INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
Architecting the Control Infrastructure of Multimedia Networks

Mun Choon Chan

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Graduate School of Arts and Sciences.

Columbia University

1997
Abstract

Architecting the Control Infrastructure of Multimedia Networks

Mun Choon Chan

This thesis addresses some of the fundamental problems related to architecting the control infrastructure of a broadband Asynchronous Transfer Mode (ATM) network. Specifically, the issues of service provisioning and signaling performance are addressed. For service provisioning, the focus is on developing an ATM Virtual Private Network (VPN) architecture. The objective is to find the appropriate trade-off between user-network interaction complexity and the degree of user control. For signaling, using an open network control model, various approaches to increasing the performance and scalability of the signaling infrastructure are studied. The objective is to design a high performance connection management framework on top of a flexible distributed processing platform.

Although an ATM VPN is not the same as an end-to-end Virtual Path (VP) or Virtual Circuit (VC), much of today’s providers provision these services the same way. As a result, customers have limited control over the VPN and cannot exploit knowledge about their own traffic statistics. The proposed solution is a Virtual Path Group (VPG)-based ATM VPN service, which reduces the customers’ dependency on the provider and allows the customers to implement control objectives and schemes according to their own requirements. A VPG-based VPN provides a very good intermediate point between the use of leased line, where there is minimum network and user interaction, and the use of VC setup, where every change in the user domain must be known to the network provider. The management and control architecture of the resulting ATM network is designed so that it is independent of the characteristics of the underlying public network services, and guarantees quality of service. The control architecture of the VPN, which runs within the customer domain, is structured into
three layers of control, according to different time-scales. Each control layer is modeled by a generic controller design concept which allows a large class of control objectives and control schemes to be implemented.

The VPN control architecture has been implemented on a network emulator that contains a parallel simulation kernel. The network emulation platform supports real-time visualization and interaction, and an implementation using up to 128 processors was evaluated. Using a prototyping concept that allows better integration between simulation studies and software development, a significant amount of the software developed on the network emulator are re-used on the target platform, which consists of a network of heterogeneous ATM switches.

Using a network programming environment where there is an open and uniform access to abstractions that model the local states of networking resources, the work on signaling focuses on designing a high performance connection management framework which exploits the programmability of the network. The amount of remote invocations is reduced drastically by caching of network states and by aggregation of access to remote objects. Further, the impact of latency incurred in remote invocations is reduced through parallelization. For various system configurations, parameter values for controlling the amount of caching and aggregation are proposed so that the system can be dimensioned appropriately for the desired trade-off between call setup latency and throughput. The system is implemented in C++ and runs on general purpose UNIX workstations. Its performance is comparable to the switching performance in a backbone network. For heavy traffic loads, call throughput of approximately $10^6$ calls/hr can be attained, and for light traffic loads, call setup latency of about 10ms can be achieved.
Table of Contents

CHAPTER 1: INTRODUCTION

1.1 Status of the Control Infrastructure for Multimedia Networks ............ 1
1.2 Provisioning of Virtual Private Networks ............................. 2
1.3 Engineering the Control Infrastructure for Performance ................. 5
1.4 Outline of Dissertation and Main Contributions ......................... 7

CHAPTER 2: VPN ARCHITECTURE FOR CUSTOMER CONTROL AND MANAGEMENT

2.1 Broadband VPN Services ............................................. 10
2.2 Customer Control ...................................................... 15
   2.2.1 Customer Control and Management Objectives .................. 16
2.3 The Architecture for Customer Control and Management ................. 18
   2.3.1 Controller Design ............................................. 22
   2.3.2 Enabling Management Objectives ............................. 26
2.4 QOS Guarantees and VPN Control Algorithms ......................... 29
   2.4.1 Constructing a Call-Level Abstraction with Cell-Level QOS Guarantees ............................................. 30
   2.4.2 Cell-Level QOS Guarantee through Schedulable Region ..... 31
   2.4.3 Constructing a Schedulable Region using CBR VP ........... 33
   2.4.4 Review of VP Control Algorithms ............................. 37
2.5 Realization of the Control System ................................... 39
   2.5.1 VP Controller .................................................. 40
   2.5.2 VPG Controller ............................................... 45
   2.5.3 Exact Approach to Weight Computation ....................... 47
   2.5.4 Heuristic Approach to Weight Computation .................. 50
2.6 Evaluation of the Control System .................................... 51
2.7 Conclusion ............................................. 55

CHAPTER 3: PROTOTYPING NETWORK CONTROL ARCHITECTURE

3.1 The Prototyping Approach ............................... 56

3.2 Design of the Emulation Platform ...................... 59
  3.2.1 The Parallel Simulation Kernel .................... 60
  3.2.2 Emulation Objects .................................. 65
  3.2.3 Emulation Support .................................. 70
  3.2.4 Real-Time Visualization and Interaction ............ 71

3.3 Realization of the VPN Architecture .................... 77
  3.3.1 Porting to a CORBA-based Platform ................. 77
  3.3.2 Realization on a Heterogeneous ATM LAN ........... 79
  3.3.3 Support Needed from the Provider .................. 81

3.4 The High Performance Platform for Experimentation .... 82
  3.4.1 Building an Emulation Platform on the KSR-1 .......... 83
  3.4.2 Benchmark Measurement on the KSR-1 ................. 84
  3.4.3 Experiences with Emulation of a Traffic Control System . 86

3.5 Porting the Emulation Platform from KSR-1 to SP-2 ...... 96

3.6 Conclusion ............................................. 98

CHAPTER 4: DESIGNING A HIGH PERFORMANCE ATM CONNECTION MANAGEMENT SYSTEM

4.1 Introduction ........................................... 100

4.2 Description of the Connection Management Framework .... 102
  4.2.1 The Extended Reference Model ..................... 103
  4.2.2 Object Interaction Model ........................... 104
  4.2.3 Model of Connection Setup ......................... 108
  4.2.4 Experimental Configuration ......................... 114
  4.2.5 Reference Measurements ............................ 115
4.3 The Design of the Connection Manager ................. 118
   4.3.1 Caching of Network States ..................... 119
   4.3.2 Aggregate Access to the SwitchServer Object .......... 124
   4.3.3 Hiding Latency Through Parallelization .......... 125
   4.3.4 Functional Design of the Connection Manager .... 127

4.4 Performance Measurement of xbind Connection Manager .... 130
   4.4.1 Impact of Message Threshold Parameter .......... 132
   4.4.2 Dimensioning the Switching Identifier Cache Size .... 135

4.5 Conclusion ............................................. 141

CHAPTER 5: CONCLUSION AND DIRECTION FOR FUTURE RESEARCH

5.1 Conclusion ............................................ 142

5.2 Direction for Further Research .......................... 144

References ................................................. 146

iii
# List of Figures

1.1 A Virtual network interconnecting isolated component networks .................. 2  
2.1 Customer’s view of a virtual private network ........................................... 11  
2.2 A VPG-based virtual private network ....................................................... 13  
2.3 A functional model of the customer control system .................................... 19  
2.4 Network views the controllers operate on ................................................. 20  
2.5 Functional model of a call admission controller interacting with a VPG controller ................................................................. 22  
2.6 Functional model of a generic controller .................................................... 25  
2.7 Framework for customer management ....................................................... 27  
2.8 The customer management console for a VPG-based VPN service. The upper layer represents the VP network, the lower layer the VPG network. The vertical bars on the VP network indicate the utilization, the vertical bars on the VPG network the allocation of VPG bandwidth to VPs .................................................. 28  
2.9 Schedulable region of a 40 Mb/s link ......................................................... 32  
2.10 Architecture of a typical broadband switching node in the customer network ........................................................................................................... 34  
2.11 Construction of scheduleable region using CBR VP ..................................... 36  
2.12 Approximating the scheduleable region with a hyperplane ......................... 42  
2.13 Projection of thresholds ........................................................................... 43  
2.14 Network topology used in the evaluation ................................................... 52  
2.15 Performance of VPG Algorithms for Class 1 Traffic ................................... 54  
2.16 Performance of VPG Algorithms for Class 2 Traffic ................................... 54  
3.1 Building blocks of the interactive emulation platform ................................... 59  
3.2 Realization of the parallel simulation kernel. (interaction between 2 local simulation kernels). ................................................................. 62  
3.3 Performance of simulation kernel ............................................................... 64  
3.4 Scalability of a 50 node experiment running on the simulation kernel .......... 65  
3.5 Inheritance relationship between simulation and emulation object classes ..... 66
3.6 Generic monitoring and access of a resource controller. ..................... 68
3.7 Software architecture for the emulation support module .................... 70
3.8 Generation of the processes in two steps ..................................... 72
3.9 Processes and communication means .......................................... 73
3.10 (a) Floating Sphere ............................................................... 76
3.10 (b) Stacked Cylinder .............................................................. 76
3.11 Topology of ATM LAN network ................................................. 79
3.12 Topology of VPG network ......................................................... 80
3.13 Hardware configuration of the interactive emulation platform .......... 82
3.14 Hardware configuration of KSR-1 ................................................ 84
3.15 Communication pattern among network mechanisms ..................... 87
3.16 Steps of a call setup .................................................................. 88
3.17 Impact of serialization of memory management system calls ........ 90
3.18 (a) Mechanisms interact by asynchronous message passing based on mailboxes .......................................................... 92
3.18 (b) Mechanisms interact by asynchronous message passing through a communication kernel ......................................................... 92
3.19 Impact of using communication kernel for emulation .................... 93
3.20 Impact of increasing communication resources (constant system size) . 94
3.21 Impact of increasing system size (constant communication resources) . 95
3.22 Impact of influencing thread scheduling ....................................... 96
4.1 Overview of the RGB Decomposition of the XRM .......................... 104
4.2 Object Interaction Model .............................................................. 106
4.3 Interaction between ConnectionManager and SwitchServer ............ 107
4.4 Examples of point-to-point connection services ............................... 109
4.5 Re-routing from connection(A,D) to connection(A,C) ..................... 113
4.6 Execution of a connection set up request ..................................... 115
4.7 Throughput-delay curve of (a) sequential (b) threaded connection manager using one or more clients ........................................... 117
4.8 Setting up the switching tables in an ATM network .................... 121
4.9 Parallelization using oneway object call invocations .................... 126
4.10 Design of the Connection Manager ............................................. 128
4.11 Interface design of SwitchServer .................................................. 129
4.12 (b) Throughput-delay curve of xbind connection manager .......... 131
4.12 (a) Throughput-delay curve of all three connection managers (one client) . ........................................................................ 131
4.13 Remote invocation load on the ConnectionManager .................. 133
4.14 Impact of threshold parameter on throughput-delay characteristics of xbind connection manager ........................................... 134
4.15 Impact of message threshold parameter on sample path of VCI cache usage ........................................................................ 137
4.16 Modeling the processing on the ConnectionManager as a M/G/∞ queue . ................................................................................................................................. 138
4.17 Comparison of measured and estimated amount of VCI cache usage distribution (a) low load (l=10, m=0.2, T=5) (b) medium load (l=50, m=0.2, T=5) (c) high traffic load (l=100, m=0.2, T=5) (d) high traffic load (l=100, m=0.2, T=15) ................................................................. 139
# List of Tables

2.1 Summary of Approaches to Broadband VPN Provisioning ............ 14
2.2 QOS parameters for physical and logical links. .................. 35
2.3 Mapping between Link Bandwidth and Size of Schedulable Region .. 37
2.4 Performance of LAPACK on a IBM/390 .......................... 50
3.1 Shared Memory vs. Distributed Memory .......................... 97
3.2 Task Involved in Porting the Emulation Platform ................ 97
4.1 State Caching on a Connection Manager ......................... 124
4.2 Difference between estimated and measured cache hit frequency .... 140
Acknowledgments

First and foremost, I would like to thank my thesis advisor, Professor Aurel A. Lazar, for his support and guidance. Working with Aurel is both demanding and rewarding. Aurel always stresses the importance of understanding the essence of a problem, and how such insights should be used to guide the direction for our research work. His vision as a researcher has been very influential towards my intellectual development.

Special thanks to Dr. Rolf Stadler, whom I have the opportunity to work with for a very significant part of my thesis work. It was a very rewarding experience and I have benefited from his knowledge and expertise. I will always remember the long meetings we used to have arguing over the finer points of a technical argument.

Past and present members of the COMET group, in particular my office-mates and the xbind core group, have made my stay in Columbia a very memorable and enjoyable one. The experience has been particularly enriching due to the breadth of the work covered. In addition, during the course of my Ph.D. studies in Columbia, which included summer internships in GTE Laboratories and AT&T Bell Laboratories, I was very fortunate to have the opportunity to work with many researchers from all over the world. The interactions have been most valuable.

I would also like to thank the members of my defense committee, Professor Andrew Campbell, Professor Alexandros Eleftheriadis, Professor John Pavlik and Professor Henning Schulzrinne for accommodating my thesis defense in their very busy schedule.

Finally, I would like to express my deep gratitude towards my family, without whom I would not have the opportunity to pursue my graduate studies. Their unyielding support has been the source of energy that keeps me through the period of seemingly never ending student life.
To my parents, Aunt Yuet-Ming and Kymie.
Chapter 1  Introduction

1.1 Status of the Control Infrastructure for Multimedia Networks

Emerging multimedia networks use Asynchronous Transfer Mode (ATM), a technology standardized by the International Telecommunications Union (ITU) as the transport vehicle of wide area Broadband Integrated Digital Services Networks (B-ISDNs). The major objective of ATM is to support the diverse requirements of heterogeneous traffic sources, such as voice, video and data in an efficient and cost-effective way. Scalability, statistical multiplexing, traffic integration and network simplicity are cited as the primary merits of ATM. However, in spite of these merits, ATM is gaining ground only in backbone networks. Its role in local area networks is not clear.

