not. In a commercial implementation a more advanced capability scheme may be needed. To accommodate this, it is possible to specify, upon instantiation of a Sandbox, one's own procedure for verifying capabilities.

At a coarser level, access to DLA granules is restricted by the interface references. DLAs or remote processes can only communicate with a granule which exported an interface reference, if they can obtain the interface reference. Storing the interface reference in an access-restricted place limits the clients that interact with it to some privileged DLAs and processes.

4.5 Implementations

The current implementation, called the *Gobi* Sandbox, treats all parameters as a single unstructured sequence of bytes. The sequence is unpacked and restructured by adaptors, as explained in Section 3.3.2.3. By convention, whenever parameters are marshalled, all implementations first place a length field indicating the total length in bytes of the parameter sequence. This allows different implementations of the Sandbox to inter-operate. Suppose, for example, that a more advanced implementation of the Sandbox is able to treat all parameters individually, rather than as a single byte sequence. This Sandbox understands invocations from *Gobi* perfectly (type checking only consists of ensuring that the invocation's parameter type is in fact a byte sequence). Invocations coming from this advanced Sandbox to *Gobi* DLAs will be handled correctly, as long as the appropriate adaptors are installed, because its parameters are simply passed up into *Gobi* space as a single byte sequence. Currently no common format for marshalling specific types has been defined. The correct handling of parameters is the responsibility of the adaptors.

How the adaptor is installed is also not specified. It could be that the procedure's signature is described in an interface definition language (IDL) and that an IDL compiler automatically creates adaptors for both sides. If all communicating parties have agreed to adhere to the conventions defined by the IDL representation, such adaptors would work across all Sandbox implementations. Moreover, they could be generated on the fly, similar to the dynamic stub-generation in Nemesis [**Menage99a**].

4.5.1 Tcl Sandbox

The implementation for Tcl (over OmniOrb) is the only one that offers a complete *Gobi* Sandbox to its DLAs. The SUFI *Threads* submodule is fully exposed to the dynamically loadable code. Each Sandbox thread maps directly onto one native (POSIX-style) thread. The thread model is non-preemptive and detached. Parameters to the constructor allow programmers to specify both whether the Sandbox should be safe and whether the *Tk* toolkit should be supported. The implementation includes trust establishment as described in Section 4.4.1.

4.5.2 Java Sandbox

The Java Sandbox offers only part of the SUFI functionality, but not all. For example, it currently does not fully support remote evaluation. Instead, standard RPC is used for remote interaction. The *Threads* interface is also not exposed to the DLAs (instead, Java threads are used) and safety and trust establishment are not implemented. It is possible to call operations in Tcl Sandboxes from Java DLAs and *vice versa*. The Java implementation served to demonstrate that it is not difficult to build Sandboxes in other environments for other languages.

4.6 Is dynamically loadable code slow?

It may be thought that dynamic evaluation of code in the Sandbox, e.g. by an interpreter, will always be detrimental to the performance of a process. This dissertation argues that this need not be the case and that DLAs may in fact result in improvements in performance. There are two issues here. The first has to do with executing code in an interpreter as opposed to running native code. The second concerns the transfer of part of the client code to the server side, e.g. to reduce the number of remote invocations.

4.6.1 Moving client code to the server side

Dynamically loadable code can constitute a performance gain in cases where part of the client code is transferred to the server side in order to reduce the amount of network traffic. This reduction may be both in number of invocations one has to make across the network, or the amount of data that is sent back as result value by the server. Probably the best example of such a speedup is when a client performs a large number of invocations on a server, or needs to search a large database for a specific data item. Rather than sending the entire contents of the database across the wire, or making a large number of RPC calls, it is often cheaper to move (part of) the client to the server and only send back the result. Similar examples exist in network control, for example, when a large number of connections must be set up, or a large amount of management information has to be searched for one particular entry. In such a case, moving client code to the server will lead to considerable performance improvements.

4.6.2 Evaluating interpreted code

The overhead of executing the interpreted code depends on platform, language and implementation, etc. For example, the top half of 4.2 shows the duration of a single assignment in a Tcl Sandbox using three different methods of evaluating Tcl code. All measurements were taken on a Sun UltraSparc running Solaris 2.5. For reference, the execution time for native C code is also shown (compiled without optimisation using Sun's *cc* compiler). The first and slowest implementation of Tcl code interpretation, `Tcl_Eval`, represents the Tcl C library procedure used in old versions of Tcl. The line below that, `Tcl_Invoke` represents an optimisation suggested by David Nichols [**Welch97**], which only works for very simple Tcl commands. `Tcl_EvalObj` corresponds to the evaluation method introduced in Tcl 8.0, where the script is first compiled into bytecode before it is executed. On average, a single assignment (measured from start of invocation in C code till return to C code) costs approximately 7200 nanoseconds. For native C such an assignment takes 135 nanoseconds.

Program (pseudo code)	Evaluation method	Time (nsec)
x := 0	Tcl_Eval	149718
	Tcl_Invoke	24760
	Tcl_EvalObj	7240
	Compiled C code	135

proc mysquare (int x) {
return x*x;
}
for (i=0; i<100; i++)
x := mysquare(i);
	Tcl_Eval	312498

	Tcl_EvalObj	115726
	Compiled C code	48654

Figure 4.2: Performance of different types of evaluation

The bottom half of the table shows the execution time for more complicated code, containing a loop, function calls and arithmetics. Only `Tcl_Eval` and `Tcl_EvalObj` (and C code) are considered, because `Tcl_Invoke` is not general enough (e.g. it doesn't allow variable substitution). In the fastest case, the average time for this program is slightly over 312 μs in a Tcl DLA, which is good considering that each invocation consists of many function calls, product calculations and assignments. It can also be observed that the relative difference between `EvalObj` and equivalent native C code, is now significantly smaller.

More interestingly, the table shows that the speed of evaluation of Tcl code is improving considerably as new versions of the interpreter appear. A similar test using Java resulted in invocation times of a few tens of microseconds for a single assignment (using the Java Developers Kit version 1.1.2). Java code evaluation, however, can be sped up considerably using just-in-time compilers. Moreover, really fast dynamically loadable code can be created using the "RISC"-like virtual machine instruction set as done in OmniWare [**Adl-Tabatabai96**]. A comparison of the performance of different types of interpreted code is given in [**Romer96**]. In conclusion, dynamically loadable code need not be much slower than static code. Its use may even lead to a speedup when a transfer of client code to the server side reduces the amount of data to send over the network.

4.7 Active trading

A *trader* is a very common and useful component in distributed computing. It allows servers to register offers for services in the form of interface references at a central location, which can subsequently be picked up by clients [**Bearman91**]. For example, a video source may register an offer for its service under the name `Video_1`. A client that wants to display the video can query the trader and ask it to return an offer called `Video_1`. A problem with this approach is that there may be multiple instances of a service (e.g. replicas of the same video). It is desirable that the interface reference of the *best* service instance is returned (e.g. closest to the client, or least loaded). This is difficult to do using traditional trading. As an alternative, this section proposes *active trading*, which allows registration not only of interface references, but also of simple DLAs that enable one to find the best interface reference. When a client requests a service by name, the DLA is executed and an interface reference specified by the DLA is returned. In the network control model of Section 3.1, the trader is part of the *generic services*.

In the current implementation, the evaluation of the DLA takes place in the trader and only the interface reference is returned to the client. This makes *active trading* completely transparent to the client. The active trader behaves exactly like a normal trader and clients need not have instantiated a Sandbox for the DLA's encoding to find their interface references. An alternative would be to offload this task from the trader and move it to the client. In this case, instead of returning an interface reference, the trader returns a program that, when invoked, will return an interface reference according to some policy. These *active interface references* have been experimented with in the context of *Noman* control architectures, discussed in Section 5.5.

As an example, consider a service which is available at two places, A and B. The implementers of the service would like to register the service under a single name, governed by the policy that if on the last client request it returned the interface reference for A, it should now return the interface reference for B and *vice versa* (thus implementing a crude load-balancing scheme). In the active trader, this can easily be specified by registering the following algorithm (in pseudo-code):

```
if ( variableNotDefined (LastIrefReturned) ) LastIrefReturned = IREF_A
else if ( LastIrefReturned == IREF_A ) LastIrefReturned = IREF_B
else if ( LastIrefReturned == IREF_B ) LastIrefReturned = IREF_A
return LastIrefReturned
```

4.8 Related work

Mobile agent technology is a very active field and many implementations exist. The next two sections discuss work on remote evaluation and mobile agents.

4.8.1 Remote evaluation

Related work on remote evaluation was presented in [**Stamos90**]. An important difference with work discussed here is that the remote evaluation (REV) prototype does not allow remote clients to install new procedures at the server. Also, the implementation is tied to a homogeneous language environment. In the paper, an extended version of CLU is described, but it is observed that similar extensions could be built in other languages as well. Mixing languages, however, is not allowed. A distinct compile phase for the entire program is assumed which allows rigorous checking prior to execution. Section 3.3.4 shows that the Sandbox SUFI retains an amount of language independence.

Early work on remote evaluation includes PostScript [**Adobe85**], which is used for example to dynamically load PostScript code into printers. Building on this, research into a special-purpose REV mechanism was conducted in the context of the SunDew distributed window system [**Gosling86**], where PostScript programs are transmitted between processes. At DEC, a system known as the network command language (NCL) was developed that implements remote evaluation and breaks the traditional client-server model by allowing clients and servers to exchange LISP expressions [**Falcone87**]. Even older than this is work in the context of operating systems described in [**Gaines72**] where individual applications are able to extend the functionality of the kernel.

4.8.2 Roaming agents

Remote evaluation in the form of mobile agents is provided by the *TUBE* [**Halls97**]. Code mobility is made straightforward by the ability to *freeze* running code and continue its execution somewhere else. The TUBE is language-dependent: properties of the *Scheme* language are used for saving the program state. The TUBE has been used in experiments with network control. By linking its libraries with stubs for an advanced control architecture it was possible to create mobile code that set up and tore down connections over which it transmitted *itself*. The TUBE has also been used in management experiments by making TUBE sites interact directly on the same host with *Ariel* interfaces (i.e. the TUBE site assumes the role of control architecture).

Telescript was one of the first implementations of mobile agents and denotes both the language and the agent environment [**White97a**]. The key concepts in Telescript are *agents* and *places* (virtual locations that can be occupied by agents). Agents in different places are able to connect to each other across a network. Agents in the same place are able to *meet*. Recently, Telescript has been replaced by the Java-based Odyssey. Agents in *Agent Tcl* are programs that are supported by a common services package (enabling mobility) implemented as a server [**Gray96**]. The assumption is that all functionality that an agent may ever want is available in the server. Explicit commands are defined to capture and define a program's state. Originally geared towards the Tcl scripting language there is now some support for Java and Python as well[3]. *ARA* is similar to Agent Tcl (and to the Sandbox) in that it provides a core service layer [**Peine97**]. The core service layer supports multiple languages through interpreters. State capture, security and correctness checking is not specified. *Aglets* are IBM's approach to implementing mobile agents [**Lange96**]. The approach is language-specific: mobile agents (or aglets) are Java programs that run in the *Aglet Workbench* environment. There is a layered security model, where at the lowest layer the Java virtual machine is always trusted. [**MAF96**] describes how aglets are CORBA objects that are allowed to migrate. In Tacoma agents are migrating *processes* [**Johansen95**]. The agent's state is stored in what is called a *briefcase*. State in *Tacoma* has to be handled explicitly by the programmer and security is weak [**Pham98**]. The agents are currently specified in Tcl. In the realm of telecommunications, some groundwork is expected to be covered for mobile agents in TINA by the MAGNA project (TINA is discussed in Section 5.6.1.6). *Obliq* [**Cardelli95**] is an untyped, lexically-scoped language with direct support for distributed computation. Obliq objects have internal state and are local to a site, while computations can roam over the network (their network connections are maintained). ObjectSpace's *Voyager* is another Java-based agent system [**ObjectSpace97**]. It is built around an ORB and offers integrated CORBA support. Agents in Voyager are autonomous and can be made to move to new sites to resume execution.

All these systems offer advanced agent mobility, but are considerably more complex than the implementations described in this chapter. Note that the Sandbox framework does not prescribe any one implementation of the dynamic execution environment. Each of the above systems can be used as a Sandbox, as long as they allow implementation of the SUFI. An overview of agent projects is given in [**Pham98**]. In [**Harrison97**], the idea of extending a server's functionality using DLAs (as proposed in the *Haboob*) is assessed as "a very valuable new capability".

4.9 Summary

In this chapter implementation details of the Sandbox were discussed. In particular, SUFI communication and security issues of a Sandbox implementation called *Gobi* were described. *Gobi* was implemented for Tcl and (partly) for Java. *Gobi*'s performance was evaluated and it was argued that loadable code need not be much slower in execution than static code and can even improve performance in certain cases. The Sandbox is a generic DLA platform, i.e. it is not specific to network control. In the next chapters, it will be shown how the Sandbox is applied to network control. For demonstration purposes, a generic service called the *active trader* was implemented.

[3] Agent Tcl has recently been renamed *D'Agents*

Chapter 5

Elastic control architectures

In the previous chapter an implementation of the Sandbox was discussed. In this chapter, it is shown how Sandbox elasticity can be used in network control. Specifically, this chapter deals with what is probably the most complex component of the network control model, namely the control architecture. Two types of elastic control architecture are discussed. The first, called *Sandman*, constitutes the bulk of this chapter. The Sandman is a complete control architecture that offers advanced functionality such as reservation in advance, measurement-based admission control, and more. Flexibility can be added in the form of DLAs. The second type of control architecture, called *Noman*, is a Sandbox with a module that allows programmers to quickly build control architectures using DLAs.

The Sandman's discussion is divided in four parts: Section 5.1 discusses its core functionality, Section 5.2 explains how traffic measurements are used for admission control, Section 5.3 describes a video server that was built using the Sandman to validate its functionality, and Section 5.4 shows how DLAs can be loaded into the control architecture to make it elastic. Noman is discussed in Section 5.5 and related work in Section 5.6.

5.1 The Sandman: basic functionality

This section explains the core functionality of the Sandman. This includes resource reservation in advance, recursive repartitioning of allocated resources, and call admission control based on measurements. Sandman is an example of a control architecture that was not designed with sandboxed DLAs in mind: elasticity was added to an existing piece of software. Before the Sandman is discussed in detail, the next three sections consider problems to do with connections that depend on other connections, advance reservations and multi-point communication.

5.1.1 Connection dependence

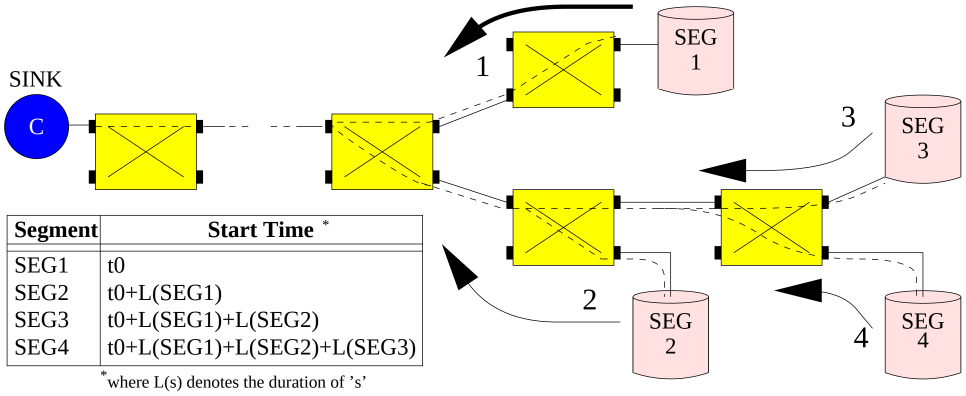
Current technology allows users to access continuous media data as well as traditional types of data simultaneously across a network. Problems arise when resources (e.g. file servers or network links) are overloaded and either degrade the system as a whole or cause new requests that want to share the resources to be rejected. It will be shown that these problems are exacerbated when communication involves multiple parties distributed over a network. This is not uncommon. Continuous media sources may be distributed for a variety of reasons:

- Continuous bit rate (CBR) encoded HDTV video requires approximately 17.5 MB of storage per second, a few hours of which will not fit completely on the disks of most clients and may well need to be segmented and distributed.
- Load-balancing is another reason for source distribution. [**Dan95**] proposes to balance the load on file servers by chopping up continuous media files in segments, which are distributed over the network, depending on the load. Furthermore, audio, video and other types of data may be stored on specialised servers for efficiency [**Lo94**].
- Many sources are distributed by nature. Cameras and microphones for example, are attached to workstations or, in the case of security cameras, distributed over an organisation's site.

Source distribution becomes a problem if the acceptance of a call to one source depends on the acceptance of calls to a set of other sources. This will be called *connection interdependence*. A connection is dependent on other connections if its establishment is only meaningful in the context of the (possibly future) establishment of a number of other connections. The connection dependencies will be called *temporal*, if the acceptance of a connection is only meaningful in combination with the acceptance and establishment of certain other calls *at a certain time.*

For example, if a video file is distributed over four nodes (see Figure 5.1), the playback of the entire video requires that segment 1 is played first, immediately followed by segment 2, etc. It is not acceptable that the first three connections are accepted while the last one is rejected (i.e. the client would not be able to watch the end of the movie). It is probably also not acceptable to set up all four connections in advance for the entire duration of the video, since this would be a needless waste of bandwidth, as all four connections would be idle for 75% of the time. Instead, there is need for an admission control algorithm that guarantees that if the connection to source 1 in a sequence is accepted, the connections to all subsequent sources are also accepted at the appropriate times. So connections 1 to 4 in Figure 5.1 are said to be temporally interdependent.

These sorts of guarantees are required in all systems with temporal connection interdependencies. A 'live' example might be a video conference where a number of speakers have been allocated speaking time in advance. Henceforth, the establishment of a set of interdependent connections will be called a *session*.



Segment	Start Time *
SEG1	t0
SEG2	t0+L(SEG1)
SEG3	t0+L(SEG1)+L(SEG2)
SEG4	t0+L(SEG1)+L(SEG2)+L(SEG3)

*where L(s) denotes the duration of 's'

Figure 5.1: One video file consisting of four segments

5.1.2 Reservation in advance

An active research topic in the field of resource management is the problem of resource reservation in advance (both in IP [**Degermark95**], and in ATM [**Ferrari97**, **Schill97**, **Hafid98**, **Bos98c**]). As to whether advance reservations are needed at all, as questioned by [**Baker97**], this dissertation follows [**Degermark95**] by observing that this first of all depends on the scarcity of resources. If resources are not scarce at all, not even immediate reservations are necessary. If resources are scarce, however, there exists a class of applications which would benefit from advance reservations. Whether advance reservations are needed, depends furthermore on whether the value of the advance reservation outweighs the cost of making the reservation. Another way of saying this is that it depends on the loss (or cost) that will be incurred if the resource cannot be reserved at the appropriate time. This may be independent of the value of the resource itself. Analogies can be found for reservations in other areas. For example, it is worth the cost to reserve a seat on a plane, but it makes no sense to reserve in advance for the subway. In networks, there are applications where the value of the reserved bandwidth is sufficiently high to justify the cost of the advance reservation.

Advance reservation can be defined in a general way as the reservation of resources for some future time period(s) during which these resources will be allocated to the requesting application. The amount and nature of the resources reserved is determined by some *policy* (which in the most general case may depend on time, load in the network, etc.).

5.1.2.1 Resource control and advance reservation

Solving the connection dependence problem in an environment where bandwidth is scarce, requires the ability to make advance reservations. For example, if a client C requests the playback of the video file of Figure 5.1, a connection has to be set up from server 1 to C for interval $[t_0,\ t_0 + L(seg_1)]$ and from server 2 for $[t_0 + L(seg_1),\ t_0 + L(seg_1) + L(seg_2)]$, etc. All these calls require resources, such as bandwidth, for which a reservation is needed. Without advance reservation, it is impossible to guarantee, while efficiently using the network resources, that all segments will be able to obtain the appropriate resources at the appropriate times. Advance reservation decouples the time of request for resources from the time of allocation of resources. It will be shown next that the desired functionality goes beyond a simple end-to-end advance reservation scheme.

5.1.2.2 Problems with traditional end-to-end reservation

Suppose that two source disks, connected to the same switch, contain two consecutive segments of a video (e.g. file servers 3 and 4 in Figure 5.1). The connections from both sources to the sole sink are almost identical and follow one another seamlessly with the same QoS. End-to-end advance reservation of connections, however, precludes the reuse of part of an existing connection. In all probability, this results in a very large handoff overhead, because a new connection has to be set up from endpoint to endpoint which may involve many switches. In the worst case, it means that all resources upto and including the sink are involved in the setup of a connection that is exactly the same as its predecessor. There is a need for less restrictive connection types[1].

In most advance-reservation models, the only way to give the required guarantees for connection interdependence is to make n separate end-to-end reservations, where n is the number of connections needed to play an entire video [**Ferrari97**, **Schill97**, **Hafid98**]. This not only introduces a large, unnecessary overhead, maintaining n separate connections also presents a considerable administration task. It would be easier if a new type of connection was introduced which can have multiple sources (in sequence) or multiple sinks or both. One tear-down request would then remove all state corresponding to the session from the network. In the next sections, two new connection types are introduced.

5.1.3 Multipoint connections

Like most control architectures for ATM, the Sandman supports simple point-to-point (unicast) connections, as well as single-source multicast connections. Joining a multicast can be initiated by leaf nodes, source nodes or third parties.

[1] This is a 'normal' signalling problem, not specifically related to the problem of connection interdependence.

Besides these relatively standard operations, the control architecture employs two new types of connection to connect multiple sources to one or more sinks. 'Multiple source' does not mean multiple connections. Instead, one connection is *time-shared* by several sources. In other words, these connections types implement *sessions*. This addresses to a certain extent the problem of multipoint communication in ATM. In ATM, it is difficult to implement multipoint-to-point and multipoint-to-multipoint connections. The problem is that whenever multiple sources share a connection, their higher-level ATM Adaptation Layer (AAL) frames are first segmented into ATM cells and then transmitted on the wire in any order. Such practice may easily lead to the interleaving of AAL frames. This is unacceptable for non-multiplexing AAL types such as AAL5.

AAL types such as AAL3/4 have been said to remedy the interleaving problem. AAL3/4 supports multiplexing AAL connections into a single ATM connection using the 10 bit multiplexing identification field (MID). Unfortunately, its resource utilisation is very poor. For example, the effective payload of an AAL3/4 ATM cell is at best 83%. The utilisation of AAL5 which does not support source multiplexing is approximately 90.5%. Moreover, AAL3/4 by itself does not solve the interleaving problem when the frames are sent from different sources. For this, MID value coordination between the sources needs to be added, which adds an extra level of complexity. As a result, AAL5 is favoured in practice and it would be useful to implement multipoint connections that handle AAL5 packets well.

Existing mechanisms for multipoint communication are summarised in [**Diot97**]. Most implementations of multipoint-to-multipoint communication for n parties employ either n sessions of $1 : n$ multicasts (e.g. [**Bettati95**]), or use a special *multicast server* (MCS), e.g. [**LANE95**]. In the latter solution, all parties establish a point-to-point VC to the MCS. When a party wants to send data to the group, it sends the data to the MCS instead, which subsequently multicasts it to the destinations. Sandman adopts these models only when multiple sources are active simultaneously. Otherwise a more optimal solution is used which involves connection sharing, as shown in the next section.

5.1.3.1 Reusing resources of previous connections

The two new connection types provide a limited form of multipoint-to-point and multipoint-to-multipoint communication support. In case there is only one sink (multipoint-to-point), the connection behaves like a rattlesnake, with its head at the sink and its tail at one of the sources: the tail may flip rapidly from one source to the next, but the rest of the connection's body remains unchanged. In fact, the sink is not even aware of the hand-off of the sources. It just listens on the same VPI/VCI pair and the connection is never torn down until all sources have finished.

For example, in Figure 5.2 a sink requests video-conference access to two

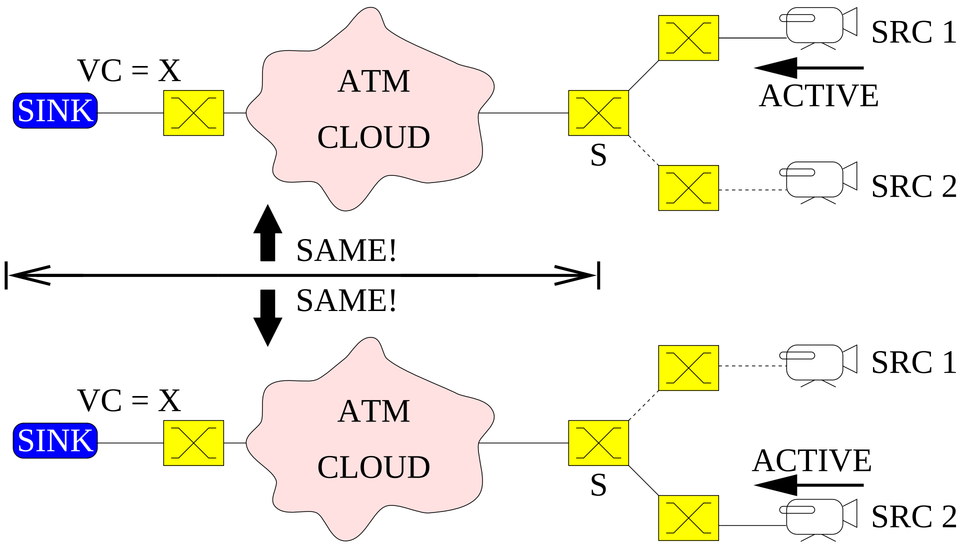


Figure 5.2: Multipoint-to-point connection: only the source moves

cameras in sequence. The cameras are connected to two switches that connect to the same switch S. The paths from cameras to sink have an arbitrary number of switches in common. At the top the connection is from camera 1 to the sink, while at the bottom the source has changed from 1 to 2. For the connection, however, hardly anything has changed. Only in the very last switch connections (from S onwards) the change has been made from camera 1 to camera 2. The overhead of setting up the entire connection from source to sink, each time the source moves, is no longer needed. Resources are also not wasted, for example by maintaining multiple connections to the same sink in parallel. A final advantage is that the sink need not be aware of the handoff. Something similar could be used in the hand-off process of mobile systems, although it is currently not used for this purpose.

5.1.3.2 Sequences and patterns

Concretely, to deal with temporal connection interdependency, the Sandman introduces the following two new connection types:

1. *Connection sequence*: this type has an arbitrary number of sources that follow each other seamlessly and a single sink that may not even be aware of the number or location of the upstream sources or hand-offs. Identical portions of the connections of subsequent sources are automatically reused. The operation to reserve and create this type of connection takes as arguments a reference for a sink (e.g. a display program on a workstation), a start time, and a list of source records each of which specifies a source, the resources required (e.g. bandwidth) and an end time. The

end time for source i is the time until which i is active and serves as the start time for source $i+1$.

2. *Connection patterns*: this type has the same source sequence, combined with an arbitrary list of (possibly overlapping) sinks. The operation is largely the same as a connection sequence, but the hand-off process is more complicated. The operation takes the same arguments as the connection sequence except that there is now a random list of sink records each of which contains a sink reference as well as a period $[t_{start}, t_{end}]$ during which this sink is active. Overlapping sink intervals indicate multicasts during the overlapping periods.

Both types of connections (as well as the more common point-to-point and multicast connections) can be reserved in advance, modified and made resilient to failures (see Section 5.1.5). Connection sequences are essentially restricted multipoint-to-point connections, while connection patterns are restricted multipoint-to-multipoint connections[2]. There is no danger of AAL frame interleaving because there is only a single source that is active at the same connection at any time. All connection types in the Sandman can be initiated either by the source or the sink, or even by third parties.

5.1.4 Sandman components

The Sandman was designed following the control architecture model described in Section 3.4.1. The most important entities of the distributed control architecture as well as their interactions will be described next. The Sandman uses the following entities (see also Figure 5.3): *local host manager*, *local trader, connection manager* and *federated trader*. The local host manager provides an interface to the control architecture. Generally, there will be a local trader and a local host manager associated with each host that wants to communicate (where the local trader and local host manager may or may not run on the same machine).

As an example of the interaction between these entities , consider a connection setup between a source and a sink application. Assuming that the setup request was initiated by the sink (indicated by `start()` in the application on the right in Figure 5.3), the sequence of interactions is as follows:

i. Before communication can start, the source must export a service offer, which is registered with its local trader, e.g. an offer for the service to send video. The trader also registers a callback function with this offer, which is the operation (e.g. "`SendVideo`") that will be executed when a client binds to the offer. Likewise, the sink registers an offer with its local trader, e.g. an offer to display video that is received from the network ("`Display`").

[2] In the proof-of-concept implementation, heterogeneity of sink QoS was not addressed

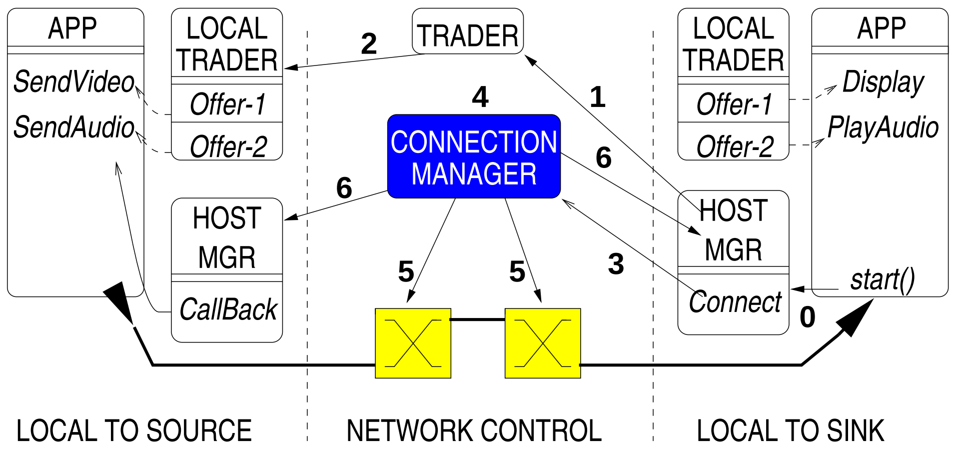


Figure 5.3: Interactions between the various Sandman components

ii. A client who wants to access the service, tries to find an offer for it using the local trader and federated trader which in turn uses the other local traders (actions **1** and **2** in Figure 5.3).

iii. It also obtains a handle on its local sink offer (“`Display`”).

iv. The client, via the local host manager, requests the connection manager to connect source offer to sink offer for the desired time interval[3] (action **3**).

v. The connection manager tries to reserve the appropriate resources on the data-path and if this completes successfully, it sends back an acknowledgment to the local host manager.

vi. The request sleeps (**4**) until t_{start}, when the connection is set up (**5**).

vii. If successful, the connection manager tells the local host managers on both sides to invoke the callback operations (**6**): the source is told to send video and the sink is told to receive and display frames.

viii. Teardown at the end of the interval is analogous: the source and sink are told to stop sending and receiving respectively and all state corresponding to the connection is removed.

5.1.5 State and failure

When resource reservations are made, *state* is introduced into the network. The nature of this state is important and requires the following questions to be answered: (1) how should one specify what portion of a shared resource needs to be reserved (e.g. how much *bandwidth* is needed for a connection), (2) should the state be *soft* or *hard*, and (3) what should happen in the event of failures?

[3] It is assumed that a clock synchronisation mechanism provides global time.

5.1.5.1 Specification of required resources

Reservation requests must contain a specification of the resources required for the connection. The question is how detailed such a specification should be. It is possible to take a simplistic approach, which assumes no knowledge whatsoever about the behaviour of the sources and the best that can be done is to provide the peak rates. Alternatively, one can use a more sophisticated scheme, whereby a very detailed characterisation of the source behaviour is supplied, possibly using parameters such as peak and sustainable cell rate, burst length, etc.

In Sandman, the former option was chosen: resource specifications consist of *peak rate* only. This dissertation argues that accurate source characterisation (modelling) is prohibitively difficult. The problem with peak-rate reservations is that it leads to poor resource utilisation. This problem will be addressed in Section 5.2.

5.1.5.2 Soft state vs. hard state

Another issue is whether one should use *soft state* or *hard state*. RSVP uses the concept of soft state, which exists only as long as periodic messages are sent along the datapath [**Zhang93**]. If at some node, the messages fail to arrive, the soft state is removed. This is very attractive in certain aspects, but it is not very suitable for advance reservations, because it requires nodes to be up all the time. This is an unreasonable requirement (when users reserve a video conference for next week, they don't care whether all nodes on the datapath go down overnight). Instead, the Sandman uses "hard" state, i.e. state that is only removed as the result of an explicit release operation. Hard state comes at the price of more complicated releasing of resources. However, the state is not extremely hard, i.e. it will disappear of its own accord after the reservation interval. In other words, it is covered by its own timeout mechanism. The nature of this mechanism, however, is different from the soft-state approach.

5.1.5.3 Failure and recovery

Failures may occur at any time and at any location in the network. Failure handling and recovery procedures in Sandman are simple but adequate as a proof-of-concept. For example, the way the connection manager deals with the failure of an operation (e.g. a reservation or setup failure, because a remote host crashed) is by throwing an exception and releasing all resources allocated to or reserved for this request. It also notifies the host managers involved.

A more interesting problem arises when the connection manager itself crashes. Consideration should be given to what effect this should have on the connections. In the Sandman, a distinction is made between two types of sessions: *persistent* and *volatile*. The volatile sessions crash with the connection manager, i.e. their connections will never be completed. If the crash happened before the start of the reservation interval, no connections were ever set up and never will be. Otherwise as soon as a connection manager comes up again, the connections' resources will be released. Persistent sessions on the other hand, are written to stable storage at reservation time. If a crash occurs before the start of the reservation interval and the connection manager comes up again before the start of the interval as well, then all reservation state is simply restored and the client is not even aware of the crash. Otherwise, if the connection manager comes up in the middle of a session, it figures out what connections should have been active at that moment, and if they are not active, sets them up. It also performs all pending notifications and cleanup operations.

As an experiment, finally, a *mirror* connection manager was run. The primary connection manager still takes care of setting up connections, but also sends all requests and replies from clients and host managers to the mirror, along with heartbeat messages. The mirror runs in *emulation mode*: it doesn't set up connections, but it keeps state as if it does. If the heartbeat messages fail to arrive, the mirror assumes that the primary connection manager has died and takes over its role automatically. All parties that communicated originally with the primary manager are notified of the change. Recently, a similar strategy was proposed for services deployed in active networks [**Kulkarni99**].

The most difficult failure, when a network is partitioned for a long time and the connection manager cannot reach one of the partitions to release resources, is the subject of future research.

5.2 Call admission control in the Sandman

Call admission control (CAC) on the one hand aims at providing applications with QoS guarantees and on the other strives to make optimal utilisation of the network resources. Even if matters such as *fairness* are left out of the equation, these are two somewhat conflicting goals that are difficult to reconcile.

5.2.1 Introduction

Admission control and resource reservation are closely related. Based on the resources that have been reserved already for existing connections and the resources that are needed for a new connection, it is decided whether or not this new connection should be accepted. In other words, it's decided whether the QoS requirements of this new connection can be met, without jeopardising the QoS of previously accepted connections.

In ATM, a wide variety of CAC algorithms has been proposed. Some of these algorithms use very clever estimation of the total bandwidth that is required for the multiplexing of n connections of which some properties (e.g. peak rate, average rate, burst length) are known. Gaussian Approximation and Equivalent Capacity are examples of these algorithms [**Guerin91**].

Most of these algorithms suffer from relying on a (static) model of the traffic, while, as mentioned before, it is often impossible to accurately characterise sources. An approach that is therefore frequently taken is the division of calls in a small set of Quality of Service (QoS) classes. This technique is problematic also, since one can never define classes that fit all possible types of traffic (including those of future services). Finally, most existing CAC algorithms differ from the policy proposed here in that they deal with whether a call can be accepted *now* rather than at some time in the future, which is needed for the guarantees mentioned above.

Recently, a promising new approach has emerged which uses on-line measurements to overcome some of these problems [**Jamin95**, **Crosby95b**, **Gibbens97**]. This is generally known as measurement-based admission control or MBAC. Instead of trying to model *a priori* the behaviour of a source, it is proposed to *measure* its resource utilisation and use the knowledge that can be derived from these measurements as the basis for the CAC algorithm. Using traffic measurements, the aim is to find an estimate of the minimum resource capacity that should be allocated to a connection to satisfy its QoS guarantees with high probability. Limiting the scope to bandwidth, this estimate is called the estimated *effective bandwidth*.

Informally, the effective bandwidth of a connection is a measure of the capacity that a connection really needs, i.e. the bandwidth effectively used by the connection. More concretely, the term *effective bandwidth* (EBW), as used in this dissertation, denotes the switch buffer service rate required to keep the cell loss ratio (CLR) due to queue overflow under a specified target bound[4]. The estimation of the effective bandwidth based on online measurements is derived in Appendix A.

5.2.2 Alleviating a conservative CAC algorithm

The default CAC algorithm in the Sandman simply checks the reservation schedules to see if enough capacity is available to accept the new request and if so, updates the schedules. Reservation schedules are employed for a whole range of resources, e.g. service access points (SAPs), capacity (bandwidth), channels (VPI/VCI values), and others. In the present discussion, the focus will be on bandwidth. If reservations are entered in schedules based on peak rates alone, this may result in very poor resource utilisation, because the resulting CAC is

[4]It seems that the only QoS metric dealt with here is bandwidth corresponding to a desired CLR. However, QoS parameters are not orthogonal and other QoS parameters such as delay and delay variance also depend only on the probability distribution of the queue length (just like the CLR).

extremely conservative. Statistical multiplexing for a time interval in the future is difficult due to the unknown behaviour of future flows.

5.2.2.1 Admission control based on schedules

The default CAC uses only reservation schedules. The algorithm to decide whether a new request should be accepted is as follows:

i. Let B_{tot} be the total capacity/bandwidth of the resource.

ii. For the time interval that is specified for the new request, determine B_{sched}, the maximum amount of bandwidth reserved by adding all the peak rates of the reservations.

iii. Let b_{new} be the new request's peak rate.

iv. If $B_{sched} + b_{new} \leq B_{tot}$ accept the request. If not, reject.

This is a very conservative admission scheme: the resource utilisation with this CAC will be low, but the resource guarantees are relatively hard.

5.2.2.2 Combining peak rates with measurements

The default CAC in the Sandman is extended to use the observed traffic during a time interval in the past. From these observations the effective bandwidth is estimated and thus the amount of traffic that can be expected if a new request is accepted. If the result does not exceed the total capacity, the call can be accepted. By definition, the EBW lies between the mean and the peak cell rate. Hence, CAC based on EBW gives better resource utilisation than CAC based on peak-rate[5]. At first sight, it seems that measuring traffic is unsuitable for Sandman, because requests are generally made for an interval in the future: at call admission time, and possibly for a long time after, there is nothing to measure. On the other hand, there is knowledge about the behaviour of flows that are currently active. This can be used for CAC in the near future.

CAC in the Sandman is an attempt to bring together the strictness of peak-rate schedules and the resource utilisation resulting from measurements. Starting with CAC based on measurements and looking further into the future, the Sandman's CAC grows progressively more conservative. As time goes by, more is learned about the flows in a specific interval that were admitted with the conservative CAC because some reserved calls will have started by then. The CAC algorithm can now make less conservative decisions for new requests for that interval.

[5] The EBW estimators in the Sandman were developed as part of the *ESPRIT Measure* project. The *Measure Toolkit* written by Horst Meyerdierks was used as the starting point.

For this purpose, measuring components are connected to the resources. For example, code is added to the switch to periodically send various cell counters for each active connection to a generic service component, called the *traffic server*. In this section, only the counter denoting the number of cell arrivals on a connection's output port is considered. The traffic server uses these counters to compute estimates for the effective bandwidth (EBW) of each of them. It also obtains an estimate for the *aggregate* EBW.

In the Sandman's schedules, besides the peak rate, the EBW of the flows is also tracked. Initially, the EBW is set to the peak rate. For each new request, the CAC asks the relevant traffic servers for the EBWs of the active connections and also for the aggregate EBW.

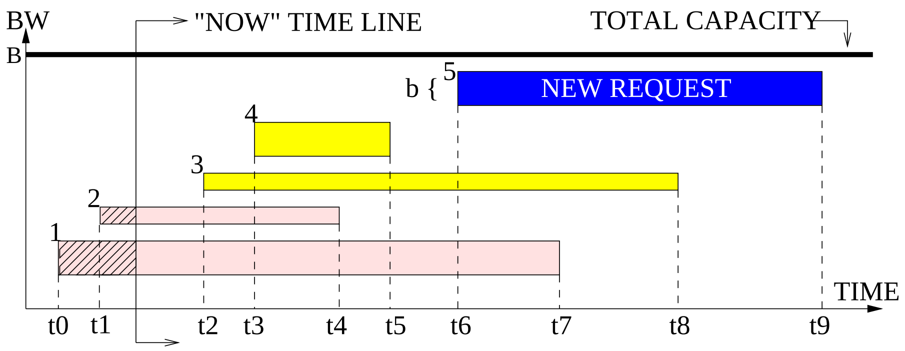


Figure 5.4: A new reservation request arrives

The process is illustrated in Figure 5.4. At $t = now$ a request arrives for bandwidth in $[t_6, t_9]$. At $t = now$ there are two active connections (1 and 2) of which EBW estimates exist. Connection 2 finishes *before* the new request starts, while the other active connection overlaps with the new reservation's interval. Between *now* and t_6 two more connections are set up (3 and 4) one of which finishes before t_6. The new CAC algorithm becomes:

i. Let B_{eff} be the aggregate EBW that is used on the resource, $b_{eff}(n)$ the EBW of connection n and $peak(n)$ the peak rate of connection n.

ii. At $t = now$ a reservation request r arrives for $[t_{start}, t_{end}]$ with $peak(r) = b$.

iii. Let E_{BW} be the estimated maximum of the bandwidth in $[t_{start}, t_{end}]$ (i.e. the bandwidth that we use to decide whether the request should be accepted or not). Initialise E_{BW} to B_{eff}.

iv. For all active flows x that finish *before* t_{start} do: $E_{BW} = E_{BW} - b_{eff}(x)$.

v. For all reservations y of which the connections have not started yet and which *overlap* with $[t_{start}, t_{end}]$, do: $E_{BW} = E_{BW} + peak(y)$.

vi. If $(E_{BW} + b \leq B_{total})$ accept, otherwise reject.

The above CAC algorithm is not optimal. It may be acting too conservatively for future reservations, because (as indicated in step (v)) the summation of the peak rates of all future reservations overlapping with the new request's interval is added to E_{BW}, regardless of whether these reservation overlap themselves. An optimal algorithm uses the *maximum overlap*, i.e. the maximum sum of peak rates of mutually overlapping connections that also overlap with $[t_{start}, t_{end}]$. This was not done in the current implementation as it would add significantly to the CAC overhead.

The more conservative treatment for time intervals that lie furter in the future may not be a problem at all, assuming that there are relatively few clients that make future reservations for a particular time interval. In that case, there will be few rejects, despite the conservative CAC algorithm. As the CAC becomes gradually less conservative for this time interval, it may be expected that by the time the conservative algorithm would have reached the capacity, a number of the streams will have started (allowing less conservative CAC decisions to be taken).

5.2.2.3 Traffic servers

Sandman provides hooks for plug-ins like traffic servers which allow the default CAC algorithm to be easily extended. Traffic servers are independent processes that fit in the generic services level in the network model of Section 3.1. They talk to the switch on one end and the control architecture on the other. Traffic servers receive raw statistics from the switch, process them and send updates of the EBW to the control architecture. The updates are generally sent on request, but traffic servers can also be instructed to send update messages periodically. For the Sandman, the traffic server is optional and can be added and replaced at runtime. If a switch cannot provide the statistics, the control architecture still works, albeit more conservatively. In fact, it is conceivable to control a heterogeneous network where some switches (or indeed some of the ports of these switches) have traffic servers, while others do not. Also, the implementation may vary from switch to switch, allowing vendors to differentiate.

The new set of components which includes the traffic servers is shown in Figure 5.5. Section 5.4.2.3 shows how traffic servers can be made elastic. This allows the traffic servers to be programmed at a fine granularity, so that they can be used for other tasks such as policing, as well.

5.2.3 Discussion

Measurement-based admission control is not without its problems. If N accepted connections are silent or near-silent for a long time, the CAC mechanism will be inclined to accept more and more connections, as their effective bandwidth estimates are low. Problems arise when suddenly the N sources become very active and start transmitting at their peak rates. This would jeopardise the QoS guarantees of all connections. A scenario in which this might happen

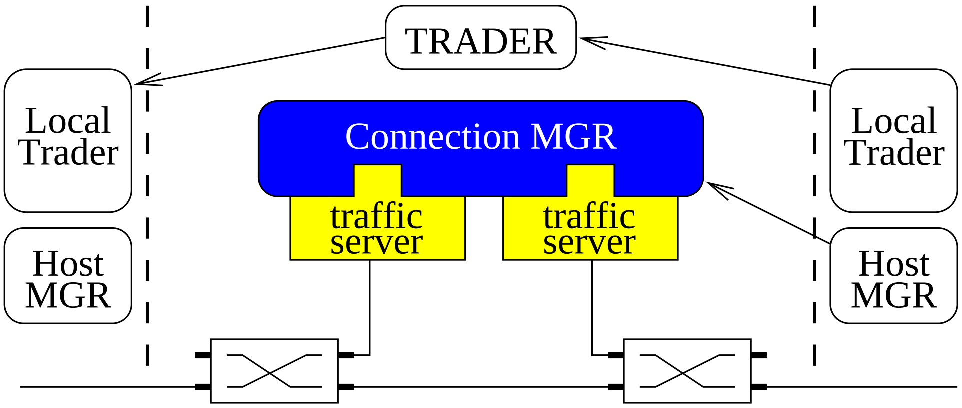


Figure 5.5: Sandman extension hooks: traffic servers can be dynamically plugged in

is when many connections are waiting silently for some video transmissions to begin, e.g. the news at nine, from different channels. Although the connections have been set up, little or no data is sent on these connections. This changes when the video transmissions start. The CAC algorithm will need to be refined to cope with these situations. For example, if there is a large discrepancy between the declared peak rate and the measured effective bandwidth, the algorithm could become more conservative, by limiting the number of connections that it is willing to accept. The more connections with a very high peak to EBW ratio, the fewer new connections such an enhanced algorithm would admit.

The results of the call admission control algorithm, as described in this chapter will be discussed in Section 8.4. Because the effective bandwidth lies between the average rate and the peak rate, CAC based on estimates of the effective bandwidth will be shown to be significantly less conservative than CAC based on the peak rate alone. In addition to this, it should be noted that because of the solid mathematical derivation of the effective bandwidth estimation, it is still possible to give probabilistic QoS guarantees. This is almost impossible with less rigidly determined estimates such as [**Jamin95**]. Recently, however, it was suggested that, from the viewpoint of optimising resource utilisation, further research on even better admission control equations is fruitless, as the differences in resource utilisation is minimal [**Breslau99**].

5.3 Building a distributed video server

Although audio and video sources and sinks are rapidly becoming ubiquitous in computer networks, the recording and playback of continuous media data, is by no means a solved problem. As a validation of the Sandman, this section will discuss the implementation of an experimental distributed video server.

5.3.1 Introduction

Storing large video files on disk is problematic due to two problems. First, most users do not have the storage space to record large high-quality video files. For example, the recording of 1 hour worth of HDTV-quality video (CBR encoded) requires more than 60 GB of diskspace [**Jardetzky95**]. Second, even when storage space is abundant, we find that most present-day disks are only able to serve a small number of simultaneous continuous media streams without compromising QoS guarantees. These systems (or the network access to them) are likely to become bottlenecks. Load balancing, therefore, becomes important.

In the next few sections, a prototype distributed video server (DVS) called *BigDisk* is discussed that addresses both of these problems. It places few demands on the machines in the network (indeed, these could even be low-end PCs with small disks). The load on the disks is automatically balanced and a solution is proposed that both improves latency considerably and enables prefetching for continuous media.

5.3.2 Recording and playback with the DVS

In this discussion of BigDisk, the focus is not so much on the problem of guaranteed rate storage servers[6]. Whether the lower-level storage server is capable of delivering guaranteed rate is considered an endpoint problem. It is assumed that the request that is sent to the DVS is based on the endpoint QoS constraints that are only known at the application level[7]. In cheap systems, users will simply use the cheap disks in their PCs for storing and retrieving. High-end users may prefer to use guaranteed-rate storage servers for high-quality video. Both types of users are supported.

For now, the principal aim of the DVS is to provide users with limited diskspace with a virtual 'Big Disk', which may comprise all disks on the network on which the user has write permission. In the DVS each designated disk stores a segment (e.g. 10 minutes) of a large video. Consider again Figure 5.1 for an illustration of how a video file has been segmented and distributed. The front-end of the DVS is a graphical user interface, while the backend consists of calls into the Sandman. The front-end enables users to inspect the various disks on the network, add new disks to the system and select the mode for this DVS session. The BigDisk DVS has two modes: recording and playback.

In recording mode, shown in Figure 5.6, users select the disks on which to record the continuous media data. Even if a video fits entirely on the local drive, users may still choose to distribute it over a number of disks to balance the load. After users have selected the disks that act as sinks, and the video

[6]Nevertheless, a simple continuous media storage server was built for experimentation purposes. More advanced systems are described in [**Anderson92**, **Lo94**, **Jardetzky95**].

[7]The rate of the storage server *should* be taken into account for end-to-end QoS guarantees

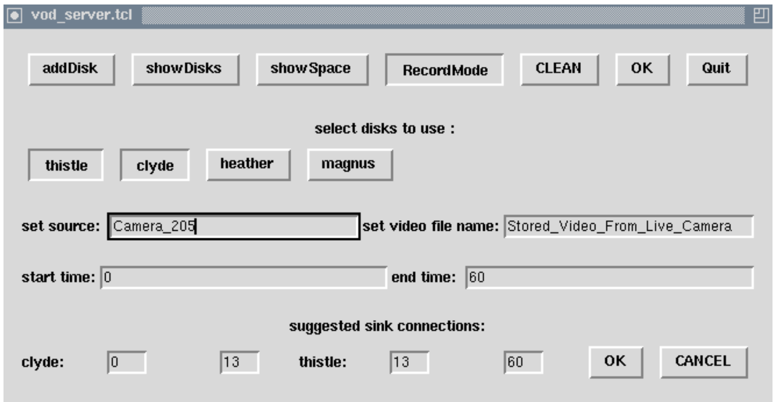


Figure 5.6: BigDisk recording mode

stream that acts as source, they specify the name for this video and the start and end time for the stream (e.g. in case of a recorded television program, these can be the start time and end time of the program). The DVS then suggests a particular segmentation and distribution corresponding to the user's choices. The suggestion is based on a load balancing policy. For example, in the prototype, the DVS tries to guarantee the same relative disk usage on all disks involved. The suggestion can be overruled if the user so desires.

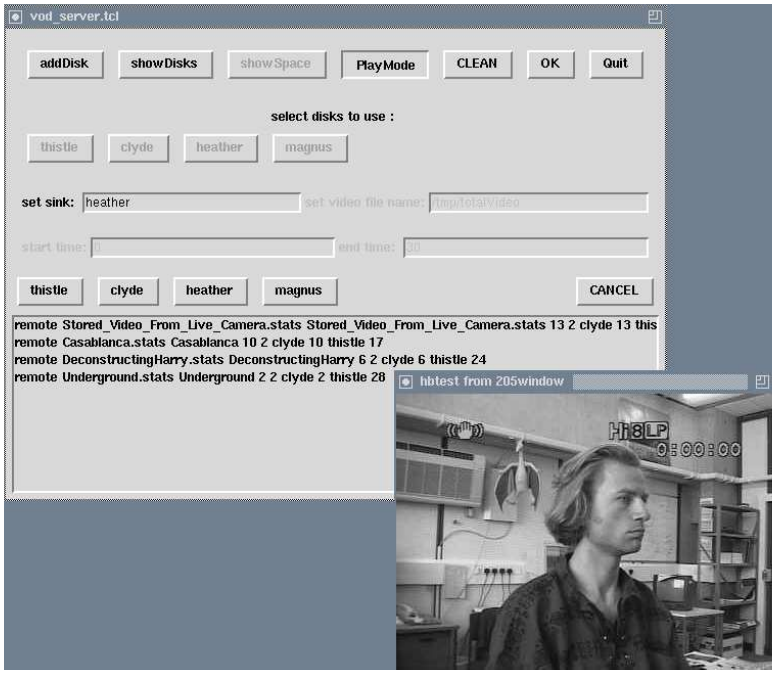


Figure 5.7: BigDisk playback mode (with a snapshot of the video).

In playback mode, shown in Figure 5.7, users select a video to play, on any of the disks in their system. When the video is selected, the DVS queries a small video database to find out which segments make up this video and where they are stored, and starts playing them in the right order and at the appropriate times.

5.3.3 Implementation of the DVS

Acting as a client of the Sandman and using the *connection sequence* and *connection pattern* types described in Section 5.1.3.2, the DVS is able to provide the necessary guarantees. The operation of the BigDisk backend upon a request from the DVS client is summarised in Figure 5.8. Because of the availability of the connection *sequence* and *pattern* operations, BigDisk itself can be very simple. Furthermore, this functionality could never be provided by the current signalling standards such as UNI [**UNI4.0:94**]. The rest of the DVS consists of a small distributed database to manage the video segments and a storage server that is CORBA aware and can be prompted to play or record video. The control part of the DVS consists of less than 800 lines of Tcl.

IMPLEMENTATION OF BIGDISK OPERATIONS		
PLAYBACK	a)	make an advance reservation for a connection sequence involving the appropriate disks for the appropriate (segment-)times and the sink for the entire duration of the video;
	b)	if successful, return the reservation identifier to the DVS (which enables the DVS to change the reservation);
	c)	at t(start), set up the first connection (allocating the appropriate amount of bandwidth) and call back the sink and the first source disk server in the sequence (essentially telling them to start receiving and sending, respectively);
	d)	at the end of each segment, flip the connection to the next source and call its callback function (also: tear down the dangling connection to the previous source and notify it of the fact) and so on until all sources have finished;
	e)	at that point, free all resources, notify the currently active sink and the currently active source and remove all state from the system.
RECORDING	a)	reserve a connection pattern with one source (e.g. the t.v. program you want to record) and n sinks (the disks on which you want to spread this video);
	b)	if succesful, return the reservation identifier and wait until the program starts;
	c)	set up the connection and notify the source and the first sink;
	d)	at the end of the pre-reserved time of segment 1, flip the connection to the next sink, clean up dangling connections, and repeat this process until all segments are stored;
	e)	free all resources, remove all state.

Figure 5.8: BigDisk implementation of recording and playback

5.3.4 Latency

Latency is an important aspect of BigDisk. Latency control is crucial to ensure the smooth playback of all segments. *Initial latency* is defined as the time it takes before the first video data starts arriving at the sink, measured from the time the connection request was issued. The *follow-up latency* is the time between the arrival at the sink of the last cell of one source and the first cell of the next during a source handoff in playback mode. Initial latency is not a problem in video playback, as clients generally don't mind when a video starts a few milliseconds or even seconds later. Reservation in advance provides control over the lower bound of the initial latency, as it can be specified that playout shouldn't start until a specific time t_{start}. At t_{start}, the latency incurred is lower (compared to traditional control architectures), because the request has already been sent to the Sandman and CAC and resource allocation have already been done.

This leaves the follow-up latency. The follow-up latency means that if one simply were to play the data immediately when it arrives at the sink, there would be glitches in the playout at the time of handoffs. The glitches have a length of the follow-up latency. To avoid them, a small buffer is placed at the sink, aimed at absorbing the latency (as well as possible jitter). The question is: how small can this buffer be? In the case of BigDisk, it is adequate to use a buffer that is simply 'big enough'. In other words, buffering a few seconds of video, will be plenty to absorb all follow-up latency and all jitter (at the expense of the initial latency). Although adequate, this is not a very general solution. Therefore, an experimental version of Sandman was built, which is capable of giving probabilistic upper (and lower) bounds on the latency. An outline of this approach is given in appendix B.

5.3.5 Segment replication

A BigDisk client expects the server to meet its QoS requirements. What if BigDisk finds that the network cannot provide the desired QoS? There are two options: (1) reject the request, (2) try and *enable* the network to accommodate the request. BigDisk attempts the latter. The reasons for not accepting a request can vary greatly. It may be that the advance reservation fails because the network's capacity is already fully reserved for the desired interval. This is an example of a rejection due to other traffic in the network. It may also be that a request would be rejected because the storage server is simply too many hops away to meet the QoS requirements, *regardless* of the presence or absence of other traffic. This request could never be satisfied.

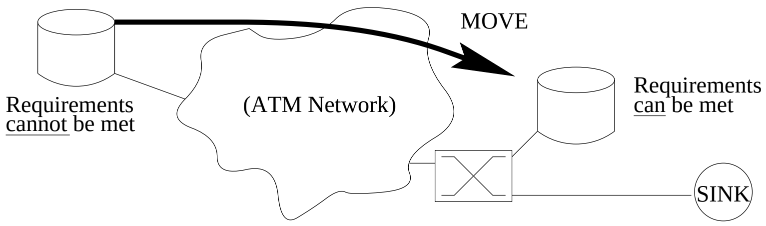


Figure 5.9: Moved segment is able to meet QoS requirements

Both problems can be addressed if the segment stored on the problematic file server is moved to a location from where the required QoS can be provided, e.g. to a disk on the same switch as the sink, as in Figure 5.9. Suppose that segment i in a connection sequence is the segment that is causing the problems (i.e. the call to this disk would be rejected because the QoS requirements can't be met). Define t_i as the time that the play-out of segment i should start. Observing that we have from $t = now$ until $t = t_i$ to move things around in the network in such a way that at t_i we will be able to play segment i, the following admission control policy is implemented:

1. If the request can be admitted by the network *as is*, i.e. without shuffling seg-

ments around, admit the request.

2. Else, if segment i can be replicated on another disk from where the QoS can be provided (before t_i), admit the request[8].

3. If not, reject the request.

Segments are automatically replicated on a lightly loaded or better-located disk F_{light}, if this enables BigDisk to meet the required guarantees. The play-out will then take place from F_{light}. All this is transparent to the client. A simple simulation with Poisson distributed request arrivals and exponentially distributed video popularity, suggested that this policy increases the acceptance ratio significantly [**Bos97**].

5.3.6 First segment replication (FSR)

The segment replication policy of the previous section yields lower latency than would be possible if the original placement of segments was used. This can be exploited in a different way as well. Consider a system with a large number of video files. Although replication is often proposed to improve availability and latency for these essentially write-only files, it makes no sense to replicate even a subset of these videos due to their enormous sizes.

A solution for this problem is to replicate only the first (relatively small) segment of each video over many network nodes, while making progressively fewer copies of the following segments, e.g. by keeping only a single replica of the last few segments. Also, the first segments will be copied 'further', i.e. to more remote sites, while the later segments are more bound to the local network. This scheme will be called first segment replication (FSR). FSR is illustrated in Figure 5.10. The initial latency to the video file in FSR will be low (there is always a first segment nearby) and the time to play this first segment can be used to move the remaining segments close enough to satisfy the QoS requirements. For this purpose, the user interface shown in Figure 5.6 has an extra option which enables users to replicate specific segments on other disks. The replication of the first segment over many nodes is essentially a form of cacheing.

Potentially, these techniques could also be used to implement prefetching for continuous media. Consider a Web page with a large number of links to very large video files. It makes no sense to prefetch all the data for all links that a user might follow, but it may be possible to prefetch the first segment of each these files. This allows users to quickly browse through a large number of videos without incurring large latencies each time a new video button is clicked.

[8] Currently, only file servers on the same switch as the sink are considered as destinations.

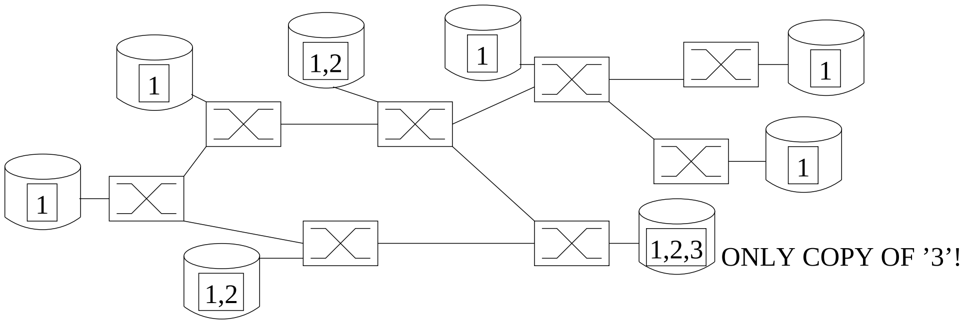


Figure 5.10: The first of three segments of video is heavily replicated

5.4 Adding granules to the Sandman

In the previous sections, the Sandman was discussed in terms of its basic functionality. Next, it is shown how DLAs can be loaded into the heart of the control architecture. Examples will be given of how this allows for functionality that would be difficult or impossible to implement otherwise, thus supporting the thesis that elastic control eases the development and introduction of new and innovative network control.

5.4.1 Introduction

Resource management in networks involves first and foremost reservation and allocation of resources to specific applications, for example in the form of connections. It will be shown how DLAs constitute a new paradigm in resource management and control. The generic nature of high-level primitives prevent applications from exploiting application-specific knowledge. For example, consider the application description in Figure 5.11. The active source changes according to an application-specific algorithm. The application may want to leave all connections to and from switch S in place, so all that is needed to change sources is changing the switch connection in S. High-level end-to-end primitives are incapable of exploiting this. This section presents a solution for this problem.

5.4.2 Recursively partitioning networks using netlets

The infeasibility of a one-size-fits-all solution applies also to the control architecture itself. Therefore, the idea of switchlets is extended into the control architecture itself by partioning virtual network resources and enabling individual applications to specify their own policies for reserving and allocating these resources. In this way, one can really speak about *open control*: flexible control that is not dictated by any one organisation, vendor or network operator. For this, the Sandman currently supports the following basic operations:

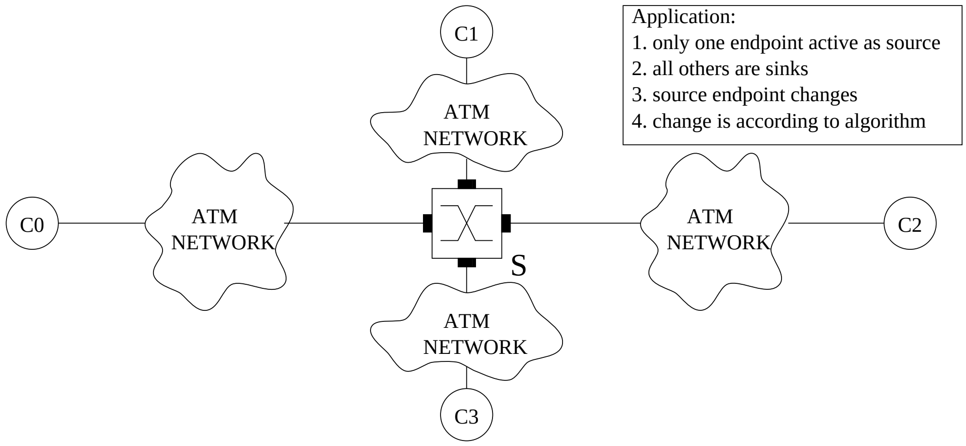


Figure 5.11: Application-specific knowledge

1. *Unicast, multicast, connection sequence* and *connection pattern.*
2. *Information gathering.* This is a wide-ranging class of operations to discover certain things about the state of the network, the topology, routes, available bandwidth on switch ports, etc.
3. *Reservation of arbitrary sets of resources.* The reservation of arbitrary sets resources in the network by partitioning (and repartitioning) existing sets will be described shortly.
4. *Loading application-specific code.* Allowing users or applications to load their own code into the control architecture allows them to exploit application-specific knowledge at a very low-level.

The first category contains the most commonly used operations. They are called the *primary* operations. All other operations mentioned above are called *secondary* operations. The *primary* operations are expected to be sufficiently expressive for a large number of applications. Some applications, however, have very specific needs so, in order not to restrict them, it is proposed to give these applications a set of resources which is theirs to use as they please (i.e. without any connections imposed on them). This is also useful for certain network management tasks. For example, it has been suggested in [**Schill97**] to partition resources in the network, so that immediate reservations are shielded from advance reservations and *vice versa.* For this purpose, Sandman allows one to make (advance) reservations for what are called *netlets*, i.e. partitions of a larger virtual network.

5.4.2.1 Netlets

Netlets consist of an arbitrary set of resources within the encompassing virtual network. For example, for switch port resources, a *netlet element* is specified which consists of the switch name, the port number, the direction (i.e. in or out), the number of channels (e.g. VCIs in ATM) and the amount of bandwidth shared by these channels. The netlet elements need not be adjacent as one netlet may consist of multiple unconnected sub-partitions (see Figure 5.12). Netlets resemble virtual networks of switchlets, but are more light-weight. For example, it will be shown how a netlet could simply be a single connection of a new user-defined type. Also, netlet partitions *need not be* hard partitions; they could be "logical" only (and used for instance for admission control).

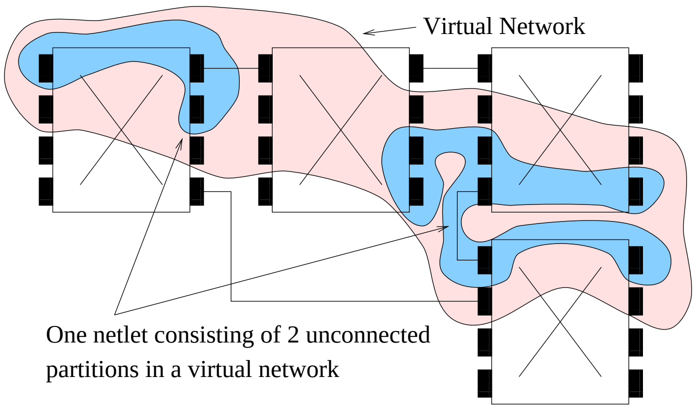


Figure 5.12: Netlet in virtual network

Netlets can be created recursively. In other words, it is possible to create netlets in netlets, which enables applications to repartition network resources in an unrestricted manner. In fact, the encompassing virtual network of Section 2.2 itself can be thought of as a netlet: the so-called *level-0* netlet[9]. Repartitioning network resources extends the idea of switchlets into the control architecture. There are a number of control and management advantages that result from doing this, including:

1. *Policing differentiation.* Level-0 netlets (virtual networks of switchlets) must be policed, because misbehaviour in one level-0 netlet N should not affect connections in any of the other level-0 netlets. Given level-0 policing, however, we can decide not to police at a higher-level netlet, because even if connections in the netlet misbehave, the problems will be limited to N only and not propagate to the outside world.

   It is now possible to differentiate the policing policy in the network, e.g.: given that there is hard (in-band) level-0 policing, we can decide to police specific higher-level netlets only very loosely (e.g. by periodically taking measurements from switches to see if they have exceeded their allocated bandwidths) and certain other netlets not at all. In fact, the *looseness* may vary from netlet to netlet. In other words, netlets are light-weight virtual networks (in this sense, the relation between a higher-level netlet

[9] As a convention, each time a netlet is repartitioned, the level number increases by one.

and a lower virtual network is similar to that between a thread and a process).

2. *Interoperability differentiation.* Applications that own netlets may have different requirements regarding the interoperability between control architectures. Interoperability will be discussed in detail in Chapter 6.

3. *Partitioning.* Using netlets, it is easy to separate a control architecture's immediate and future reservations as proposed in [**Schill97**] (as well as any other type of connection or reservation).

4. *Extended functionality.* A netlet can be used to implement new control architecture functions whose scope is limited to a particular netlet. For example, it will be shown how new application-specific connection types can be introduced. This is discussed in more detail in Section 5.4.3. Also, in a control architecture it may be desirable to implement functions that are not (and should not be) provided by the netbuilder. For example, one may want to change the resource allocation to groups of applications at very short time scales (making calls to the netbuilder expensive).

5. *Application grouping.* In a virtual network, there may be more control over resources than in the network-at-large, because potentially the behaviour of applications within the virtual network is known, while this is not true at a global scale. For example, if it is known that a set of applications will never be active at the same time, this allows one to assign a logical portion of the resources to these applications that is as large as the maximum of the resource requirements of these applications (as opposed to the sum of the resource requirements).

At the start of the reservation interval, the netlet resources are allocated to the requesting application. There are simple operations which the client can use to set up and tear down end-to-end connections in netlets. At the end of the interval, the control architecture automatically tears down all connections belonging to the netlet and releases the corresponding resources. However, since the resources of a netlet are said to *belong* to a specific application (and nobody else), such an application should be able to manipulate these resources in *any* way it wants to, not just by setting up connections between endpoints. For this, control at a finer level of granularity than end-to-end connections is needed. For example, applications should be allowed to set up a (possibly multicast) connection across an individual switch from a specific input (port, VPI, VCI) to a specific output (port, VPI, VCI). This allows applications to build their own connection types and setup mechanisms. Such low-level operations will be called *tertiary* operations.

5.4.2.2 Taxonomy of virtual networks

Netlets were defined as light-weight virtual networks. Section 5.4.2.1 explained in what way netlets are light-weight. Conceptually, however, there is little to distinguish a higher-level netlet from a level-0 virtual network, except in the entities that know of their existence. This suggests a more complete classification of virtual networks as shown in Figure 5.13. In the figure *"Restricted"* indicates whether or not the virtual networks are policed (e.g. whether there is a mechanism that ensures that virtual networks never use more bandwidth than was reserved for them). The next three columns indicate whether the virtual networks exist at *netbuilder*, *switch divider* and *control architecture* level. The typename is a simple abbreviation of the characteristics of the virtual network. Not all combinations are expected to be useful.