One of the main problems hindering ATM deployment is the inflexibility and complexity of current ATM signaling protocols. Existing standards for connectivity services on B-ISDN networks are based on the User/Network Interface (UNI) and Network/Node Interface (NNI) defined for the plain old telephone services (POTS) network. In order to allow for the rapid creation and deployment of sophisticated network services, the signaling infrastructure of an ATM network needs to be designed for flexibility and scalability.

This thesis addresses some of the issues related to service provisioning and performance of the ATM signaling infrastructure. For service provisioning, the focus is on developing an ATM Virtual Private Network (VPN) architecture. The objective is to find the appropriate trade-off between user-network interaction complexity and the degree of user control. For signaling, using an open network control model, various approaches to increasing the performance and scalability of the signaling infrastructure are studied. The objec-
tive is to design a high performance connection management system on top of a flexible
distributed processing environment.

1.2 Provisioning of Virtual Private Networks

The Virtual Private Network (VPN) is a communication service that allows the creation of
a virtual enterprise network for a large, geographically distributed corporation. The enter-
prise network is made up of islands of private component networks connected using the
public network infrastructure (Figure 1.1).

![Figure 1.1: A Virtual network interconnecting isolated component networks.]

VPN-related research found in the literature can be broadly divided into two cate-
gories, management and control.

As a VPN spans multiple network domains, including one or more public network
domains, there are many issues related to the development of a multi-domain management
system. Many of these issues are highlighted in [LEW95] and [TSC95]. The design of a
common management information model is covered in [SCH93]. In [PAR97] and [FEL97],
design of integrated customer and provider management systems, so that the enterprise net-
work appears as a single virtual network entity, are described.

The issues of control include both signaling interaction between user and network,
and the algorithms for resource allocation. Different signaling interaction paradigms for
VPN provisioning, in a more traditional telecommunications framework, are described in
[TRO92] and [SCO92]. Approaches based on the assumption of access to all network re-
sources and controllers are proposed in [Mou95] and [KOB97]. Depending on how the
VPN is being provisioned (e.g., Virtual Circuit based or Virtual Path based), different re-
source control algorithms can be used. Some of these algorithms are presented in [ATS93],
[DZI96], [FOT95] and [YAM91].

The focus of this work is on developing an VPN control architecture from the cus-
tomer’s perspective, in particular, a VPN service that allows more customer control. Spe-
cifically, the objective is to develop an architectural framework in which solutions to the
following questions are given:

- What is the appropriate trade-off between user-network interaction complexity and
  the degree of user control?
- What is the control and management architecture for such a network?
- How should end-to-end quality of service be provided over the private components
  and the public networks?
- How should this architecture be evaluated and realized?

Although an ATM VPN is not the same as an end-to-end Virtual Path (VP) or Vir-
tual Circuit (VC), much of today’s public network providers provision these services the
same way. As a result, customers have limited control over the VPN and cannot exploit the
knowledge about their own traffic statistics. The proposed solution for the first question is
a Virtual Path Group (VPG)-based ATM VPN service, which provides a very good intermediate point between the use of leased line, where there is a minimum network and user interaction, and the use of VC setup, where every change in the user domain must be known to the network provider. A VPG-based VPN reduces the customers’ dependency on the provider and allows the customers to implement control objectives and schemes according to their own requirements. A VPG can be conceptually explained as a “Big VP” that contains a bundle of VPs, connecting VP switches in a public broadband network or its termination points, forming a virtual ATM network within the VPN.

The control and management architecture of the resulting ATM network is designed so that it is independent of the characteristics of the underlying public network services, and guarantees quality of service. The control architecture of the VPN, which runs within the customer domain, is structured into three layers of control, according to different timescales. Each control layer is modeled by a generic controller design concept which allows a large class of control objectives and control schemes to be implemented. The concept of QOS guarantee is based on constructing a capacity abstraction over Constant Bit Rate (CBR) VP. This approach requires the minimum guarantee from the network providers and at the same time satisfies the need for better efficiency through multiplexing.

As a proof of concept, the VPN control architecture is implemented on a network emulator which runs on a KSR-1 and on an IBM SP2. Both supercomputers are located at the Cornell Theory Center, Ithaca, New York. The emulation system is controlled by and visualized on an Indigo2 workstation at Columbia that is connected to Cornell over high-speed ATM links. The Network visualization interface is based on OpenGL, a 3D graphics library. The emulation environment allows us to experiment with the functionality and dynamics of virtual networks, with greater flexibility and lower cost than implementing components on a real testbed.
The VPN control architecture implemented on a network emulator uses a prototyping concept that allows better integration between simulation studies and software development. The prototyping approach has proved to be rather useful. A significant amount of the software developed on the network emulator has been re-used on the target platform, which consists of a network of heterogeneous ATM switches.

1.3 Engineering the Control Infrastructure for Performance

One of the most fundamental objectives in networking is the provisioning of connectivity, i.e., the establishment of a communication path between two end-points. In the case of an ATM network, which is a connection-oriented network, setting up a communication channel requires a signaling system that coordinates operations among a set of distributed software modules.

The signaling system for emerging ATM-based multimedia networks has to meet two requirements. First, because of the need to be able to rapidly create and deploy new services, it must be very flexible. Second, in anticipation of the large volume of users, it must display high performance and be scalable.

Unlike existing standards for connectivity services, which are essentially extensions of the User/Network Interface (UNI) and Network/Node Interface (NNI) standards designed for the plain old telephone service (POTS) networks, modern approaches to provisioning of network services (e.g., xbind [CHA96d], TINA [TIN95] and DCAN [KOB97]) exploit advances in distributed system technologies to provide more flexibility for service creation and deployment. In these approaches, signaling entities run on a general purpose distributed computing platform. Interactions among these signaling entities are expressed in terms of high-level operations over an infrastructure that provides an open and uniform access to abstractions that model the local states of networking resources.
Given such a platform, the signaling infrastructure has to be carefully engineered so that the performance of the connection management system is capable of handling the large volume of expected user requests. The focus of our work is on performance. In addition, in order to meet the flexibility requirement, the signaling system is designed to run on the xbind platform, a broadband kernel for multimedia networks [LAZ96].

Performance of distributed computing platforms has been extensively studied in the literature. A number of studies have been specifically performed for the underlying transport system. In [SCH96b], the performance of two popular Common Object Request Broker Architecture (CORBA) [OMG93] implementations for bulk data transfer have been studied. However, for signaling purposes, bulk data transfers are not an appropriate benchmark because signaling messages tend to be short, and the performance objective caters around providing low latency for short exchange of messages, rather than high throughput for large data sets. This observation is taken into account in [BLA96], where changes to the network processing software are proposed. These changes allow the processing of many short messages to be performed much more efficiently. On a higher level, in [OLI97], a large number of distributed platforms (implemented using C, C++, CORBA, Java, etc.) are benchmarked for exchanges of short messages. In [TAM97], the performance of various connection management algorithms are studied. Finally, [VEE95] describes a system-level design for improving the performance of a broadband signaling system.

Our work differs from most of the other works mentioned above in that we propose design enhancements on the system design level, instead of the protocol level. Improvement in the protocol layer will independently help boosting the performance of our system. The closest work to ours is that of [VEE95]. Compared to [VEE95], we believe that we have gone much further in terms of exploiting the network as a programmable entity and have applied well-known techniques to our scheme in a domain-specific way with substantial performance gains.
In the design of the connection management, the amount of remote invocations is reduced drastically by caching of network states and by aggregation of access to remote objects. Further, the impact of latency incurred during remote invocations is reduced through parallelization. With different system configurations, parameter values for controlling the amount of caching and aggregation can be changed so that the system will be dimensioned appropriately for the desired trade-off between call setup latency and throughput.

1.4 Outline of Dissertation and Main Contributions

The thesis is organized in the following way. In Chapter 2, we describe the design of the virtual private network architecture, the issues of QOS guarantees and control algorithms. Our approach to evaluation and realization of the proposed architecture is presented in Chapters 3, where we also describe the high performance platforms used for network emulation and the lessons learned. In Chapter 4, we describe the design of the high performance connection management platform and related measurements.

The major contributions of this thesis are:

ATM VPN Architecture

- A VPN service based on the concept of Virtual Path Group, which provides a high degree of customer control, is proposed.

- A VPN control architecture, which runs within the customer domain, is designed. The control architecture is structured into three layers of control, according to different time-scales. Each control layer is modeled by a generic controller design concept we developed that allows us to implement a large class of control objectives and control schemes.

- A set of performance management capabilities has been realized for this service, including QOS management, VP management and priority management.
• A concept for end-to-end Quality of Service (QOS) in the customer network over a provider network that supports Constant Bit-Rate (CBR) Virtual Paths (VP) has been developed.

**Network Software Prototyping**

• A concept of "evolutionary" prototyping is developed for the development of network software. The process of writing simulation software is integrated with the process of building the software on the target platform.

• As a proof of concept, the control architecture for virtual private networks has been developed on a prototyping platform. Performance of algorithms running in the system were evaluated through simulation, and a substantial amount of the code written for simulation was re-used on the target platform, which consists of a network of heterogeneous ATM switches.

• The implementation of a large network control software (using up to 128 processes) on a parallel machine was evaluated.

• A concept for real-time interactive visualization in parallel simulation has been developed and demonstrated.

**Engineering the Control Infrastructure for Performance**

• A very high performance connection management framework, that runs on a flexible distributed processing environment, was designed and demonstrated.

• The design of the connection management system allows the system to be dynamically re-configured for the desired trade-off between latency and throughput.

• It is demonstrated that by partitioning the network states and resources in various ways, different interaction models can co-exist in the same network control system.
Part of this dissertation have appeared in [CHA96a], [CHA96b], [CHA96c], [CHA96d], [CHA97a], [CHA97b] and [CHA97d]. A patent has also been filed for the design of the high performance connection management framework outlined in Chapter 4. Additional work that was done during my Ph.D. studies, and not included in this thesis, may be found in [CHA95a], [CHA95b], [CHA96e] and [CHA97c].

Finally, a working prototype of the emulated system was shown in the demonstration program of ACM Multimedia, San Francisco, CA, November 1995, and a prototype of xbind was shown in the demonstration program of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5), which was held in Syracuse, NY, from August 5 to August 9, 1996.
Chapter 2  VPN Architecture for Customer Control and Management

2.1 Broadband VPN Services

Broadband technology has the potential to change corporate networking in major ways. Broadband networks are aimed at providing quality-of-service (QOS), thus making it possible to support real-time services like voice and video communication, in addition to best-effort data delivery. Due to their ability to integrate different services on the cell-level, they provide a promising platform for distributed multimedia applications that are emerging today. Furthermore, the advent of broadband technology will enable the integration of today’s separate corporate networks (voice network, data network), which often rely on different public services (e.g., leased lines for voice traffic and LAN interconnection, frame-relay service for low-volume data exchange) into a single enterprise network, using a single Virtual Private Network (VPN).

A broadband virtual private network (VPN) is a service that provides broadband transmission capability between islands of customer premises networks (CPNs). It is a central building block for constructing a global enterprise network (EN) which interconnects geographically separate CPNs.

A VPN service involves several administrative domains: the customer domain, the domain of the VPN service provider—also called “value added service provider” (VASP)—
, and one or more carrier domains [SCH93]. As a result, it is necessary to address the aspects of multi-domain management in the context of VPN service management and provisioning ([HAL95], [LEW95], [TSC95]). The scope of this chapter is limited to the customer domain and the interaction between the customer domain and the VPN provider domain.

![Diagram of a virtual private network](image)

Figure 2.1: Customer's view of a virtual private network

In the following, we briefly describe different types of broadband VPN services, taking the customer's point of view (Figure 2.1). Traditionally, leased line circuits based on STM (SDH/SONET) technology have been used for providing VPN services [YAM91]. The speed of the circuit can be changed by customer-provider cooperative control. However, dynamic bandwidth adjustment for leased line circuits is inefficient and costly compared to ATM-based services, which place no restriction on the line speeds the customer can choose from [HIS94].

Service providers are beginning to offer broadband VPN services using ATM transport networks. Two common approaches are VC-based VPN services ([MOU95], [SAY95], [FOT95]) and VP-based VPN services [ATS93]. These services provide ATM logical links
between separate CPNs. In the case of a VC-based VPN service, the customer requests a new VC from the provider for every call to be set up over the VPN. Bandwidth control and management between customer and provider is performed per VC.

In the case of a VP-based VPN service, customers can perform their own call and resource control for a given VP, without negotiating with the VPN provider. Bandwidth control and management between customer and provider is performed per VP.

VC-based and VP-based VPN services replace today's leased line services. They offer customers more flexibility in dynamically requesting adjustments in the VPN capacity. Since networks typically exhibit a dynamic traffic pattern, such a technique of rapid provisioning will result in lower cost for the customer, because pricing is expected to be based on the VPN capacity per time interval allocated to the enterprise network.