VN = Virtual Network
NB = NetBuilder
SD = Switch Divider
CA = Control Architecture

TYPE NAME	Restricted	Known by NB	Known by SD	Known by CA	EXAMPLE
R-NBSDCA	✓	✓	✓	✓	Top-level VN: created by NB, protected + policed by SD, controlled by CA
U-NBSDCA		✓	✓	✓	Top-level VN as above, without policing (dangeous option when bandwidth is limited)
R-SDCA	✓		✓	✓	Netlet which has requested the SD to police its connections
U-SDCA			✓	✓	Same as above, without policing (usefulness doubtful)
R-CA	✓			✓	Netlet with out-of-band policing (e.g. by extended traffic servers)
U-CA				✓	Netlet only known to CA: very light-weight

Figure 5.13: A classification of virtual networks

All virtual networks are known by a control architecture (otherwise they can not be controlled). The virtual networks described in Section 2.2.3 are of the R-NBSDCA category (or U-NBSDCA if bandwidth is unlimited). Higher-level netlets fall in the bottom four categories. We speak of U-CA networks, if the netlets only exist at a logical level in the control architecture. In this case, the netlet is generally used for CAC in an environment of well-behaved applications. This is the most light-weight of virtual networks. R-CA is more restrictive than this. The switch divider is still not aware of the fact that a switchlet has been repartitioned, but there is *a posteriori* policing. The policing could be implemented by out-of-band monitors that periodically measure the amount of data sent on connections belonging to the netlet. If the monitor detects a violation of the traffic contract (i.e. the reservation that was made) for this netlet it alerts the control architecture. Out-of-band policing is discussed in Section 5.4.2.3. In principle, hybrid forms are also possible. For example, it may be that a particular virtual network is mostly U-CA, except for one or two switch ports which are to be controlled in the R-CA manner. Whenever in this dissertation the term *netlet* is used, a higher-level netlet of the U-CA or R-CA category is meant unless explicitly stated otherwise. Whenever the term *virtual network* is used, this generally refers to a level-0 netlet of the R-NBSDCA category. The usefulness of U-SDCA netlets is questionable, and R-SDCA is probably also redundant. Although it may be useful to allow the switch divider to repartition the resources on a switch and police (and protect) each of the partitions separately, there seems no clear advantage in bypassing the netbuilder while doing so. Furthermore, it makes the model simpler to request the netbuilder to repartition the resources.

5.4.2.3 Policing by traffic servers

It was shown that networks can be recursively repartitioned into netlets, creating a hierarchy of resource partitions, the root of which is the physical network. Moreover, it was suggested that policies such as policing are a property of the individual levels in the hierarchy, i.e. of netlets. Level-0 requires strict partitioning and hence, in-band policing, but higher level netlets could differentiate the policing policy, e.g. to provide *a posteriori* policing via measurements. It will now be shown how this was achieved in the Sandman.

Recall that the CAC algorithm was enhanced with *traffic servers* that periodically measure the traffic on connections that belong to the control architecture for EBW estimation. The next step is to enhance the traffic servers to do policing as well. At the same time, it is important to recognise that this is just one way in which traffic servers can be extended; there may be many others. For this reason, the traffic server was extended with a Sandbox and a small module that allows authorised DLAs to access specific measurements. Next, DLAs were added to monitor and police individual netlets at different time scales[10]. For example, one netlet was policed in one second intervals, another at five second intervals, etc. This is illustrated in Figure 5.14. Recently, something resembling this monitoring function of the traffic server was described in [**Huard98**], where it is known as a *transport monitor*.

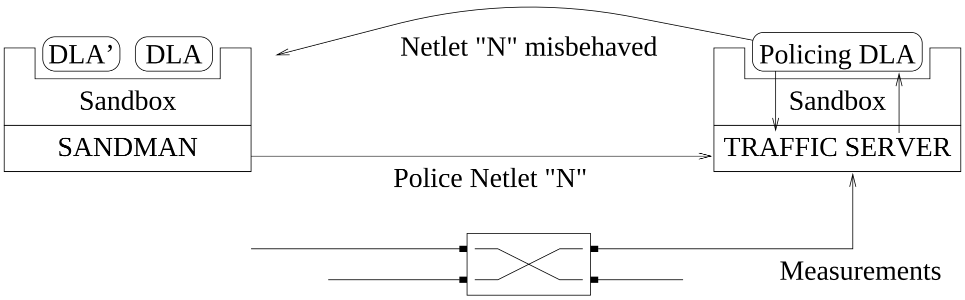


Figure 5.14: Slow policing by DLA

5.4.3 Loading application-specific code

One problem with giving fine-grain control over network resources, e.g. with netlets, is that because of the distributed nature of the interaction between

[10]The policing was implemented as follows: each netlet was assigned a total amount of bandwidth B for its policing interval. If the total traffic during an interval exceeded B, the control architecture was alerted.

client and Sandman, it may take a long time to do simple things such as setting up a connection across a large number of switches (each low-level operation travels across the network). An elegant solution is to enable the application to pass its own management and control policy into the control architecture and have it interact locally with the low-level control operations. The interaction takes place within the same address space, making it very fast. In other words, applications are enabled to program the network using DLAs.

5.4.3.1 Code and available operations

To allow clients to load code into the network, a Sandbox is implemented in the heart of the control architecture. Sandboxes are only instantiated in the connection manager and not in all possible components. However, the DLAs in these Sandboxes are given *controlled* access over the other components via the tertiary operations contained in what is called the Sandman module.

As shown in Figure 5.15, a special load operation in the secondary interface allows applications to specify a policy to run in the Sandbox, as well as a time t_{start}, when the application wants it to be scheduled for the first time[11]. At start of day, the Sandman has two interfaces: one for the primary operations and one for the secondary operations. The tertiary interface is not publicly accessible. The usual Sandbox SUFI enables remote applications (e.g. the parent application) to communicate with the DLAs. Operations available to the DLAs include the primary, secondary, and the low-level tertiary operations. This allows DLAs to extend the core functionality of the Sandman arbitrarily.

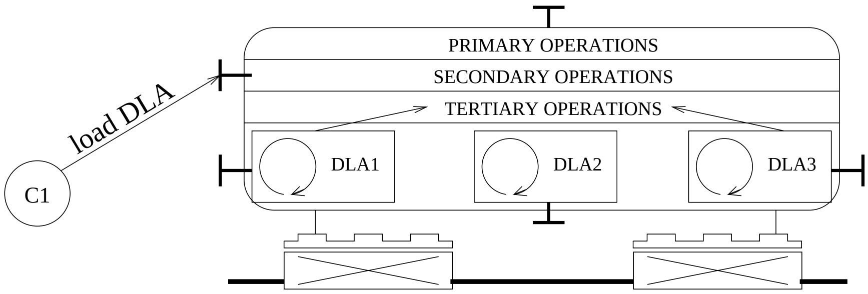


Figure 5.15: Application code injected in the network

5.4.3.2 Combining resources with policies

With netlets and DLAs, it is possible to associate application-specific behaviour with (sets of) netlets. This creates something akin to a *connection closure* [**Rooney98**]: a set of resources together with a policy that manipulates these resources. In addition to the normal Sandbox capabilities, the Sandman hands

[11]Restrictions may be placed on how many DLAs are allowed to run in the control architecture as well as on how long they are allowed to run.

the DLA a simple *netlet capability* that corresponds to the resources in the netlet. DLAs must present the capability in order to manipulate the resources. The netlet can be extremely light-weight and only correspond to a new type of connection (e.g. an application-specific multipoint-to-multipoint connection). In that case, DLA can export an interface for the new connection type, so that all external clients with the appropriate capability can make use of the extension. On the other hand, a netlet can also represent something more like a virtual network of switchlets (complete with policing).

The point is that the approach of netlets controlled by DLAs is much more general than the connection closures of [**Rooney98**]. It allows recursive repartitioning of the resources, dynamic changes and extensions to the policies and the overall control architecture functionality, and also provides a more complete security and access control model in the way of the Sandbox design. The DLA itself is also able to create new netlets (if need be recursively), which it can control itself as separate light-weight networks, or associate with DLAs other than itself. Experience with the loadable code feature of the Sandman has shown that it is extremely useful in prototyping and testing.

5.4.4 Applications, manipulations and reservations

It was shown that it is possible to extend the functionality of the control architecture with application-specific policies and to partition resources recursively. This section uses elastic functionality to implement truly *arbitrary* advance reservations. Arbitrary reservations may be periodical and changing, as shown in Figure 5.16.

ORGANISATION X REQUESTS THE FOLLOWING RESOURCE RESERVATION POLICY:

1. Reserve a netlet N every day from 9am until 7pm
2. Except on Sundays when N should be reserved from 10am till 4pm
3. Except when that Sunday falls on May 1 in which case the reservation should be for 24 hours
4. The bandwidth reserved on N should be B
5. If, after 3 months, it turns out that N's utilisation was never more than 70%:
   update the capacity to reserve: B := 0.9B
6. Repeat (5) every three months

Figure 5.16: Example of a reservation profile

Simple reservations (such as a reservation for a specific connection tomorrow from 9am to 5pm), which are traditionally used in advance reservations, are not sufficiently expressive to deal with this example. Due to the long time scale it is not possible to make a separate reservation for every single day the netlet is needed because the number of entries in the reservation schedules would explode. However, as shown in Figure 5.16, the policy is easily expressible in a simple algorithm. So again, the solution adopted for Sandman is to use DLAs.

5.4.4.1 Admission Control

In this section a distinction is made between an application's *resource manipulation behaviour* and its *resource reservation behaviour*. Resource manipulation behaviour is defined by the application's actions and operations on resources under the implicit assumption that the resources are available for it to use. This type of behaviour includes allocating resources to connections, freeing a connection's resources, etc. In resource manipulation behaviour, applications generally don't worry about the availability of the resources. Resource reservation behaviour on the other hand concerns itself solely with ensuring that certain resources are available at certain times. In general, it does not care what the resources are used for (or whether they are used at all). Figure 5.17 illustrates the separation between the two types of behaviour. Note that many operations encapsulate both resource manipulation and resource reservation behaviour. An example is the operation to make an advance reservation for an end-to-end connection. This will both reserve the resources and manipulate them when the connection is set up.

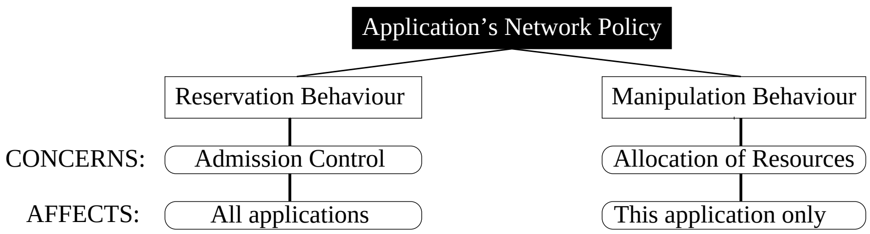


Figure 5.17: Separating resource reservation and resource manipulation behaviour

An important difference between the two types of behaviour is that manipulation behaviour does not affect any other applications, whereas reservation behaviour does. For example, setting up a connection for application 1 has no effects on application 2 as it does not change the state of the resources application 2 expects to control. Reserving resources on the other hand, affects all other applications in the system as it prevents them access to these resources during the interval specified. Reservation behaviour is essentially an input to the CAC algorithm, whereas manipulation behaviour is not.

So far, it was shown that DLAs and netlets give total freedom in the resource manipulation behaviour. Applications can load their code into the network and let it set up connections and manipulate resources in any way they see fit. The reservation behaviour, however, is still more or less fixed. Clients are able to reserve connections, or netlets, for specific time intervals. This type of reservation behaviour, where all bookings are entered in the reservation schedules, is called *static*. All advance-reservation control architectures to date offer static reservation.

However, resource reservation behaviour can be opened up as well. This is done by allowing applications to load code into the control architecture that is capable of directly influencing the CAC decision. This type of reservation behaviour will be called *dynamic*. Recognising that it is essentially a CAC problem that we are addressing, the DLAs used in this context will be called CAC DLAs. The Sandman offers an interface with an operation that is capable of installing CAC DLAs for a particular port of a switch[12]. Figure 5.18 shows the two types of DLA running in the heart of the control architecture.

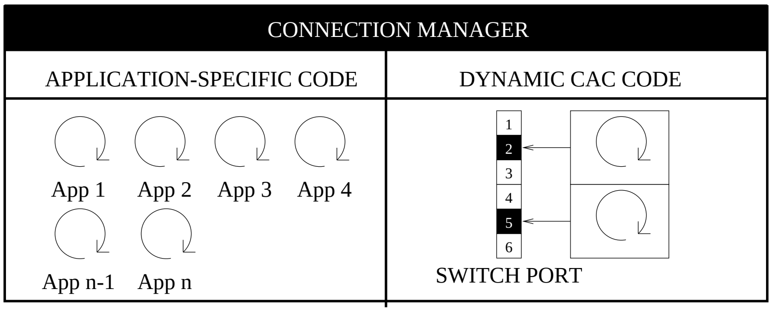


Figure 5.18: Two types of DLA running in the connection manager

5.4.4.2 Temporary reservations at CAC time

The loadable CAC code can influence the CAC decision as follows. Whenever the Sandman needs to make a CAC decision for a switch port (i.e. it tries to make a reservation R for a certain amount of bandwidth for a time interval $[T_a, T_b]$), it checks whether a CAC DLA was installed, and if so, executes it. The loadable CAC code receives from the control architecture the time interval $[T_a, T_b]$ corresponding to reservation R. It then makes *temporary reservations* according to its own reservation policy for this time interval. After making all the reservations it wants to make for this interval, it simply returns. The CAC algorithm now makes its decision based on all reservations currently in the schedules, including the temporary ones. The temporary reservations disappear automatically when the CAC procedure returns. An example of the CAC process is given in appendix C.

The loadable CAC code is called each time a CAC decision is made for this port in this virtual network. This adds some overhead to the CAC procedure. In a commercial situation applications requiring such flexibility in their resource reservation behaviour might be charged a premium rate for their reservations, while applications that don't need it are charged lower rates.

[12]The granularity of switch port is an implementation detail. It could be changed to providing DLAs per switch, or per small number of VCIs on a particular port, if so desired.

5.4.4.3 Combining application-specific CAC programs

CAC DLAs are different in nature from the DLAs in Section 5.4.3 in that it affects the CAC for all applications. The CAC DLA describes reservation behaviour which cannot be specified using higher-level operations. This introduces the following problem: how do we decide whether or not the reservation behaviour as specified in the CAC DLA of one application will never clash with the reservation behaviour specified in the CAC DLA of another application? In other words, how do we decide that two CAC DLAs are mutually *feasible*? If Turing-complete languages are used for the CAC DLAs, this is a very hard problem to solve. A simple solution is to restrict the CAC DLAs. For example, the DLAs may be restricted to a limited execution time (e.g. the code has to make its reservations within 5 ms, or within 100 instructions), and/or it may be restricted in the expressiveness of the language. Instead of Tcl or Java, the CAC DLAs could use a language that is easy to check on the fly, e.g. a limited number of statements of the form: "`if <expression> then <temporary_reservation_list>`" (with `<expression>` and the amount to reserve simple functions of the time).

5.5 Noman

The second type of control architecture in this dissertation is potentially much simpler and yet more flexible than the Sandman. Control architectures from the *Noman* family consist solely of dynamically loadable code. Noman is defined as a programmable control architecture *framework*. The Noman environment itself provides no control architecture functionality. Instead, it offers a simple API that allows DLAs to build their own control architecture. The Noman is centred around a Sandbox. The Noman control architectures are also called *dynamically loadable control architectures* (DLCAs). Noman represents a different approach from the Sandman. In the latter, elastic execution environment was added to an existing, full-grown, control architecture. In the former, the code in the elastic execution environment *is* the control architecture. In other words, Noman is an extreme case of elastic control architecture. The Noman Sandbox is extended with a *NetControl* module that allows it to: (1) create and destroy virtual networks, (2) add switchlets to the virtual networks, and (3) control the switchlets (e.g. set up and tear down connections).

5.5.1 Implementing different types of control architecture

The Noman is illustrated in Figure 5.19. Noman allows any type of control architecture model to be implemented. For example, a completely centralised Noman control architecture was implemented, where one DLCA in one Sandbox controls the entire network, as well as a completely distributed Noman control architecture whereby every component (router, CAC, host manager, etc) of the DLCA is a separate granule in a different Sandbox on a different host. Noman also allows for what may be called 'open vertical integration', whereby applications and control architecture are integrated. The integration is 'open' in the sense that functionality of both control architecture and application may be added on the fly by any party with the appropriate access rights.

Even though interpreted code is generally slower than optimised native code, this may allow for a speedup, as there is virtually no communication overhead between application and control architecture. The functionality is quite flexible: a complete control architecture, simpler than, but comparable to the *Hollowman* control architecture in [**Rooney97**], took less than 700 lines of DLA code in a Noman control architecture[13]. Like the Hollowman, it provides end-to-end connection types (both point-to-point and multicast), that can be set up by the source, the sink, or some third party. The structure of the Hollowman is roughly that of Figure 3.5 and this structure is also followed by the Noman control architecture. Where Hollowman supports its own trading service by which applications can register and deregister service offers, the Noman control architecture uses the active trader of Section 4.7 for this purpose. The Hollowman offers limited programmibility in the form of *connection closures*, which allows clients to provide a control policy to be associate with simple set of resources. The Noman control architecture offers complete programmibility using the normal Sandbox-SUFI procedures. Failure handling and performance optimisations, however, are dealt with more extensively in the Hollowman.

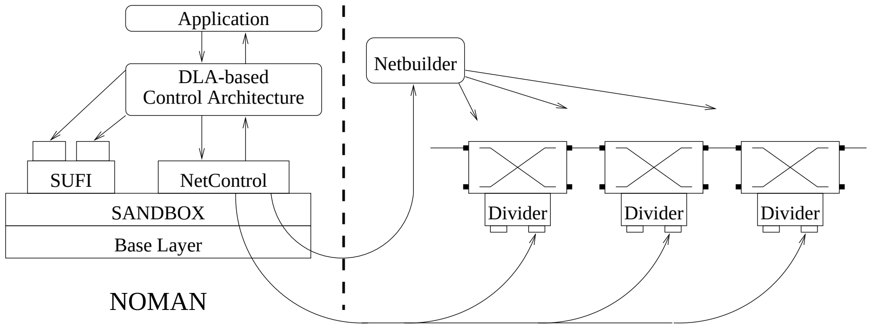


Figure 5.19: Noman control: switchlets are created, controlled and destroyed by DLAs

5.5.2 Example: Noman-based bandwidth speculation

Noman has been used in bandwidth management experiments aimed at using a speculative free-market model for optimal resource allocation. Resource allocation based on a free-market model is an active research topic, advocated for example by [**Miller96**]. In the model, applications are charged for the bandwidth they use, where the price of bandwidth fluctuates according to supply and demand (e.g. bandwidth will be cheaper late at night than during office

[13]using a *Gobi* implementation of a Tcl Sandbox

hours). In the Noman implementation, a control architecture called the *Noman-auction*, allows clients to make two types of reservations: (1) fixed bandwidth (FB), which, if granted, guarantees the amount of bandwidth that an application receives, regardless of cost, and (2) fixed price (FC), which keeps the price per time unit constant, while adjusting the amount of bandwidth provided to the application[14].

Clients are charged for the bandwidth they use, according to the current price[15]. So if a connection is active for three time units, while the price is p for two time units and $3p$ for one time unit, the amount charged to the client will be $5p$. Bills are paid to the network operator. However, there is a way for clients to make money out of the network as well. It is based on the possibility of bandwidth speculation. The idea is that a client A can offer for sale part of the bandwidth that was reserved in its FB reservations. Next, if client B needs some FB bandwidth, but none is currently available (i.e. it is all used up by other FB reservations), it may *buy* the bandwidth directly from other clients. The transaction involves selling the bandwidth to B (to be used at the normal market rate), for a fixed amount that is paid only once, the beneficiary of which will be A. This is implemented via the accounts that both applications have with the network operator. The amount is debited to B's account and credited to A's account.

This allows for bandwidth speculation, where bandwidth is bought when it's cheap (e.g. in the middle of the night), only to be sold when there is a shortage. The cost of the speculation is the market rate that is charged for the duration of the reservation. The profit is made by selling bandwidth at a price that is higher than this cost. Clients are allowed to load their own market policies in the Noman-auction (i.e. policies that handle the clients' bandwidth management). These DLAs are able to monitor the current prices and resource availability, and act by buying or selling bandwidth whenever they think appropriate. [**Steiglitz96**] suggests that allowing speculators in the market has a stabilising effect on price (and hence resource utilisation).

5.6 Related work

In this chapter many issues to do with control architectures were condensed into the design of two types of control architectures. This section briefly discusses some other approaches.

[14]It is arranged that a fraction of the bandwidth is always available for FC reservations.

[15]Whether the price should be set in real money or in the form of credits is beyond the scope of this dissertation.

5.6.1 Control architectures

5.6.1.1 Xunet 2

Xunet 2 [**Kalmanek97**] on which work was started as early as 1989, was one of the first wide area prototype implementations of ATM. Many of the experiences and recommendations derived from Xunet 2 made their way into standards defined by the various standard bodies. Xunet includes hardware design, interoperability and distributed control and management. An abstract switch model is designed which hides details of the hardware, so that switch hardware and control software can develop independently. Unlike *Ariel*, however, the switch control interface was not designed to be a generic switch control interface and the switches were custom-built.

A simple signalling protocol was designed to establish, maintain and clear ATM connections. Most of the network control software is designed as a client-server based system based on a DPE. For convenience, the switch controller was run off-switch, in a general purpose workstation. There is one switch controller per switch and one switch per switch controller. Like the basic Sandman control architecture (and unlike standard protocols such as Q.2931 [**ITU-T94**]), Xunet provides unidirectional connections. The bootstrap network in Xunet consists of a set of pre-established PVCs over which an implementation of IP is active.

5.6.1.2 ATMF and ITU-T standards

Standards developed by ITU-T and the ATM Forum deal with most levels of network control, including user-network signalling (defined in the user-network-interface, or UNI) and internetwork control and routeing (defined in network-network-interfaces, or NNIs). In general, the ATM Forum concerns itself with private networks, while ITU-T standards pertain more to the public network.

5.6.1.2.1 Signalling System Number 7 SS7, a clear channel signalling specification published by the ITU-T, is the prevalent signalling system in the modern public switched telecommunications networks (PSTN) [**ITU-T93**]. It defines procedures for set-up, management and clearing of calls between telephone users. For this purpose, SS7 employs telephone control messages that are exchanged between SS7 components that support end-user connections. The SS7 signalling data links are 64 kbit/s full duplex, digital transmission channels, dedicated to SS7 signalling. Figure 5.20 depicts a typical SS7 topology.

Subscriber lines are connected to the SS7 network through the service switching points (SSPs), which receive signals from the customer equipment and perform call processing on the user's behalf. SSPs are the source and destination for SS7 messages, initiating SS7 message exchanges either with another SSP or with a signalling transfer point (STP). STPs are responsible for trans-

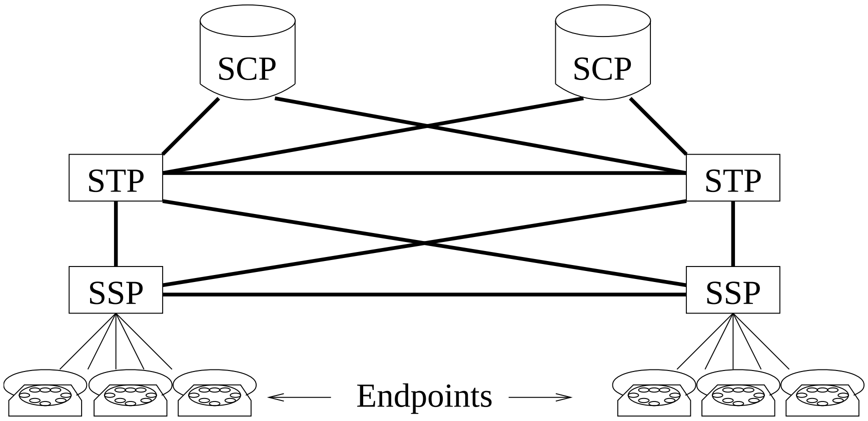


Figure 5.20: SS7 topology

lating the SS7 messages and routeing them between nodes and databases. In other words, they are switches that relay between SSPs, STPs and service control points (SCPs). SCPs contain software and databases for the management of the calls. This is explained in Section 3.8.1, on intelligent networks. SS7 is a layered suite of protocols that uses, for the lowest layers, the message transfer part (MTP) layers 1-3, while employing protocols such as the ISDN user part (ISUP) and the transactions capabilities part (TCAP) for the higher layers. TCAP is an application-layer protocol which is used for such things as the exchange of non-circuit related data (e.g. routeing information for 0800 numbers in intelligent networks). As shown in Section 5.6.1.2.3, SS7 is used by ITU-T and the ATM Forum as the model for signalling.

5.6.1.2.2 The user-network interface (UNI) The UNI [**UNI4.0:94**] connects an end system to an access switch. The UNI specifies the signalling functionality that is offered to the user. Therefore, users cannot benefit from innovations in functionality until this has been incorporated in a standard and the standard has been introduced in the network. UNI also defines a number of service categories, such as: variable bit rate (VBR), constant bit rate (CBR), available bit rate (ABR) and unspecified bit rate (UBR, or best-effort). Finally, it defines the protocol messages, information elements, timers and procedures to establish, maintain and tear down connections. Signalling is based on the Q.2931 protocol. Resource reservation is carried out at connection setup time. All point-to-point connections are bidirectional, while multicast connections are unidirectional.

5.6.1.2.3 Network-network interface (NNI) NNIs are divided into two categories: the private network-network interface or P-NNI [**PNNI1.0:96**] and the public network-network interface. Of these, the ATM Forum's private network-network interface is intended for private networks and contains interfaces both for the exchange of routeing information and for connection control

[**PNNI94**]. B-ISUP, the public NNI developed by the ITU-T serves as a demarcation point between two public networks. It is based on a modified version of SS7 and uses preassigned VCIs for signalling [**Onvural95**]. The ATM Forum version of the public NNI, known as the broadband inter-carrier interface (B-ICI), gives ATM carrier networks interoperability, assisting them in transporting different services across each other [**BICI94**]. The B-ICI specification includes the physical layer, the ATM layer and service-specific functions above the ATM layer. Also addressed are such issues as traffic management, network performance and operations and management specifications. B-ICI only exists at the boundaries of networks. It specifies a wide range of physical layers over which the ATM layer can run and also particular adaptation layers for common inter-carrier services such as Frame Relay.

Unlike B-ICI, one of P-NNI's goals is to make switches from different vendors work together. Accordingly, it can be used as an inter-network solution as well as for intra-network control. In routeing, P-NNI uses information aggregation, where groups of network nodes are collapsed into a single aggregation node, to achieve scalability. For end-to-end signalling, UNI signalling is mapped onto NNI signalling inside the network. At the remote endpoint the NNI signalling message is mapped back to the corresponding UNI signalling message. The mapping is straightforward, since P-NNI and UNI signalling are very similar. P-NNI includes a link-state routeing protocol that is similar to OSPF (open shortest path first) in that extensive flooding is used for sending topology and utilisation updates.

5.6.1.3 OPENET: enhanced PNNI

OPENET aims at providing a common, portable, open and high-performance control platform based on enhancements of the P-NNI standard [**Cidon95**]. In this sense, it is not a radically new approach to network control. Directly at odds with this dissertation, OPENET embraces wholeheartedly the idea that one particular control architecture, e.g. enhanced P-NNI, should become the universal ATM control platform. The changes that are proposed to P-NNI are: (1) native ATM switching for dissemination of utilisation updates, (2) lightweight connection setup and teardown, and (3) a rich signalling infrastructure which enables the development of augmented services. The 'openness' in OPENET consists of providing open APIs at various levels, as well as a highly modular design to ensure easy module replacement.

It is shown how using P-NNI as both an inter-domain and intra-domain solution, presents severe limitations. For example, current P-NNI is based on extensive flooding of topology and utilisation information, using fairly large packets of up to 300 bytes for every link state update. This either places a heavy burden on network control processors or results in limitation of speed and accuracy of information updates, if adopted as a universal solution. Similar drawbacks arise from P-NNI's use of UNI signalling (over the complex reliable data-link protocol known as Service Specific Connection Oriented Protocol), which yields low throughput and high latency. Although the focus is on P-NNI alone, OPENET advocates open control, observing that:

> P-NNI lacks a publicized open interface, which makes it difficult for switch vendors to incorporate extensions and advanced features that they believe should be provided [**Cidon95**].

OPENET offers unrestricted multipoint-to-point and multipoint-to-multipoint functionality. Multipoint-to-point connections take the form of *source trees*, which multiplex cells of different sources at a single destination which receives all of them with the same VCI value. This implies that an ATM adaptation layer (AAL) that relies on VPI/VCI values alone (such as AAL5) cannot be applied. The tree prescribes the use of unicell messages or of an AAL that is able to deal with frame interleaving. Multipoint-to-multipoint in OPENET works similarly and the same multiplexing drawbacks hold.

5.6.1.4 Xbind

In *xbind* [**Lazar96a**] a framework is presented for the creation, deployment and management of multimedia services on ATM-based broadband networks. Attention is given to end-to-end QoS guarantees. Services are created by interconnecting (binding) abstract representations that model the local states of network resources including links and switches. The implementation of the binding architecture, which provides an open programmable environment, is essentially a toolkit for building programmable ATM networks. 'Programmable' in xbind means that the architecture supports functional APIs for developing useful services, which should be 'high-level' enough to allow the service specification and creation process via a high-level programming language. Xbind service abstractions also contain a so-called 'Service Programming' interface, which allows customisation or modification to be made on the algorithmic component of a service or service instance. It is unclear to what extent this functionality has been implemented, how deep the customisability reaches and how access control is executed.

The binding architecture consists of an organised collection of interfaces, called the Binding Interface Base (BIB). BIB interfaces provide an open and uniform access to the abstract representation of the networking resources. Service creation consists of binding network resources to create a service using the corresponding binding algorithms. Xbind provides extensive modelling of the various entities related to network control via the layered Extended Reference Model (XRM). The model seems to include abstractions for almost any entity and relationship in the network, from the physical resources and their operations, to 'logical' resources such as VCIs and elaborate network control and management services.

QoS is modelled explicitly in the architecture via a set of abstractions that characterise the multiplexing capacity of networking and multimedia resources under QoS requirements. These abstractions are derived from the physical layer of the XRM in the form of *multimedia networking capacity graphs.* These graphs consist of *schedulable regions.* For each QoS class that is supported, the schedulable region indicates how many bindings of this class can be supported by the resource, given that some possible combination of all other QoS classes has been scheduled already. It is not clear how schedulable regions are derived from the resources.

5.6.1.5 The Genesis Kernel

In [**Campbell99b**], the recent Genesis project is discussed which enables recursive partitioning of network resources in a way that strongly resembles netlets[16]. The partitions of network nodes, called "routelets", are created by a *parent* control program, but controlled by an independent *child* control program. However, the Genesis model is much more complex than that of netlets. Also, since Genesis is related to xbind and uses the BIB to link together components that make up the control code, routelets suffer from the same shortcomings as xbind. It is necessary to specify in advance which resource abstractions and service model the new control code will use.

More serious is that the hierarchy of nested routelets is reflected in the datapath. Each routelet consist of a myriad of components and services and part of this comprises the routelet's transport module, which resides on the datapath. The transport module includes a virtual input port, a forwarding engine (that processes the data) and a virtual output port. An incoming packet is classified, for example, as belonging to a specific (higher-level) routelet and demultiplexed to the routelet's transport module accordingly. It arrives at the routelet's virtual input port, is processed in the forwarding engine and sent out on the virtual output port. However, it is not sent to the next node yet. It first arrives at the virtual input port of its parent routelet. Here the process is repeated and so on, until it arrives at the root routelet's input port. Here it is processed for the last time and finally forwarded to the next node in the network. A similar repartitioning model, focused on active networks, is described in [**Brunner99**].

This is very inefficient as the data is touched, not just once, but potentially many times. This is an example of multiplexing at many layers. Contrast this with the "flat" model of netlets. Here netlets form simple partitions of lower-level netlets that can be controlled in arbitrary ways, e.g. by setting up connections from one endpoint to another. The data simply flows through these connections and is not touched at all. Multiplexing happens once and at the lowest possible level.

[16] The first implementation of netlets was described in [**Bos98a**].

5.6.1.6 TINA

An initiative of the telecommunications industry, the Telecommunication Information Networking Architecture (TINA) [**Nilson95**], aims at developing an open control architecture for telecommunications in broadband services, whilst preserving compatibility with existing ITU-T standards. For control it proposes to use a DPE based on CORBA [**Pavon98**]. TINA extends ideas of the Information Networking Architecture (INA) [**Rubin94**]. Similarities exist between TINA and the *Haboob*, but TINA's goals are much more ambitious. It appears to try and develop a model that is able to incorporate all entities, parties, and services, relevant to telecommunication as well as their relationships. Rigorous partitioning and layering techniques are applied at numerous levels to separate amongst others:

- functionality from engineering,
- data management functionality and also human interaction functionality from other application-level functionality,
- equipment (network element) control functions (e.g. port connection/disconnection, collection of resource usage data for a switch port, etc.) from service-related functions (e.g. connection setup and release, billing and routeing),
- resource management from signalling.

Using extensive modelling, TINA approaches the problem of network control and management from a very high level. Instead of giving control over the individual resources to the entities eventually using them, i.e. individual applications, TINA assumes that complete models of the relevant components and their interactions are first developed, which then represent a service to be used by applications. For network management, TINA uses the managing/managed object paradigm and management protocols. A managed object is an abstraction of a resource for management purposes. Managing a resource therefore, involves representing the resource by a managed object and performing management operations on the managed object. TINA complies with the layers of the Telecommunication Management Network (TMN) model [**Raman99**, **Sidor98**].

TINA services consist of service components which communicate using the DPE. The service components include *user* and *terminal agents* that manage profiles of terminals and users (and perform negotiation/session setup and endpoint session setup) respectively. Also, defining a session as the activity of originating or terminating a call, service components include *service session managers* which manage sessions at user level and *communication session managers* which manage sessions at terminal level and provide network connections. The TINA network information model consists of a number of levels of abstraction of network resources. Implementations of individual resources are described as simple elements, while at the highest level of abstraction the connectivity between media flows is defined in terms of connection graphs. TINA's all-encompassing approach based on high-level modelling is very ambitious. Where other, simpler approaches are being pushed into mainstream network control, TINA, for all its complexity and extensive modelling, has not "yet produced aggressive market deployments" [**Decina97**]. More critical still is [**Redlich98**]:

> The most dramatic appearance of distributed object technology in networking is, however, in the Telecommunications Information Networking Architecture (TINA). [...] Unfortunately, in order to present a complete structure TINA has become extraordinarily complex. [...] The desire for clean separation is so strong that there are more levels than network and service designers may wish to implement.

5.6.1.7 UNITE

In ATMF signalling, every flow suffers the overhead of end-to-end connection setup (i.e. a full round-trip delay), even connections for best-effort traffic. The fundamental contribution of UNITE is the separation of connection setup from QoS control [**Hjalmtysson97b**]. Data is allowed to follow the connection setup message almost immediately (i.e. before the end-to-end connection has been set up). This way the round-trip delay associated with connection setup is eliminated. Things are optimised even further by using a single cell for connection setup signalling messages. This results in light-weight signalling, with very fast setup times for what is expected to be the most common case: best-effort traffic. QoS negotiation, if needed, is done later. An additional advantage of this approach is that different QoS (re-)negotiation schemes can be easily incorporated. Routeing is based on very coarse QoS classes.

5.6.1.8 IP Switching

IP Switching [**Newman97**] is one of many approaches that use the fast switching technology offered by ATM to leverage IP performance. It combines IP routeing with ATM switching by turning long-lived IP flows into ATM connections. The IP switch controllers (augmented IP routers) have the task of identifying flows in the IP traffic passing through the node. After flow identification, the down stream switch controller tells the upstream node that from now it on it should stop sending the IP packets corresponding to the flow on the default VC for IP traffic, providing it with a new, dedicated VC instead. For this purpose, it employs the Ipsilon Flow Management Protocol (IFMP). IFMP allows adjacent switch controllers to exchange information about the mapping of IP flows onto ATM connections. IP Switching uses GSMP to set up and tear down the actual connections on the switch.

5.6.1.9 Hollowman

The Hollowman open control architecture described in [**Rooney98**] strongly influenced the development of Sandman. Its component model (which in turn resembles somewhat the model proposed in [**Rubin94**]) formed the basis of that of Sandman. Many of the components (e.g. the connection manager) were rewritten from scratch, while others were incorporated almost unchanged (e.g. naming and addressing). Hollowman provides operations for immediate setup of best-effort unicast and multicast connections. The only resources that are controlled are the VPI/VCI spaces on the switch and the Service Access Points (SAPs) on the hosts. Hollowman further allows a limited amount of programmability by enabling applications to load (Java) code controlling the resources of a connection, into the network. For interoperability, Hollowman employs a mechanism very similar to the simple hop-by-hop solution of Section 6.3. For more details of Hollowman, see also Section 5.5.

5.6.2 Reservations in advance

Reserving resources in advance is a very active field in network control. The related work presented here represents a cross-section of research in this area.

5.6.2.1 RSVP

The resource reservation protocol RSVP [**Zhang93**] was not designed for reservations in advance. With minor modifications, however, RSVP's functionality could be extended for this purpose [**Degermark95**, **Schill97**]. RSVP shows that concepts that were part of ATM from the outset are now finding their way into IP as well. RSVP offers receiver-initiated reservation with support for heterogeneous receivers, different reservation styles and route changes. The nature of the reservations is defined by *soft state*, i.e. state that disappears if nodes do not periodically receives messages. In RSVP periodic messages flow both downstream (from source to sinks) in the form of path messages and upstream in the form of reservation messages. Path messages update path state, which contains information about the incoming link upstream and the outgoing links downstream from the node. The reservation messages update the reservation state associated with the path state. This includes the amount of resources reserved, the packet filters, the reservation style, the sender of the reservation message, etc. Section 5.1.5.2 describes why soft state is not very suitable for advance reservations.

5.6.2.2 NAFUR

Negotiation approach with future reservation (NAFUR) is the name given to a QoS negotiation mechanism which allows clients to specify the desired QoS for a connection for a specific time interval in the future [**Hafid97**]. If the requested resources are not available in the requested interval, NAFUR offers counter proposals which consist of lower quality service in the requested interval, as well as an indication of the time at which the requested quality would be available.

5.6.2.3 Distributed Advance Reservation in Tenet

In [**Ferrari97**] an advance reservation service is discussed within the context of the Tenet Real-Time Protocol Suite 2, which is being developed for multi-party communication. The reservations are implemented as interval tables which represent resource schedules. This is similar to the way resource schedules are stored in the Sandman. The tables are stored in a distributed fashion: each server handles the interval tables for its own resources. The distributed state, as acknowledged by the authors, makes fault recovery more complicated. There is partitioning between immediate reservations and more efficient advance reservations.

5.6.2.4 ReRA

A good model for advance reservation is given in [**Wolf95**] and [**Wolf97**]. Here, as in [**Ferrari97**], immediate reservations are separated from advance reservations and reservations of finite durations are separated from reservations of infinite duration. It is concluded that advance reservations are more efficient than immediate reservations. One of the most difficult topics in resource reservation in advance, according to [**Wolf95**], is the occurrence of failures. Failures that occur after resource reservation but before resource usage are distinguished from failures during resource usage.

5.6.2.5 Advance reservation in heterogeneous networks

In [**Schill97**] the case for advance reservations from the viewpoint of efficiency is made once again. It presents implementations of RSVP and IP version 6 over ATM. After that, design issues are discussed for an appropriate solution, including: signalling, admission control and duration of flows. Finally an extension to RSVP for reservation in advance is outlined. Advance reservation in RSVP enables users to make advance reservations across heterogeneous networks.

5.6.3 Measurement-based admission control

Much current research on MBAC, including the admission control scheme of Section 5.2, is based on a branch of statistics known as large deviation theory (LDT) [**Weiss95**]. An example of a mathematical derivation of admission criteria is given in [**Gibbens97**]. In [**Tse97**], an MBAC scheme is analysed and by simulation it is shown that the MBAC is a promising approach to admission control. The goal for these solutions is to be able to estimate the EBW as accurately as possible, without requiring a precise description of the sources *a priori*, as was needed in the original effective bandwidth (and Gaussian approximation) schemes [**Guerin91**]. LDT represents a rigorous basis for such CAC mechanisms, as it allows one to give probabilistic guarantees, e.g. about CLR, delay, etc. It should be pointed out, however, that early work on MBAC did not employ such rigorous mathematical techniques. [**Jamin95**], for example, proposes to measure at fixed intervals how much bandwidth a connection had used in that interval. This value, X say, is then stored in a circular buffer. The EBW estimate was taken to be the maximum value of X in the buffer.

In IP, an MBAC scheme which also deals with advance reservations is described in [**Degermark95**]. It is based on the simple estimates of [**Jamin95**]. This is rather different in scope and objectives from the mechanism described in Section 5.2, in that it focuses on bounding delay for *predictive services* in the Internet, while the MBAC mechanism in the Sandman aims to manage bandwidth in a general way. A comparison of various MBAC schemes for controlled-load services is given in [**Jamin97**].

In [**van der Merwe98**], estimates of effective bandwidth are used for the management of resources of virtual networks. As part of this, the effective bandwidth estimates are used for a "lower-level" admission control, namely to decide whether or not new virtual networks can be added to the system. The server that calculates the effective bandwidth of connections and groups of connections is integrated with the switch divider.

5.6.4 Continuous media file servers

Most continuous media file servers are concerned with the problem of providing guaranteed data rate and possibly even end-to-end QoS guarantees as described in [**Anderson92**, **Jardetzky92**, **Chaney95**, **Holton95**, **Neufeld95**, **Bosch96**] and many others. In [**Dan95**] an interesting segment replication scheme is proposed for load-balancing purposes. Here, overloaded file servers replicate some of their segments on lightly loaded servers which will serve all future requests for that particular segment. [**Federighi94**] proposes a hierarchical video distribution, whereby the most popular films are highly available on fast and expensive storage servers such as disks, less popular videos reside on slower CD-ROM and titles that are rarely selected are stored on very slow but cheap tape archives. This is a distribution mechanism that is quite different and indeed orthogonal to first-segment replication (FSR) as described in Section 5.3.6. Both policies could be employed in the same system.

Few papers recognise the problems associated with replication schemes when dealing with potentially very large continuous media files. A notable exception is [**Lougher94**], where it is suggested that video files are encoded using multi-resolution encoding algorithms and where only the lower-resolution versions are replicated (propagated) to remote sites, while the higher-resolution versions are limited to the local network only. Again, this policy should be easy to combine with FSR to achieve even greater scalability.

5.7 Summary

Two types of elastic control architecture were discussed. The first, known as Sandman, offers very advanced functionality in the form of advance reservations, measurement-based admission control, resource repartitioning, etc., while also allowing clients to specify application-specific policies regarding network control. It was shown how a simple distributed video server was built on top of the Sandman. A different approach was taken in the Noman type of control architecture. Noman is not a control architecture in its own right, but it offers an API that allows DLAs to build their own control architecture (all in dynamically loadable code).

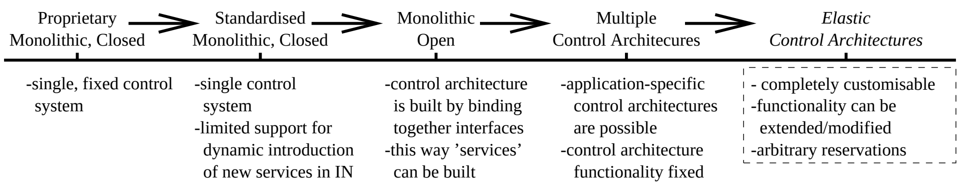


Figure 5.21: The control architecture in the evolution of network control

Figure 5.21 shows how the open, customisable control architecture fits in the evolution of network control. So far, control architectures have only been considered in isolation. However, when multiple control architectures are active simultaneously, interoperability between different control architectures becomes an important issue. This is not a trivial problem, as two control architectures may have very different control primitives that must somehow be mapped onto each other in order to let the two domains cooperate. The next chapter will therefore look at the problem of interoperability.

Chapter 6

Interoperability

In an environment where multiple management and control architectures are active simultaneously, interoperability between these control architectures becomes an important issue. For example, it should be possible to access services that are located in another domain, controlled by an incompatible control architecture. It is remarked in [**Decina97**] that:

> Interoperability is the most important requirement of the infrastructure in an information and communication environment that is rapidly evolving towards heterogeneity.

This chapter presents a solution to the problem of interoperability that is different from other such solutions in that it takes into account that primitives in one control architecture often do not map nicely onto those of another and moreover, that there may be multiple possible mappings from one set of primitives to another. The solution is only possible because of the elasticity of the *Haboob* components and thus supports the thesis that elastic network control eases the introduction of new functionality in the network. Given interoperability, it is shown how network control and management policies can be established that span multiple control and management domains.

6.1 Introduction

In Chapter 5, it was shown how clients were allowed to load application-specific policies into the heart of the control architectures. Many applications, however, extend beyond the boundaries of a single control and management domain. A *control architecture domain* is defined as a set of switches controlled by a specific control architecture. It is important that these applications should be supported in a similar fashion. The challenge here is twofold.

Firstly, different types of control architecture should be enabled to interoperate. Ideally, this should be possible without degrading the functionality of two communicating feature-rich control architectures A and B, only because an interconnecting control architecture C, located between A and B, does not provide this rich functionality.

Secondly, clients should be allowed to take their policies across the domain boundaries. This way clients are able to exploit their specific knowledge about the nature of their applications throughout the network, i.e. to install application-specific policies spanning multiple control domains. One problem here is that, although applications may be assumed to have knowledge about the local domain (e.g. about the topology), no such knowledge can be assumed for remote control architecture domains. Some support for enabling such policies to find out information about their new environments is therefore required.

6.2 Assumptions and configuration

Sandman will be used to demonstrate the principles and as a proof-of-concept implementation. It should be stressed, however, that the issues and solutions are not specific to Sandman. They apply to interoperation between any two control architecture domains. Rather than a discussion of a particular implementation of interoperability, this chapter is meant to be a description of a principles for advanced interoperability between control architectures that can be applied to the design and implementation of many different types of control architectures.

Consider Figure 6.1, which shows four different control architecture domains, three of which are controlled by instantiations of the Sandman, while the one in the middle is controlled by some other control architecture, e.g. P-NNI. This is called the CA_X domain. There are only four types of inter-domain interaction that are relevant:

1. both endpoints are in neighbouring Sandman domains;
2. the originating endpoint is in a Sandman domain while the other is in a neighbouring CA_X;
3. the other direction: the originator is in CA_X, while the destination is in the Sandman domain;
4. both endpoints lie in Sandman domain, which are separated by one or more CA_X domains.

For these four types of interaction, it is sufficient to consider only the cases where communication originates in Sandman-1 and CA_X. It is assumed that Sandman has only partial domain-level knowledge about the network topology

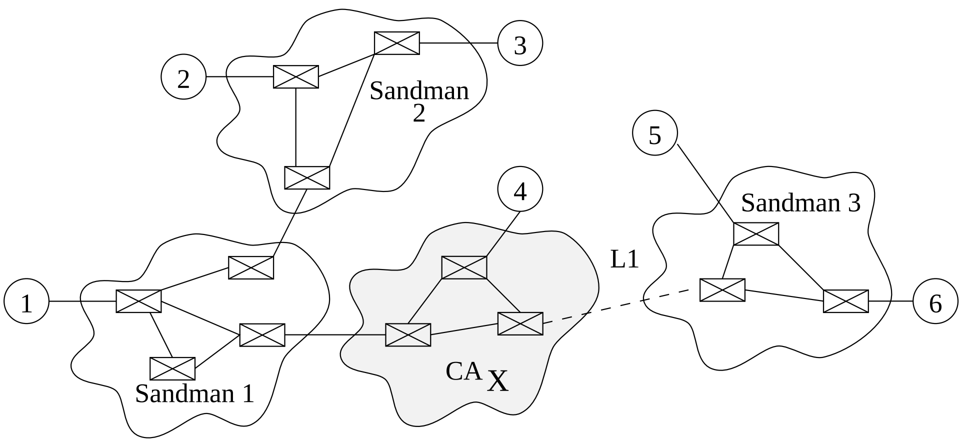


Figure 6.1: Multiple interconnected control architecture domains

This means that Sandman-1 knows, or is able to find out, that endpoint 2 is connected to Sandman-2, but not what the exact topology is within Sandman-2 (i.e. how many switches there are, what switch is connected to what switch, etc.). Similarly, it knows that endpoint 6 is connected to the network controlled by Sandman-3 and that it can be reached through CA_X. The dashed line $L1$ between the Sandman-3 and CA_X domains indicates that there might be other domains between Sandman-3 and CA_X of which Sandman-1 has no knowledge. When communication originates in CA_X, it is not even required that CA_X has partial knowledge. It will be shown that to CA_X, Sandman-1 behaves exactly like another instantiation of CA_X, so that it can use its own proprietary signalling to set up connections to endpoints in Sandman-1.

6.3 Simple interoperability between domains

As a start, a simple solution for interoperability between domains is discussed, based on the one proposed in [**Rooney98**]. The next step is to identify the problems associated with this solution and propose a new solution that overcomes these problems. The simple solution is to associate gateway code with a *pseudo-endpoint* that corresponds to the link connecting the two control architectures. A *pseudo-endpoint* is a *control gateway* that translates signalling messages from one control architecture into those of another. In Sandman domains it takes the role of an endpoint, while to a neighbouring CA_X domain, it may look like a native CA_X switch controller.

As shown in Chapter 5, when both endpoints lie in the same domain, the Sandman sets up a connection from one endpoint to another and then notifies the endpoints that the connection is in place. Things change if one (or more) of the endpoints lie outside the local domain. As an example, consider a point-to-point connection connecting a local endpoint A with a remote endpoint B. This is illustrated for two interoperating Sandman domains in Figure 6.2. When a request for such a connection arrives at the Sandman, the pseudo-endpoint C of the appropriate outgoing link automatically takes on the role of the remote endpoint B. In other words, whenever Sandman tries to set up a connection to a remote endpoint, it really sets up a connection within its own domain, to the pseudo-endpoint corresponding to the outgoing link and then notifies the pseudo-endpoint that the connection is in place (and which VPI/VCI values are associated with it).

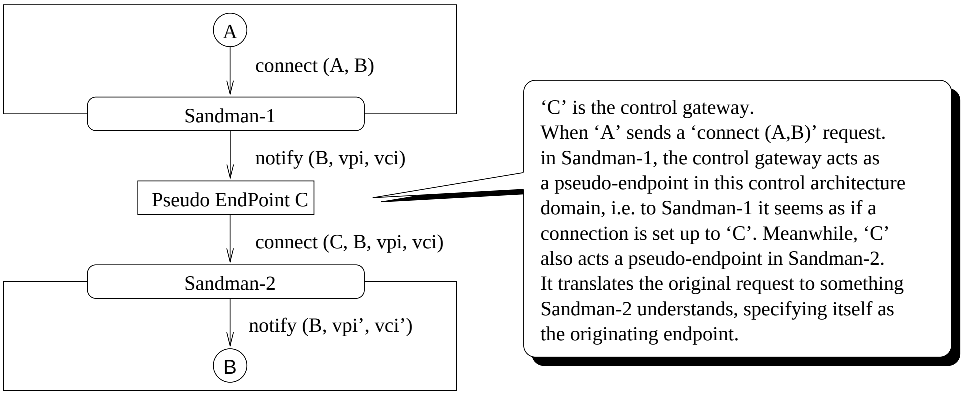


Figure 6.2: Pseudo-endpoints as control gateways

Upon receiving the notification, the pseudo-endpoint translates the setup request to whatever signalling protocol is used in the neighbouring domain (this includes translation of addresses if necessary, e.g. into UNI4.0's NSAP address format [**UNI4.0:94**]). If the neighbouring domain happens to be another Sandman domain, it simply repeats the connection request, this time assuming the role of endpoint A. If the neighbouring domain succeeds in setting up the rest of the connection, the pseudo-endpoint returns the boolean value *true* to the first Sandman. If not, all actions performed by Sandman-1 are rolled back also and the initiating endpoint is notified of the failure.

Connections from CA_X to the Sandman could be set up in the same way (if such a control gateway has been implemented in the CA_X domain). Alternatively, it is possible to let the Sandman offer the same sort of interface to the CA_X domain that would have been offered by another CA_X domain. This is illustrated in Figure 6.3. In this case, the CA_X domain cannot tell that it is actually communicating with a different type of control architecture. For example, many control architectures have well known signalling channels for setting up and tearing down connections, etc. ATMF UNI signalling uses a dedicated point-to-point signalling VC with VCI = 5 and VPI = 0. It is not difficult to direct this signalling channel to the control gateway which in turn translates the incoming signalling into Sandman control messages (this includes address translation, if necessary).

This solution covers all four cases of interoperability mentioned in Section 6.2. Henceforth, it will be called the *hop-by-hop solution* for interoperability because each control architecture only communicates with its immediate neigh-

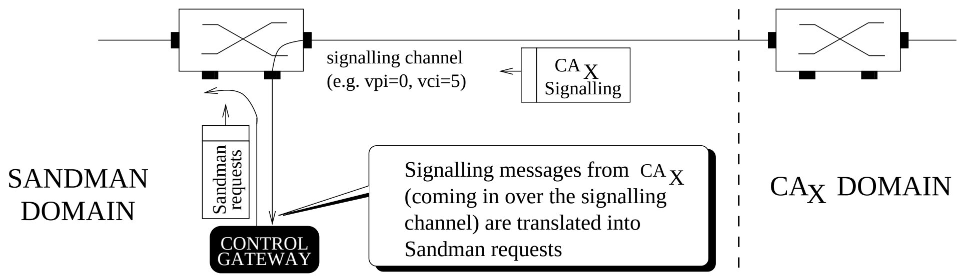


Figure 6.3: Control gateway translates signalling messages

bour, translating each control message from its own domain directly into that of the neighbouring control architecture.

6.4 Shortcomings of hop-by-hop solution

The simple hop-by-hop solution discussed in Section 6.3 provides very basic interoperability between multiple control architecture domains. The solution is attractive because of its simplicity but for the same reason limited in its usefulness.

The main problem is that the signalling gateways reduce all possible interconnection to the lowest common denominator in terms of control architecture functionality. Consider, for example, the case of interoperability between two Sandman domains, connected by a CA_X domain, such as between Sandman-1 and Sandman-3 in Figure 6.1. Although both Sandman control architectures support the use of future reservations (as discussed in Section 5.1), it is impossible to make use of this functionality in an inter-domain connection (assuming that CA_X does not support such service). This is because at the control gateway between Sandman-1 and CA_X, the future reservation request is translated into the type of setup request that CA_X understands, probably an immediate setup. After that it will never be 'promoted' to future reservation again. Instead, at the boundary between CA_X and Sandman-3, the CA_X immediate setup request is translated into a Sandman immediate setup request. In other words, any advanced functionality is reduced to the simplest common service on the path between Sandman-1 and Sandman-3. This will be called the problem of *functionality degradation*.

An additional problem is that the nature of the interoperation between two domains is fixed. This makes it hard to exploit application-specific knowledge. Taking again the example of future reservation, consider the case where endpoint 1 in Sandman-1 wants to reserve in advance for a connection from itself to endpoint 4 in CA_X. The Sandman domain first makes all local future reservations for an interval $[T_{start}, T_{end}]$ and then forwards the request to the CA_X domain via the control gateway.

The control gateway has to translate the request into control operations that CA_X understands. One option would be to simply allocate the resources (i.e. set up the connection) in the CA_X domain immediately and keep it in place, so that at least the future reservation is guaranteed. This is the right solution if the guarantees regarding the availability of resources in $[T_{start}, T_{end}]$ are important and the resources in CA_X are scarce. Alternatively, it may decide not to allocate any resources in CA_X at all and simply *try* to set up the connection when it is needed (i.e. at T_{start}). This may be the right solution if there is little risk of other applications using the required resources in the meantime. The point is that the gateway uses a static method of translating the requests. This method may be acceptable in certain cases, but not in others. This will be called the problem of *fixed interaction*. If the application itself were able to decide on the nature of the interoperation between two domains, then it could exploit application-specific knowledge that is impossible to support otherwise.

6.5 Sandman control channels and tunnels

The problem of Sandman functionality degradation is addressed first. Assuming that Sandman is the more feature-rich control architecture it can be observed that functionality degradation only occurs when multiple Sandman domains are on the paths between the endpoints and when these Sandman domains are separated by non-Sandman control architectures.

6.5.1 Inter-Sandman signalling

When multiple Sandman domains are involved, it is possible to implement a so-called inter-Sandman signalling channel (ISSC) between each two adjacent Sandman domains (which may be separated by a number of unknown control architecture domains). This is illustrated in Figure 6.4. As indicated in the figure, there is no need to dedicate a well-known VPI/VCI value to the signalling channel. The channel can be set up using the hop-by-hop inter-domain communication as described in Section 6.3 (lowest common denominator is good enough for setting up signalling channels). All intermediate domains simply pass on the Sandman control messages without understanding them or even looking at them (tunnelling). As usual, the ISSC finds its endpoints in the control gateways of both Sandman domains (i.e. the control gateways are the entities that signal to each other).

Next, if Sandman-1 wants to communicate with Sandman-2, it sets up a data connection between the two control architecture domains. Ostensibly, the data connections also find their endpoints in the pseudo-endpoints described in Section 6.3, so that the pseudo-endpoints (i.e. the control gateways) get notified when connections are set up, which allows them to handle these connections further (outside the local domain). The pseudo-endpoints take care of the

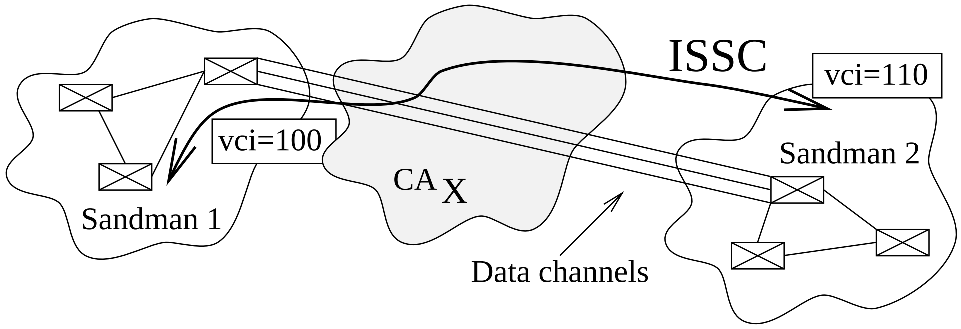


Figure 6.4: Interdomain signalling channel

administration and maintenance of these connections. Inside the two Sandman domains however, these inter-domain data channels can be connected in any way the control architectures want to. So the data channels are really data tunnels connecting two Sandman domains. The further connection of these data channels on the remote side is controlled by signalling over the ISSC[1]. It is possible to set up data channels in advance or leave them in place after a certain application is done with them. This will be called *tunnel cacheing*.

So to take up the example again of a reservation in advance for a connection from endpoint 1 in Sandman-1 and endpoint 6 in Sandman-3 (see Figure 6.1), this now becomes a matter of grabbing (or setting up) a data channel between the two Sandman domains and then making a local reservation in advance in Sandman-1 which gets transferred over the ISSC to Sandman-3 (because Sandman-1 recognises that the destination endpoint lies in Sandman-3 and therefore lets the control gateway towards the destination domain handle the reservation for the remote part). Sandman-3's control gateway picks up the reservation request and tries to make an advance reservation from link L1 to the eventual destination. If successful, it sends the boolean value *true* back to Sandman-1. If not, it returns *false* which indicates that the actions in Sandman-1 should also be rolled back. At the start of the reservation interval, the connections are set up locally on both sides and connected to the VPI/VCI of the data channel. This will be called the *local extension* of the connection.

6.5.2 Implementation

The way interdomain signalling has been implemented is illustrated in Figure 6.5. Control gateways run on general purpose workstations connected to the switches. As described above, a control gateway serves as the *pseudo-endpoint* of a switch port that connects the local domain to a neighbouring domain. Such a switch port will be defined as *gateway port*. By definition this is never the switch port to which the control gateway's own host machine is

[1]However, it is still not necessary for one Sandman domain to have precise knowledge of the topology of the remote domain: all routeing is local to the domains themselves.

connected.

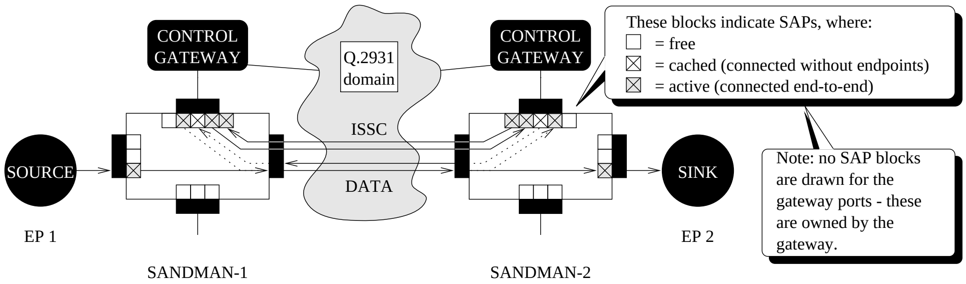


Figure 6.5: Interdomain signalling implementation

Each control gateway owns a number of slots representing *service access points* (SAPs) for the gateway port it manages. Each SAP allows for a single unidirectional connection. In other words, SAPs have a *direction* attribute which indicates whether the connection represents a *source* to the domain (data on the connection flows from the neighbouring domain *into* the local domain), or a sink (the direction of the data is *away* from the local domain). It should be noted that SAPs are not only allocated to control gateways, but also to every other endpoint connected to the network. For endpoints, the available SAPs determine how many connections can be set up simultaneously.

The control gateway is only unique in the sense that it also owns the gateway port's SAPs, i.e. SAPs for a switch port that is not the same as the one connecting it to the switch. This is indicated in the figure by the fact that there are no slots drawn for the gateway ports (instead, these are owned by the control gateway). The SAPs allocated to the control gateway are shared between the SAPs that are used for connections to and from the control gateway (e.g. ISSCs) and those that are used for connections between endpoints that cross the domain boundary (at the gateway port). For each unidirectional connection across the gateway port a single SAP slot is taken.

Gateway SAPs have four possible states, which are illustrated in Figure 6.6. If SAPs are *free*, they are not associated with any connection whatsoever. If the state is *reserved*, it means that the SAP is associated with a connection in progress. This is a temporary state. The connection can either be aborted (or fail), in which case the state returns to *free*, or succeed, in which case the state becomes either *cached* or *active* depending on whether the local extensions of the connection exist or not. Active connections are connected end-to-end, as far as the local control architecture domain is concerned. In reality, however, their endpoints may lie in remote domains beyond the scope of the local control architecture. So, an *end-to-end connection* in this context is defined as: a connection from one link at the edge of the network to another link at the edge of the network, where the links are connected either to real endpoints/hosts or to neighbouring domains. Whether or not the connection is really end-to-end, i.e. connecting endpoints, depends on the semantics of the control messages that are sent over the ISSC. Removing and adding local extensions changes a

SAP's state from *active* to *cached* and back, respectively. Both the *active* and the *cached* state reduce to *free* if the resources corresponding to the SAP are explicitly released.

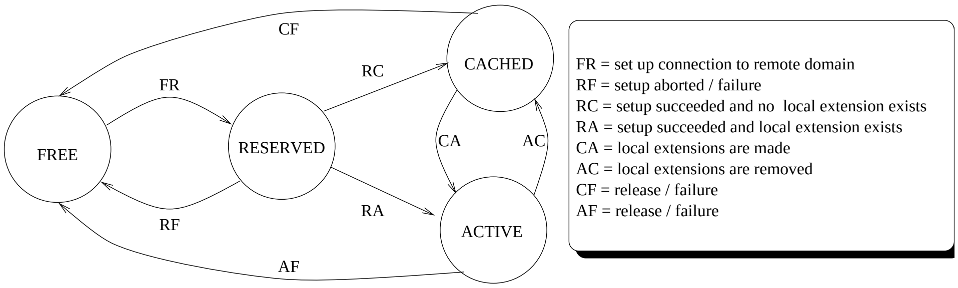


Figure 6.6: Gateway SAP finite state machine

The state transition diagram of Figure 6.6 is oversimplified in two ways. First, it ignores failures and errors. Currently all failure indications lead to a teardown of the connection, and a return to the *free* state. Second, all SAPs have an extra attribute indicating whether they have been reserved for some time interval in the future. It contains a pointer to the advance reservation schedule of the SAP. This allows for the possibility to make advance reservations for SAPs as well.

As an illustration, consider again Figure 6.5. For simplicity, all connections are assumed to be best-effort. In the figure, a bidirectional ISSC connects the two Sandman domains across a Q.2931 domain [**ITU-T94**]. This takes up two SAPs at the control gateways (one *source* and one *sink*). At some point, EP1 wants to set up a unicast connection from itself to the remote endpoint EP2. For this purpose, the control gateway has to set up a data channel between itself and the remote Sandman domain. Now suppose that, because the control gateway expects that a return channel will be needed shortly as well, and since the connections in Q.2931 are bidirectional anyway, the control gateway requests the Q.2931 domain to set up a bidirectional connection with the same metrics for the return channel as for the outgoing connection.

If successful, there are now two unidirectional data channels. Ostensibly, these data channels are connected to the pseudo-endpoints (indicated by the dashed lines). In reality, local extensions are made for the outgoing channel to connect EP1 with this connection at the gateway port, and on the other end, to EP2 via the gateway port in the second Sandman domain. This becomes an active connection. The return channel, however, is not used at the moment. It enters the *cached* state. Whenever a best-effort connection in the reverse direction is needed, this cached data channel (or: tunnel) can be used. All that is needed to complete the connection are the local extensions in the two Sandman domains.

There is now full interoperation between islands of Sandman domains, providing the full functionality of the control architecture, while being interconnected by simple connections that act as tunnels. It is not necessary to set up ISSCs from an originating Sandman domain to all other Sandman domains on a path between a source and destination. Instead, hop-by-hop interconnectivity can be used for this, albeit of a somewhat coarser granularity. Each hop is now a Sandman domain. Setting up connections end to end is done by sending the appropriate control message along the ISSCs from one Sandman hop to the next. Tunnelling of signalling messages has also been proposed as a solution to connect two private ATM networks via a public network [**Alles95**]. This is different from the ISSC solution in at least two ways:

1. it assumes the pre-existence of a private virtual path (PVP) connecting the private ATM network across the public network (this PVP is set up manually);
2. all traffic (both signalling and data) from private ATM network 1 to private ATM network 2 is sent through this PVP tunnel.

In contrast, the ISSC solution is not only capable of creating its own inter-domain signalling channels, but also of creating separate channels for individual data streams. This offers more perspective for offering QoS to individual connections.

6.6 Loadable interoperability

The ISSC solves the problem of *functionality degradation* between cooperating domains, but does not address the problem of *fixed interaction*. For example, if a future reservation is made for a connection between a Sandman-1 endpoint and a Sandman-3 endpoint, it was assumed that the data channel between the two domains was set up immediately. This may be the right solution in certain cases but not necessarily in others (as shown in Section 6.4). A connection's QoS metrics may be equally hard to translate, as QoS may have different meanings in different domains [**Cidon95**]. Furthermore, setting up a specific return channel with specific properties, may be useful in some cases, but not all.

This section proposes a simple solution to this problem. Essentially, applications are permitted to define their own pseudo-endpoints, if necessary with their own ISSC and data channels. For this purpose, the technique of dynamically loading code (in the form of DLA granules) is used. In other words, users are allowed to load up their own gateway code dynamically. Of course, some restrictions are necessary to avoid that an application-specific (and maybe even faulty) implementation of a control gateway becomes the only available option for *all* applications. This is solved by associating an application's control gateway DLA with a particular *netlet* (assuming the netlet contains the necessary resources on the gateway port). Then, whenever a connection to a remote endpoint is made, the application's own netlet control gateway is used to communicate with the neighbouring control architecture (as well as with the remote Sandman, using the netlet ISSC).

This allows applications to specify exactly the mapping between Sandman operations and the operations supported by the neighbouring domain. For example, a netlet gateway may decide not to map a future reservation onto an immediate connection in the neighbouring domain, but instead to wait until the start of the reservation interval with setting up the connection (for example, because it knows that there is plenty of bandwidth available in CA_X).

6.7 Global policies

Section 5.4 described how applications can inject application-specific code into their local connection manager. In Section 6.5 it was shown how neighbouring control architecture domains are able to interoperate. This is still not sufficient, however, for implementing truly global application-specific policies. In the interaction between domains so far, it was only possible to use the basic operations of the control architectures. In other words, functionality *degradation* across multiple domains has been prevented (where possible), but we still have not enabled applications to implement their own functionality *upgrades* across multiple domains.

For example, assume that two Sandman domains are interconnected, possibly via other domains, as illustrated in Figure 6.7. The nature of a simple application that spans the two Sandman domains may be such that at all times only one of the six endpoints shown in the figure is active as source, while all the others are active as sinks. Which endpoint should be source changes over time, according to an application-specific algorithm. Such multi-domain applications could be easily implemented if the application-specific code could be distributed over the two domains. This section shows a simple way of doing this.

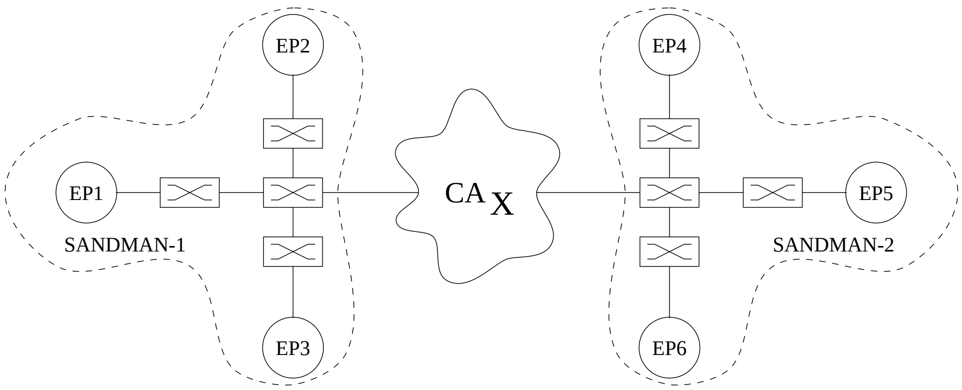


Figure 6.7: Multi-domain applications

6.7.1 Policy migration and replication

An interesting detail of the support for DLAs as discussed in Section 5.4 is that it allows the loadable code itself to inject granules in other control architectures. This follows from the fact that starting DLAs/granules is part of the Sandman's secondary interface and the secondary interface is publicly accessible. So if there are multiple Sandman control architectures in a large network, each is capable of sending DLAs across the wire which will then be run in the remote control architecture. In this way the DLA is able to migrate or replicate itself across a larger network. This is illustrated in Figure 6.8. The various incarnations of the DLA distributed over the network, as well as different DLAs, are able to communicate using the SUFI.