In spite of the fact VC- and VP-based VPNs allow dynamic bandwidth adjustments to be performed, these services are still essentially point-to-point services, and not a virtual network service. In this chapter, we will introduce a VPN service that provides more customer control over the VPN by allowing a virtual ATM network to be constructed over the public domain networks. This service is based on the Virtual Path Group (VPG) concept, which has been introduced in [HIS89] to simplify virtual path dynamic routing for rapid restoration in a carrier network.
A VPG is defined as a logical link within the public network provider's ATM network. Figure 2.2 shows a VPG-based Virtual Private Network connecting 3 CPNs. A VPG is permanently set up between two VP cross connect nodes or between a VP cross connect node and a CPN switch that acts as a customer access point for the VPN service. Any ATM switch that supports VP switching can be used as a VP cross connect node, including an ATM LAN switch such as FORE ASX-100. A VPG accommodates a bundle of VPs that interconnect end-to-end customer access points. The VPN provider allocates bandwidth to a VPG, which defines the maximum total capacity for all VPs within the VPG. A VPG-based VPN consists of a set of interconnected VPGs.

VPs and VPGs are set up by the network management system of the VPN provider during the VPN configuration phase. Only the network management systems must know about the routes of the VPGs, their assigned bandwidth, and the VPs associated with them. The use of VPGs has no impact on cell switching, as cells are transmitted by VP cross connect nodes based on their VP identifier. However, in order to guarantee cell-level QoS in
the carrier's network, policing functions (Usage Parameter Control) are required at the entrance of each VPG.

The VPG concept enhances the customer's capability for VP capacity control. It allows transparent signalling and dynamic VP bandwidth management within the customer domain. A customer can change the VP capacities, within the limits of the VPG capacities, without interacting with the provider. As a result, the VPG bandwidth can be shared by VPs with different source-destination pairs. Furthermore, customers can independently achieve the optimum balance between the resources needed for VP control (e.g., the amount of user-network signaling) and the resources needed to handle the traffic load.

A summary of the three approaches to VPN provisioning is given in Table 1.

<table>
<thead>
<tr>
<th>Bandwidth Sharing Level</th>
<th>VP-based VPN [ATS93]</th>
<th>VPG-based VPN</th>
<th>VC-based VPN [FOT95]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Among different customers</td>
<td>VP capacity control with customer-network negotiation</td>
<td>VPG capacity control with customer-network negotiation</td>
<td>VC set up with customer-network negotiation</td>
</tr>
<tr>
<td>Among different source-destination pair of the same customer</td>
<td>None</td>
<td>VP capacity control by customer</td>
<td></td>
</tr>
<tr>
<td>Among different users within the same source-destination pair.</td>
<td>VC set up by customer within VP</td>
<td>VC set up by customer within VP</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of Approaches to Broadband VPN Provisioning
2.2 Customer Control

Corporations want to control and manage their enterprise networks according to their own control objectives and management strategies. This implies that a corporate customer, using a VPN service, needs the capability to control and manage its traffic on the VPN—possibly in cooperation with the provider. For the designer of an enterprise network, the question of which functions are performed by the customer alone, which by the provider alone, and which in the form of customer-provider cooperative control arises.

There are strong reasons for customer control, i.e., for running traffic management functions in the customer domain. Firstly, different customers pursue different control and management objectives while running their enterprise networks. For example, customer requirements concerning the traffic carried in a VPN are very diverse with respect to supporting multimedia traffic with different performance characteristics and performance requirements. Some customers may want to operate a multiclass network with several traffic classes for both real-time and non real-time traffic and a high degree of cell-level multiplexing; others may want to support just one class of traffic with peak rate allocation. Some might want to implement a call priority scheme which enables calls of higher priority to pre-empt those of lower priority when the network is congested; others may want to apply other control schemes in case of congestion. etc., etc. Providers face difficulties to support such diverse requirements. Customers who know their requirements better than the providers may be in a better position to execute control according to their objectives.

Also, operations under customer control can be executed faster than those performed in cooperation with the provider, since no negotiation is required. For example, setting up connections over a VPN can be done by the customer in a distributed way, based only on local information. This allows customers to engineer or configure their traffic control systems in such a way that short connection set-up times can be achieved, which is required by some applications.
Secondly, customers want provider-independent control in order to meet special requirements for the enterprise network [ZER92]. For example, usage collection that permits billing at a level of detail beyond the provider’s capability, such as billing at an application level, may be needed. Furthermore, the partitioning of the VPN by the customer may be required to implement sophisticated access control mechanisms, which prevent unauthorized access to certain partitions of the network. Also, automatic fall back mechanisms may be desirable for critical applications that need high network reliability.

Finally, moving the responsibility for VPN traffic management from the provider to the customer accelerates the introduction of Broadband VPN services. Specifically, public VPN services based on CBR VPs can be provided efficiently today [ATS93, FOT95]. However, such a service requires resource control by the customers, since they will be billed based on allocated bandwidth—even if they do not use it.

Obviously, raising the level of customer control increases the complexity of the customer control system. However, recent advances in distributed object-oriented technology make it easier to build network control systems with a rich functionality.

In the next section, we present an architecture for management and control of a broadband VPN service. The architecture is operated by the customer and emphasizes the concept of customer control. We outline how different control and management objectives can be achieved with this architecture. An element of this architecture is the design of a generic resource controller, which can be specialized in order to realize a large class of control schemes, following a customer’s specific requirements.

2.2.1 Customer Control and Management Objectives

From the perspective of traffic control, the customer wants to achieve two sets of objectives. The first set relates to end-to-end QOS requirements for the traffic on the enterprise network, which translates into QOS objectives for the traffic that traverses the VPN. QOS objectives on the cell level are usually expressed in terms of bounds on end-to-end delays and
error rates. On the call level, QOS objectives include call blocking constraints and bounds on call set-up times. The second set relates to efficient use of VPN resources, primarily trunk bandwidth.

Efficient use of the VPN bandwidth is very much related to the QOS requirements. On one end of the spectrum is peak rate allocation in the form of CBR VPs, where the provider guarantees upper bounds for delays and loss rates on the VPs [ATS93]. Based on this information, the customers can choose to run their own multiplexing schemes on various levels if more efficiency is desired and less stringent QOS requirements can be tolerated. On the cell-level, exploiting multiplexing among calls with the same source-destination pair in the VPN can be performed using the schemes described in [HYM91, ELW93]. Cell multiplexing among calls with different source-destination pairs can be achieved using the contract region concept [HYM94]. On the call-level, schemes used for VP control (e.g. [OHT92]) can be used to exploit multiplexing among calls with the same source-destination pairs. Finally, the schemes described in [FOT95] and [CHA96a] can be used to multiplex calls with different source-destination pairs. Depending on the type of VPN service the provider offers, the customer can choose to implement a combination of the above described multiplexing schemes in the customer control system.

In terms of managing the enterprise network, customers want capabilities to control the bandwidth cost of the VPN service, define QOS objectives and set preferences and priorities for resource allocation to deal with congestion situations. These management objectives apply to the customer domain only and are different from customer to customer. They define the policies according to which the customer control system operates. Management capabilities can be realized by tuning controllers in the customer control system (Section 2.3.2). For illustration purposes, we describe below some of the management capabilities we have implemented in our prototype system.
Cost management allows the customer to define the maximum average cost of the VPN communication resources over a specific period of time. This capability is realized by setting constraints on the negotiation of VPN bandwidth between the customer and the provider. VP management allows the customer to directly manipulate VP bandwidth. Operationally, the control of the VP bandwidth can be executed either automatically by the customer control system or under direct control of the operator of the enterprise network. The operator can allocate a fixed amount of bandwidth to a VP, which must be respected by the control system. QOS and priority management operations define how calls are handled in the enterprise network. In our specific implementation, every call is characterized by a performance class and a priority class. Both classes represent independent concepts. The performance class of a call determines its QOS requirements. QOS management deals with managing the level of service provided to different performance classes. In particular, the customer can modify the blocking objectives of calls belonging to a performance class. The level of priority determines the relative importance of a call. In our scheme, a high priority call can preempt a call of lower priority in case of congestion. The customer can enable and disable priority control and can set blocking objectives for priority classes. The above described management capabilities are orthogonal in the sense that they can be applied independently of one another.

2.3 The Architecture for Customer Control and Management

In this section, we describe the design of a control system for a VPG-based VPN service. Since a VPG-based VPN service can be seen as an extension of a VP-based VPN service, its control system contains an additional layer of functionality, the VPG control layer, which operates on a medium time-scale.

The customer control system is operated by the customer and runs in the customer domain. In our design, the primary interaction between the customer and the VPN provider relates to negotiation of VPN bandwidth. This gives customers a high degree of control over
the traffic that is carried over the VPN. Specifically, customers can perform all aspects of
call control and VP control independently of the provider, according to their own objectives
and requirements.

Figure 2.3: A functional model of the customer control system

Figure 2.3 shows the systems involved in the provisioning and operation of a VPG-
based VPN. In the provisioning phase, information concerning the VPG topology, the VP
topology and the mapping between them is exchanged and stored in the management sys-
tems of the customer and the provider. Knowledge about the VPGs is also required in the
provider's control system, which performs Usage Parameter Control (UPC) per VPG. The
use of VPGs has no influence on cell switching and transmission, since cells are switched
according to the VP identifiers in their headers.
Figure 2.3 also shows the organization of the control system according to time-scales. The customer control system contains three classes of controllers: VP controller, VPG controller, and VPN controller. These controllers operate on different time-scales and run asynchronously.

![Diagram showing network views of controllers](image)

Figure 2.4: Network views the controllers operate on

We illustrate the interaction among these controllers with an example. Assume that one of the VPs experiences a sudden increase in traffic load. The VP controller that is associated with this VP admits calls as long as there is sufficient capacity. If there is insufficient capacity available, calls are blocked. On a slower time scale, the VPG controller detects the congestion in this particular VP and attempts to allocate additional bandwidth to this VP. If the increase in traffic load is transient and, therefore, the demand for band-
width drops after some time, the interaction stops here. Otherwise, if the congestion persists, the VPN controller, which runs on an even slower time-scale, will request additional VPN capacity from the provider.

For the purpose of dynamic bandwidth control, a VPG-based VPN can be compared to an ATM network in which the link size can be varied. Therefore, controllers in the customer domain operate on two views of the network (Figure 2.4). The view on the left side of Figure 2.4 shows a network of end-to-end VPs which connect a set of CPNs. The view on the right shows a VPG network, which connects the same set of CPNs. The relationship between VPs and VPGs defines the mapping between both views.

The VP controller, which participates in call setup and release in the enterprise network, operates on the left view. The controller decides whether a call can be admitted into the VPN, based on the VP capacity, its current utilization and the admission control policy. The VP controller behaves like an admission controller. It ensures that enough capacity is available, such that cell-level QOS can be guaranteed for all calls that are accepted. The controller runs on the time scale of the call arrival and departure rates (seconds or below). There can be one VP controller per VP, or one for a set of VPs.

The VPG controller operates on both views. Depending on the state of the VPs (in particular, traffic statistics and VP size) and the control objectives, it dynamically changes the amount of VPG bandwidth allocated to associated VPs. This controller enables customers to exploit variations in utilization among VPs that traverse the same VPG, allowing bandwidth between VPs of different source-destination pairs to be shared without interacting with the provider. In order to guarantee QOS, the sum of the VP capacities must be less than or equal to the capacity of the VPG link. The controller runs on a time-scale of seconds to minutes.

The VPN controller operates on the right view. It is the only controller which interacts with the provider, and it runs on the slowest time scale of all the controllers (minutes
or above). The VPN controller dynamically negotiates the bandwidth of the VPG links with the provider, based on traffic statistics and control objectives (e.g., minimizing the VPN cost), while observing the customer’s QOS requirements.

### 2.3.1 Controller Design

Figure 2.5 shows the functional design of a VP controller and a VPG controller according to our implementation. In this design, the VP controller includes two objects: a VC capacity allocator and a coordinator. The allocator receives requests from a VC connection manager in the customer domain. The coordinator changes the capacity of the VP upon request from the VPG controller. It changes the capacity of the VP only when the bandwidth requirement of the active calls in the VP does not exceed the new capacity.

![Functional model of a call admission controller interacting with a VPG controller.](image)

The VPG controller includes four objects. The trigger object periodically initiates the VP capacity allocator to run the VP allocation algorithm. The coordinator sends the new VP capacities to the coordinators of the associated VP controllers, using a synchronization
protocol. Finally, an estimator object collects statistics from the VP controllers. This data is used by the capacity allocator.

Obviously, there exist many ways of realizing the above design, with respect to control algorithms, mechanisms for trigger realization, synchronization protocols, and centralized or distributed implementation of the controllers. For example, the control system may include one VP controller per VP or one centralized controller for the whole VPN. The same applies for VPG control. Also, VP controllers can send bandwidth requests to VPG controllers, triggered by a pressure function, or a VPG controller can periodically recompute the VP capacities and distribute them to VP controllers. Similarly, the synchronization protocols between the VC admission controller and the VPG controller can be realized in different ways. One possibility is that the VP controller, upon receiving a request to change the VP size, checks whether the current utilization is above or below the new size. If the utilization is below, the VP size is changed and a confirmation is sent to the coordinator of the VPG controller. If it is not below, the VP size remains the same and a failure reply is sent instead. In another possible implementation, when the attempt for changing the VP size is not successful, the VP controller waits and blocks further calls from being admitted. As a result, the utilization of the VP can only decrease, as calls can leave but no new calls are admitted. When the utilization drops below the new size, the VP size is updated and the reply sent to the VPG controller.