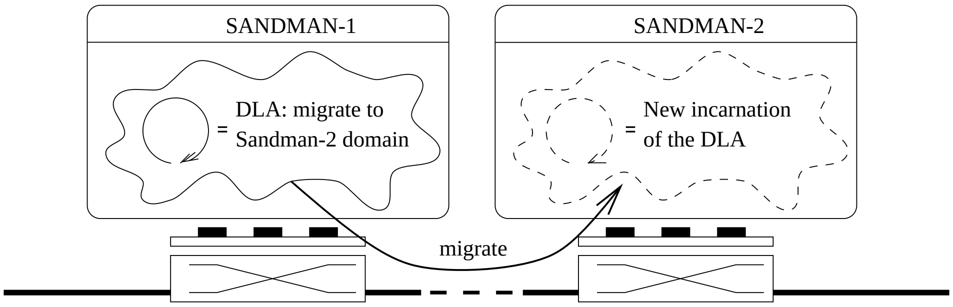


Figure 6.8: Policy migration

It can be argued that a network operator running a control architecture in a particular administrative domain may not want to permit DLAs from applications in different administrative domains to be loaded inside the heart of its control architecture. Nevertheless, there are advantages in doing precisely that and there is no intrinsic risk in doing so, provided the security issues described in Section 3.3.3 are addressed. But even if DLAs are not allowed to spread accross multiple administrative domains, this does not mean that they are not allowed to spread over multiple control architecture domains, as these are very different things. A control architecture domain only consists of an instantiation of the control architecture together with one or more switches which it controls. In traditional systems there is a one-to-one relationship between the switch controller and the switch. The Sandman control architecture is not so different from traditional control architectures, except that it gives network operators a *choice* of how many switches should be in a control architecture domain. This could be a single switch as in traditional (integrated) systems, or small clusters of three or four switches. This is illustrated in Figure 6.9. As a result, there will generally be more than one control architecture domain in a single administrative domain (which can be as large as a department or campus). Within the adminstrative domain, it may be perfectly permissible to have DLAs cross control architecture domain boundaries.

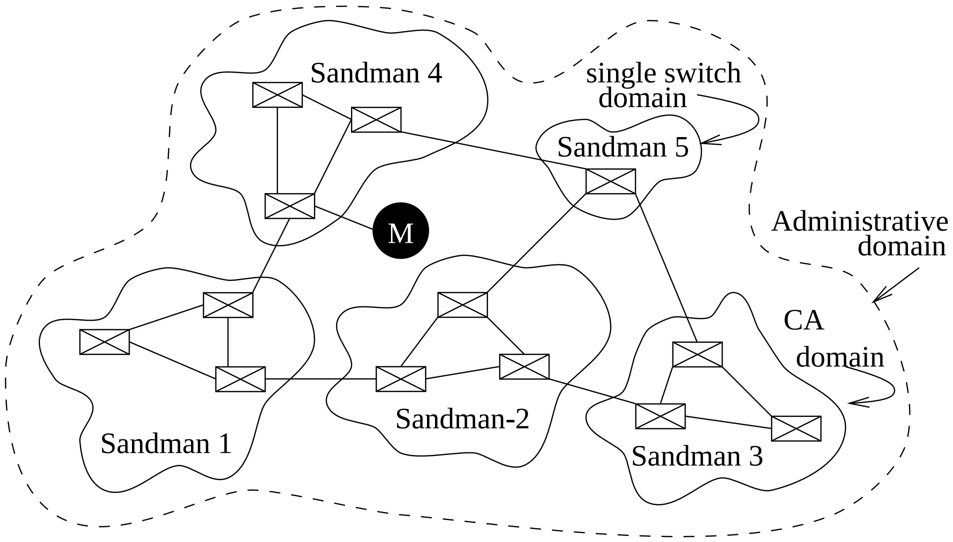


Figure 6.9: Administrative domain consisting of five control architectures

6.7.2 Environmental awareness

It may be assumed that a DLA has rather extensive knowledge about the domain where it was created. For example, the topology of the network may be known by the application injecting the DLA and hence programmed into it (hardcoded topology knowledge). This assumption, however, is no longer true when DLAs migrate through the larger network in order to implement global policies. A DLA may know that certain endpoints are connected to a particular control architecture domain, but it generally has no knowledge of the switches and interconnections on the paths between these endpoints. This makes it impossible to exercise low-level control over the resources in this domain (using netlets), unless the DLA has a way to learn about its new environment.

For this purpose, the Sandman's tertiary interface contains a number of operations that enable DLAs to acquire knowledge about the new domain. For example, one of these operations returns to the DLA a detailed description of a path between two endpoints (including switch names, port numbers, etc.). In this way, the DLA learns about the part of the control architecture domain that is of interest. And using this knowledge, the DLA can create netlets consisting of these paths, allowing it the low-level control that it may require to implement application specific policies in the new domain.

6.7.3 Example: mobile agents for mobile computing

To demonstrate the usefulness of global policies, consider the case of mobile computing. A client may have very specific knowledge about the route followed by, or the communication pattern associated with a mobile system M (see Figure 6.9). Assume that there are multiple control architecture domains and that the mobile system roams among these domains. Using the technique described in the previous sections, it is now easy to install the client's application-specific knowledge across the entire administrative domain. It is even possible to have a DLA 'follow' the mobile system as it travels from domain to domain (which means it would not burden those parts of the network that mobile system M is not even close to). The DLA can set up connections for the mobile system (ensuring no loops arise in the connection), work out optimal routes and also collect billing information. This is an example of application-specific control using roaming policies. In a different context something similar was proposed by [**Biswas97**]. As a demonstration, a very simple roaming mobile tracker was built which migrates from Sandman domain to Sandman domain according to some policy. Since no real mobile computing systems were available to demonstrate the roaming policy, the mobiles were emulated using roaming agents.

6.8 Testing interoperability in practice

The implementation of the interoperability design uses the Sandman as the feature-rich control architecture. For CA_X an implementation of Q.2931, called Q.Port, was used [**Bellcore97**]. The Q.Port software consists of a front-end, which clients connect to, and a back-end which was modified to set up connections across a switch by calling the appropriate operations over the *Ariel* interface.

In the tests, Q.Port controlled a single switch, located between two other domains. The two other domains were controlled by Sandman control architectures. Since the address format in Sandman is not the same as the format used in Q.Port, an address translation step is needed whenever inter-domain communication is required. In the proof-of-concept implementation, address translation is performed by an address-translation server which, given an address in one format, returns the address in the other.

The easiest way to test the simple hop-by-hop solution is by establishing an ISSC from a Sandman-1 to Sandman-2, across the Q.Port domain. This is done automatically: whenever an endpoint in Sandman-1 attempts to establish a connection to an endpoint in Sandman-2, an ISSC is set up. The existence of the ISSC proves that the hop-by-hop solution works. Next, the ISSC is used for signalling between the Sandman domains. As an example, it was demonstrated how an advance reservation for a multicast connection in a netlet was established across the two Sandman domains, while the reservation was mapped onto either an immediate reservation, or no reservation at all, depending on the interoperability policy corresponding to the netlet (and specified by a client).

6.9 Related work

Efforts within the ATM Forum and the ITU-T have led to the definition of signalling interfaces between switches called the Network-to-Network Interfaces (NNIs). Of these, the ATM Forum's Private Network-to-Network Interface is intended for private networks and contains interfaces both for the exchange of routeing information and for connection control [**PNNI94**]. The B-ISUP public NNI developed by the ITU-T serves as a demarcation point between two public networks. The ATM Forum's version of the public NNI is called B-ICI. Both private and public NNIs are discussed in Section 5.6.1.2.

The cooperation of various standards is the topic of the REFORM project [**Georgatsos99**]. REFORM is part of the European ACTS programme and combines elements of CORBA, TINA, PNNI, TMN, and ATMF UNI, which all interact in the REFORM environment. The focus is more on making various solutions for different problems (e.g. PNNI routeing, TMN management, UNI signalling, etc.) cooperate in a single framework, than making different incompatible control architecture domains interoperate.

6.10 Summary

Interoperability was achieved between incompatible control architecture domains. The starting point was a conventional hop-by-hop design, where signalling messages from one control architecture are translated into those of another at the domain boundaries. Functionality degradation at the domain boundaries and rigid function translation were identified as problems inherent to the design. The hop-by-hop solution was therefore rejected, except as an underlying mechanism to set up very simple connections (e.g. best-effort across multiple domains). Inter-domain signalling channels were used to solve the problem of functionality degradation. The problem of rigid function translation was solved by allowing clients to specify the appropriate mappings themselves. This is an example of functionality that would be impossible to support without dynamically loadable code.

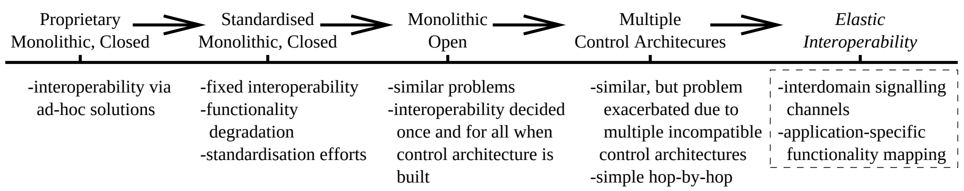


Figure 6.10: Interoperability in the evolution of network control

Figure 6.10 shows how such elastic interoperability fits in the evolution of interoperability in network control. So far, however, the discussion has focused mainly on control architectures and their interoperability. As the goal was to introduce elasticity to all aspects of network control and management, the next chapter will look at other components in the *Haboob*.

Chapter 7

Elastic switch interfaces, dividers and netbuilders

It was shown in Chapter 2, that the open switch control interface, known as *Ariel*, allows non-proprietary solutions to the problem of switch control. Furthermore, it was observed that in an environment where multiple control architectures are expected to be active simultaneously, switch dividers play an essential role. The switch dividers partition resources on a switch on behalf of a netbuilder, which combines the partitions to form virtual networks. This chapter talks about all three of these aspects of network control and management and shows how allowing extensibility of these components eases the introduction of new network control mechanisms.

7.1 Introduction

Switch dividers have two primary tasks. Firstly, they partition resources on the switch into switchlets, so that for example, each switchlet has its own VPI/VCI range, buffer space, etc. Secondly, they instantiate *Ariel* implementations for each of the switchlets. The *Ariel* instantiations are exactly the same as the *Ariel* interface to the physical switch, except that access to resources is limited to resources owned by the switchlet.

This chapter begins with a discussion of a switch divider in which the functionality of both divider and switch interface is no longer fixed. The former will be called *elastic divider* and the latter *elastic Ariel.* Similarly, netbuilders to which code can be added dynamically will be called *elastic netbuilders.* Extensions and changes can be made on the fly, when and where appropriate. It will be shown that this allows for functionality that would be very difficult to achieve otherwise.

7.2 Elastic *Ariel*

Ariel has been extended in two ways. The first allows applications to use what are called *native* methods, i.e. methods that are not part of standard *Ariel*, but which are supported by the switch. The second enables one to control and modify *Ariel* operations from granules running in a switchlet's Sandbox.

7.2.1 Native methods

The first extension that was made to *Ariel* in the course of this work concerned the support for implementation-specific functionality. This is referred to as the exposure of *native methods*. It entails the ability of applications to call functions that are not in *Ariel* but are part of the API of the protocol over which *Ariel* is implemented[1]. For example, if *Ariel* is implemented using GSMP, the GSMP `MoveBranch` operation [**Newman96**] may be exposed as a native method.

The *conformance check* is the function of the divider that ensures that a control architecture only accesses resources that belong to its own switchlet. Such a check must also be applied to native methods. Since native methods are provided by the implementation of *Ariel*, this implementation should also provide the conformance checks. There is a logical separation between native methods and *Ariel* operations and the conformance check for native methods is encapsulated in what is called a *native object* corresponding to a switchlet. To allow exposure of native methods, it is required that their conformance checks are implemented in the native object. Although the current implementation does not support this, the conformance checks could be dynamically loadable in the form of DLAs. In that case, only native methods for which a conformance function was loaded, can be exposed. In the current proof-of-concept implementation, only a small number of GSMP operations, such as the `MoveBranch` operation, has been made part of the native interface.

Native methods should be used with care. In particular, control architectures that may be used in many different environments with different equipment, should be written so that they do not depend on native methods. This is similar to the use of native methods in languages like Java in that whenever a Java applet depends on native methods, it can no longer be considered portable. To prevent naive use of native methods, they must be explicitly exported before they can be used.

7.2.2 The *Ariel* Sandbox

One of the functions exported in the divider's management interface allows the creation of a switchlet with a Sandbox. Compared with the standard operation

[1] Or any other protocol that both switch and switch divider support.

to create switchlets, it takes as extra argument the granule or DLA to run in the Sandbox. The Sandbox is extended with an *Ariel* module, which allows granules in the switchlet's Sandbox to call any *Ariel* function that is available to external clients. The Sandbox's *Ariel* module also contains an operation called `override`, which enables granules to override any of *Ariel*'s operations. The `override` operation specifies both the operation to override and the override procedure to replace it with. If an overridden operation is called, e.g. from an external client, the override procedure is called instead.

7.2.2.1 Example: an application-specific viewer

As a demonstration, an *Ariel* Sandbox is loaded with a DLA that overrides the operations for creating and deleting connections. The new `createVC` and `deleteVC` operations still set up and tear down connections on the switch, but also send event notifications corresponding to these operations to a DLA running in a remote Sandbox. The remote DLA acts as a *viewer* and puts up a graphical representation of a switch, showing all ports on which the control architecture owns resources. Whenever it receives connection setup or teardown notifications it updates the view and thus displays graphically the currently active connections on the switch[2].

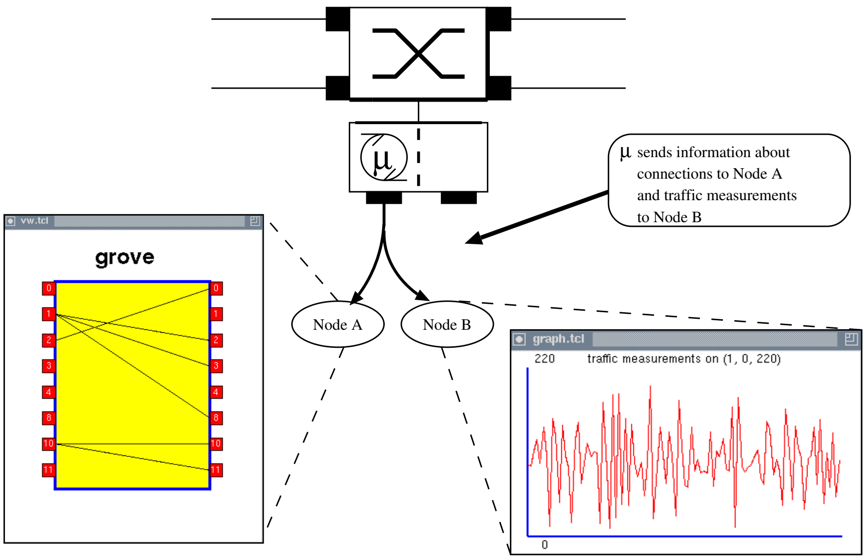


Figure 7.1: Remote monitoring with micro control achitectures

The demonstration DLA which runs in the *Ariel* Sandbox is called a micro control achitecture. It implements functions that can be called from external clients—in short, it acts exactly like the control architectures of Chapter 5, except that it is completely loaded into the switchlet Sandbox. Henceforth,

[2]A more general viewer, showing the switchlets and connections of all control architectures in the physical network, was implemented by Rebecca Isaacs of the Computer Laboratory

this particular DLA will be referred to as micro control achitecture μCA-1. It is shown graphically on the left in Figure 7.1 with an actual screenshot of the viewer.

7.2.2.2 Virtualising the VPI/VCI space

The `override` operation, present in all Sandboxes discussed in this chapter, is a powerful operation that allows for customisation and specialisation of operations that were previously fixed. Recall that switchlets constitute a partitioning of (among other things) the VPI/VCI space. This means that control architecture A obtains a different VPI/VCI range than control architecture B. However, certain control architectures may want to use specific VCIs inside the network. In that case, problems arise when a control architecture wants to use a VPI/VCI value V that was already assigned to another switchlet for some switches inside the network. This can be solved using the `override` operation in the *Ariel* Sandbox, by overriding all control operations to map V onto one of the switchlet's own VPI/VCI values V^*. This is transparent to the control architecture which continues to exercise control over the usual VPI/VCI value V.

In fact, it is possible to completely virtualise the VPI/VCI space[3]. A control architecture can be given the entire spectrum of VPI/VCI values supported by the switch and provide (or have provided for it) a simple mapping function, which translates these virtual VPI/VCI values to real VPI/VCI values supported by its switchlet. The virtual VPI/VCI space is logical only and may even include values outside the range supported by the switch, e.g. negative VPI/VCI values. At the edges of the network, it is possible to "pin down" real VPI/VCI values. The concept of virtual VPI/VCI spaces is almost identical to that of virtual memory in operating systems.

7.2.3 Adding new operations to *Ariel*

Ariel can be easily extended with new operations. The operations are called over the SUFI (unless they override existing *Ariel* functions in which case they are invoked over *Ariel*). In the simplest case, one only uses the original *Ariel* API. For example, it is possible to add an operation which creates a batch of connections, by taking as arguments the connection details for a set of connections, which are subsequently translated into normal *Ariel* connection setups. Such operations can improve performance considerably because each setup request no longer has to travel over the network separately. However, it is also possible to use native methods to provide new operations. In fact, new operations can be built that consist of a combination of *Ariel* and native methods[4].

[3] Thanks to Richard Mortier and Rebecca Isaacs of the Computer Laboratory for pointing out the usefulness of virtual VPI/VCI spaces

[4] Use of native methods does not necessarily mean that the new operation is not portable. For example, a DLA may probe the native methods to see if a particular method exists and

For demonstration purposes, an operation was introduced to micro control achitecture μCA-1 of Section 7.2.2.1 which, when invoked, periodically sends statistics of the traffic on connections or ports to a DLA running in a remote Sandbox. The remote DLA displays a scrolling graph of the statistics returned, as shown on the right in Figure 7.1. Since such operations can be added to the *Ariel* micro control achitecture from a remote client (in the form of granules which are added to the micro control achitecture), this lends itself well for use in remote monitoring, in the spirit of RMON [**RMON97**]. For example, if there is an indication of network problems such as congestion, it allows one to quickly add a granule that gathers statistics, possibly analyses them locally and sends back results. This avoids having to send a lot of statistics over an already overloaded network. If it is decided that different statistics (or different correlations of the statistics) are needed, one simply replaces the granule with another. The granule may even be able to solve the problem locally, e.g. by taking down certain connections, or destroying switchlets. This is a simple implementation of network management by delegation [**Goldszmidt95b**].

7.2.3.1 Vertical integration

The above permits new forms of vertical integration. Since it is possible to load both the control architecture and the application into an *Ariel* Sandbox, and functionality to govern the working of the switch divider into a management Sandbox, it is also possible to integrate these three functions into the same address space. If on top of that, the divider is integrated with the switch this allows for tight integration of network control, while retaining a modular design.

7.3 Elastic divider management

The divider elasticity is created by instantiating a Sandbox in the management component of the divider. It is possible to distinguish between two possible types of *Management* DLAs. The first type allows clients to customise or override operations for the entire divider. If a granule in such a DLA overrides the operation to create a switchlet, then the DLA is triggered, each time a switchlet is created. In other words, this type of *Management* DLA has far-reaching capabilities that affect all switchlets. The ability to install such DLAs should be restricted to privileged clients (e.g. the system administrators). On request, a single Sandbox is created for such DLAs. The Sandbox and its DLAs are persistent in the sense that whatever switchlet management operations are executed (e.g. to create or destroy a switchlet), the Sandbox will not be destroyed.

The second type only concerns the management operations for one particular switchlet. This type of DLA can be loaded as switchlet-specific manager,

use it if this is the case, while falling back on *Ariel* methods if not.

before the actual switchlet is created. It is able to use and override the management operations of this particular switchlet. Because this type of *Management* DLA will not affect any other switchlets, its access control requirements are much less strict. In principle, even the requesting control architecture could install such a DLA. The Sandbox and its DLAs are transient, i.e. they are destroyed with the switchlet. As an example, consider a *Management* DLA that overrides the method to create a switchlet with an operation that not only creates a switchlet, but automatically loads this switchlet with *Ariel* granules as well, e.g. to override in turn the operations to create and delete switch connections. The next section describes another example of *Management* DLAs.

7.4 Aggregate switchlets and delegated control

This section contains an example that addresses the problem of scalability. It uses both *Ariel* Sandboxes and management Sandboxes. Remember that creating a switchlet normally comprises the execution of a single predefined function: a switchlet is created and an interface reference to control it is returned. Now consider Figure 7.2 in which a wide area virtual network is shown with two main centres: one in London and one in Glasgow. In theory, it is possible to control this network with a single controller (connection manager)[5]. As shown in the figure, this presents the problem of overstretched control paths: each of the switchlets in Glasgow is controlled separately by the single control architecture in London. This results, for example, in very long connection setup times.

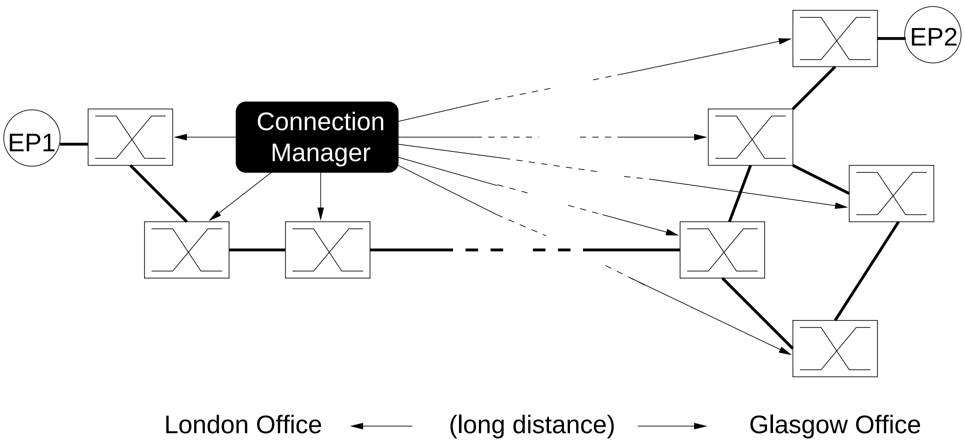


Figure 7.2: Large network: overstretched control paths

[5]For clarity reasons, the example is somewhat extreme; for really long distances it is probably better to use multiple interoperating control architectures, as discussed in Chapter 6.

7.4.1 Aggregating resources

There are many ways to solve this problem. A simple solution that does not require changing the existing network control software (and can be added on the fly) uses *aggregate switchlets*. This is illustrated in Figure 7.3. Aggregating resources is a well-known technique to address scalability and is employed, for example, in the P-NNI routeing algorithms [**PNNI1.0:96**]. In the example, the resources of the five switchlets in Glasgow are aggregated into one meta-switchlet called the *aggregate switchlet*. The London-based control architecture only controls the local switchlets and the one aggregate switchlet. The aggregate switchlet, meanwhile, picks up the requests sent by the control architecture and translates them into the appropriate requests for its local constituent switchlets. To the control architecture it seems as if it is communicating with a single (albeit rather large) switch. Another way of saying this is that control is delegated from the central site to the aggregate switchlet.

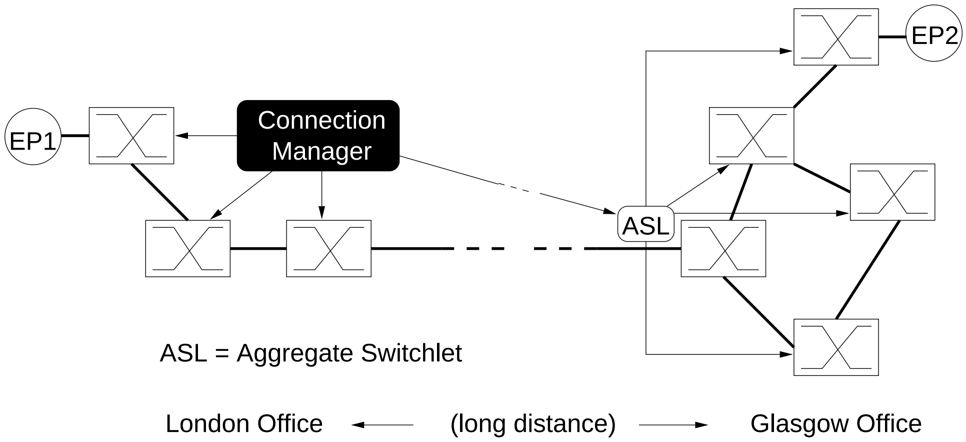


Figure 7.3: A more scalable solution using resource aggregation

The aggregate switchlet is created automatically by a switchlet-specific *Management* DLA. Whenever a switchlet is created by an entity from London, the DLA not only creates a switchlet on its own local divider, but on all other dividers for the Glasgow part of the network as well. Additionally, the DLA instantiates a DLA in the *Ariel* Sandbox which overrides all *Ariel* functions so that when they are called by the control architecture, each request is decomposed into the appropriate sub-requests for the constituent switchlets. Destroying the aggregate switchlet will result in the destruction of all its constituent switchlets. This is transparent to the control architecture (as well as to the constituent switchlets). In this way a hierarchy of resource aggregation may be created that is arbitrarily deep (i.e. aggregate switchlets may themselves be aggregated). As an experiment, an aggregate switchlet was built, which controlled all five switches of the local ATM testbed, while offering only a single switchlet interface to a Noman control architecture.

7.4.2 Remote monitoring and management by delegation

A slightly more conventional use of aggregate switchlets is remote monitoring and "management by delegation" [**Goldszmidt95a**]. Instead of, or in addition to, controlling the aggregate switchlet, the DLA for the aggregate switchlet can monitor the individual switchlets under its control. For example, it can obtain statistics periodically, or ping individual components to see if they are alive. When there are problems, the DLA can generate alarms, or even try to solve the problem independently. This is useful for two reasons. Firstly, the response time to network problems is shorter, because the managing entity is close to the switches. Secondly, if there is a network problem, it may be that part of the network becomes unreachable, so that it becomes *impossible* to solve the problem from the remote control architecture. As pointed out in [**Goldszmidt95a**], making the control loop shorter by placing managing entities in the vicinity of the switches, increases the probability that the management messages actually get to the appropriate places. Next, a simple example of such network management which implements a degree of fault-tolerance is discussed.

Consider an aggregate switchlet that consists of a set of sub-switchlets in such a way that all or most endpoints can be reached through multiple paths. An example is given in Figure 7.4, where an aggregate switchlet spans five physical switches. Suppose that in the aggregate switchlet a connection was set up between link A and link B via switch $S1$. If, for some reason, it is necessary to take switch $S1$ out of service (e.g. for maintenance), the aggregate switchlet can decide to reroute the call via $S1$, without informing the control architecture. In fact, the control architecture is aware neither of the fact that one of the switchlets went down, nor of the fact that its connection was rerouted. The problem of fault tolerance and automatic rerouteing is the topic of ongoing research in the Computer Laboratory.

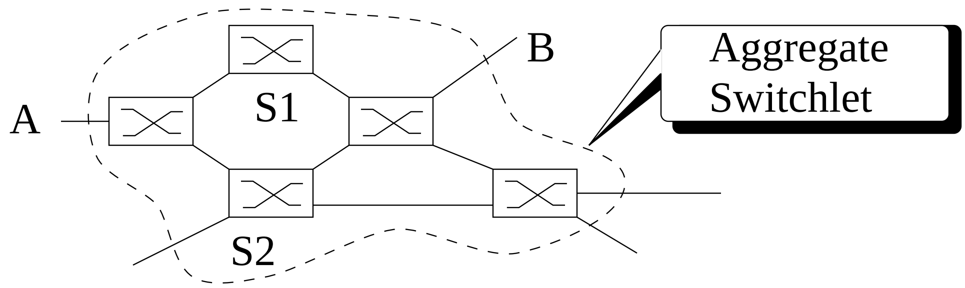


Figure 7.4: Fault tolerance using aggregate switchlets

7.5 Elastic netbuilders

The netbuilder not only implements the procedures to create virtual networks, it also provides a host of functions that allow control architectures to discover useful information about the network, e.g. the addresses of endpoints, the topology of a switch, etc. Until now, the exact operation of the netbuilder procedures was decided once and for all at implementation time.

7.5.1 Shared and individual Sandboxes

In the *Haboob*, the netbuilder was extended with a Sandbox that is shared by all clients of the netbuilder. In addition, individual Sandboxes that are owned by specific control architectures can be created on request. The only difference between shared and individual Sandboxes is the access control and scope of the `override` operation. In the shared Sandboxes, access to the `override` operation is restricted to certain privileged clients, e.g. the system administrator. An override at this level will modify the behaviour of the overridden function for all clients. Individual Sandboxes, by contrast, may grant access to their `override` operation to any client. The scope of the override is limited to this Sandbox.

The scope of the override in a shared Sandbox extends to the individual Sandboxes as well. So, if the system administrator modifies the behaviour of an operation F and a granule in an individual Sandbox calls this operation, the modified version will be invoked (unless the individual Sandbox also overrides operation F). Override operations in an individual Sandbox pertain only to calls made from the Sandbox itself or calls over the Sandbox's SUFI. All other external invocations (i.e. invocations made from remote control architectures) are only influenced by overrides in the shared Sandbox.

Both the individual and the shared Sandboxes have been extended with two modules. The first module provides a set of operations that allow clients to find out useful information about the network, e.g. about the topology of a switch or the addresses of endpoints. The second allows DLAs to call (and override) operations to create and destroy virtual networks, add switchlets to and remove switchlets from existing virtual networks, etc.

7.5.2 Automatic aggregation of switchlets

To demonstrate the usefulness of elastic network building, a DLA was loaded in the netbuilder's shared Sandbox which redefines the operation to add a switchlet to a virtual network. When the procedure to add a switchlet is called, the DLA checks the name of the switch on which to create a switchlet and, depending on the result, either simply creates a switchlet, or initiates the creation of an aggregate switchlet as described in Section 7.4. The next two sections will consider this in a little more detail.

7.5.2.1 Implementation

Overriding the operation to create a switchlet allows for the use of *logical switch identifiers*. These are names that seem to identify a single switch but in reality map onto a number of physical switches. In this dissertation, the set of network resources corresponding to such a logical switch will be defined as an *aggregate switch*. The mapping to the physical network resources is transparent to the clients of the netbuilder, who only "see" a single switch.

Suppose, for example, that a client requests the creation of a switchlet on a switch called `Aggregate_1`. This is not a real switch. Instead `Aggregate_1` serves as a logical switch identifier that triggers special actions in the netbuilder. In the example implementation, the method to create a switchlet was overridden, so the netbuilder executes the specified override function. The override function recognises that `Aggregate_1` is really an aggregate switch that maps onto a specific set of interconnected physical switches. It subsequently initiates the necessary actions to create the corresponding aggregate switchlet. For this purpose, it installs the switchlet-specific *Management* DLA described in Section 7.4 in one of the dividers (the one that will handle all requests for the logical switchlet). This DLA in turn creates all the other switchlets that correspond to `Aggregate_1` and instantiates a DLA in its local *Ariel* Sandbox to override all *Ariel* functions in the appropriate ways. Finally, the netbuilder returns the interface reference of the aggregate switchlet to the requesting control architecture. This completes the discussion on aggregate switchlets.

7.5.2.2 Distributed netbuilders

The previous section assumed that all switches lie in the domain of a single netbuilder. This will generally not be the case. Currently, netbuilders do not provide automatic support for this problem. Therefore, a proof-of-concept solution that was achieved in the *Haboob* makes extensive use of DLAs to make up for this lack of functionality. To illustrate the point, it is assumed that the physical network is partitioned into two domains, each managed by a different netbuilder. It is also assumed that both netbuilders have been loaded with the network-building DLA described below.

The DLA consists of two granules. The first granule, called the *state exchange* periodically sends state information to the remote netbuilder. This information generally consists of topology information that it wants the remote end to know about. For example, in the proof-of-concept implementation, both DLAs tell the other side about the endpoints reachable through, as well as the switches in their respective domains. These state messages are sent every five seconds and also serve as so-called 'heart-beat' messages (if no state message is received within a certain period, it is assumed that the remote side is down).

The second granule overrides the operation to create a switchlet on a switch.

If it finds that the switch on which the switchlet is to be created lies in the remote domain, it forwards the request to the remote netbuilder, together with the VPI/VCI space needed on the switchlet (determined by the VPI/VCI space of the adjacent switches). The remote netbuilder will try to create a switchlet on the adjacent switch, provided that an overlap in VPI/VCI space between adjacent switches is found.

This is a simple solution for the distributed netbuilder problem, which doesn't scale well to a large number of netbuilder domains. If the number of domains grows, it will become too much of a burden to have all netbuilders exchange such detailed topology information. Instead, some sort of information aggregation is needed, e.g. similar to that used in P-NNI routeing [**PNNI1.0:96**]. A general solution for the problem of distributed network building is beyond the scope of this work. Nevertheless, the simple solution just described demonstrates that it is possible to build virtual networks across multiple netbuilder domains and also that elastic netbuilders allow very useful functionality to be added to the existing implementation.

7.6 Example: active networks

This dissertation agrees with the active networks community that instead of standardising the computation performed on every packet, standardisation should focus on the computational model [**Tennenhouse96**]. It disagrees, however, about where to apply this solution. The active networks' approach is to treat *every* packet on the datapath as a program [**Tennenhouse96**], or at least allow *some* of the packets to carry program code [**Alexander98a**]. Instead, this dissertation proposes not to interfere with the data path at all. If computation on the data packets is necessary this should be treated as the exception rather than the rule (other packets should simply be switched at the highest possible speed). Advantages of separating control from data are highlighted in Section 3.8.4.

Despite these misgivings, the value of in-band computation, as advocated by the active network community, in *certain* cases is recognised. In-band processing is useful for the purposes of filtering, compression, encoding, transcoding, protocol enhancement and other such tasks that require manipulation of the data itself. For example, [**Marcus98**] shows as an example how a protocol "booster" can extend the functionality of a protocol on the fly, by adding forward error correction (FEC) on the datapath when it is found that one of the links is very slow and unreliable. The booster only affects that particular link. The idea is as follows. Whenever communication takes place across an unreliable link L with a slow (or non-existing) return channel (e.g. a satellite connection), it is often cheaper to deal with data loss and corruption by employing FEC than by using the more standard method of retransmissions. The default protocol has no FEC functionality. Instead, "booster code" is installed in the active network nodes just before and just after link L. In the former, the FEC is added to the data on the data path, while in the latter it is removed again. The FEC information only traverses link L. The endpoints still communicate via the base protocol and need not even be aware of the fact that FEC has been employed.

Similarly, [**Alexander97**] shows how an "active bridge" between two Ethernets segments allows for upgrading to new versions of network software with minimal disruption. Related work at the University of Lancaster implements what are called *propagating filters* [**Yeadon96**], i.e. filters that can be allocated to network nodes dynamically. The idea is to allow transcoding of continuous media streams and variation of QoS in multipoint communication in order to support heterogeneous clients.

The problem addressed in this section is what to do when processing is needed on the datapath. In the *Haboob*, this problem is dealt with in a simple way: if and when necessary, an active network node is built on the fly. Following the model of Section 3.1, active network nodes are just special cases of *datapath components*. This section shows how *Ariel* DLAs, together with an additional datapath component running off-switch, are used to implement an active network node.

Consider the example of the protocol booster. Whenever the decision is made that a particular link is unreliable and needs FEC, it is simple to install an *Ariel* DLA that reroutes the connection to a DLA in a Sandbox near the switch[6]. The DLA adds the FEC and returns data and code to the data path. The protocol booster mechanism is illustrated in Figure 7.5. This does not differ in any way from the protocol boosters in [**Marcus98**]. In fact, a very simple (off-switch) active node has been built. Packet classification is simple and only based on the ATM cell's flow label. Protocols such as ANEP [**Smith99**] are not needed when cells or packets are switched into an active node based on flow labels.

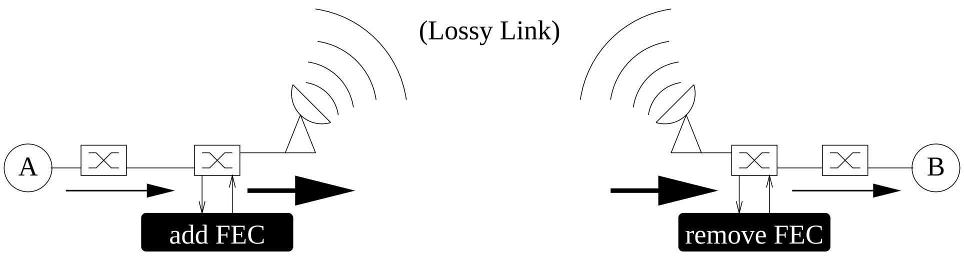


Figure 7.5: Active protocol booster in the *Haboob*

For the purpose of building an active node, an ATM enabled datapath component was implemented that includes a Sandbox with the means to manipulate a connection's transmit and receive buffers directly. An ATM data module was added to the Sandbox, which allows DLAs to send and receive data on ATM connections. DLAs are given access to the transmit and receive buffers *directly*,

[6]This FEC DLA can be automatically loaded by the *Ariel* DLA.

i.e. without copying the data to DLA space (which would be unacceptably expensive, especially if the data concerns continuous media). This is a delicate operation, because giving DLAs direct access to memory introduces the risk that they will access and/or manipulate memory that is not theirs. This could have disastrous results and completely destroy the Sandbox's safety and security mechanisms. It was solved by strictly limiting the DLA's direct memory access to those buffers that were explicitly created for it. Next, a DLA was loaded in the Sandbox which adds FEC to the data on specific connection and sends the FEC enhanced data back to the switch on the same VCI. In the current implementation a simple Hamming code is used for FEC [**Hamming80**].

The conclusion is that, although technology and mechanism in both approaches are different, it is possible to build active networks with the *Haboob*. Note that the inverse is not necessarily true: (capsule-based) active networking technology is tied to the integration of control and data path. This makes it impossible to build services that rely on the separation between these two tasks.

7.7 Summary

This chapter has shown some of the limitations of 'static' switch interfaces, dividers and netbuilders. It proposed to solve these problems by allowing code to be added to these components dynamically, making them *elastic*. The advantages of such an approach are that (1) functionality that is currently missing can be added on the fly, without having to wait until it is implemented in the components (e.g. distributed network building), (2) environment-specific knowledge and functionality can be exploited (e.g. by using *native* methods in the interface to the switch), and (3) functionality that is currently offered by the components can be overridden to perform customised tasks (e.g. virtualising the VCI space, or building aggregate switchlets). The viability of this approach has been demonstrated by a working system. In particular, it was shown how active network nodes can be built on the fly.

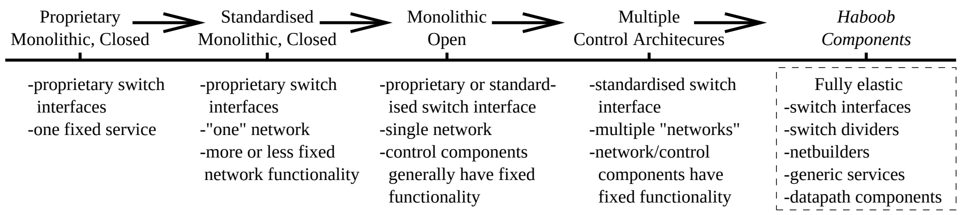


Figure 7.6: Remaining elastic components in the evolution of telecommunications network control

Figure 7.6 shows how the elastic components of the *Haboob* form the next step in the evolution of network control. Compared to the OSSA components, which allowed multiple control architectures to be active simultaneously, but were themselves closed, all of the components in the *Haboob* are completely elastic, so that the process of partitioning resources, building virtual networks, etc., can be customised. Whichever technology one chooses for the control and management of networks, however, it will probably not be very successful if it performs poorly. So, the next chapter will look at the *Haboob*'s performance.

Chapter 8

Performance evaluation

The previous chapters discussed in detail the way in which all components of the *Haboob* have been made elastic. Some of the advantages of such an elastic approach to network control were shown by various example applications. In this chapter the performance of the *Haboob* is evaluated.

8.1 Introduction

As a measure of performance, the connection setup times incurred by the control architectures described in Chapter 5 will be compared with those published in the literature. It will also be shown how using effective bandwidth (EBW) estimation for CAC improves the resource utilisation. The ATM testbed used to test and develop the network control and management implementations discussed in this dissertation currently consists of five ATM switches, namely one FORE ASX-1000, two FORE ASX-100s and two FORE ASX-200s [**FORE95a**]. Connected to the switches are a number of SUN UltraSparc workstations running Solaris (versions 2.5 and 2.6), as well as a number of machines running the *Nemesis* operating system [**Leslie96**]. Also connected are a number of cameras plugged into FORE ATM Video Adapters (two AVA-200s and an AVA-300 [**Pratt94**]) and an ATM video display device (ATV). Figure 8.1 illustrates the topology of the ATM testbed used in the experiments. Not all workstations and equipment connected to the switches are shown, only those used in the experiments. Since the switches are fully connected, it was possible to experiment with networks of any topology by excluding certain links from the physical network.

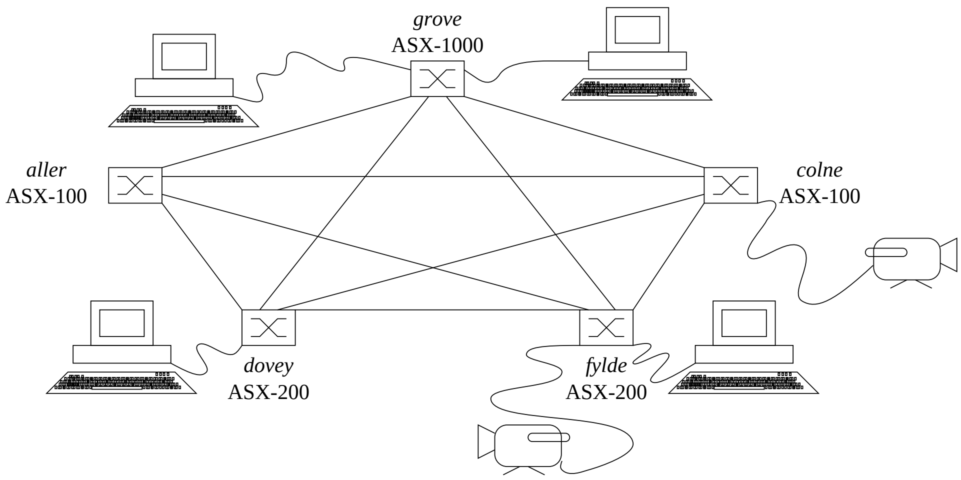


Figure 8.1: Topology of the ATM testbed

8.2 SUFI overhead

First, the overhead associated with the SUFI is considered. The performance measurements all used a Tcl implementation of the *Gobi* Sandbox (see Section 4.5). In the experiments, the client and the server ran on different machines (167 MHz UltraSparcs), which were shared by other users. The communication component of the Sandbox is built on top of OmniOrb [**ORL97**]. This means that the native OmniOrb RPC times must be subtracted from the SUFI invocation time, to get the overhead incurred by the SUFI administration.

Table 8.1 shows the invocation times of OmniOrb RPC and SUFI remote evaluation as measured. The results were obtained by averaging over 100 invocations. The best and worst invocation times that were measured are also included. Null-RPC comprises the invocation of a remote operation with no arguments, which returns immediately. A null-RPC call took approximately 0.8 ms on a reasonably loaded machine. It is not quite fair to use null-RPC values for the analysis, because the SUFI provides additional functionality in the form of capability-based access control. The second row in Table 8.1 shows the communication overhead when a compensation is made for the capability (e.g. the same capability is sent in the RPC call as well). The last row of Table 8.1 shows the time it takes from the moment a DLA initiates a SUFI remote evaluation call with no arguments, until control returns to the DLA. It turns out that the SUFI remote evaluation operation is approximately five times slower than the OmniOrb RPC with the same arguments. The overhead comprises access control and other administration, as well as evaluating interpreted code.

Type of call	Min (ms)	Aver (ms)	Max (ms)
OmniOrb null-RPC	0.7	0.8	1.1
OmniOrb RPC with SUFI data	0.9	1.2	1.8
SUFI ping from DLA to DLA	5.9	6.0	6.2

Table 8.1: Communication overhead of the SUFI and OmniOrb RPC

8.3 Connection setup and teardown times

This section looks at the connection setup latency, a metric that is often used to compare control architectures, e.g. [**Battou96**]. The workstations used were all 167 MHz UltraSparcs, while the DPE used for communication is OmniOrb. Long connection setup times make a control architecture unsuitable in environments where short-lived connections are expected to be common. End-to-end connection setup latency is comprised of (1) the time it takes to process the call in the control architecture, (2) the time it takes to communicate with the switches to actually set up the connections and (3) the time it takes to communicate within the control architecture.

8.3.1 Call processing overhead

The time it takes to process a call depends on the control architecture and on the service requested by the client and is further influenced by the number of simultaneous requests that are made to the control architecture. For example, it makes a difference whether an advance reservation for a multicast connection is requested or an immediate point-to-point call. To illustrate the point, the call processing time of Sandman, the most complex control architecture discussed in this dissertation, and a trivial Noman control architecture are compared. The Noman control architecture hardly does any call processing at all. It checks a static routeing table to find the switch ports corresponding to the connection, makes sure that VPI/VCI pairs are available on them (and marks them as used) and enters the connection details in a connection table. If the connection table is empty initially, the call processing time for setting up a single connection across a switch is 0.6 ms. In the Sandman experiment, the call processing time was measured for an advance reservation for a connection across a single switch. On average the overhead for making the advance reservation is approximately 1.4 ms (see Table 8.2).

8.3.2 DPE overhead

The time it takes to communicate with and within the control architecture (the *DPE time*) varies depending on which DPE is used. For example, OmniOrb claims an inter-machine null-RPC time of 0.7 ms [**ORL97**], while DIMMA [**Li95**] does not get much faster than 1.5 ms. The DPE times measured here

Operation	**Overhead (approx.)**
Noman call processing	0.6 ms
Sandman call processing	1.4 ms
DPE time for notification of endpoint	0.85 ms
DPE time for immediate setup RPC	1.4 ms
DPE time for loading simple DLA	6.2 ms
Minimum connection setup time (ASX-1000, GSMP)	15 ms
Minimum connection setup time (ASX-200, SNMP)	6.7 ms

Table 8.2: Summary of *Haboob* performance figures

were all achieved using OmniOrb. The optimal RPC figures published in [**ORL97**], were never quite achieved. Table 8.1 shows some typical OmniOrb RPC figures (as measured).

In Noman control architectures DPE time is strongly related to the SUFI overhead of Section 8.2. Because the SUFI allows for remote evaluation, in addition to traditional RPC, it is more difficult to estimate the DPE time. For example, where a traditional RPC style interaction with a server (e.g. the control architecture) required n invocations, the client in the *Haboob* can decide to invoke only one remote evaluation of a granule, which from then on makes all n invocations locally. The time to send a granule for remote evaluation will be denoted by T_{REV}, while the overhead of an RPC will denoted by T_{RPC}. Although the one remote evaluation might be more expensive than a single RPC, the total DPE time can be significantly reduced, if $T_{REV} \ll n \times T_{RPC}$.

Figure 8.2 shows the DPE time as function of the number of connections that were created of both an RPC style interaction (which rises approximately linearly with the number of connections) and a single remote evaluation to create the same number of connections (which remains constant). The RPC style interaction invokes a separate `createVCI` operation for each switch connection. The remote evaluation creates all switch connections by local calls in the same address space. T_{REV} was measured to be 6.2 ms on average, while T_{RPC} averaged only 1.4 ms. The figure shows that the break-even point for setting up multiple connections across a single switch (as far as DPE time is concerned), lies around four connections.

The overhead incurred by the DPE depends on the number of DPE invocations made. The number of remote invocations can be less in a Noman control architecture, because control architecture and application can be integrated to a large degree (vertical integration). However, some DPE invocations are hard to avoid. For example, it will always be necessary to notify endpoints of the existence of a connection (providing the VPI/VCI values of the connection). This notification time is the same for Sandman and Noman and averaged 0.85 ms.

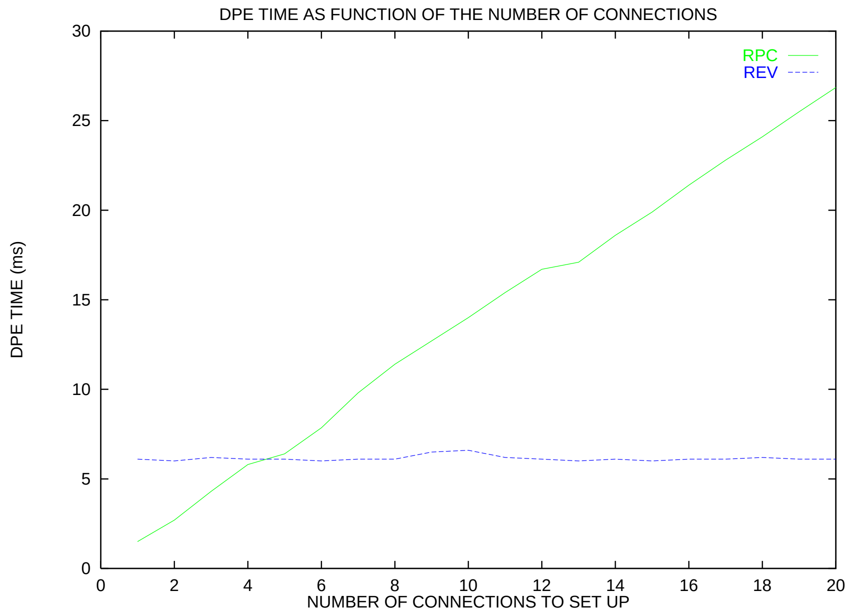


Figure 8.2: Remote evaluation outperforms RPC when the number of connections increases

8.3.3 Creating connections on a single switch

This section considers the time it takes to create the actual connections, i.e. the overhead generated by communication with the physical switches. For this purpose, a number of experiments was conducted in which both the *Ariel* implementation as well as the location of the initiating client was varied. The various possibilities are illustrated in Figure 8.3, the letter codes of this figure correspond to the first two characters of the scenario names in Figure 8.4.

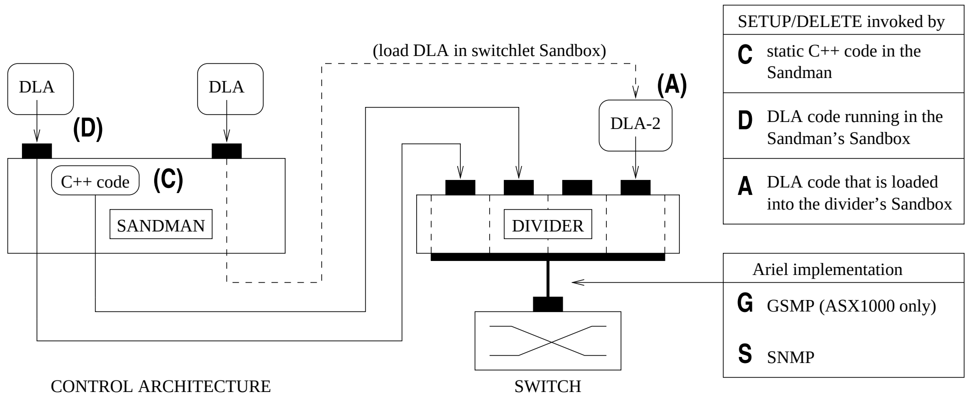


Figure 8.3: Options for client location (A, C, D) and implementation of *Ariel* (G, S)

8.3.3.1 The experiments

Each experiment consisted of the setup and teardown of a thousand connections across a single switch. The average duration of each create/delete pair was recorded. Each experiment was repeated ten times and the best case, worst case and average of these values was recorded. The experiment scenarios are named as indicated in Figure 8.4. The switches used were a FORE ASX-1000 switch and a FORE ASX-200. The Sandboxes were all based on Tcl.

SCENARIO NAME	DESCRIPTION
G-C-1	-Switch communication: Ariel over GSMP to ASX-1000 -Control architecture and divider independent processes -All processes run on the same machine -Time measured: from the moment 'createVCI' is called in static C++ code, until the 'deleteVCI'operation returns to the C++ code (after 1000 setups and teardowns)
G-D-1	-same as G-C-1, except that the time is measured from the moment 'createVCI' is called in the DLA, until the 'deleteVCI'operation returns to the DLA (after 1000 setups and teardowns)
G-A-1	-Same as G-C-1, except that the time is measured from the moment that the control architecture DLA initiates the remote evaluation procedure to load DLA-2 into the switchlet Sandbox (this includes creating the Sandbox) until 1000 connections have been setup and torn down by DLA-2
G-C-2	-Same as G-C-1, except that client and divider run on different machines.
G-D-2	-Same as G-D-1, except that client and divider run on different machines.
G-A-2	-Same as G-A-1, except that client and divider run on different machines.
S-C-2	-Same as G-C-2, except that Ariel is implemented over SNMP
S-D-2	-Same as G-D-2, except that Ariel is implemented over SNMP
S-A-2	-Same as G-A-2, except that Ariel is implemented over SNMP
S-C-200	-Same as S-C-2, except that the experiment is now run on a FORE ASX-200
S-D-200	-Same as S-D-2, except that the experiment is now run on a FORE ASX-200
S-A-200	-Same as S-A-2, except that the experiment is now run on a FORE ASX-200

Figure 8.4: Experiment scenarios for measuring setup/teardown times

Figure 8.5 shows the results of these experiments. It is striking how much faster the connection setup/teardown time is for an ASX-200, compared to the ASX-1000. No GSMP server was available for the ASX-200 switch, so only the SNMP implementation could be tested. In experiment *S-A-200* on switch *fylde*, the total time taken by a connection set up and teardown was on average only 8.7 ms. Of this combined figure, 6.7 ms were taken up by creating the connection, while tearing it down again took approximately 2 ms.

The are two reasons why the ASX-200 is so much faster than the ASX-1000. Firstly, the nature of the control software on the ASX-1000 is dubious. The switch image that was loaded into the ASX-1000 is quite flaky. The reason why it was used is that it offered GSMP support. However, many spurious non-GSMP messages are created and sent also (e.g. FORE SPANS messages), which hampers the performance. Secondly, the control processor in the ASX-200 is a Pentium II, which is more powerful than the ASX-1000's Intel i960. As expected, on the ASX-200 switch, scenario *G-C-200* performed a little worse than *C-A-200*, because there is an extra communication with the switch. *G-D-200* was slightly slower still, because it suffers both from the extra communication overhead and from the overhead incurred by running interpreted code. Even so, the performance is still very good (as will be shown in Section 8.3.3.2).

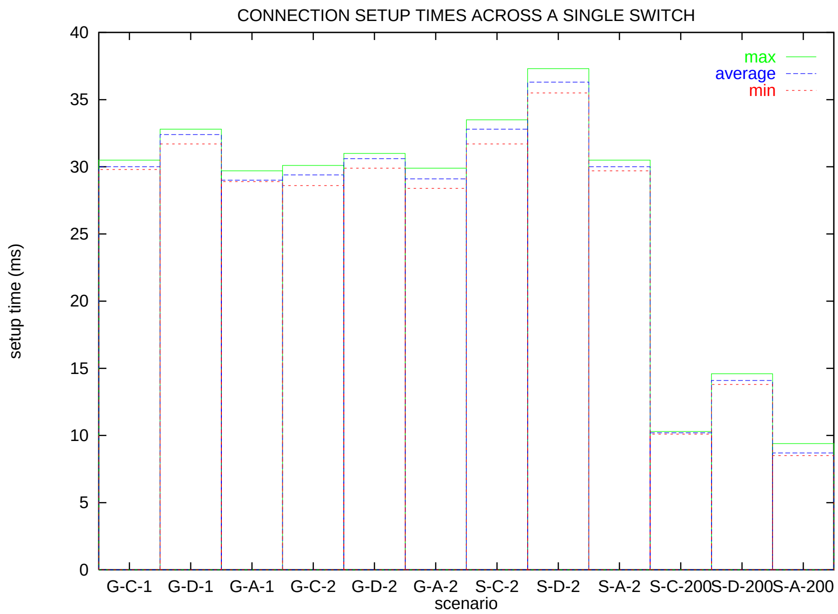


Figure 8.5: Total time for connection setup + teardown of single switch connection

On the ASX-1000, the average time for a combined connection setup and teardown lies around 30 ms. Again, the fastest times, approximately 28 ms, were achieved by the DLA running on the divider itself, as was expected. In this case, the DPE time was negligible. Breaking down this figure revealed that connection creation took approximately 15 ms, while tearing it down required 13 ms. In all other cases, the overhead includes DPE time incurred by the communication with the divider process. This overhead outweighed that of evaluating interpreted code. The results do not include the time needed to notify the endpoints. Table 8.2 shows that notification takes 0.85 ms.

Another interesting result is that setups and teardowns where client and divider ran on different hosts turned out to be slightly *faster* than if they ran on the same host. This in spite of the extra communication overhead incurred by the former. A possible explanation is that the DPE did not optimise well for same-host communication, so that the calls had to travel all the way up and down the protocol stack. Additionally, running two communicating processes on the same host involves a context switch each time an RPC is made. This is not necessary when the processes run on different machines.

On the ASX-1000, the SNMP implementation is a few milliseconds slower than the GSMP version. This is not surprising, as GSMP was explicitly developed for switch control, while SNMP connection setup and teardown involves walking the switch's management information base (MIB) with SNMP `get` and `set` operations. In fact, it was somewhat of a surprise that the difference between GSMP and SNMP was so small. Again, the reason can probably be found in the very poor implementation of the GSMP server on the switch.

8.3.3.2 Comparison

The connection setup times are compared with published performance figures. The results are summarised in Table 8.3.

Xbind is somewhat comparable to the *Haboob* in that it uses an IP bootstrap network and a CORBA DPE to implement off-switch control [**Lazar97**]. An intrinsic call setup time of 16 ms is claimed with a GSMP implementation of the signalling software running on a Sun SparcStation 10. It is not clear on what type of switch this was achieved, or indeed what 'intrinsic' call setup time means.

UNI signalling performance was tested for both wide area networks (WANs) and local area networks (LANs) in [**Battou96**]. On a single switch experiment in the Naval Research Laboratory, a extremely long connection setup time of 53 ms was measured. The author speculates that the poor results was due to the low priority that given by the switch to signalling requests, which meant that most of the resources were used by the routeing daemon. This hypothesis was supported by subsequent experiments. The switch hardware is not specified.

A first attempt at developing performance benchmarks for ATM signalling is presented in [**Niehaus97**]. Although the work is aimed at providing a general benchmarking framework for many different types of hardware and signalling software, the figures presented so far only address UNI signalling on a small number of switch types. Indeed, to present the control software with signalling requests, a call generator was implemented which is built around a core formed by the Q.Port signalling software [**Bellcore97**]. It shows that off-board control using Q.Port over GSMP on a Linux machine proved to be as fast as on-board signalling on a FORE ASX-200 Work Group switch. The ASX-200 BX represents newer hardware. The authors speculate that the significantly faster signalling results of the BX were due to increased processing power of the CPU supporting signalling.

Switch type	**Signalling software**	**Type of signalling**	**Setup time (ms)**
Unknown	Xbind/GSMP	off-switch	16
Unknown	UNI	on-switch	53
DEC AN/2	UNI (Q.Port/GSMP/Linux)	off-switch	21
FORE ASX-200WG	UNI	on-switch	20
FORE ASX-200BX	UNI	on-switch	10
FORE ASX-1000	Sandman/GSMP	off-switch	15
FORE ASX-200	Sandman/SNMP	off-switch	< 7

Table 8.3: Comparison of connection setup times across a single switch

8.3.4 Multiple switches

Finally, the connection setup and teardown times across multiple switches are considered. For this purpose, the topology of Figure 8.1 is arranged as a line of switches, each of which has a potential endpoint connected to it. The source endpoint is connected to *grove*. The first sink endpoint is connected to *grove* as well, the second to *fylde*, the third to *dovey*, the fourth to *aller* and the fifth to *colne*. Now the combination of a connection setup and teardown from the source to each of the sinks is measured. The experiments involve one thousand connections that are set up and torn down and all values are averages of these thousand connections. The experiments are repeated ten times. The best, worst and average case values are shown in Figure 8.6. For *grove* an implementation of *Ariel* over GSMP is used, while all other switches are controlled using SNMP. The results are shown in Figure 8.6.

Also shown in the same figure are the performance figures of setup times across multiple switches using ATMF UNI published in [**Battou96**]. Note that these figures represent connection *setup only*, while the Sandman figures include *both a setup and a teardown*. In other words, creating a connection in the Sandman and subsequently tearing it down again, is approximately twice as fast as only creating a connection in [**Battou96**]!

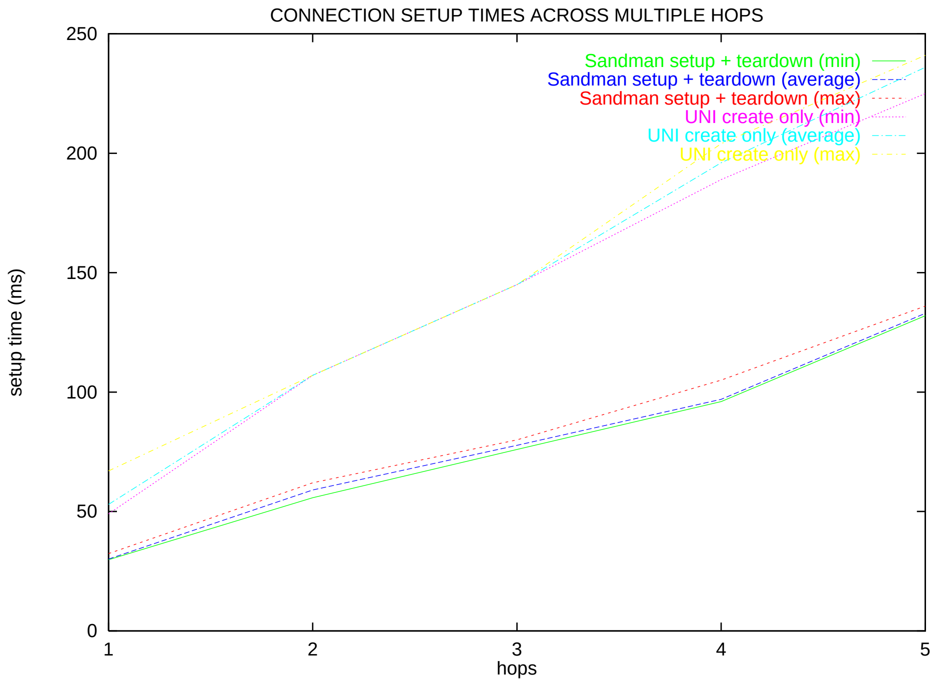


Figure 8.6: Connection setup and teardown times for multiple hops

8.4 Traffic servers and admission control

In this section, the traffic servers proposed in Section 5.2 and extended in Section 5.4.2.3 are evaluated. Traffic servers are generic servers that can be used (and extended) by any application. In this section, it will be shown how they are used successfully for admission control. For this purpose a variety of different sources is used, including JPEG video streams at various frame rates, as well as data streams created by a *traffic generator*. The latter allows more precise control over the burstiness and general shape of the arrival process.

8.4.1 Effective bandwidth of a single source

To see how estimating the effective bandwidth helps resource utilisation, consider Figure 8.7. The figure shows a thirty seconds trace of a bursty source (created by a traffic generator running on a Sun UltraSparc). The connection is created approximately at $t = 12.5$ and becomes active at $t = 14$. All measurements were conducted on a FORE ASX-100 switch with a buffer size of 256 cells. The switch is loaded with special-purpose software that periodically sends measurements via the switch's control port to a traffic server running on a general-purpose workstation. In the first experiment, this period was set to 75 milliseconds. The traffic server calculates the effective bandwidth (EBW) on connections of which it was previously notified that they are active. It also calculates the aggregate effective bandwidth. The results of these calculations are sent to the Sandman control architecture to be used for CAC decisions.

Theoretically, the peak rate for this source reaches 4700 cells per seconds, but in practice this rate is seldom reached. Indeed, as shown by the trace, during long intervals the rate stays well below 4000 cells per seconds and bursts are relatively rare. The mean cell rate seems to lie somewhere around 3500 cells per second. Instead of using the peak rate, it is more efficient to use the effective bandwidth, which, by definition, lies somewhere between the mean cell rate and the peak rate. For illustration, Figure 8.7 also shows the connection's estimated effective bandwidth corresponding to a cell loss ratio of 10^{-2}. In this case, when the effective bandwidth is used for admission control instead of the peak rate, the gain in resource utilisation approaches 15%.

It is interesting to note that like the measured traffic, the estimated effective bandwidth starts at 0, but rises very rapidly the moment the source becomes active at $t = 14$. Since the data rate jumps from 0 to almost 4000, the effective bandwidth makes a large jump as well. After the initial jump, however, the effective bandwidth remains within a small range, not far above the mean rate and well below the peak. Where the effective bandwidth estimate stabilises is a function of the target CLR. A smaller target CLR would push the effective bandwidth estimate closer to the peak. After 43 seconds, the connection is torn down. As the control architecture explicitly tells the traffic server that

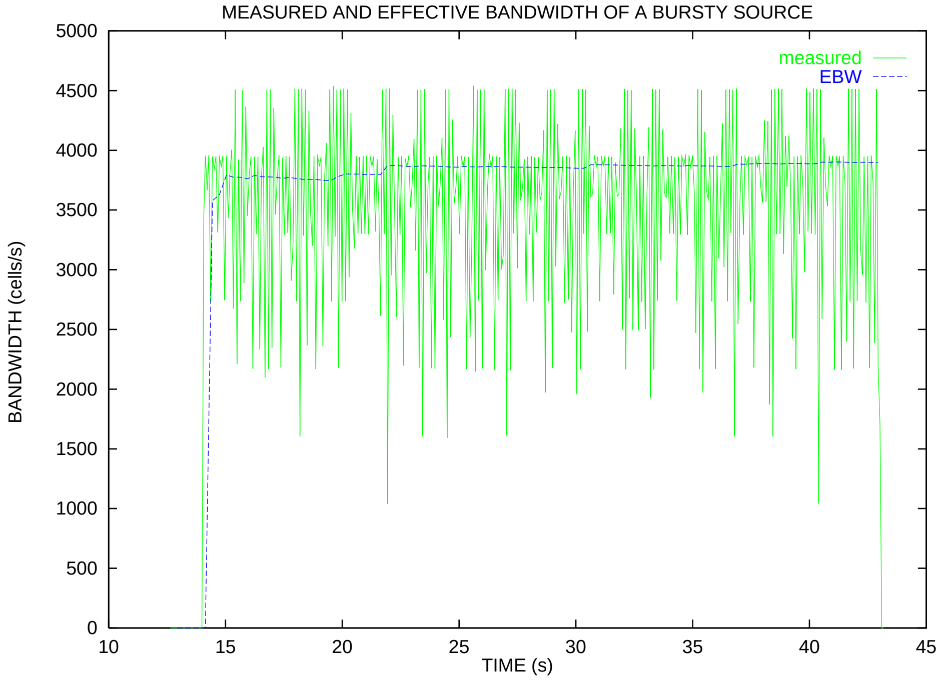


Figure 8.7: Effective bandwidth of a single bursty source

the connection has left, the measurements from the switch for this VPI/VCI pair are now ignored and the effective bandwidth calculation for the connection stops.

Figure 8.8 shows a similar trace for a JPEG video transmission. The video data was generated by an ATM camera (FORE AVA-200 [**Pratt94**]). The video was a full colour video stream with a frame size of 768x288 pixels and a Q factor of 32. The frame rate was 10 frames per second. All video transmissions that were used in the experiments had these same properties, except for the frame rate, which was varied to differentiate between video streams. Again the effective bandwidth stays well below the peak rate.

8.4.2 Effective bandwidth of multiple sources

Next, the aggregation of multiple sources is considered. For example, the short trace in Figure 8.9, shows the measured and effective bandwidth of two JPEG video streams. Both video streams are generated by ATM cameras with the same properties as the video stream of Figure 8.8. The first stream had a rate of 12 frames per second, while the second ran at 5 frames per second. The first stream is active between $t = 1$ and $t = 15$, while the second stream is added at $t = 5$ and removed again at $t = 10$. Again, it is noteworthy that the effective bandwidth estimation shoots up quickly, the moment the second stream is added and drops immediately after the second stream has left. This is exactly the desired behaviour.

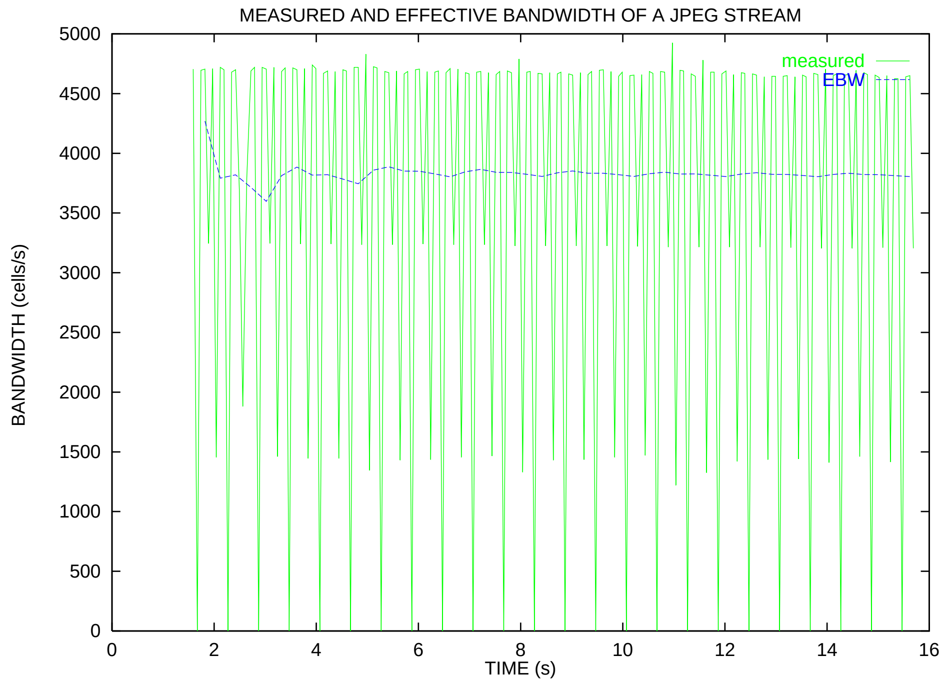


Figure 8.8: Effective bandwidth of a single JPEG video stream

8.4.3 Using effective bandwidth for CAC decisions

As a simple demonstration of how the EBW estimates by the traffic servers may influence CAC decisions consider the following scenario. A set of requests is submitted to the Sandman. The resource of interest is a switch port shared by all calls. The relevant parameters are shown in Figure 8.10. All three data streams are created by traffic generators.

The sum of the peak rates of source 1 (5000 cells/s) and source 2 (4000 cells/s) exceeds the total capacity of the virtual network for this port (8000 cells/s). At $t = 0$, an attempt is made to reserve bandwidth for three calls, but since at this point the CAC is based on peak rates only (no traffic has been seen yet), the request for source 2 (which overlaps with source 1) is rejected. Figure 8.11 shows the actual traffic of the flows[1].

The total traffic on the port is shown in Figure 8.12. Also shown is the aggregate EBW which lies well below the peak rates. Since the CAC algorithm uses the EBW of active calls rather than their peaks, it makes a less conservative admission decision some time after the first connection starts. For example, it can be observed in Figure 8.10 that when another attempt is made to reserve bandwidth for source 2, this time at $t = 50$, the request is accepted (see also Figure 8.12). The EBW of the active connection (< 4000) plus the peak of the new request (4000) does not exceed the resource capacity. Figure 8.12 shows

[1]Although there are three connections, only two VCI values are used. Since the third connection starts after the first connection has ended, it reuses its VCI.