A customer's choice for a specific design of the control system is based upon its control objectives and requirements for the control system, which relate to system size, expected traffic and signalling load, efficiency of resource control and robustness of the control system. In order to enable the realization of a large class of control objectives and control schemes, we have designed a generic controller as one of the building blocks of a customer control system. This generic controller enables many interaction patterns among controllers and is constructed in a modular way.
Figure 2.6 shows a functional model of the generic controller, which includes two sets of subcontrollers in a symmetrical design. One set of subcontrollers regulates the access to the resource, and the other set controls the size of the resource. The two sets of subcontrollers cooperate by accessing a shared data object, the resource graph. Each set of subcontrollers is made up of three functional components: trigger, allocator, and coordinator. The trigger decides when a computation should be done. The allocator performs the computation, which can be initiated by an external event or by the trigger. The allocator that controls the access to the resource computes the amount of the resource that should be given to a particular request. The allocator that controls the size of the resource determines the resource capacity. A change of the resource capacity is coordinated by the coordinator object, which facilitates the interaction with other controllers. In particular, it implements the synchronization protocol needed to ensure that state changes among distributed controllers do not violate a set of resource constraints. The resource graph is modeled as two sets of weighted graphs, one representing the resource allocation and statistics, the other the resource capacity. Interfaces are provided to access and modify the relationship among these graphs. The statistics on the graph are collected and updated by the State Estimator.
In our implementation, a generic controller is realized using C++ templates, where the actual classes for the subcontrollers, trigger, allocator, coordinator, etc. must be specified when the template class is instantiated. Interfaces offered by these subcontrollers are implemented as virtual functions that are overloaded for a specific realization of the controllers.

The design of the generic controller shown in Figure 2.6 has brought us the following benefits. First, it was possible for us to design and implement all three classes of controllers --VP controller, VPG controllers, and VPN controller-- as a refinement of the generic controller class. For example, the VP controller in Figure 2.5 has two “non-trivial” controller objects --the VC resource allocator and the coordinator-- and five “trivial” controller objects. (Trivial controller objects can be thought of as objects which perform no ac-
tion except that of forwarding data to another object. They are not shown in Figure 2.5). The VPG controller contains four non-trivial controller objects and three trivial objects.

Second, based on the generic controller design, we were able to realize different control schemes that attempt to achieve different control objectives for the customer control system. Realizing different control schemes is often possible by exchanging a set of subcontrollers in the system. For example, we implemented two classes of VC capacity allocators, realizing different VP schemes. One scheme aims at achieving call blocking objectives related to performance classes. The other scheme realizes call preemption in case of congestion, taking into account the priority of a call. In the same way, we have realized different synchronization protocols by building different classes of coordinator objects. One of these protocols is designed towards efficient use of VPN bandwidth, another towards guaranteeing fair access to the VPN capacity in case of overload conditions.

2.3.2 Enabling Management Objectives

The customer operates a management system to control and monitor the traffic on the enterprise network. A part of this system manages the traffic over the VPN. Examples of management capabilities that are related to the VPN service include controlling the bandwidth cost of the VPN service, VP bandwidth management, and QOS management. In the following, we describe how the management objectives outlined in Section 2.2.1 can be realized.
Figure 2.7 shows our framework for implementing management capabilities. In this framework, management parameters, which directly relate to management objectives, are mapped onto control parameters, which influence the behavior of the controllers, and are subsequently distributed to the controllers in the customer control system [PAC95]. In our implementation, management parameters are made available to the operator of the enterprise network through the management console (Figure 2.8).

A management parameter can be mapped onto control parameters for one or more classes of controllers. For example, cost management operations affect only the VPN controller. Allocating a specific capacity to a VP through a VP management operation affects both the VPG and the VPN controllers. QOS management operations, such as setting call blocking objectives, generally affect all classes of controllers. In response to a change in
blocking objectives, the VP controller adjusts its VP policy, the VPG controller changes the VP allocation strategy, and the VPN controller negotiates the VPG sizes according to the new bandwidth requirements.

Figure 2.8: The customer management console for a VPG-based VPN service. The upper layer represents the VP network, the lower layer the VPG network. The vertical bars on the VP network indicate the utilization, the vertical bars on the VPG network the allocation of VPG bandwidth to VPs.

Figure 2.8 shows the screen of the management console that we have implemented for customer management of a VPG-based VPN service. Both layers of the VPN are visible. The upper layer represents the VP network, the lower layer the VPG network. The vertical bars on the VP network show the current utilization of the VPs. The three segments of
a particular bar correspond to the three traffic classes supported in our particular system. The outline of the cylinders indicate the currently allocated VP capacities. The vertical bars on the VPG network give the allocation of the VPG bandwidth to the VPs. A "cloud view" on the lower left corner shows the number of active calls in the VPs. Each axis corresponds to a traffic class. The interface in Figure 2.8 allows an operator to perform management operations and observe the reaction of these operations on the global state of the system. The interface will be described in greater detail in Section 3.2.

2.4 QOS Guarantees and VPN Control Algorithms

From a performance point of view, there are two goals:

- **End-to-end QOS**: The network must guarantee cell-level QOS end-to-end to all connections it admits. The QOS should, as far as possible, be independent of the characteristics of the underlying public network services.

- **Efficient use of resources**: The architecture must support statistical multiplexing of the customer traffic within the VPN, and it must allow dynamic bandwidth renegotiation on all control layers in order to accommodate changes in the traffic characteristics within the customer network.

The first goal is closely related to the problem of defining an abstraction of the VPN capacity for the customer network. Here we understand the term capacity as a call-level abstraction, defining how many connections of a given class can be supported by the VPN, while observing the cell-level QOS requirements of each such connection. VPN services may be provided by multiple providers and it is likely that each of these providers use different QOS guarantees. As a result, we argue that the requirements on the provider necessary to construct such a capacity abstraction should be minimum, for example, based on services that are already commonly available today. Our approach is presented in Section
2.4.1. The second goal is achieved by implementing a set of control algorithms according to the control architecture outlined in Section 2.3. The algorithms implemented are described in Section 2.4.4 and Section 2.5. Finally, in Section 2.6, we present an evaluation of the performance of a system prototype implemented based on simulation.

2.4.1 Constructing a Call-Level Abstraction with Cell-Level QOS Guarantees

The network model we assumed is a multiclass network supporting a finite number of traffic classes like voice, video, and data, each of which is defined by two sets of parameters: the traffic characterization and the QOS requirements. The QOS requirements can be seen as constraints under which the real-time control system of the customer network must operate. They include parameters like the maximum cell delay, the maximum error rate, and the minimum average throughput per connection.

Different customers have different requirement on their networks, including defining their own real-time and non real-time traffic services (or classes) with specific cell-level QOS requirement. The customized QOS requirements are built upon services provided by multiple providers. Therefore, an approach is required to handle the different QOS schemes between the customer domain, and the VPN domains.

We approach the problem in two steps. First, we define a call-level capacity abstraction called the Schedulable Region [HYM93]. Use of schedulable region allows the customers to define their own traffic classes according to their network requirements and provides a unified notion of resource capacity abstraction independent of cell-level algorithms and hardware specifics. Second, we describe how such an abstraction can be built using CBR VP services. We believe that the use of CBR VP is a reasonable approach because this service is one of the basic ATM services defined by the ATM Forum and is relatively easy to provide. In fact some public network providers have already begin to offer such services [ATS93].
2.4.2 Cell-Level QOS Guarantee through Schedulable Region

The task of the admission controller is to accept or reject calls so as to maximize some utility function under the constraint that the required QOS at the cell-level to every call admitted into service can be met. From the point of view of the admission controller, the link capacity can be expressed by its schedulable region $S$, which defines how many calls of a given class the link can support, while guaranteeing the appropriate cell-level QOS to each class. The cell-level QOS for a traffic class is specified in terms of bounds on loss probability and delay experienced by cells through a network multiplexer. The schedulable region $S$ is an $N$-dimensional space, where $N$ is the number of classes (such as voice, video, data) recognized by the link controller. The resource state is defined by the occupancy vector $x$, which represents the number of calls of each class currently active in the link. In order to guarantee cell-level QOS to each class of traffic, the occupancy vector can assume only values that are inside the schedulable region. In general, the size of the region will depend on the statistical characteristics and cell-level QOS constraints of each class of traffic, as well as on the details of the scheduling policy in use.

For illustration purposes, we estimated the size of the scheduclable region for a 40 Mb/s link through simulation. In the simulation, we defined two traffic classes. The first traffic class (Class I) is for video. It is characterized by 24 frames per second, peak rate of 6Mb/s, average rate of 1.32 Mb/s and the cell-level QOS constraints are maximum delay of 1 ms through the multiplexer and no cell loss. The second traffic class (Class II) is defined for voice traffic and is modeled as on-off source with constant arrivals with an exponential distributed active period and 64Kb/s peak rate. The cell-level QOS requirements for class II are maximum delay of 1 ms and no more than 2% cell loss. A static priority scheduling is used. The schedulable region of a multiplexer with a 40Mb/s link is shown in Figure 2.9. The line in the figure delimits the schedulable region of the multiplexer.
Figure 2.9: Schedulable region of a 40 Mb/s link

The admission controller must ensure that the number of class I and class II calls admitted onto the multiplexer is a combination below the boundary of the schedulable region. For an end-to-end connection, cell-level QOS guarantee is satisfied if at each multiplexer, the operating point (the combination of the class I and class II after the call is accepted) is below the schedulable region and if the following constraints are satisfied.

**Delay Constraint**

$$\sum_i s_{\text{multiplexer}_i} + \sum_j s_{\text{link}_j} < s$$

**Cell Loss Constraint**

$$\sum_i \epsilon_{\text{multiplexer}_i} + \sum_j \epsilon_{\text{link}_j} < \epsilon$$

where $s_i$ is the maximum cell delay at the multiplexer (or link) $i$, $\epsilon_i$ is maximum cell loss at the multiplexer (or link) $i$, and $s$ and $\epsilon$ are the end-to-end delay and loss constraints respectively.

The concept of a schedulable region allows a separation between cell and call control. As a result, the call admission control policy used in this framework operates only on
call level abstractions. Cell-level QOS guarantee is enforced by restricting the occupancy state of the link to below the schedulable region. The admission control policy specifically deals only with call-level QOS like blocking probability. This approach is very different from many existing admission control algorithms where the admission controller operates on both call and cell statistics (see for example [LEE96]).

2.4.3 Constructing a Schedulable Region using CBR VP

In order for a scheme that uses schedulable regions to work, the traffic classes defined by the customer must be recognized by all of the multiplexing units. However, in the case of a Customer Network (CN), it is unlikely that this is true in the Virtual Private Network (VPN) domain, which is built on top of the public network. In the rest of the section, we will show how a homogenous view of cell-level QOS guarantee can be built even though the Customer Premise Network (CPN) cell-level QOS guarantee uses a cell multiplexing scheme different from the Public Network (PN) using Constant Bit Rate (CBR) Virtual Path (VP).

To discuss the abstraction of communication resources within the CPN domains, we refer to a general architecture of a broadband switch that is based on a non-blocking switch fabric (see Figure 2.10). Traffic arriving at an input link of the switch is routed onto one of the output links. Note that the critical resources in the switch are the output ports, each of which is controlled by a link resource controller.
Given the cell-level QOS constraints, the size of the schedulable region is determined by the following:

- size of the output buffer
- the buffer management algorithm
- the capacity of the output link
- the scheduling algorithm
- the cell-level traffic statistics

We exploit the fact that both physical link and CBR VP are characterized by the same parameters, namely: bandwidth, cell delay and cell loss ratio. In both physical link and CBR VP, delay can be caused by propagation (which depends on the length of the cable) and transmission (which depends on the size of the packet and the speed of the link); cell loss on a physical link can be caused by errors in transmission or synchronization. For a CBR VP, delay can also be caused by switching and queuing, and cell loss through con-
tention for buffer space in the public network domain, all of which are under the control of the network provider.

The following table summarizes the differences in factors contributing to the loss and delay.

<table>
<thead>
<tr>
<th>Physical Link</th>
<th>Delay</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Propagation Delay (Dp1)</td>
<td>Cell loss due to physical error (Ep1)</td>
</tr>
<tr>
<td></td>
<td>Transmission Delay (Dp2)</td>
<td></td>
</tr>
<tr>
<td>CBR VP</td>
<td>Propagation Delay (Di1)</td>
<td>Cell loss due to physical error (El1)</td>
</tr>
<tr>
<td></td>
<td>Transmission Delay (Di2)</td>
<td>Cell loss due to buffer contention (El2)</td>
</tr>
<tr>
<td></td>
<td>Switching Delay (Di3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Queuing Delay (Di4)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: QOS parameters for physical and logical links

For the purpose of constructing a schedulable region, the important differences between a physical and a CBR VP is the difference in delay and cell loss characteristics and that the capacity of a CBR VP (therefore the capacity of the output link) can be changed dynamically.

Thus, given a CBR VP (of a specific bandwidth) with bound on maximum delay and loss, construction of schedulable region can proceed as if the output link is a physical link with the same bandwidth. The delay and loss characteristics of the CBR VP is given by the network provider as part of the service contract. The size of the schedulable region of the combination of the multiplexer and logical link of 45 Mb/s will be the same as if we have a 45 Mb/s physical link, with modifications made to the cell-level QOS provided to take
into account the additional delay and loss due to queuing, transmission and switching introduced by the CBR VP.

![Diagram](image)

**CPN**  
Switch output port (multiplexer)  

**PN**  
Logical Link

Figure 2.11: Construction of scheduleable region using CBR VP

Using such an approach, the customers can choose traffic classes with QOS specifications that are totally independent of the provider. Negotiation of resource between CPN and PN is a matter of finding the logical link bandwidth corresponding to the desired scheduleable region size.

As it is expected that bandwidth negotiation is done in discrete step size, the mapping between logical link size and scheduleable region size can be computed off-line by the customer and put into a table. An example of such a table is given in Table 2.3. The table has two columns. The right column contains possible bandwidth of the logical link, in this case measured in terms of megabits per second (Mb/s). The left column contains the corresponding size of the scheduleable region for each logical link size. For example, if the size of the logical link is 15 Mb/s, the size of the scheduleable region is Region15. By constructing such a table, dynamic bandwidth negotiation between the CPN and PN can be done in units of Mb/s, even though within the customer domain, bandwidth negotiation is interpreted in terms of scheduleable region.
Table 2.3: Mapping between Link Bandwidth and Size of Schedulable Region

<table>
<thead>
<tr>
<th>Logical Link Capacity (MB/s)</th>
<th>Schedulable Region of Multiplexer</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Region10</td>
</tr>
<tr>
<td>15</td>
<td>Region15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>Region100</td>
</tr>
</tbody>
</table>

With the resource abstraction given in terms of schedulable region, control algorithms can now be designed so that they do not deal with the specifics of cell-level multiplexing and instead deal only with call-level capacity abstractions. This is the topic of the next section.

2.4.4 Review of VP Control Algorithms

Many classes of control algorithms can be implemented using the framework provided by the functional model and generic controller design described in Section 2.3. In particular, a large class of VP bandwidth allocation algorithms that was previously used mainly in the provider's domain can now be applied in the customer domain. Using the architecture framework described in Section 2.3, we characterized these VP algorithms according to the time-scale and the controller on which they run on.

Algorithms in the VPG and VPN controllers run on medium or slow time-scale. These algorithms tend to be centralized, and capacity distributions are computed based on medium- to long-term measured traffic statistics. For a survey of some of these algorithms see [ANE96]. The time-scale on which these algorithms run is very important to their design and usually statistics have to be collected over a sufficiently long period for these algorithms to be applied. In fact, some of these algorithms are used by the network provider for dimensioning of the network, which occurs on a very slow time-scale. In [LOG95] a “medium time-scale” VP redistribution algorithm is proposed. In this paper, the author
highlights the importance of choosing the correct redistribution period. It is argued that the bandwidth reallocation time (BRT) should be large enough so that existing traffic has the time to leave the network. For traffic with exponential holding time of mean 100s, they suggest a BRT of 30 minutes (1800s).

Another class of VP bandwidth reallocation algorithms are state-based, therefore changes in the bandwidth are triggered by the current state of the system. Due to their fast time-scale reaction, they usually run in the VP controllers, and sometime in the VPG controllers. For example, [OHT92] and [ORD96] describe two such state-based, dynamic VP bandwidth renegotiation algorithms that run in the VP controllers. In [OHT92], a fixed amount of bandwidth is requested when the utilization of the VP crosses a high "watermark". On the other hand, bandwidth is released when the utilization crosses a low "watermark". A similar approach is used in [ORD96]. The difference (and improvement) comes from requesting and releasing variable amounts of bandwidth, of which the amount of bandwidth changes depends on the call arrival and departure statistics. The algorithm described in [ORD96] is more efficient but is significantly more complex to implement than the algorithm in [OHT92]. The time-scale on which these algorithms run depends on the call arrival and departure statistics and the trade-off between efficiency in bandwidth usage and signalling load.

Control across different control layers can be coordinated in order to achieve a possibly better system performance by organizing the system as a hierarchical structure. This is the approach taken in [PIT95], where multilevel optimal control is performed. A three-level hierarchically organized control structure is used. The overall objective is to optimize an objective function that minimizes the sum of the differences between the desired and measured delay and switching bandwidth allocated to a specific traffic class within a VP. On the lowest level, the local controller solves a set of low order linear difference equations that assumes that the predictions on the queue state and allocated service rate are exact. On
the higher level, the controllers attempt to force the prediction of the delayed terms equal to their true values. Successive iterations (if they converge) will ensure that the predicted trajectories are the same as the true trajectories. It should not be difficult to see that this control structure can be implemented in a rather straightforward way using our architectural framework.

2.5 Realization of the Control System

In this section, we provide a detailed description of an implementation of the control system architecture described in Section 2.3. As we would like to highlight the new possibilities available due to the introduction of the VPG concept, we chose to implement a state-based VP allocation algorithm. Specifically, since VP bandwidth allocation can be performed entirely within the customer domain without interacting with the provider, we implemented a centralized VP algorithm that exploits multiplexing on the VPGs among VPs from different source-destination pairs. This algorithm runs on the VPG controller and periodically redistributes the VP bandwidth based on the current state of the VPs, and the measured call arrival and departure statistics. The bandwidth of all VPs is changed simultaneously, and needs to be performed frequently for efficient distribution of VP capacities. In a VP-based VPN, such an algorithm will be considered impractical due to the frequent interactions it needs to make with the public network provider management system. It is, however, a reasonable choice in a VPG-based VPN, since all control interactions are performed within the customer domain.

This algorithm is different from those described in [OHT92] and [ORD96] in the following way. First, although redistribution is based on the current state of VPs, the algorithm is centralized and runs in the VPG controller, instead of being distributed and running on the VP controller. Second, redistribution is performed periodically, every T seconds, instead of being triggered by a change in the state of the VP. The parameter T provides an
explicit and direct way in which the management system can influence the behavior of the VPG controller.

2.5.1 VP Controller

The admission control algorithm implemented on the VP controller is based on the Multi-dimensional Threshold Control policy (MDT), a form of complete partitioning (CP) policy [HYM91]. This policy is designed to achieve the blocking constraints for each class by the imposition of thresholding rules. Let \( z \) be the threshold vector, \( S \) be the schedulable region and, \( x = (x^1, ..., x^n) \), where \( x^i \) is the number of calls in the system of class \( i \); then define the policy \( u \) as

\[
u^i(x) = \begin{cases} 
1 & \text{if} \left( \{ x^i < z^i \} \land \{ x + e^i \in S \} \right) \\
0 & \text{otherwise}
\end{cases}
\]

where \( e^i \) is the elementary vector with 1 at position \( i \) and 0 elsewhere and \( u^i(x) = 1 \) if a call of class \( i \) can be accepted, and 0 if otherwise.

In this policy, each class, in effect, is allocated its own dedicated bandwidth, and no call-level interference between classes takes place. Modeling call arrivals as Poisson processes and assuming that the holding time of calls is exponentially distributed, the blocking probability \( p^i \) for class \( i \) is given exactly by \( p^i(u) = E(\lambda^i / \mu^i, z^i) \) where \( E(A, N) \) is the one-dimensional Erlang-B formula:

\[
E(A, N) = \frac{A^N / N!}{\sum_{i=0}^{N} A^i / i!}
\]

where \( A^i \) is the blocking objective for class \( i \). The threshold \( z^i \) is thus chosen as the smallest (or minimum) amount of bandwidth \( N \) such that the blocking probability calculat-
ed using the one-dimensional Erlang-B formula is less than or equal to the blocking constraint $\kappa^i$. Mathematically, this is expressed as the following equation:

$$z^i = \min_{E(\lambda^i/\mu^i, N) \leq \kappa^i} N$$

The CP admission control algorithm used has the advantage that it tends to divide the resources "fairly" among the traffic classes according to the blocking objectives. By computing the thresholds used in the policy dynamically during run-time, the algorithm is also able to adjust to variations in traffic load. Perhaps, more importantly, the behavior of this algorithm can be influenced by management operations by changing the values of $\kappa^i$, the blocking objective for class i traffic. Changing $\kappa^i$ causes the corresponding threshold $z^i$ used by the MDT algorithm, and thus the admission policy, to be changed.

Conceptually, the surface of the schedulable region can be highly irregular. However, the storage and manipulation of a general N-dimensional space can be rather expensive. Therefore, all surfaces are approximated as a N-dimensional hyperplane, which can be expressed in the form $\sum x^i / N^i = 1$, where $N^i$ is the maximum number of calls of traffic class i that can be allowed into service. Algebraically, the approximation involves finding all the $N^i$. There is, however, no unique solution to the approximation. As shown in Figure 2.12, more than one possible approximation is possible. (In fact, there are infinitely many of them). One way to decide on an approximation is to fix the ratio $N^i/N^j$, for all i not equal to j. Some possibilities for these ratios are to let $N^i$ equal to the average bandwidth, peak bandwidth, utility or some combinations. Another approach is to choose the surface that maximize the utility function $\sum u^i N^i$, where $u^i$ is the utility of class i calls. Figure 2.12 shows two possible approximations for a 2-dimensional schedulable region.
Figure 2.12: Approximating the scheduleable region with a hyperplane.

Depending on the size of the scheduleable region, it is possible that the hypercube formed by the boundaries of all the thresholds will be either under or above the boundary of the scheduleable region. In the case where the hypercube is below the boundary of S, the thresholds are increased to allow for better utilization of the resources; in the case where the hypercube is above the boundary of S, the thresholds are decreased so as to maintain some sort of “preference” among traffic classes. The heuristic in this case is to use a new set of thresholds \( z^i_p \) that forms the largest hypercube that is below the boundary of S and such that the ratios among thresholds are maintained. Since the scheduleable region is represented as a hyperplane, the new thresholds can be computed using a simple vector projection operation given as follows:

\[
F = \frac{1}{\sum_{i}^{i} \frac{z^i_p}{N^i}} \quad z^i_p = F \times z^i
\]

Figure 2.13 illustrates the projection operation F graphically. In Figure 2.13(a), the thresholds calculated form a rectangle below the boundary. In order to increase utilization, the thresholds are increased by the projection operation into a larger box. Figure 2.13(b)
shows the reverse case where the rectangle forming the computed threshold is above the boundary. The thresholds are decreased using the projection operation so that the new thresholds are smaller and the rectangle is below the boundary surface.

![Diagram](image)

**Figure 2.13: Projection of thresholds**

During durations of overload, when the demand for bandwidth is much higher than the amount of bandwidth available, the control objective could be different from control under normal conditions. In particular, during periods of overload, it might be more important to guarantee the blocking constraints of certain classes of traffic than for others. Such control objectives can be achieved through the use of a priority scheme.

A straightforward implementation of a priority scheme is to accept a call of higher priority as long as there is sufficient capacity available. Pre-emption of calls of lower priority is allowed if necessary. Such an implementation poses a problem in terms of blocking objectives in the sense that calls of high priority do not experience blocking at all, as long as there are calls of lower priority in the system. The blocking probability of calls in a certain traffic class then depends only on the capacity of the VP, and has nothing to do with the blocking constraint of the class.
One possible solution is to define two separate objectives. One objective deals only with blocking constraints and the other deals only with priority. Admission control can thus be executed with either one of these objectives. Blocking constraints can be used for example in normal load conditions, and priority used in overload conditions.

The approach we took is to integrate both objectives into the same framework. Each call is associated with a performance characterization and a priority. Priority is considered purely as a policy (when there is no spare capacity, high priority pre-empts low priority calls). Blocking remains then as the only objective to be met, and the priority of a call implies how faithfully (or with what priority) its blocking objective should be respected. A variation of the MDT algorithm enhanced with the notion of priority, is described as followed:

(1) When a call with blocking objective $K^i$ and priority $L^i$ arrives, if the link utilization is less than $\alpha$ (< 1.0), accept the call or else go to (2).

(2) Accept the call with probability $(1-K^i)$ and reject the call with probability $K^i$. If the call is to be accepted and the VP is full, pre-empt the minimum number of calls with lower priority such that this call can be accepted. If there are not enough calls of lower priority in the system such that this call can be accepted, the call is blocked.

The algorithm implemented has the following characteristics. First, in times of overload, calls of higher priority will experience actual blocking probabilities close to the specified objectives (as long as there are calls of lower priority to pre-empt). Second, when the value $\alpha$ is set to a value less than 1.0 and the system is not heavily utilized, calls of higher priority can take advantage of the availability of resources and will experience blocking probabilities less than the specified objective. This algorithm has the advantage that it works with any level of priorities. Experience from the simulation shows that the algorithm is simple to implement, executes very efficiently and, the blocking experience of high priority traffic converges to the blocking objective quickly in overloaded conditions.
2.5.2 VPG Controller

In order to exploit statistical multiplexing among VPs from different source-destination pairs, the VPG controller periodically redistributes the capacities among the VPs. The algorithm, which is state-based, takes into account the blocking objectives, the period of redistribution, the current operating point of the VPs, and the call arrival and departure statistics measured.

The implemented centralized algorithm runs periodically. The algorithm executed in each cycle is as follows:

Step (0): Check if the previous computation has terminated. If it has not, go to (5)

Step (1): For each VPG, calculate the weight of VP j on VPG i. Let this weight be $W^j_i$. Based on this weight, calculate the bandwidth of VP j ($BW^j_i$) is getting from VPG i, whose bandwidth is $BW_i$ using the equation

$$BW^j_i = \frac{W^j_i}{\sum_j W^j_i} \times BW_i$$

Step (2): For each VP j, calculate the amount of bandwidth it is allocated, $BW^j$, using the equation:

$$BW^j = \min(BW^j_i)$$

Step (3): If there is any unassigned bandwidth left in any VPG, form a candidate list containing all the VPs. Pick a VP at random and assign it the maximum possible bandwidth. Remove this VP from the list. This step ends when either all VPG bandwidth has been assigned or there are no more VPs left in the list.