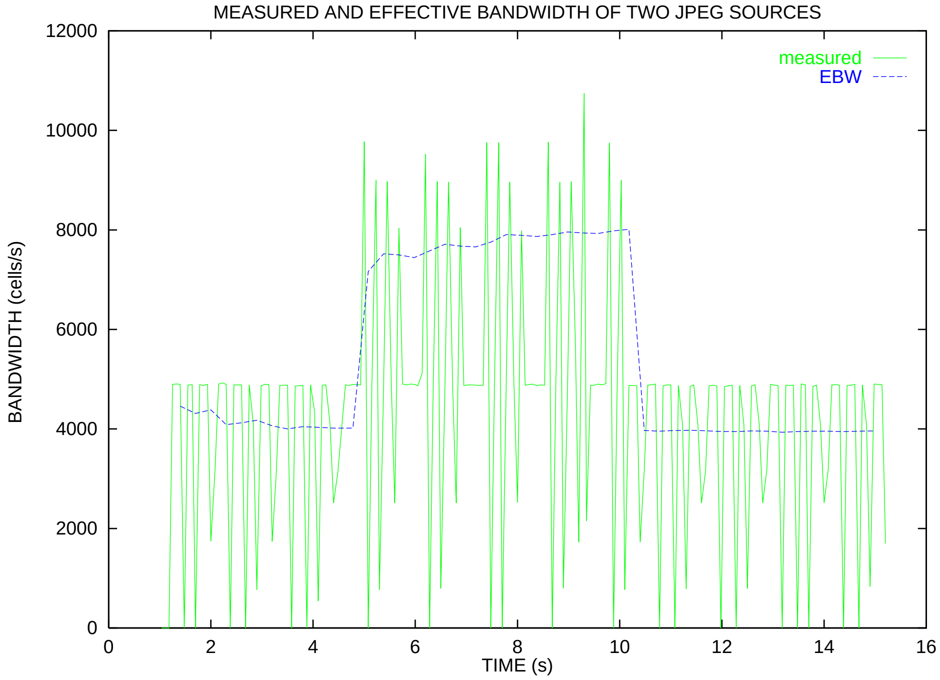


Figure 8.9: Effective bandwidth of two video streams

TIME (s)	RESERVATION	INTERVAL	CAC RESULT
0	SOURCE 1	[1, 300]	ACCEPT
	SOURCE 2	[150, 300]	REJECT (!)
	SOURCE 3	[300, 450]	ACCEPT
50	SOURCE 2	[150, 300]	ACCEPT (!)

Resource capacity: 8000
Peak (source 1): 5000
Peak (source 2): 4000
Peak (source 3): 7000

Figure 8.10: Requests, peak rates and capacity

that the acceptance is justified—the total traffic never exceeds the capacity and, even for such a small number of sources, resource utilisation improves considerably.

Observe that the aggregate EBW rises logarithmically, when the second source is added. If this plot is compared to previous diagrams, one might suspect that the estimated aggregate effective bandwidth is not the same as the sum of the effective bandwidths of the two individual connections. Indeed, it is not: the aggregate EBW lies significantly below this sum. This constitutes the multiplexing gain. Another interesting thing is the decay of the EBW in the trace of the third source. This source starts with a burst of traffic, which pushes the effective bandwidth up to well over 6500 cells per second. After this burst, the data rate of the source drops. What is interesting is that the effective bandwidth drops in an exponential way. The first drop is by a large step, but after a while this tails off. This is similar to the way the effective bandwidth is pushed up when the data rate increases.

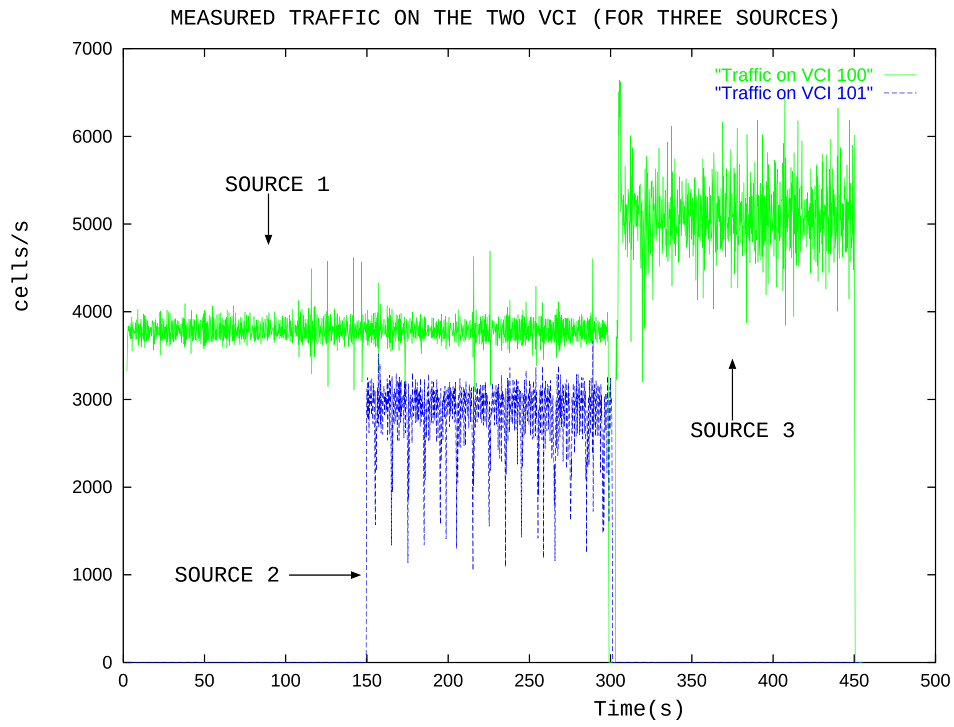


Figure 8.11: Trace of three traffic sources

Using an estimation of effective bandwidth based on online measurements to make CAC decisions is an active research topic. Related work in the field is discussed in Section 5.6.3. It has been shown that a traffic server's effective bandwidth estimates can help improve the resource utilisation considerably, especially in systems that would otherwise use the declared peak rate of connections for CAC. Making traffic servers elastic allows a client to specify exactly what sort of measurements and estimates it wants and when it wants them.

8.5 Summary

The performance results discussed in this chapter show two things, namely that: (1) running network control off-board need not be slower than running the controller on the switch, and (2) control performed by (interpreted) DLAs need not be slower than control performed by static C/C++ code, due to the possibility to apply application-specific optimisations on the fly. The connection setup times measured for the *Haboob* were faster than even the best results published in the literature for comparable equipment. The call admission control experiments show that using effective bandwidth estimates instead of peak rates for CAC improves resource utilisation considerably.

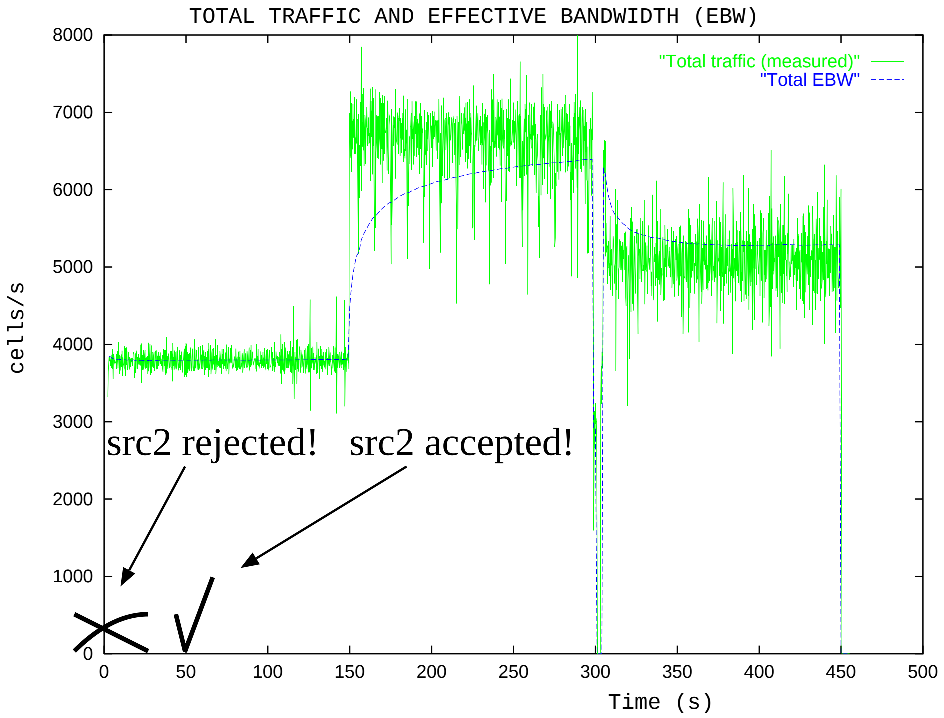


Figure 8.12: Total traffic and effective bandwidth

Chapter 9

Conclusion

In this dissertation, an open and extensible way to control and manage ATM networks is presented. The aim is to allow for easier development and introduction of innovative network control and services than hitherto possible. This is done by providing a set of basic components which users can combine, modify and extend on the fly, according to their own needs.

9.1 Summary

The dissertation concerns itself with elastic network control, which is defined as network control software that is distributed across a network and consists of multiple interacting components. The interaction takes place across well-defined interfaces, which are open and public. *Elasticity* means that the components can be dynamically modified and extended. An implementation of elastic network control has been described in the form of the *Haboob*.

In Chapter 1 the inadequacies of existing approaches to network control are pointed out. Standards-based approaches are shown to be both too complex and too limited. A more promising approach is what is known as open control with support of multiple control architectures in the same physical networks. A first attempt at such control is made with the open system support architecture (OSSA). Although the OSSA and related projects elsewhere are recognised as important steps towards open network control, they lack flexibility in that it is hard to extend or override the functionality of the components. However, this type of flexibility is slowly emerging in active networks technology. This dissertation can be viewed as a bridge between active networks and open signalling. This is illustrated in Figure 9.1. Chapter 2 summarises the *Tempest* and the research context.

In Chapter 3, both the network model and the computational model for dynamically loadable code are discussed. The network model consists of de-

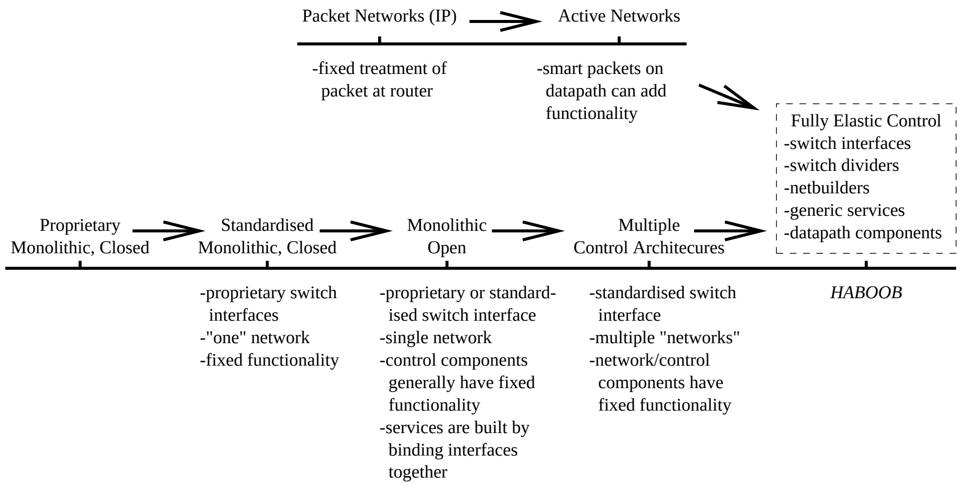


Figure 9.1: The *Haboob* in the evolution of communication networks control and management

composing control in six independent levels, namely:

1. a generic switch control interface, known as *Ariel*;
2. *switch dividers* that partition switch resources into *switchlets*;
3. *netbuilders* that combine switchlets into virtual networks;
4. *control architectures* that perform the actual control over the virtual networks;
5. *generic services*, such as traders and traffic servers, that can be used by all other components in the system;
6. *datapath components*, which are either endpoints of data streams partaking in control, or data processing units located on the datapath (in-band control).

The computational model is described following the Sandbox framework. The Sandbox is an execution environment in which dynamically loadable agents (DLAs) can be loaded and executed. Functionality in the Sandbox is provided by modules that have well-defined interfaces. The basic execution environment provides one standardised module, known as SUFI (for "simple uniform framework for interaction"), which enables DLAs in different Sandboxes to interact. Only interfaces are defined, so that as much implementation independence as possible is retained. Inter-DLA interaction takes place either via RPC or remote evaluation. An implementation of the Sandbox is discussed in Chapter 4. DLAs execute in sand-boxed environments: their access to resources is limited and they are not able to interfere with either the underlying system or DLAs in different Sandboxes.

In Chapter 5, elastic control architectures are discussed. Control architectures can be extremely complex and many issues are touched upon in this chapter in the context of an advanced home-grown control architecture, called

Sandman. They include advance reservation of resources, call admission control based on measurements, and the problem of interdependent connections. The call admission control algorithm admits or rejects requests based on declared peak rate only, but uses an estimation of the effective bandwidth on active connections as well. New connection types are introduced for restricted multipoint-to-point and multipoint-to-multipoint connections. It is shown how the resources owned by a control architecture can be repartitioned recursively using *netlets*. An application that creates a netlet is considered the 'owner' of that netlet and is given low-level control over the netlet's resources.

It is also permitted to load control code in a Sandbox in the control architecture, in the form of a DLA that corresponds to a netlet. Netlets can be seen as light-weight virtual networks. To prove the validity of the Sandman, a distributed video server, called *BigDisk*, is implemented. *BigDisk* allows owners of cheap and limited disk space to record feature-length video files, by chopping the file up in segments and spreading the segments across several disks on the local network, according to a load-balancing policy. The second type of control architecture in this dissertation are control architectures of the *Noman* family, consisting solely of dynamically loadable code. Noman is defined as a programmable control architecture *framework*, which offers a simple API that allows DLAs to build their own control architecture.

In Chapter 6, the problem of control architecture interoperability is tackled. Traditional approaches towards interoperability were shown to suffer from two majors flaws. First, there is a problem of functionality degradation which arises when requests from a feature-rich control architecture are translated into those of a simpler control architecture at the domain boundary. As a solution, simple interoperability is used to establish an inter-domain signalling channel between two feature-rich control architectures. The second problem consists of fixed functionality mapping. There are often multiple ways to map a request of a control architecture into that of another. In traditional solutions for interoperability this mapping is predetermined and fixed. However, only end applications know what the best mapping in their application area is. To solve this problem, applications are allowed to specify their own code to perform the interoperability mapping.

Elastic versions of the generic switch interface, the switch divider and the netbuilder are discussed in Chapter 7. Starting with the switch interface, in the *Haboob*, applications are permitted to load DLAs that run directly on top of their switchlets, making calls to switchlet operations (over the *Ariel* interface) in the same address space as the divider itself. It is also possible to extend *Ariel* itself, using *native methods*. At the divider level, two types of DLA are permitted. The first type controls the behaviour of the entire switch divider and only privileged users should be allowed access to this Sandbox. The other type is more forgiving, corresponding only to the management operations of a particular switchlet. It is shown how aggregate switchlets could be built using DLAs. The elastic netbuilder allows both system administrators and clients with fewer privileges to influence the way virtual networks are built. As an example it is shown how "normal" active network nodes were built on the fly.

In Chapter 8, the proof-of-concept *Haboob* is tested for performance. It is shown that the connection setup times of the *Haboob* compare favourably with those of existing, commercial control architectures. It is also shown that using effective bandwidth for call admission control increases network utilisation significantly.

9.2 Future work

One particular problem that needs to be addressed thoroughly concerns off-switch call admission control. Vendors are generally reluctant to publish the internals of their switches. This makes it hard to map QoS onto resources, when performing CAC off-switch. This is a problem that is not specific to the *Haboob*. All open signalling approaches that perform CAC externally suffer from it. Furthermore, the call admission control experiments show that there is a lot to be gained by using effective bandwidth estimates instead of peak rates for CAC. Open issues that need to be resolved, however, include pathological, but still realistic, scenarios, e.g. in which many sources increase their resource utilisation all at once and by a large amount.

It would be interesting to port the *Haboob* to other connection-oriented network technologies. In the Internet, the RSVP protocol allows for resource reservation on the datapath. It should be possible to allow networks supporting a modified version of RSVP to be controlled by an IP version of the *Haboob*. Similarly, IPv6 flows could be controlled with control components in the *Haboob* [**IETF98**]. In this case, it would be much harder to give strict QoS guarantees. However, it would still allow the exploitation of application-specific knowledge. As a first step in this direction, new versions of the *Ariel* and divider interfaces have been designed, that allow a more general treatment of QoS issues and are less ATM specific.

9.3 Conclusion

It is the thesis of this dissertation that open elastic network control provides users and application programmers a convenient means to build applications and control services that hitherto were difficult or impossible to realise. This has been achieved in the sense that network control policies can be modified and extended dynamically, down to the processing of individual packets (as in active networks), while retaining in general the clear separation between control and datapath. Recognising the limitations of previous approaches to open signalling, it is an attempt to completely open up the control of networks. This includes not only connection setup and such operations, but also partitioning of resources, building virtual networks, design and implementation of control architectures, generic services and even datapath components.

In particular, a carefully designed Sandbox, that is independent both of the underlying implementation of the execution environment and of the language used for the dynamically loadable code, allows elastic behaviour to be introduced in many aspects of network control. Sandboxes contain a simple standardised module which allows them to interoperate. This functionality can be extended easily with new modules. All interaction takes place via well-defined interfaces, which allows independent development of the individual components. This permits application-specific knowledge to be exploited at various levels of network control. A number of examples, that to the author's knowledge cannot be implemented using existing network control, were built to support the thesis. These include the possibility to program application-specific policing, interoperability and reservation behaviour, of partitions of a network. It was further shown, that performance need not suffer from elastic network control. Furthermore, the additional functionality that is gained, equals or exceeds that of other approaches to programmable network control.

It is concluded that the dissertation supports the thesis. Furthermore, adding such flexibility as offered by the *Haboob*'s elastic network control is essential, if the rapid development and deployment of application-specific services and control mechanisms is important. This work is strongly at odds with, and more flexible than, standardisation efforts which aim to provide an all-encompassing control solution, using fixed functionality that is set in stone.

Appendix A

Estimating the effective bandwidth

In Section 5.2, a call admission control (CAC) algorithm was described, which was based (partly) on the estimation of the effective bandwidth (EBW) using online measurements. This was called measurement-based admission control (MBAC). The next two sections will mathematically derive an expression for the EBW corresponding to a desired cell loss ratio (CLR).

A.1 Introduction

The Sandman EBW estimation is based on a branch of statistics known as Large Deviations Theory [**Weiss95**]. The EBW estimate is similar to [**Crosby95b**] with a few modifications. In this section, the relevant results are summarised. It is important to realise that various other MBAC schemes have been proposed in the literature. A discussion of related work on MBAC is given in Section 5.6.3.

In the following discussion, the next few parameters are very important. The aim is to find r, the effective bandwidth of a connection, or group of connections, which corresponds to a target CLR p. The target CLR p represents the probability that a cell is lost due to buffer overflow. The size of this buffer is denoted by b (for example, in a FORE ASX-100, the size of the switch buffer is 256 cells).

A.2 Effective bandwidth estimation

In an environment where a buffer is served at constant rate r_{tot}, the workload process W_t (which essentially denotes the total number of cell arrivals minus the number of departures) is defined as:

$$W_t = \sum_{i=1}^{t} X_i - t \times r_{tot}$$

where $\{X_i\}$ is the number of arrivals in interval i and $\{X_i\}$ independent and indentically distributed (iid)[1]. It is a well-known result from queueing theory that the queue length Q of a buffer depends on W_t as follows: $Q = max\,\{W_t : t \geq 0\}$.

Under general conditions, a single-server queue has a queue length distribution (which is derived from the workload) with asymptotes of the form:

$$\begin{aligned} &P(W_t > x) \asymp e^{-tI(x)} \\ &\Rightarrow P(Q > q) \asymp e^{-\delta q} \end{aligned} \tag{A.1}$$

where $I(x)$ is the rate function of the workload process and δ is called the decay rate. It is not hard to see that δ is a function of r. Note that setting $q = b$ in equation A.1 yields the CLR, provided b is sufficiently large. Measurements on real switches (with finite buffer space) have shown that for realistic buffer sizes, the log linear approximation of equation A.1 holds. In that case, equation A.1 gives a good approximation for the CLR. Now, the rate r that corresponds to a specific CLR can be calculated. Define λ, a transform of I called the *scaled cumulant generating function* (SCGF), of the workload process as follows:

$$\lambda(s) = \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{sW_t}\right) \tag{A.2}$$

which is related to I by the Legendre transform:

$$I(x) = max_s\,\{xs - \lambda(s)\} \tag{A.3}$$

Note that r is constant. In other words, $W_t = \sum_{i=1}^{t} X_i - r \times t$. Then:

$$\begin{aligned} \lambda(s) &= \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{s(\sum X - r\times t)}\right) \\ &= \lim_{t\to\infty} \frac{1}{t} \ln E\left(e^{s(\sum X)}\right) - s \times r \\ &= \lambda_A(s) - s \times r \end{aligned} \tag{A.4}$$

[1] in fact, this can be relaxed to 'weakly dependent and stationary', but this is beyond the scope of this discussion

where λ_A is the SCGF of the arrivals process. So, given the arrivals SCGF we can calculate δ as function of r as follows:

$$\delta(r) = max\{s : \lambda_A(s) \leq s \times r\} \tag{A.5}$$

This result can be used to find r, the service rate corresponding to a particular δ, i.e. to the target CLR. The CLR is really the probability of buffer overflow. Assuming the buffer size is b, we can use (A.1):

$$\begin{aligned} P(Q > b) &\asymp e^{-\delta(r)b} \\ \Rightarrow r = min &\left(\delta(s) \geq -\frac{\ln CLR}{b}\right) \end{aligned}$$

From (A.5) we know that $r \geq \lambda(\delta)/\delta$. Substitute $\delta = -\frac{\ln CLR}{b}$:

$$r = \frac{\lambda\left(-\frac{\ln CLR}{b}\right)}{\left(-\frac{\ln CLR}{b}\right)}$$

and it is easy to prove[2] that $lambda(-x) = -lambda(x)$. So for a target CLR of p, the required service rate can be calculated as follows:

$$\rightarrow r = (\lambda((\ln p)/b))/((\ln p)/b) \tag{A.6}$$

The function $\lambda(\theta)/\theta$ is called the effective bandwidth. To estimate λ we use the following: for a large class of arrival processes it is possible to find a block length T such that the aggregated arrivals A_T are approximately iid. The arrival process is therefore broken up into intervals of length T. The number of arrivals in the i^{th} block of length T is denoted by $A_T^{(i)}$. Then:

$$\begin{aligned} \lambda(s) &\approx \frac{1}{T}\ln E\left(e^{sA_T}\right) \\ \text{so}: \quad \hat{\lambda}(s) &= \frac{1}{T}\ln\frac{1}{N}\sum_{i=1}^{N} e^{sA_T^{(i)}} \end{aligned} \tag{A.7}$$

gives an estimation λ and hence of the EBW. The choice of block size T is important. It should be large enough to make arrivals in the interval independent, but not so large that short bursts are smoothed out. It is beyond the scope of this dissertation to derive the ideal value for T. In the experiments, T was set to values between 75 and 200 milliseconds.

[2] X_n iid, so simply work out $lambda(-x)$ using equation A.2, in the same way as in A.4

A.3 Discussion

From equation A.7, it becomes clear that according to theory the effective bandwidth estimation relies on the complete history of all arrivals. In other words, the aggregate EBW is estimated from all arrivals in all blocks of length T since the start of day (observe though, that it is sufficient to store only the partial sum, instead of the entire history). However, this means that also measurements from connections that are really no longer active (even though the switch connection may still be there) are included in the calculation of the aggregate EBW. To remedy this, measurements are taken per *active* connection, where the client of the traffic server determines when connections are activated and deactivated. In this case, this client will be the Sandman control architecture, but the traffic server is a generic service and could serve other clients as well. Summarising, the traffic servers receive statistics from the switch for each active connection individually (and also for the aggregate of all connections). When connections leave, the measurements corresponding to them will be discarded.

Appendix B

Estimating the follow-up latency

In Section 5.3.4, the latency in the BigDisk distributed video server was discussed. It was mentioned there that the estimation of the follow-up latency was important to prevent glitches in the playback of a video. A brief sketch of the mechanism for this estimation and its assumptions is given below.

The approach taken in an experimental version of the Sandman is to give probabilistic upper (and lower) bounds on the follow-up playback latency, across a single switch and potentially even end-to-end. What this means, is that BigDisk clients are allowed to specify a latency probability bound (lpb) which they submit to the control architecture. The control architecture is then able to calculate the corresponding delay D_{lpb}. For example, an lpb of 99% means that the client wants to know the delay bound that 99 % of the traffic stays under. In other words: $P(delay > D_{0.99}) \leq 0.99$. This delay bound can be used to determine the size of the buffer required to guarantee smooth playout. The next few sections will detail how this is done.

B.1 Model and assumptions

Figure B.1 shows what components make up the follow-up latency. It is assumed that the network-related follow-up latency is comprised of the following components:

1. teardown time of the first connection on the cut-off switch;
2. communication delay for teardown message;
3. setup time on the cut-off switch (the connections on the other switches on the path to the new source are set up just prior to the handoff);

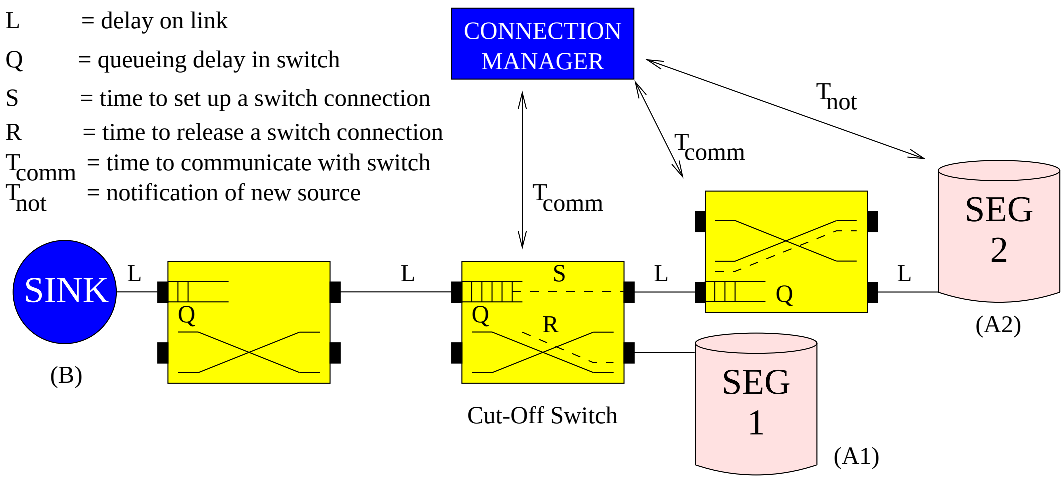


Figure B.1: Follow-up latency model

4. communication delay for setup message;
5. delay incurred by notifying the new source;
6. end-to-end network delay of data:
   (a) queueing delay in the switches;
   (b) delay 'on the wire'.

Consequently, the latency in the endpoints (e.g. of getting the data from disk), is not included. This is only known at the application level and should be dealt with there also. The following simplifying but not unreasonable definitions and assumptions are made.

1. The time it takes to set up or tear down a connection on a switch i is bounded by $T_s(i)$.
2. The delay on link i is bounded by $T_l(i)$.
3. The communication delay for communication with a specific host/switch i (or notification of host i) is bounded by $T_{rpc}(i)$.
4. The queueing delay on switch i is denoted by $T_q(i)$ and is unknown.
5. There are $S_{tot}(i,j)$ switches between endpoints i and j (and $S_{tot}(i,j)+1$ links).

In reality, T_{rpc} depends on the delay in the switches as well. Things have been simplified here to keep the equations short, but it is not difficult to extend the model to take this into account. Also, in reality the notification should take only $0.5T_{rpc}$, as the new source will start sending immediately upon being called back[1].

B.2 Deriving latency

The teardown of the old source connection beyond the cut-off switch takes zero time, because it is deferred until after the new source has started sending. We can now define the latency L incurred by a source replacement. In Figure B.1, source $A1$ is replaced by $A2$ at cut-off switch s_c and the sink is called B. Define $path_{[a,b]}(i)$ as the i^{th} element on the path from a to b (this can be either a switch or a link, depending on the context). The latency L is defined as follows:

$$\begin{aligned} L = {} & T_{rpc}(A2) + 2T_{rpc}(s_c) + 2T_s(s_c) \\ & + \sum_{i=1}^{S_{tot}(A2,B)+1} T_l(path_{[A2,B]}(i)) \\ & + \sum_{i=1}^{S_{tot}(A2,B)} T_q(path_{[A2,B]}(i)). \end{aligned} \tag{B.1}$$

The value of the variables in equation B.1 are constant and known, except for the values of the queueing delay, which depend on the other traffic. In other words, $L_{total} = L_{const} + L_{queue}$. For L_{queue}, a target probabilistic bound will be derived, i.e. a bound of the form: 'less than 100 ms with probability 0.95.'

B.3 Large bounds on queueing delay

There are many ways to derive bounds on the queueing delay on switches. Often a specific traffic model is presumed. This dissertation argues against the validity of such an approach and in section A.2, a method to arrive at values for effective bandwidth corresponding to a desired CLR was discussed, which does not need such accurate source characterisation. The method was based on large deviation theory, as described in [**Weiss95**]. A similar approach is used here, to obtain probabilistic bounds on the queueing delay. Note that both CLR and queueing delay depend on the queue length distribution. Define Q to be the length of a queue, D to be the delay that a cell in this queue experiences, B_i to be the size of the buffer[2] on switch i, and $r_{svc}(i)$ to be the service rate of this queue. As described in A.2, large deviations theory states that under very general circumstances (and when q is sufficiently large):

[1]In fact, a similar optimisation could be made for communication with the switches, if simple non-blocking messages are sent to the switches

[2]for the particular connection class of interest

$$
\begin{aligned}
& P(Q > q) \approx e^{-\delta q} \\
\Rightarrow\ & P(D > d) \approx e^{-\delta \cdot d \cdot r_{svc}}
\end{aligned} \tag{B.2}
$$

So, if we pick d_i^*, the lower bound for delay switch i, reasonably large (large enough for large deviations theory to be applied), we can obtain the probability distribution of the queueing delay across a single switch[3]. Define $\hat{d}_i$ of a switch i as the maximum queueing delay on this switch. It can be easily seen that: $\hat{d}_i = B_i / r_{svc}(i)$. And so the maximum queueing delay over n switches can be defined as $\hat{D} = \sum_i^n \hat{d}_i$.

Observe that the queueing process here is essentially a single server FIFO queue. Things change somewhat if per-VC (or per connection class) queueing is used. It is assumed that in such a case, each connection (class) has a relatively fixed buffer size that is FIFO in nature and the service rate at which the queue is drained is a relatively constant fraction (weight) of the total capacity. Under these assumptions, the same methods can be applied as in the case of a single FIFO queue (albeit with different buffer size and service rate).

It is now possible to get a handle on the probability distribution of the end-to-end queueing delay. A simple demonstration for providing latency guarantees was built, which made the simplifying assumption that the probability distribution of the queueing delay in the switches is independent[4]. Then the probability that the delay across n switches is greater than t_Q, where $t_Q > \sum_i^n d_i^*$ follows:

$$
P(D_{tot} > t_Q) \leq \sum_{t_Q \leq d \leq \hat{D}} \left\{ \prod_{d1 + \ldots + d_n = d} P(D_i = d_i) \right\} \tag{B.3}
$$

where

$$
P(D = d) = P(Q \geq d \times r_{svc}) - P(Q \geq d \times r_{svc} + 1) \tag{B.4}
$$

Using (B.2) and the derivations of δ and λ from appendix A, the desired probabilistic upper bound on the end-to-end delay has been found. All variables can be determined (see Appendix A.2). An application can tune its buffer to a particular delay probability. It will look at the total latency and determine the desired probability for buffer underflow, i.e. the buffer should be big enough

[3]It's beyond the scope of this dissertation to determine the minimum value for d_i^*. Interested readers are referred to [**Weiss95**].

[4]Although an over-simplification, it is not as unreasonable as it may seem: it can be shown that it holds for both M/M/1 and Geom/Geom/1 queues (as demonstrated in [**Hui90**], Chapter 7).

to hold at least L_{const} milliseconds of video plus the L_{queue} that corresponds to this desired underflow probability. Note that since d, the end-to-end delay bound, should be larger than $t_q > \sum_i^n d_i^*$, we are only able to determine the probability distribution for short end-to-end delays in connections with a small number of hops. For connections with many hops, we only get the distribution for relatively long delays. This is probably reasonable, as it matches the delay we are likely to experience in the real world. The derivation of the probability distribution of short delays (where large deviations theory cannot be aplied), is the topic of ongoing research.

Appendix C

Example of temporary reservations

In Section 5.4.4.2, CAC DLAs are introduced, which make temporary reservation at CAC time. A brief example illustrating the process is presented here. When a new request arrives at the Sandman, i.e. when Sandman needs to make a CAC decision for a switch port, it checks whether a CAC DLA was installed, and if so, executes it. The loadable CAC code receives from the control architecture the time interval $[T_a, T_b]$ corresponding to the new reservation request. The DLA then makes *temporary reservations* according to its own reservation policy for this time interval. After the loadable code has made all the reservations it wants to make for this interval, it simply returns. The CAC algorithm now makes its decision based on all reservations currently in the schedules, including the temporary ones. The temporary reservations disappear automatically when the CAC procedure returns.

As an example, consider a contrived application A_1 that wants to reserve bandwidth proportional to what day of the month it is. So, if it is the first of that month, the application reserves B bandwidth, next day it reserves $2B$, the day after that $3B$ and so on, until it wraps back to B, when the month changes. Also assume that the application's request to install loadable code corresponding to this resource reservation behaviour on a certain switch port was accepted. Suppose finally that the total capacity of that switchport is B_{tot}, that the amount of bandwidth reserved statically (in reservation schedules) is B_{resv} and that the available bandwidth at some point in time is B_{avail}. Now, a new application A_2 submits a request to reserve statically B_{req} bandwidth on this port from the morning of July the 5th until the night of the July the 6th. What happens is the following:

1. The CAC procedure discovers that loadable CAC code exists for this port and executes it.

2. The loadable CAC code makes temporary reservations for $5B$ bandwidth

on the 5th and $6B$ bandwidth on the 6th of July and returns control to the default CAC procedure.

3. The default CAC procedure now simply checks its schedules and accepts the new reservation request if at all times during the two days of interest $B_{avail} = B_{tot} - Bresv \geq B_{req}$, and rejects it otherwise.

4. The CAC result is returned (the temporary reservations disappear automatically).

In this way, any application-specific reservation policy can be implemented. The DLAs for different applications are independent and can be executed in any order. An interesting problem is how one should charge for these reservations. One option is to charge client charged a predetermined fee that is determined when the DLA is loaded, pretty much like a contract. Alternatively, one could opt for more adventurous schemes, e.g. to charge clients according to how often their reservations overlap with those of other requests.