Step (4) (a): Divide the VPs into two groups: “small” and “large”. VPs whose new allocation is less than the current allocation is in the “small” group, or else it is in the “large” group.
Step (4) (b): Send out an allocation of VPs in the “small” group and wait for acknowledgments from all these VPs.

Step (4) (c): All acknowledgments from the “small” group have been received. If the allocation fails on any of the VPs, the allocation fails. The original VP bandwidth is sent out to all VPs in the “small” group to execute a rollback operation. If all allocation on “small” VP succeeds, send bandwidth to “large” VPs.

Step (5): End of computation cycle.

Step (4) is necessary due to the fact that the states of the VPs are distributed. Changing these states has to be coordinated so that at any time, the sum of the capacities allocated to the VPs do not exceed the capacity of the VPGs they pass through. The solution is to divide VPs into two groups. The first group (“small” group) consists of VPs whose new allocations are smaller than the current allocation; the second group (“large” group) consists of VPs whose new allocations are larger or equal to the current allocations. In the first phase, only allocations of VPs in the “small” group are sent, in the second phase allocations of VPs in the “large” group are sent. Note that VP controllers who receive allocations in the second phase have larger or equal allocation and therefore have no problem.

When the VP controller receives a request for a change in bandwidth and cannot execute the request immediately, it can react in two ways. It can either reply immediately with an unsuccessful message or it can stop accepting new calls, while allowing existing calls to depart. Eventually, its utilization will go below the new allocation, and when this happens, the VP controller replies with a successful message and can again accept new calls. The first solution has the advantage that the duration of an allocation cycle is bounded and is usually very short. It can, however, fail to change the VP bandwidth frequently when the system is heavily loaded. As a result, VPs that experience a heavy load earlier tend to have more bandwidth allocated. On the other hand, the duration of an allocation cycle in the second
solution can be long. However, even with a heavy load, the allocation will eventually succeed.

Using a reasonable value for the period of computation between 10 to 30 seconds, a computation cycle in both algorithms usually terminates before the start of the next cycle. If the previous computation has not been completed, a new allocation is not allowed to start so that only one allocation process is allowed at a time (Step 0).

If the VPG controller receives one or more unsuccessful messages, it aborts all changes performed in the first phase. This operation is similar to the roll-back operation in a 2-phase commit protocol used in database management system. The second phase will not be executed.

Note that it is not specified in step (1) how the weight of a VP should be computed. In the following, we present two possible approaches to computing the weight of a VP. The first approach is computationally intensive and is not suitable for real-time control. It can, however, be considered as the reference case for the second approach, which is based on a heuristic that attempts to approximate the first approach.

2.5.3 Exact Approach to Weight Computation

We formulate the exact approach as follow. For a given traffic class \( x \), we model the VP as a M/M/C/C queue where \( C \) is the maximum number of calls of class \( x \) it allows into service. The dynamic behavior of the states of the VP is described by the Chapman-Kolmogorov equation for continuous-time Markov chains [KLE75]:

Let \( P_{ij}(t) \) be the probability that the state of the VP goes from initial state \( i \) to state \( j \) in time \( t \). The set of differential equations that describe the transition from state \( i \) to state \( j \) in time \( t \) are:
\[ \frac{dP_{i0}(t)}{dt} = -\lambda P_{i0}(t) + \mu P_{i1}(t) \]

\[ \frac{dP_{ij}(t)}{dt} = \lambda P_{i,j-1}(t) - (\lambda + \mu)P_{ij}(t) + \mu(j+1)P_{i,j+1}(t) \]

\[ \frac{dP_{ic}(t)}{dt} = \lambda P_{i,c-1}(t) - \mu CP_{ic}(t) \]

\[ P_{i,c+n}(t) = 0, n > 0 \]

where \( \lambda \) = arrival rate of class \( x \) calls and \( \mu \) = service rate of class \( x \) calls. These equations can be rewritten in matrix form as:

\[ \frac{dP(t)}{dt} = AP(t) \]

where \( P \) is the matrix whose element at the \( i \) row and \( j \) column is \( P_{ij}(t) \). Solving these differential equations translates into finding the eigenvalues and eigenvectors of a CxC tridiagonal matrix. The solution is given in [RIO62] as:

\[ P_{ij}(t) = \frac{\rho^j / j! + C^{-i} \sum rD_i(r)D_j(r)e^{r\mu}}{\sum \rho^j / j!} \sum R \sum D_C(r)D_C(r+1) \]

where \( \rho = \lambda/\mu \), \( r \) is an eigenvalue of \( A \) and \( D_i(r) \) is defined recursively as:
\[ D_0(r) = 1 \]
\[ D_1(r) = \rho + r \]
\[ D_{n+1}(r) = (\rho + n + r)D_n(r) - \rho n D_{n-1}(r) \]

The eigenvalues of \( A \) are real and simple (pp. 82, [RIO62]). The solution to \( P_{ij}(t) \) nicely captures the dynamics of the transitional behavior of the multi-server queue. The first term of the equation is the equilibrium probability \( p_j \), forming a truncated Poisson distribution. The largest negative eigenvalue, say \( r_1 \), governs the rate of approach to the steady state. For \( \rho \) much larger than \( C \), this root is near \(-\rho\); for \( \rho = C \), it is \(-2\); it approaches \(-1\) as \( \rho \) approaches 0.

Let \( K^x \) be the blocking constraints associated with each traffic class \( x \). Given that the state of the VP at time \( t \) is \( i \), and the period of bandwidth recomputation is \( T \), the smallest amount of bandwidth needed to satisfy the required blocking constraints is the smallest \( C \) that satisfies the following equation:

\[ \frac{1}{T} \int_{t}^{t+T} P_{iC}(t)dt \leq K^x \]

The value of \( C \) for each traffic class is computed separately. The weight of a VP is the bandwidth that corresponds to the smallest schedulable region that contains the hypercube formed by these thresholds.

Note that by using these weights, if sufficient capacity is available in the network, the VP controllers will always be able to guarantee the cell- and call- level QOS guarantee.
In order to show that the blocking probability can be calculated this way, we need to show that the PASTA property holds in a finite time. This is in fact true. The proof can be found on pp. 295 of [WOL89] and will not be reproduced here.

On an IBM/390 series workstation, using the LAPACK software[AND94], the computation time for finding the eigenvalues and eigenvectors of a C x C matrix are as follow:

<table>
<thead>
<tr>
<th>Dimension (C)</th>
<th>Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 x 100</td>
<td>0.26</td>
</tr>
<tr>
<td>250 x 250</td>
<td>2.44</td>
</tr>
<tr>
<td>500 x 500</td>
<td>17.8</td>
</tr>
<tr>
<td>1000 x 1000</td>
<td>226</td>
</tr>
</tbody>
</table>

Table 2.4: Performance of LAPACK on an IBM/390

Numerical instability, in the form of complex and non-simple eigenvalues, is observed when the ratio $C/\rho$ is too big. For example, if $\rho = 30$, all eigenvalues computed are real and simple only if $C$ is less than or equal to 113.

Based on the actual eigenvalues computed, it is further observed that all eigenvalues must be used in the computation of $P_{ij}(t)$, even if these eigenvalues are very small (which should make $e^{\lambda t}$ very small). This is because the coefficient of the terms can be very large, and the final product is not negligible.

This exact approach to weight computation is obviously not practical for a real-time implementation, but it serves as a good reference for judging the performance of the heuristic algorithm to be presented next.

2.5.4 Heuristic Approach to Weight Computation

The heuristic weight computation is an attempt to approximate the weight computed in the exact case and is given by the equation below:
\[ Weight = \sum_{x} \frac{-\log K^x + (\rho \times T \times 0.01) + i}{N^x} \]

The heuristic is based on the assumption that when the queue is empty, a certain minimum amount of bandwidth is needed to handle the anticipated arrival within the next computation period. The additional amount of bandwidth required increases linearly with the number of calls in the system. For each traffic class, there are two components to the weight. The first is the initial offset, which depends on the blocking objective, the traffic intensity and the computation period. The second component varies linearly with the current number of calls. The denominator is the normalization constant. A comparison with the exact values obtained in Section 2.5.3 shows that this approximation is reasonably close to the exact values in the range of T between 1s and 100s.

2.6 Evaluation of the Control System

The performance of the algorithms described in Section 2.5 was evaluated in a scenario based on the topology of the NYNET testbed. Only the VPG and VP control layers were implemented. An important characteristics of our evaluation is that we run an executable model of the control system that runs on top of the emulation platform described in Chapter 3. Each controller is implemented as a C++ object, and we simulate the exchange of signalling messages. Both message passing and processing delay are modeled in the simulation.

In this scenario, a VPN service interconnects 6 CPNs. The VPN contains 14 unidirectional VPGs which support 30 unidirectional VPs, connecting the 6 CPNs in a full mesh topology. The two VPGs in the middle carry 9 VPs, the remaining VPGs carry 5 VPs each (Figure 2.14). The visualization of this network topology is shown in Figure 2.8.
We model the network traffic load, the processing time of controllers and the time delay to send a message from one controller to another. For the sake of simplicity, all message processing time and message delivery time is set to 1ms. The computation time for a new VP distribution is assumed to be 100ms in all cases.

The network traffic is composed of two classes with different bandwidth requirements. A class 1 call needs 10 units of bandwidth, while a class 2 call requires 1 unit of bandwidth. The holding time of the calls of both classes is exponentially distributed with a mean of 100 seconds, and call arrivals are modeled as Poisson processes. The arrival rate per VP for class 1 call is 0.12 call/second and class 2 call 0.50 call/second. Finally, the blocking objectives are 0.10 for class 1 calls and 0.01 for class 2 calls. All VPs in the VPG
link experienced the same offered load. The VPG links between Syracuse and White Plains have 180 units of bandwidth, and all other VPG links have 100 units of bandwidth.

We vary the bandwidth computation period (T) in the experiment and measured network blocking rate for three cases:

- Static VP bandwidth allocation. Each VP is given 20 units of bandwidth and the MDT algorithm is used for admission control.

- Dynamic bandwidth VP bandwidth reallocation using exact weight computation and the MDT algorithm is used for admission control.

- Dynamic bandwidth VP bandwidth reallocation using heuristic weight computation. A complete sharing policy is used in the admission control algorithm.

Figure 2.15 and Figure 2.16 show the blocking rates for both class 1 and class 2 calls. From the figures, it can be seen that both the exact and heuristic algorithms perform significantly better than static allocation. The blocking objectives cannot be met by static allocation but can be met by the two dynamic algorithms in most cases. As expected, the performance of the exact algorithm deteriorates as the computation period increases. In fact, when T is large enough, the exact algorithm becomes the static allocation algorithm. As a result, the performance of the heuristic algorithm can be better than the exact algorithm when T is large. From Figure 2.15, we can see that for T > 30s, the heuristic algorithm out-performs the exact algorithm for class 1 calls. On the other hand, it is also true that if it is possible to use a T that is small enough, the blocking rate of a system running the exact algorithm can be made very small. This is not true in the case of the heuristic algorithm.
Figure 2.15: Performance of VPG Algorithms for Class 1 Traffic

Figure 2.16: Performance of VPG Algorithms for Class 2 Traffic
2.7 Conclusion

In this chapter, we have presented the design of a VPG-based VPN. The advantage of a VPG-based VPN is that it allows better customer control capabilities compared to current approaches. The control architecture is organized according to time scales, and within each time scale, statistical multiplexing is exploited so that resources can be utilized more efficiently. Allowing the customer more control over the allocation of its resources also means that in times of high network load, the customers can partition the resource or prioritize the traffic according to their needs. An approach to integrate blocking objectives into a priority scheme is also proposed. In order to provide QOS guarantee over heterogeneous network, we proposed an approach for abstracting the capacity of a VP by constructing a schedulable region using CBR VP.

The performance of two set of control algorithms, each consisting of a VP admission control algorithm and a VP reallocation algorithm is compared. As a reference, the performance of static allocation is also included. The results show that a more computational intensive algorithm works well for a small recomputation period (<30s). When the recomputation is larger, a simple heuristic algorithm works better. Both algorithms are better than static allocation.
3.1 The Prototyping Approach

Two main tasks are involved in the development of a network control system: the development of a software system and the design and analysis of control algorithms. The first task focuses on the software engineering aspects that satisfy the system requirements. The second concentrates on developing control functions that meet performance objectives. So far, these tasks have often been carried out separately, using different tools and environments.

The first task is usually performed on testbeds, where the feasibility of providing high-speed connectivity and multimedia communication capabilities among small groups of users is demonstrated [CLA92, FRA92, STO92, CHA96d]. Unfortunately, the results produced on such a testbed are generally restricted to a specific system configuration and cannot be used to predict the behavior of the system when the number of users, services, and network nodes are increased. Furthermore, it is hard to perform experiments on network testbeds because of the intrinsic difficulty in instrumenting and monitoring a distributed system. Finally, it is hard to quickly modify a system on a testbed, since there is generally no homogenous development environment available. As a result, performance studies related to scalability are performed on network simulation tools (for a survey of such tools, see [COM94]), where the processing and communication delays can be controlled and modeled, and the configurations of the system varied, to a large extent, arbitrarily.
A problem with separating these two tasks is that a thorough evaluation of the performance characteristics of a network control system has to take into account both system design and control algorithms. Performance evaluation studies that model only the control algorithms do not take into account the dynamic behavior and complex interactions introduced into the network control system by software engineering decisions. The process of system development and performance evaluation should be combined and performed in an integrated way.

The advantages of having a prototyping environment that emulates the target environment and that allows the system to be developed incrementally and revised after each developmental step is well-established in hardware design. Tools like Field Programmable Logic Array (FPGA), Very High-Level Hardware Description Language (VHDL) and software emulation have been created for this purpose. We believe that a similar approach to network software prototyping would be tremendously useful.

With this goal in mind, we built a network emulation platform which allows us to develop and evaluate prototypes of network control systems and has the following characteristics:

- First, the platform includes a parallel simulation kernel which runs on two types of parallel computers—the KSR-1 and the IBM SP2. The KSR-1 used has 128 processors and the SP2 has 512 processors. The availability of large amount of processing resources allows scalability to be studied.

- Second, the platform provides a homogenous programming environment with base object classes to build network controllers and with support for communication among controllers. In order to provide a similar environment for software development, we designed the platform in such a way that the simulation is transparent to
the developer of a network architecture. This means that a user of our platform needs little (parallel) simulation knowledge. The fact that the platform contains a discrete parallel event simulation kernel is transparent to the users.

- Third, the platform provides us with flexibility for testing purposes similar to other simulation environment. Traffic generators, for example, allow us to study the system under different load patterns. Different system configurations, including the network topology, are defined in files that can be changed from run to run.

- Finally, the platform supports interactive control and dynamic visualization of the system during run-time. For visualization purpose, the evolution of simulation time can be forced to run in lock step with a (scaled) wall-clock time. We can run what-if scenarios and observe dynamic phenomena such as oscillation of the network state or transient behavior in “real-time”. The visualization front end is implemented using OpenInventor and runs on SGI graphics workstation.

This platform has been used in several projects which are aimed at developing and evaluating network architectures [CHA96a, CHA95a, PAC95] and has allowed us to evaluate the characteristics of the system before the implementation phase on a testbed. Its use has brought to our attention additional, previously unknown or unnoticed system requirements and also discovered unanticipated side effects of design decisions.

The rest of this chapter is organized as follow. The design of the prototyping platform is described in Section 3.2. In Section 3.3, the Virtual Private Network architecture presented in Chapter 2 is used an example to illustrate the prototyping process. Finally, in Section 3.4 and Section 3.5, the experience gained from using the high-performance platforms is presented.
3.2 Design of the Emulation Platform

The emulation platform consists of four building blocks: parallel simulation kernel, emulation support, real-time visualization and interactive control, and emulated system (Figure 3.1). The emulated system and emulation support modules consist of a set of objects that communicate by exchanging messages, using functions provided by the simulation kernel. The emulated system module represents the prototype of the network control system under evaluation.

![Diagram](image)

Figure 3.1: Building blocks of the interactive emulation platform.

The simulation kernel controls the execution of these objects and ensures that messages are processed in the correct order. It implements the paradigm of parallel discrete event simulation (PDES) [FUJ90], using a variation of the window-based synchronization protocol described in [NIC93]. In order to support real-time visualization and interactive control of the emulated system, the kernel controls the progression of the simulation time, constraining it by the progression of the processor time.

Objects that interact with the parallel simulation kernel require minimal knowledge about the kernel—mainly how to send and receive messages. Therefore, the design of the emulated system follows the same rules as the design of network controllers that run on a real network platform. The major difference is in how the interaction among controllers is realized. In the emulated system, interaction is performed by the simulation kernel. In a
broadband network environment, for example, the exchange of messages is provided by a signaling system.

The module for real-time visualization and interactive control contains a graphical interface which provides 3-D visual abstractions of the system state. The state of the emulated system is continuously refreshed, which allows a user to follow the system dynamics on the interface. The user is able to control the behavior of the emulated system by changing parameters through the interface.

The emulation support module coordinates the exchange of control and monitoring messages between the graphical interface and the emulated system. It reads the states of the emulated system, and performs filtering and abstraction operations before making the information available for visualization. Control information from the user is mapped to a set of control parameters that are interpreted by the emulated system.

The design of the emulation platform along the four building blocks described above serves the purpose of portability and re-usability. The platform runs on two types of supercomputers—one with a shared memory architecture and the other having a distributed memory architecture. The two implementations differ in emulation support and types of simulation kernel modules. The emulated system module needs only minor changes when porting the software from one computer to the other; the module for real-time visualization and interactive control needs no modification.

3.2.1 The Parallel Simulation Kernel

The simulation kernel controls the execution of simulation objects and routes events from one object to another. Objects that require services from the kernel are defined as a subclass of SimObject. Each SimObject (or any subclass of this object) has a processEvent method with an input parameter of type Event. Attributes of an Event include a time-stamp indicating when the event can be processed, object identifiers of source and destination, and an event-type tag. An object generates an event by invoking the sendEvent
method which has an output parameter of type Event. The simulation kernel receives this event, routes it to the destination, and invokes the recvEvent method of the destination object.

When the system initializes, a local simulation kernel is created on each of the processors available in the simulation program. Since the local simulation kernels also create all the simulation objects running in the system, the local kernels must have handles to the constructors of all simulation objects. In addition, the kernel must know how many objects of each class to create, and a globally unique identifier is assigned by the kernel to each of the objects created.

In the initial version of the simulation kernel, objects can only be created at initialization time and once created, they cannot be deleted or moved. The placement of objects are compiled into the executable and changes to this information requires recompilation. In the later versions, information relating to object naming, creation time and location can be read in dynamically during run-time. This allow objects to be created, deleted and moved while the simulation is running. A distributed protocol is used to ensure that the local directories containing object-related information seen by all local simulation kernels are consistent. Note that the kernel only ensures that the object directories are consistent. The application has to handle the failures associated with objects being deleted or moved.

Figure 3.2 shows the realization of the parallel simulation kernel.
During the execution of the simulation, the simulation kernel controls object execution using a global time $T_G$. It delivers an event to an object for execution only when the time-stamp of the event is smaller or equal to $T_G$. Otherwise, the event is buffered in the event queue.

$T_G$ is computed as the minimum of $T_S$ and $T_W$, which are explained below. $T_S$ is the simulation time as computed by the causality-control protocol, which is based on the window-based synchronization protocol described in [NIC93].

$T_W$, the scaled wall-clock time, is a linear function of the processor time taken from a reference processor, i.e., $T_W = P_{time} \cdot K_{ratio}$, where $P_{time}$ denotes the processor time and $K_{ratio}$ is a control parameter that can be changed by the user.

Performing the simulation in the above described way allows us to synchronize the evolution of the simulation time with a linear function of the processor time. Our system is therefore capable of reproducing and visualizing the dynamics of the real system, with the time scaled by a factor. By changing $K_{ratio}$, we can control the speed at which the simulated system state evolves. For example, we can slow down or “speed up” the simulation. (Speed-
ing up the simulation is obviously constrained by the performance characteristics of the hardware.)

By processing an event only when its time-stamp is smaller than $T_S$ -- i.e., when no causality violation can occur -- we follow the conservative approach to parallel simulation. We eschew the optimistic approach, which requires roll-back and state saving mechanisms, mainly because that approach makes it difficult to build simulation objects independent of the simulation kernel [NIC95]. Our objective is to have a clean separation between the emulated system and the simulation kernel, rather than improving the performance of the simulation kernel. (It is argued that, in general, optimistic simulation allows for better performance of a parallel simulation kernel [FUJ90], although our application falls into a class where conservative simulation exhibits a comparable speedup [NIC93].)

On a function level, the task performed by a simulation kernel can be described by a loop consisting of three steps. In step 1, the set of local simulation kernels runs a synchronization protocol to compute $T_G$. In step 2, each logical process is allowed to process all events with time-stamp not later than $T_G$. In step 3, when all events have been processed, the set of local simulation kernels has to make sure that all events sent previously have been received before they proceed back to step 1. Synchronization in step 1 and 3 are performed using MPI primitives on the SP-2. In step 1, a call to MPI_Allreduce() is used to find the minimum time-stamp of messages received by all logical processes. In step 3, there is one call to MPI_Barrier() and then one call MPI_Allreduce() to check that all messages sent are received. The overhead introduced by these synchronization operations are measured for 4 processes, each running on a different processor, calling MP_Allreduce() or MPI_Barrier() in a tight loop. The result shows that MPI_Allreduce() takes on the average 0.2ms to execute, while MPI_Barrier takes 0.1ms to execute.
Synchronization overhead (4 CPUs)

STEP 1 0.2 ms
STEP 3 0.3 ms

Figure 3.3: Performance of simulation kernel

The performance of a 50 node experiment running on the simulation kernel is shown in Figure 3.4. In this measurement, we ran a traffic control system, which includes traffic generators, connection managers, routers, and admission controllers. There are more than 250 objects in the simulated system. The number of processors used in the simulation is increased from 2 to 48, and the number of events processed per second by the simulator is recorded. The figure shows a reasonable increase in performance when the number of processors increases. As expected, the increase is far from linear.

An interesting question one can ask is how many processors are needed such that the simulation can run in real-time. In other words, how many processors are needed such that the simulator can process events fast enough so that \( K_{\text{ratio}} = 1 \) (\( P_{\text{time}} = T_w \)), and \( T_G = T_W = T_S \). As an approximation, we assume that events are processed at a uniform rate. Thus, the event processing rate required for real-time simulation is taken to be the ratio of
the total number of events processed and the total simulation time. In Figure 3.4, the horizontal line drawn across the figure indicates the average event processing rate required for real-time simulation. The interaction between this line and the speedup curve gives the number of processors needed. For this particular system configuration, about 26 processors are needed.

![Graph showing scalability](image)

Figure 3.4: Scalability of a 50 node experiment running on the simulation kernel

### 3.2.2 Emulation Objects

Emulation objects, i.e., objects in the emulated system, are instances of subclasses of `SimObject`. The purpose of the base class `SimObject` is to hide the simulation functionality from the user by enforcing a clean separation between `SimObject` and the emulation objects.
Figure 3.5: Inheritance relationship between simulation and emulation object classes.

Figure 3.5 and Figure 3.6 follow the notation of [RUM91]. Each object is described by its name (e.g., Simulation Object), followed by attributes (e.g., Simulation Time), and finally the methods associated with the object (e.g. sendEvent and processEvent). As shown in Figure 3.5, SimObject has two groups of methods. Group (A) consists of methods that are relevant to the emulation objects. The interface sendEvent() is a generic function used by all emulation objects to initiate the sending of messages. The interface processEvent() is used for interpreting messages that have been received, so that the appropriate methods in the emulation object are invoked. (This function is similar to the stub code function for remote procedure call (RPC) systems.) This interface is different for each class of emulation objects. The interface getProcessingTime() returns the time needed to process a particular event, drawn from a probability distribution that models the processing time. This
interface is also different for each class of emulation objects. Group (B) consists of methods necessary to support a conservative approach to parallel simulation. These methods are common to all emulation objects. They are visible only to the simulation kernel and are not accessed by emulation objects.

In order to add a new emulation class to the emulated system, the user needs to specify the implementation of two methods, `processEvent()` and `getProcessingTime()`. The rest of the methods are inherited directly from `SimObject`. Therefore, the overhead of using our platform over a general purpose computing platform is that two additional methods need to be provided. Since in order to run an object in a message passing environment, a method equivalent to `processEvent()` is required anyway, the additional interface needed when writing an emulation object which runs on our platform is the method `getProcessingTime()`.

Our design is based on the conservation approach to parallel simulation. This enables a clean separation between simulation and emulation model, because the designer of an emulation object needs minimal knowledge about the underlying simulation model. If an optimistic approach is adopted, the methods in group (B), which implement a conservative simulation protocol, must be replaced by a set of methods that implement an optimistic protocol. (The methods in group (A) remain the same.) In this case, however, the designer of an emulation object has to deal with state-saving that is needed if roll-back occurs. The designer has to specify, for each class of objects, the states to be saved. This is a non-trivial task, which requires knowledge of the simulation model in order to obtain good simulation performance. In this sense, the conservative approach provides an easier-to-use simulation environment, compared to the optimistic approach [Nic95].

A fundamental abstraction in the domain of network control and management is the concept of a `resource`. We model a resource as an object characterized by the attributes capacity and operating point, which indicates the utilization of the capacity. The capacity of a resource defines an upper bound for the operating point. Resources are modeled
as finite capacity queues without buffers. A resource controller regulates access to a resource. It receives requests for access to the resource and decides whether to accept or reject the request, based on the capacity, the current operating point, and the control policy. We model resource controllers as emulation objects with methods that allow for resource negotiation and generic monitoring (see Figure 3.6).

![Diagram](image)

Figure 3.6: Generic monitoring and access of a resource controller.

A monitoring operation on a resource controller object returns the following event counters:

- \( S(t) \) the number of successful requests at time \( t \).
- \( B(t) \) the number of unsuccessful requests at time \( t \).
- \( D(t) \) the number of departures at time \( t \).
A monitoring agent can be constructed by having the agent requests these event counters from the controller periodically. Note that unless the counter overflows, each of these counters always increases with time. The agent implements a moving window scheme by keeping the most recent \( N \) (the size of the window) samples. Let the agent queries the controller every \( T \) seconds. The duration of history stored in the moving window is thus \((N-1) \times T\) seconds.