Bibliography

[Adl-Tabatabai96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. *Efficient and Language-Independent Mobile Programs.* In Proceedings of ACM SIGPLAN'96 Symposium in Programming Language Design and Implementation (PLDI), pages 127 – 136, May 1996. (pp. 38, 59)

[Adobe85] Adobe. *PostScript Language Reference Manual.* Adobe Systems, Inc., Reading, Massachusets, USA, 1985. (p. 61)

[Alexander97] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan Smith. *Active Bridging.* In Proceedings of ACM SIGCOMM'97, Cannes, France, September 1997. (pp. 49, 134)

[Alexander98a] D. Scott Alexander, , Michael Hicks, Pkaj Kakkar, Angelos Keromytis, Marianne Shaw, Jonathan Moore, Carl Gunter, Trevor Jim, Scott M. Nettles, and Jonathan Smith. *The SwitchWare Active Network Implementation.* In Proceedings of the 1998 ACM SIGPLAN Workshop on ML, 1998. (pp. 49, 133)

[Alexander98b] D. Scott Alexander, William Arbaugh, Angelos Keromytis, and Jonathan Smith. *Safety and Security of Programmable Network Infrastructures.* IEEE Communications Magazine, 36(20):84–92, October 1998. (p. 39)

[Alexander99] Scott Alexander, William Arbaugh, Angelos Keromytis, and Jonathan Smith. *Security in Active Networks.* Secure Internet Programming: Issues in Distributed and Mobile Object Systems, Springer-Verlag Lecture Notes in Computer Science "State of the Art" Series, 1999. (p. 55)

[Alles95] Anthony Alles. *ATM Internetworking.* Internal Paper, Cisco Systems, Inc (an abridged version was published in the Proceedings of Engineering InterOp, Las Vegas, March 1995), May 1995. (p. 116)

[**Anderson92**] D. P. Anderson, Y. Osawa, and R. Govindan. *A File system for Continuous Media.* ACM Transactions on Computer Systems, 10(4):311–337, November 1992. (pp. 78, 105)

[**Appleby94**] S. Appleby and S. Steward. *Mobile Software for Control in Telecommunications Network.* BT Technology Journal, 12(2):104–113, April 1994. (p. 35)

[**Baker97**] Fred Baker, Jon Crowcroft, Roch Guerin, Henning Schulzrinne, and Lixia Zhang. *Reservations about reservations.* In Proceedings of IFIP Fifth International Workshop on Quality of Service (IWQOS '97), New York, NY, May 1997. (p. 65)

[**Battou96**] Abdella Battou. *Connection Establishment Latency: Measured Results (ATM Forum document number T1A1.3/96-071).* Technical Report, Centre for Computational Science, Naval Research laboratory, Washington, DC, October 1996. (pp. 139, 144, 145)

[**Bearman91**] Mirion Bearman and Kerry Raymond. *Federating Traders: an ODP Adventure.* In Proc. IFIP TC6/WG6.4 International Workshop on Open Distributed Processing, pages 125–141, Berlin, Germany, October 1991. North-Holland. (pp. 24, 60)

[**Bellcore97**] Bellcore. *Q.Port Portable ATM Signalling Software.* Product Information, 1997. (pp. 120, 144)

[**Bettati95**] R. Bettati, D. Ferrari, A. Gupta, W. Heffner, W. Howe, M. Moran, Q. Nguyen, and R. Yavatkar. *Connection Establishment for Multi-Party Real-Time Communication.* In NOSSDAV, pages 255–265, Durham, New Hampshire, April 1995. (pp. 12, 67)

[**Bhattarcharjee97**] S. Bhattarcharjee, K. Calvert, and E. Zegura. *Active Networking and the End-to-End Argument.* In Proceedings ICNP'97, Atlanta, GA, April 1997. (p. 50)

[**BICI94**] BICI. *Broadband Intercarrier Interface (B-ICI) Specication Version 1.1.* ATM Forum Standard, July 1994. (p. 98)

[**Birrell84**] A.D. Birrell and B.J. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1):39–59, February 1984. (p. 31)

[**Biswas97**] Subir Kumar Biswas and Andy Hopper. *An Agent-based Signaling Architecture for supporting Mobility in Radio ATM Networks.* International Journal of Communication Systems, 10:87–101, 1997. (pp. 20, 120)

[**Biswas98**] Jit Biswas, Aurel Lazar, Jean-Francois Huard, Koongseng Lim, Semir Mahjoub, Louis-Francois Pau, Masaaki Suzuki, Soren Torstensson, Weiguo Wang, and Stephen Weinstein. *The IEEE 1520 Standards Initiative for Programmable Network Interfaces.* IEEE Communications Magazine, 36(10):64–70, October 1998. (p. 15)

[**Borenstein94**] N. Borenstein. *Email with a mind of its own: The Safe-Tcl Language for Enabled Mail.* In IFIP International Conference, Barcelona, Spain, 1994. (p. 56)

[**Bos97**] Herbert Bos. *An Active Distributed File Server For Continuous Media.* In Proceedings of the 2nd European Research Seminar on Advances in Distributed Systems, pages 93–98, March 1997. (pp. i, 82)

[**Bos98a**] Herbert Bos. *ATM Admission Control based on Measurements and Reservations.* In Proceedings of the IEEE International Performance, Computing and Communications Conference, pages 298–304, Phoenix, Arizona, February 1998. (pp. i, 100)

[**Bos98b**] Herbert Bos. *Building a Distributed Video Server using Advanced ATM Network Support.* In Proceedings of the second IFIP/IEEE International Conference on Management of Multimedia Networks and Services '98, Versailles, France, November 1998. (p. i)

[**Bos98c**] Herbert Bos. *Efficient Reservations in Open ATM Network Control Using Online Measurements.* International Journal of Communication Systems, 11(4):247–258, August 1998. (pp. i, 65)

[**Bos99a**] Herbert Bos. *Application-specific Behaviour in Distributed Network Control.* In Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems, April 1999. (pp. i, 55)

[**Bos99b**] Herbert Bos. *Application-Specific Policies: Beyond the Domain Boundaries.* In Integrated Network Management VI (IM'99), pages 827–840, Boston, USA, May 1999. Chapman & Hall. (p. i)

[**Bos99c**] Herbert Bos. *Open Extensible Network Control.* Journal of Network and System Management (accepted for publiction), 1999. (p. i)

[**Bosch96**] P. Bosch and S. J. Mullender. *Cut-and-Paste File Systems: Integrating Simulators and File Systems.* In Proceedings USENIX Conference, San Diego, California, January 1996. (p. 105)

[Breslau99] Lee Breslau, Sugih Jamin, and Scott Shenker. *Measurement-Based Admission Control: What is the Research Agenda?* In Proceedings of the 7th International Workshop on Quality of Service (IWQoS), pages 3–5, London, UK, May 1999. (p. 77)

[Brunner99] Marcus Brunner and Rolf Stadler. *The Impact of Active Networking Technology on Service Management in a Telecom Environment.* In Integrated Network Management VI (IM'99), pages 385–400, Boston, MA, USA, May 1999. IEEE/IFIP. (p. 100)

[Buckley98] William Buckley. *Virtual Switch Interface (VSI) Specification.* Technical Report, Multiservice Switching Forum, 1998. (p. 15)

[Callon99] R. Callon, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan. *A Framework for Multiprotocol Switching (Internet Draft draft-ietf-mpls-framework-03.txt).* Technical Report, IETF Network Working Group, June 1999. (p. 12)

[Calvert98] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. *Directions in Active Networks.* IEEE Communications Magazine, 36(20):72–78, October 1998. (p. 30)

[Cameron93] E. Cameron, N. Griffeth, Y-J Lin, M. Nilson, W. Schnure, and H. Velthuijsen. *A Feature-Interaction Benchmark for IN and Beyond.* IEEE Communications Magazine, 31(3):64–69, March 1993. (p. 47)

[Campbell99a] Andrew Campbell, Herman De Meer, Michael Kounavis, Kazuho Miki, John Vicente, and Daniel Villela. *A Survey of Programmable Networks.* ACM Computer Communication Review, April 1999. (p. 9)

[Campbell99b] A.T. Campbell, H.G. De Meer, M.E. Kounavis, K. Miki, J. Vicente, and D.A. Villela. *The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures.* In Proceedings of OPENARCH'99, New York, USA, March 1999. (p. 100)

[Cardelli95] Luca Cardelli. *A Language with Distributed Scope.* Computing Systems, 8(1):27–59, 1995. (p. 62)

[Cardoe99] R. Cardoe, J. Finney, A. Scott, and D. Shepherd. *LARA: A Prototype System for Supporting High Performance Active Networking.* In Proceedings of IWAN'99, pages 117–131, Berlin, July 1999. Springer-Verlag. (p. 49)

[Case96] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. *Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*. Request for Comments (Draft Standard) 1905, Internet Engineering Task Force, January 1996. (Obsoletes RFC1448). (p. 15)

[Chaney95] A.J. Chaney, I.D. Wilson, and A. Hopper. *The Design and Implementation of a RAID-3 Multimedia File Server*. In NOSSDAV, pages 327–338, Durham, New Hampshire, April 1995. (p. 105)

[Cidon95] I. Cidon, T. Hsiao, A. Khamisy, A. Parekh, R. Rom, and M. Sidi. *The OPENET Architecture*. Technical Report SMLI TR-95-37, Sun Microsystems Laboratories, December 1995. (pp. 11, 98, 99, 116)

[Cisco96] Cisco. *Frame Relay and ATM WAN Technology (White Paper)*. Technical Report, Cisco, 1996. (p. 42)

[Crosby95a] S. Crosby. *Performance Management in ATM Networks*. PhD dissertation, University of Cambridge Computer Laboratory, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., May 1995. Also available as technical report TR 393. (p. 12)

[Crosby95b] S. Crosby, J.T. Lewis, I. Leslie, N. O'Connell, R. Russel, and F. Toomey. *Bypassing Modelling: an Investigation of Entropy as a Traffic Descriptor in the Fairisle ATM Network*. In Proceedings of 1st Workshop on ATM Traffic Management, pages 139–146, February 1995. (pp. 10, 21, 73, 157)

[Dan95] A. Dan, M. Kienzle, and D. Sitaram. *A Dynamic Policy of Segment Replication for Load-Balancing in Video-on-Demand servers*. Multimedia Systems, 3(3):93–103, July 1995. (pp. 64, 105)

[Decina97] M. Decina and V. Trecordi. *Convergence of Telecommunications and Computing to Networking Models for Integrated Services and Applications*. Proceedings of the IEEE, 85(12), December 1997. (pp. 11, 102, 107)

[Degermark95] M. Degermark, T. Kohler, S. Pink, and O. Schelen. *Advance Reservations for Predictive Service*. In NOSSDAV, Lecture Notes in Computer Science, pages 3–14, Durham, New Hampshire, April 1995. Springer. (pp. 65, 103, 105)

[Delgrossi99] Luca Delgrossi, Giuseppe Di Fatta, Domenico Ferrari, and Giuseppe Lo Re. *Interference and Communications among Active Network Applications*. In Proceedings of IWAN'99, pages 97–108, Berlin, July 1999. Springer-Verlag. (p. 35)

[Diot97] C. Diot, W. Dabbous, and J. Crowcroft. *Multipoint Communication: A Survey of Protocols, Functions, and Mechanisms.* IEEE Journal on Selected Areas In Communications, 15(3), April 1997. (p. 67)

[Falcone87] Jospeh Falcone. *A Programmable Interface Language for Heterogeneous Distributed Systems.* ACM Transactions on Computer Systems, 5(4):330–351, November 1987. (p. 61)

[Federighi94] C. Federighi and L.A. Rowe. *A Distributed Hierarchical Storage Manager for a Video-on-Demand System.* In Proceedings of Symposium on Electronic Imaging Science & Technology, February 1994. (p. 105)

[Ferrari97] D. Ferrari, A. Gupta, and G. Ventre. *Distributed Advance Reservations of Real-Time Connections.* Multimedia Systems, 5(3):187–198, 1997. (pp. 65, 66, 104)

[Finin98] Timothy Finin, Yannis Labrou, and Yun Peng. *Mobile Agents can benefit from Standards Efforts on Interagent Communication.* IEEE Communications Magazine, 36(7):50–56, July 1998. (p. 40)

[FORE95a] FORE. *ForeRunner ASX-200, ATM Switch User's Manual.* Fore Systems Inc., 1000 Fore Drive, Warrendale, PA 15086-7502, USA, June 1995. (p. 137)

[FORE95b] FORE. *SPANS UNI: Simple Protocol for ATM Signalling, release 3.0.* Fore Systems Inc., 1000 Fore Drive, Warrendale, PA 15086-7502, USA, 1995. (p. 6)

[Franklin96] Stan Franklin and Art Graesser. *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.* In Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages. Springer-Verlag, 1996. (p. 26)

[Gaines72] R. Gaines. *An Operating System based on the concept of a supervisory computer.* Communications of the ACM, 15(3):150–156, March 1972. (p. 61)

[Garrahan93] James Garrahan, Peter Russo, K. Kitami, and R. Kung. *Intelligent Network Overview.* IEEE Communications Magazine, 31(3):30–36, March 1993. (pp. 1, 47)

[Georgatsos99] P. Georgatsos, D. Makris, D. Griffin, G. Pavlou, S. Sartzetakis, Y. T'Joens, and D. Ranc. *Technology Interoperation in ATM Networks: the REFORM Project.* IEEE Communications Magazine, 37(5):112–118, May 1999. (p. 121)

[Gibbens97] R.J. Gibbens and F.P. Kelly. *Measurement-based connection admission control.* In 15th International Teletraffic Congress Proceedings, June 1997. (pp. 73, 105)

[Goldszmidt95a] Germán Goldszmidt and Yechiam Yemini. *Decentralizing Control and Intelligences in Network Management.* Integrated Network Management IV (IM'95), May 1995. (p. 130)

[Goldszmidt95b] Germán Goldszmidt and Yechiam Yemini. *Distributed Management by Delegation.* In The 15th International Conference on Distributed Computing Systems, Vancouver, British Columbia, June 1995. (pp. 49, 127)

[Goldszmidt98] Germán Goldszmidt and Yechiam Yemini. *Delegated Agents for Network Management.* IEEE Communications Magazine, 36(3):66–70, March 1998. (pp. 8, 49)

[Gosling86] J. Gosling. *SunDew: A Distributed and Extenisble Window System.* In Proceedings USENIX, pages 98–103, January 1986. (p. 61)

[Gosling96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley ISBN 0-201-63451-1, 1996. (p. 56)

[Gray96] Robert Gray. *Agent Tcl.* In Proceedings of Fourth Annual USENIX Tcl/Tk Workshop, pages 9–23, 1996. (pp. 53, 61)

[Greenberg98] Michael Greenberg, Jennifer Byington, and David Harper. *Mobile Agents and Security.* IEEE Communications Magazine, 36:76–85, July 1998. (pp. 37, 39)

[Guerin91] R. Guerin, H. Ahmadi, and M. Naghshineh. *Equivalent Capacity and its Application to Bandwidth Allocation in High-Speed Networks.* IEEE Journal on Selected Areas In Communications, 9(7):968–981, September 1991. (pp. 73, 105)

[Hafid97] A. Hafid. *Providing a Scalable Video-on-Demand System using Future Reservation of Resources and Multicast Communication.* In Proceedings of the 5th IFIP International Workshop on QoS (IWQOS), May 1997. (p. 104)

[Hafid98] Abdelhakim Hafid, Gregor von Bochmann, and Rachida Dssouli. *Quality of Service Negotiation with Present and Future Reservations: A detailed Study.* Computer Networks and ISDN Systems, 30(8), May 1998. (pp. 65, 66)

[**Halls97**] David Halls. *Applying Mobile Code to Distributed Systems.* PhD dissertation, University of Cambridge Computer Laboratory, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., June 1997. (pp. 33, 36, 39, 53, 61)

[**Hamming80**] R.W. Hamming. *Coding and Information Theory.* Prentice-Hall, New Jersey, 1980. (p. 135)

[**Harrison97**] C. Harrison, D. Chess, and A. Kershenbaum. *Mobile Object Systems: Towards the Programmable Internet. Lecture Notes in Computer Science No. 1222.* , chapter Mobile Agents: Are they a good idea?, pages 46–48. Springer-Verlag, April 1997. also available from: `http:/www.sics.se/ps/abc/survey.html`. (p. 62)

[**Hayton96**] Richard Hayton and Ken Moody. *An Open Architecture for Secure Interworking Services.* In Proceedings of SIGOPS European Workshop, 1996. (p. 55)

[**Hjalmtysson97a**] Gisli Hjalmtysson and S. Bhattacharjee. *Control on Demand: A Flow Oriented Approach to Active Networking.* In Proceedings of OPENSIG'97, Cambridge, U.K., April 1997. (p. 51)

[**Hjalmtysson97b**] Gisli Hjalmtysson and K. K. Ramakrishnan. *UNITE - An Architecture for Lightweight Signalling in ATM Networks.* In Proceedings of OPENSIG'97, Cambridge, U.K., April 1997. (pp. 5, 12, 102)

[**Holton95**] M. Holton and R. Das. *XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate IO.* XFS whitepaper, Silicon Graphics, Inc., 1995. (p. 105)

[**Huard98**] Jean-Francois Huard and Aurel Lazar. *A Programmable Transport Architecture with QoS Guarantees.* IEEE Communications Magazine, 36(10):54–62, October 1998. (p. 88)

[**Hui90**] Joseph Y. Hui. *Switching and traffic theory for integrated broadband networks.* Kluwer Academic Publishers, 1990. (p. 164)

[**IETF98**] IETF. *Internet Protocol, Version 6 (IPv6) Specification (draft-ietf-ipngwg-ipv6-spec-v2-02).* Technical Report, IETF Network Working Group, August 1998. (pp. 42, 155)

[**ITU-T92**] ITU-T. *Recommendation M.3010. Principals for a Telecommunications Management Network.* ITU publication, 1992. (p. 47)

[ITU-T93] ITU-T. *Introduction to CCITT Signalling System No. 7.* ITU Recommendation Q.700, ITU publication, 1993. (p. 96)

[ITU-T94] ITU-T. *Draft Recommendation Q.2931, Broadband Integrated Service Digital Network (B-ISDN) Digital Subscriber Signalling Systems No. 2, User-Network Interface layer 3 specification for basic call/connection control.* ITU publication, November 1994. (pp. 96, 115)

[Jamin95] Sugih Jamin, Peter B. Danzig, Scott Shenker, and Lixia Zhang. *A Measurement-based Admission Control Algorithm for Integrated Services Packet Networks.* In Proceedings ACM SIGCOMM'95, Cambridge, Massachusetts, September 1995. (pp. 73, 77, 105)

[Jamin97] Sugih Jamin, Scott Shenker, and Peter Danzig. *Comparison of Measurements-based Admission Control Algorithms for Controlled-Load Service.* In Proceedings INFOCOM'97, Kobe, Japan, April 1997. IEEE. (p. 105)

[Jardetzky92] P. Jardetzky. *Network File Server Design for Continuous Media.* PhD dissertation, University of Cambridge Computer Laboratory, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., October 1992. Also published as Technical Report No. 268. (p. 105)

[Jardetzky95] P. W. Jardetzky, C. J. Sreenan, and R. M. Needham. *Storage and synchronisation for distributed continuous media.* Multimedia Systems, 3(4):151–161, 1995. (p. 78)

[JavaSoft97] JavaSoft. *The Java Telephony API—An Overview.* Technical Report, JavaSoft, January 1997. (p. 48)

[Johansen95] Dag Johansen, Robert van Renesse, and Fred Schneider. *An Introduction to the Tacoma Distributed System.* Technical Report, University of Tromso and Cornell University, June 1995. (p. 62)

[Kalmanek97] Charles Kalmanek, Srinivasan Keshav, William Marshall, Samuel Morhgan, and Robert Restrick. *Xunet 2: Lessons from an Early Wide-Area ATM Testbed.* IEEE/ACM Transactions on Networking, 5(1), February 1997. (pp. 19, 20, 96)

[Kulkarni98] A.B. Kulkarni, G.J. Minden, R. Hill, Y. Wijata, S. Sheth, F. Wahhab, H. Pindi, and A. Nagarajan. *Implementation of a Prototype Active Network.* In Proceedings of OPENARCH'98, San Francisco, USA, 1998. (p. 54)

[Kulkarni99] Amit Kulkarni, Gary Minden, Victor Frost, and Joseph Evans. *Survivability of Active Networking Services.* In Proceedings of IWAN'99, pages 299–306, Berlin, July 1999. Springer-Verlag. (p. 72)

[LANE95] LANE. *LAN Emulation Over ATM Specification—version 1.* Technical Report, ATM FORUM, February 1995. (p. 67)

[Lange96] Danny Lange and Daniel Chang. *IBM Aglets Workbench – Programming Mobile Agents in Java (White Paper).* Technical Report, IBM Corporation, September 1996. (p. 62)

[Laubach94] Mark Laubach. *Classic IP and ARP over ATM.* Technical Report, Internet RFC 1577, January 1994. (p. 20)

[Lazar96a] A.A. Lazar, K.S. Lim, and F. Marconcini. *Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture.* IEEE Journal on Selected Areas In Communications, 14(7):1214–1227, September 1996. (pp. 1, 5, 12, 15, 99)

[Lazar96b] A.A. Lazar and Franco Marconcini. *Towards an Open API for ATM Switch Control.* Available from: `http://www.ctr.columbia.edu/comet/xbind/xbind.html`, 1996. (p. 15)

[Lazar97] Aurel Lazar. *Programming Telecommunication Networks.* IEEE Network, 11(5):8–18, October 1997. (p. 144)

[Lehman98] L. Lehman, S. Garland, and D. Tennenhouse. *Active Reliable Multicast (ARM).* In Proceedings of IEEE INFOCOM, page 581, San Francisco, CA, April 1998. (p. 49)

[Leslie96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal on Selected Areas In Communications, 14(7), September 1996. (pp. 22, 38, 137)

[Li95] G. Li. *Dimma Nucleus Design.* Technical Report 1553.00.05, APM, Cambridge, U.K., October 1995. (pp. 19, 54, 139)

[Lo94] S.L. Lo. *A Modular and Extensible Network Storage Architecture.* PhD dissertation, University of Cambridge Computer Laboratory, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., January 1994. Also published as Technical Report No. 326. (pp. 64, 78)

[**Lougher94**] P. Lougher, D. Pegler, and D. Shepherd. *Scalable Storage Servers for Digital Audio and Video.* In IEEE International Conference on Storage and Recording Systems 1994, April 1994. (p. 106)

[**MAF96**] MAF. *Mobile Agent Facility Specification (OMG TC Document cf/96-12-01).* Technical Report, GMD FOKUS, IBM and The Open Group, December 1996. (p. 62)

[**Marcus98**] William Marcus, Ilija Hadzic, Anthony McAuley, and Jonathan Smith. *Protocol Boosters: Applying Programmability to Network Infrastructures.* IEEE Communications Magazine, 36(10):79–83, October 1998. (pp. 49, 133, 134)

[**Menage99a**] Paul Menage. *Improved Code Sharing in Dynamically Generated Marshalling Routines.* In GLOBECOM'99 (submitted), Rio de Janeiro, December 1999. (p. 57)

[**Menage99b**] Paul Menage. *RCANE: A Resource Controlled Framework for Active Network Services.* In Proceedings of IWAN'99, pages 25–36, Berlin, July 1999. Springer-Verlag. (p. 49)

[**Miller96**] Mark Miller, David Krieger, Norman hardy, Chris Hibbert, and E. Dean Tribble. *Market-Based Control–A Paradigm for Distributed Resource Allocation*, chapter 5: An Automated Auction in ATM Network Bandwidth, pages 1–27. World Scientific, 1996. (p. 94)

[**Needham78**] Roger Needham and Michael Schroeder. *Using Encryption for Authentication in Large Networks of Computers.* Communications of the ACM, 1978. (p. 55)

[**Neufeld95**] G. Neufeld, D. Makaroff, and N. Hutchinson. *The Design of a Variable Bit Rate Continuous Media Server.* In Proceedings 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video, pages 375–378, Durham, New Hampshire, 1995. (p. 105)

[**Newman96**] Peter Newman. *GSMP Protocol Specification.* RFC 1987, August 1996. (pp. 14, 124)

[**Newman97**] Peter Newman, Greg Minshall, Tom Lyon, and Larry Huston. *IP Switching and Gigabit Routers.* IEEE Communications Magazine, January 1997. (pp. 5, 12, 50, 102)

[**Newman98**] P. Newman, W. Edwards, R. Hinden, E. Hoffman, F. Ching, T. Lyon, and G. Minshall. *Ipsilon's General Management Protocol Specification - Version 2.0, RFC 2297.* Technical Report, Network Working Group, March 1998. (p. 15)

[Niehaus97] Douglas Niehaus, Abdella Battou, Andrew McFarland, Basil Decina, Henry Dardy, Vinai Siraky, and Bill Edwards. *Performance Benchmarking of ATM Networks.* IEEE Communications, 35(8):134–143, August 1997. (p. 144)

[Nilson95] Gunnar Nilson. *An Overview of the Telecommunication Information Networking Architecture.* In Proceedings of the TINA Conference, Melbourne, Australia, February 1995. (p. 101)

[ObjectSpace97] ObjectSpace. *Voyager Core Package: Technical Overview (White Paper).* Technical Report, ObjectSpace Inc., March 1997. (p. 62)

[OMG91] OMG. *The Common Object Request Broker: Architecture and Specification.* Technical Report, The Object Management Group (OMG), December 1991. (p. 19)

[OMG95] OMG. *The Common Object Request Broker: Architecture and Specification Version 2.0 (CORBA 2.0).* Technical Report, The Object Management Group (OMG), 1995. (p. 19)

[Onvural95] R.O. Onvural. *Asynchronous Transfer Mode Networks, Performance Issues*, chapter Chapter 7: Signaling in ATM Networks, pages 305–321. Artech House, 2nd edition, 1995. (p. 98)

[ORL97] ORL. *OmniOrb version 2.* Technical Report, Olivetti/Oracle Research Centre, 24a Trumpington St, Cambridge, UK, 1997. Available at `http://www.orl.co.uk/omniOrb.html`. (pp. 19, 54, 138, 139, 140)

[Pavon98] J. Pavon, J. Tomas, Y. Bardout, and L-H Hauw. *CORBA for Network and Service Management in the TINA Framework.* IEEE Communications Magazine, 36(3):72–79, March 1998. (pp. 20, 101)

[Peine97] H. Peine and T. Stolpmann. *Mobile Agents*, chapter The Architecture of the ARA platform for Mobile Agents, pages 50–61. Lecture Notes in Computer Science. Springer, 1219 edition, 1997. (p. 62)

[Peterson99] Larry Peterson. *NodeOS Interface Specification.* Technical Report, Active Networks Node OS Working Group, February 1999. (p. 49)

[Pham98] Anh Pham and Ahmed Karmouch. *Mobile Software Agents: An Overview.* IEEE Communications Magazine, 36(7):26–37, July 1998. (pp. 26, 62)

[PNNI1.0:96] PNNI1.0:. *Private Network-Network Interface Specification Version 1.0 (P-NNI 1.0).* ATM Forum Document: af-pnni-0055.000, March 1996. (pp. 5, 97, 129, 133)

[PNNI94] PNNI. *ATM Forum contribution, Draft Specification.* 94-0471 R7, 1994. (pp. 98, 121)

[Pratt94] Ian Pratt and Paul Barham. *The ATM Camera V2 (AVA-200).* Technical Report, University of Cambridge Computer Laboratory – ATM Document Collection 3, March 1994. (pp. 137, 147)

[Raman99] Lakshmi Raman. *OSI Systems and Network Management.* IEEE Communications Magazine, 36(3):46–53, March 1999. (p. 101)

[Redlich98] Jens-Peter Redlich, Masaaki Suzuki, and Stephen Weinstein. *Distributed Object Technology for Networking.* IEEE Communications Magazine, 36(10):100–111, October 1998. (p. 102)

[Reed99] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. *Xenoservers; Accounted execution of untrusted code.* In Seventh Workshop on Hot Topics in Operating Systems, March 1999. (p. 55)

[Rekhter97] Y. Rekhter, B. Davie, E. Rosen, G. Swallow, D. Farinacci, and D. Katz. *Tag Switching Architecture Overview.* Proceedings of the IEEE, 85(12):1973–1983, December 1997. (pp. 12, 50)

[Ricciuli98] L. Ricciuli. *Anetd - Active Network Daemon V1.0.* `http://www.csl.sri.com/ancors/anetd`, 1998. (p. 50)

[Rivest78] R.L. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems.* Communications of the ACM, 21(2):120–126, January 1978. (p. 55)

[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. (pp. 47, 48)

[RMON97] RMON. *RMON Methodology: Towards Successful Deployment for Distributed Enterprise Management (White Paper).* Technical Report, McConnel Consulting, 8324 Westfork Road, Boulder, CO 80302, May 1997. Available at: `http://www.3com.com/nsc/5005251.html`. (p. 127)

[Roche98] Maureen O'Reilly Roche. *Call Party Handling Using the Connection View State Approach: a Foundation for Intelligent Control of Multiparty Calls.* IEEE Communications Magazine, 36(6):60–66, June 1998. (p. 47)

[Romer96] Theodore Romer, Dennis Lee, Geoffrey Voelker, Alec Wolman, Wayne Wong, Jean-Loup Baer, Brian Bershad, and Henry Levy. *The Structure and Performance of Interpreters.* In Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems, October 1996. (p. 59)

[Rooney97] S. Rooney. *The Hollowman: An Innovative ATM Control Architecture.* In Integrated Network Management V (IM'97), pages 369–380, San Diego, California, May 1997. Chapman & Hall. (p. 94)

[Rooney98] S. Rooney. *The Structure of Open ATM Control Architectures.* PhD dissertation, University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., February 1998. (pp. 1, 4, 18, 89, 90, 103, 109)

[Rubin94] Harvey Rubin and N. Natarajan. *A Distributed Software Architecture for Telecommunication Networks.* IEEE Network, 8(1):8–17, February 1994. (pp. 101, 103)

[Schill97] A. Schill, S. Kuehn, and F. Breiter. *Resource Reservation in Advance in Heterogeneous Networks with Partial ATM Infrastructures.* In Proceedings of INFOCOM'97, April 1997. (pp. 65, 66, 84, 86, 103, 104)

[Schönwälder99] J. Schönwälder and J. Quittek. *Secure Management by Delegation within the Internet Management Framework.* In Integrated Network Management VI (IM'99), pages 687–700, Boston, MA, USA, May 1999. IEEE/IFIP. (p. 49)

[Shenker97] S. Shenker, C. Partridge, and R. Guerin. *Specification of Guaranteed Quality of Service (RFC 2212).* Technical Report, IETF Network Working Group, September 1997. (pp. 12, 50)

[Sidor98] David Sidor. *TMN Standards: Satisfying Today's Needs While Preparing for Tomorrow.* IEEE Communications Magazine, 36(3):54–64, March 1998. (p. 101)

[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. (pp. 49, 50)

[Smith99] J. Smith, K. Calvert, S. Murphy, H. Orman, and L. Peterson. *Activating Networks.* IEEE Computer, April 1999. (p. 134)

[Stallings98] W. Stallings. *SNMP and SNMPv2: The Infrastructure for Network Management.* IEEE Communications, 36(3):37–42, March 1998. (p. 15)

[Stamos90] James W. Stamos and David K. Gifford. *Implementing Remote Evaluation.* IEEE Transactions on Software Engineering, 16(7):710–722, July 1990. (pp. 32, 61)

[Steiglitz96] K. Steiglitz, M.L. Honig, and L.M. Cohen. *Market-Based Control–A Paradigm for Distributed Resource Allocation*, chapter 1: A Computational Model Based on Individual Action, pages 1–27. World Scientific, 1996. (p. 95)

[Tennenhouse96] D.L. Tennenhouse and D.J. Wetherall. *Towards an Active Network Architecture.* ACM Computer Communication Review, April 1996. (pp. 4, 6, 48, 133)

[Tennenhouse97] David L. Tennenhouse, Jonathan Smith, W. David Sincoskie, David J. Wetherall, and Gary J.Minden. *A Survey of Active Network Research.* IEEE Communications, 35(1):80–86, January 1997. (p. 56)

[Tschudin99] Christian Tschudin. *An Active Network Overlay (ANON).* In Proceedings of IWAN'99, pages 156–164, Berlin, July 1999. Springer-Verlag. (p. 50)

[Tse97] D. Tse and M. Grossglauser. *Measurement-based Call Admission Control: Analysis and Simulation.* In Proceedings of INFOCOM'97, April 1997. (p. 105)

[UNI3.0:93] UNI3.0:. *ATM User-Network Interface Specification - Version 3.0.* The ATM Forum: Approved Technical Specification, 1993. (p. 12)

[UNI3.1:94] UNI3.1:. *ATM User-Network Interface Specification - Version 3.1.* The ATM Forum: Approved Technical Specification, 1994. (p. 12)

[UNI4.0:94] UNI4.0:. *ATM User-Network Interface Specification - Version 4.0.* ATM Forum Document: af-sig-0061.000, 1994. (pp. 8, 12, 41, 80, 97, 110)

[van der Merwe98] Kobus van der Merwe. *Open Service Support for ATM.* PhD dissertation, University of Cambridge Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, U.K., February 1998. Also available as technical report 450. (pp. 1, 3, 4, 13, 15, 21, 105)

[Weiss95] Alan Weiss. *An Introduction to Large Deviations for Communication Networks.* IEEE Journal on Selected Areas In Communications, 13(6):938–952, August 1995. (pp. 105, 157, 163, 164)

[Welch97] Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall, 2 edition, 1997. (p. 59)

[Wetherall96] David J. Wetherall and David L. Tennenhouse. *The ACTIVE IP Option.* In Proceedings of the 7th ACM SIGOPS European Workshop, Connemara, September 1996. (p. 8)

[Wetherall98] David Wetherall, John Guttag, and David Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols.* In Proceedings of OPENARCH'98, San Francisco, CA, April 1998. (pp. 25, 49)

[White97a] J. White. *Mobile Agents White Paper.* Technical Report, General Magic, 1997. `http://www.genmagic.com/technology/techwhitepaper.html`. (p. 61)

[White97b] P. White. *RSVP and Integrated Services in the Internet: a Tutorial.* IEEE Communications Magazine, may 1997. (p. 12)

[Wilkes79] Maurice Wilkes and Roger Needham. *The Cambridge CAP Computer and its Operating System. ISBN 0-444-00357-6.* Elsevier North-Holland, Inc., 1979. (p. 34)

[Wolf95] L. Wolf, L. Delgrossi, R. Steinmetz, S. Schaller, and H. Wittig. *Issues of Reserving Resources in Advance,.* In Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video, pages 27–37, Durham, New Hampshire, April 1995. (p. 104)

[Wolf97] L.C. Wolf and R. Steinmetz. *Concepts for Resource Reservation in Advance.* Multimedia Tools and Applications, 4:255–278, 1997. (p. 104)

[Yeadon96] N. Yeadon, F. Garcia, D. Hutchison, and D. Shepherd. *Filters: QoS Support Mechanisms for Multipeer Communications.* IEEE Journal on Selected Areas In Communications, 14(7):1245–1262, September 1996. (p. 134)

[Yemini96] Yechiam Yemini and Sushil da Silva. *Towards Programmable Networks.* In White Paper. IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, L'Aquila, Italy, October 1996. IFIP/IEEE. (pp. 7, 49)

[**Zhang93**] L. Zhang, S. Deering, D.Estrin, S. Shenker, and D. Zappala. *RSVP: A New Resource ReSerVation Protocol.* IEEE Network, 7(5):8–18, September 1993. (pp. 8, 12, 71, 103)