Statistics relating to a resource controller are derived from these counters. Let the most recent sample be collected at \( t_1 \) and the oldest sample collected at \( t_0 \). The number of active requests at time \( t_1 \) is given by \( S(t_1) - D(t_1) \), and the average blocking between the time interval \( t_1 \) and \( t_0 \) is \( (B(t_1) - B(t_0))/ (t_1 - t_0) \). The product of the window size and sampling period of the samples kept in memory determines thus determines how sensitive the statistics are to short-term variations.

**Emulated Objects communicate via asynchronous message passing.** For example, in one of our network prototypes, messages are exchanged among connection managers and link admission controllers during a connection setup. Logically, a mailbox is associated with each emulation object. If object A wants to send a message to object B, it invokes a send function that inserts the message into B’s mail-box. Conversely, object B receives messages from other objects by invoking a receive function that removes the messages from its mailbox.

Our platform supports asynchronous message passing, because it is a flexible communication paradigm. It can be used to emulate other paradigms, including synchronous message passing, generative communication, and remote procedure call (RPC) [AND91]. Furthermore, the Signalling System No. 7 [ADB90], which is a part of the telecommunications networks’ control systems, also uses message passing as the communication paradigm.
3.2.3 Emulation Support

Figure 3.7 shows a data-flow diagram of the software architecture for the support of real-time visualization and interactive control. Its design follows the monitoring-control paradigm. Monitoring is realized as a continuous activity, whereby a stream of data produced by the network emulator is consumed by the operator interface in the graphics workstation. Interactions among components involved in the monitoring process are asynchronous, via reading and writing shared objects, which allows them to run on different time scales. Control operations, on the other hand, are event-driven and are based on the client-server paradigm.

![Diagram of software architecture for the emulation support module.]

Figure 3.7: Software architecture for the emulation support module.

There are two types of control operations: (1) emulation control operations, which alter the behavior of the emulated system and are executed by the network emulator, and (2) monitoring control operations, which affect the monitoring activity. Emulation control operations include changing the input load on the system and modifying the control policies with which the control mechanisms run. Monitoring control operations tune compo-
ments in both the emulation system and the manager station. They regulate the part of the network state that can be visualized on the manager station, as well as the amount and the granularity of data that is carried over the network, by, for example, adjusting the sampling interval and the size of the monitoring window. Inside the manager station, the information collected from the various network objects can be abstracted, correlated and displayed on the screen in different ways. By varying the quantity and granularity of information collected and sent over the network, our system can be tailored to different computing and network platforms.

3.2.4 Real-Time Visualization and Interaction

The design of the Real-Time Visualization and Interactive Control module focuses on the selection capabilities, high-level control primitives available to the operator, and visual representation of network states. These modules allow an operator to try what-if scenarios, invoke specific network services during run-time, emulate network management operations, and visualize the effect of these operations.

Operations performed by the human operator through the interface are categorized into three groups. The first set relates to defining the input traffic characteristics for each network node. The second set enables a variety of monitoring functions. By selecting objects (switches, links, network regions, etc.) on the network map, together with the desired monitoring option, the operator can visualize, in real-time, various abstractions of the network state, including traffic intensities and network utilization. The third set of interface operations refers to changing management parameters, which allow operators to tune the behavior of mechanisms in the traffic control system.

This remaining part of this section gives an overview of the structure and the composition of the network management system consisting of a graphical user interface which allows interactive control of real-time services.
3.2.4.1 Processes and Tasks

The network emulation and visualization is done by different processes that run on different machines. The emulation and simulation of the network is computed on a parallel computer, while the visualization of the network state is done on an SGI workstation. The main process on the workstation is *Net3D*. Net3D starts the sender and receiver process, builds the user interface including the three-dimensional graphics and handles the user input.

![Diagram](image1)

**Figure 3.8: Generation of the processes in two steps**

Figure 3.8 shows the phases of the initialization process. Net3D forks a child process *kcomm* which deals with the communication tasks. *kcomm* will be divided further into two processes, one for sending and the other for receiving data to and from the network emulator.

Communication between the *kcomm* processes (sender and receiver) and the emulator is performed using TCP/IP sockets. Figure 3.9 gives an overview. The sender/receiver processes and Net3D communicate over shared memory segments. There are three different segments. The statistics data describing the current traffic load, number of calls etc. are written by the receiver process into the *monitoring* shared memory segment. Net3D reads from this shared memory and updates the graphics. The receiver process and Net3D operate
asynchronously using independent control cycles. The result is that the receiver can update the data more (or less) frequently than Net3D reads the data.

When the control parameters are changed by the user, the new values and corresponding control tags are written by Net3D into the control shared memory. The sender process reads the values and the tag from the shared memory segment and send them to the emulator. Depending on the operation, the sender may write the values to a third shared memory segment, the parameter segment. This shared memory segment stored the control parameters currently active in the emulator. Thus, when these control parameters are requested by the user, Net3D can read the values from the shared memory segment and does not have to get the values from the emulator. Some of the control parameters stored are, for example, the arrival and departure rate used by the traffic generators.

![Diagram showing processes and communication means](image)

**Figure 3.9: Processes and communication means**
3.2.4.2 Modules and Classes
The system can be divided into two subsystems which correspond to the communication functionality on the one hand and the visualization functionality on the other. The visualization functionality consists of a part that builds up the menu and handles the user events and a second part that visualizes the network and the related statistics. The visualization subsystem is implemented in C++ resulting in a modular, abstract class system. The modules dealing with sockets and shared memory are written in C and rely on global shared variables, which leads to a system that is not as modular and abstract as the graphics subsystem. However, the advantage resulting from this approach is a high performance which is fundamental for real-time visualization. The module smmgr.c containing the functionality for the shared memory management and is used by both subsystems.

3.2.4.3 Sender and Receiver
The sender and the receiver are both implemented in comm.c. The functions sendfunc() and recvfunc() correspond to the sender and receiver respectively. After initializing the shared memory segments and the connections, the sender and receiver both go into a loop. The sender waits for requests to send to the emulator. The receiver waits for incoming data from the emulator. The exchange of data between the sender and Net3D is controlled by a handshaking mechanism using a control variable placed in the shared memory segment. After a write operation the Net3D process writes 1 to the control variable. The sender periodically tests whether the control variable has changed to 1. If it has, the sender reads the data and sets the control variable to 0. Net3D can only write again, after the control variable has been set to 0.

3.2.4.4 Indexing of Monitoring Shared Memory
The shared memory management for Net3D is done in smmgr.c and in the class StatisticsManager. As mentioned above, there are three different kinds of shared memory segments. The monitoring segment stored monitoring information collected from the emulation plat-
form. For efficiency reason, monitoring is done on demand by selecting one or more objects.

Since the number of selectable objects on the interface can be very large, it may not be possible to define a monitoring shared memory segment for each object in the network. Instead, the monitoring segment is partitioned into a number of blocks. The maximum number of objects that can be monitored at the same time cannot exceed the total number of monitoring blocks, which is 256 in most of the implementations.

Net3D keeps track of the indexes of monitoring blocks that are currently in use. For each new monitoring request, an unused index is associated with it. If there are no more free block, an exception is raised. Since the association between the object(s) being monitored and the index of the shared memory segment the monitoring information will be written to is dynamic, Net3D has to keep track of the association.

The index associated with the monitoring request is sent by the sender process to the emulation platform. It is, however, not interpreted at all on the emulation platform. Instead, it serves as a kind of return address for the monitoring information send by the emulation platform to the receiver process. For example, if the receiver processes receives a monitoring packet with an index N, it simply writes the data into the Nth segment of the monitoring shared memory.

3.2.4.5 Display of Statistics

There are two main style of displaying information. Both of them are shown in Figure 3.10. On the left is the floating sphere where each sphere represents an object in the network. For such display to be visually possible, not more than three attributes per object (or sphere) can be displayed simultaneously, with one attribute per dimension. The movement of the spheres in the 3D space gives a good impression of the dynamics of the attributes under observation. When a number of spheres (or objects) are shown together, the general trend of the network behavior can be observed.
On the right of the figure is the *stacked cylinder*. The stacked cylinder is made up of a number of cylinders, each with a different color (or shade of color), stacked one on top of the other. The number of cylinders is the number of attributes monitored per object. The stacked cylinder is inherently a one-dimensional object. However, because it is one-dimensional, it can be displayed together with a 2-dimensional network map, which helps to related the statistics observed to the geographical location of the object.

Figure 3.10: (a) Floating Sphere  Figure 3.10: (b) Stacked Cylinder

3.2.4.6 User interface and interactive control

The user interface contains beside the mentioned statistics functions management simulation control functions. The emulation control function allows the user to vary the parameters determining call arrival rate and holding time. The simulation control allows the user to change the simulation speed and to monitor the advance of the real clock and simulation time. The network management function encompasses the definition of the adaptivity of the system, the definition of quality of service constraints and the capacity allocation. Other features included are mapping of the relation between the virtual paths and the virtual path groups and changing of the refresh rate for graphics updates.

[CHA97b] contains a detail description of the user interface design, including the object inheritance hierarchy derived from the OpenInventor object classes.
3.3 Realization of the VPN Architecture

We have successfully ported the VPN architecture software to xbind [LAZ96, CHA96d], a CORBA-based environment\(^1\) that controls a heterogeneous network of ATM switches.

The principal aim of the xbind platform is to provide an open programming environment that facilitates the easy creation of services by providing an open and uniform access to networking resources. By porting the emulation code to xbind, we are able to use the same control system software to perform resource control tasks on an actual ATM network.

In the following, we describe our experience in this porting process. First, we describe porting of the software from a MPI environment to a CORBA environment, followed by the hardware configurations used for building a VPG-based VPN (Chapter 2).

3.3.1 Porting to a CORBA-based Platform

The porting process is conceptually simple. The major tasks performed include mapping of message processing functions into RPC calls, object naming and system initialization.

In our design, mapping of message passing code to a CORBA-based environment involves translating the processEvent method of an Emulation Object into the appropriate IDL interfaces. As the prototyping environment allows the isolation of the emulation object design from the simulation environment, this translation process is relatively straightforward. The most time consuming task is the mapping of messages in the MPI domain into appropriate data structures used by the IDL interfaces.

Object names in the simulation domain consists of two integer fields. The first is an integer that identifies the class type, and the second is another integer that uniquely identifies an object instance within its own class hierarchy. Such a naming scheme is not appropri-

---

1. The specific implementation of CORBA we used is orbix (version 1.3), a commercial software available from IONA Technologies.
ate in the CORBA domain, where names are often represented as strings. A similar naming scheme, consisting of two strings, is used in the CORBA domain. The first string corresponds to the server name of the object class and the second string corresponds to the marker name that identifies a specific instantiating of an object class.

In the MPI domain, configuration and initialization of system parameters are easy because all processes spawned after initialization have a copy of the initialized system parameters. This is, however, not true in a CORBA environment where objects are created independently. Therefore, the programmer must very carefully decide what should be system knowledge and what should not be.

In order to minimize changes to the emulation code, two services are needed in the CORBA environment: a timer service and a name service. The timer function in the CORBA domain is provided by a TimerQueue object. Such an object is used only locally and is much more light-weight than the CORBA event services. The name service added is written specifically to locate objects based on our naming scheme and allows mapping of objects to processors in a very simple way.

We measurement the degree of code reuse by counting the number of line of code that is common to both the emulation platform and the target platform. The simulation kernel is about 3000 lines of C++ code and only runs on in the emulation domain. (The newer version, which has more features, has 8,000 lines.) The application code running on the emulation domain, which includes the VP controller and the VPG controller has 20,000 lines of C++ code. 11,000 lines of these code can be re-used with no change on the xbind platform. Approximately 3,400 lines of new code were added. 1,100 lines of these code were wrapper for the original emulation code and the rest are needed for performing monitoring, name service, timer and object initialization. The interface, which is reused, has about 14,000 lines of C++ and C code.
Our experiences has shown that by following the proper design process, moving software from the MPI domain to the CORBA domain is not difficult. The reverse is unfortunately not so easy. As this is not directly related to this thesis, we will not go into the details.

3.3.2 Realization on a Heterogeneous ATM LAN

The ATM network we experimented with have 4 ATM LAN switches, namely:

- FORE ASX-100
- ATML Virata VM 1000
- NEC Model 5
- Scorpio Stinger 1

These switches are connected using the topology shown in Figure 3.11. Based on this topology, we set up a VPG-based VPN network. A possible VP/VPG topology is shown in Figure 3.12.

![Diagram](image)

Figure 3.11: Topology of ATM LAN network
Configuration of the hardware was more difficult than anticipated. Some of the problems encountered are summarized as follow:

- **Partial use of VPI and VCI Name Space**
  
  Even though the ATM standard defines 12-bits for the VPI header and 16-bits for VCI header, switch manufacturers might not use all 28 bits for switching. For example, the NEC switch uses only 12 bits for switching. The user is allowed to partition these 12 bits among the VCI and VPI header. For example, the user can configure the switch to recognize 4 VPI bits and 8 VCI bits or 2 VPIs and 10 VCI bits. In the former case, up to 16 VPs are allowed and the valid VCI range is between 0-255. In the later case, the VCI range increases to 0-1023, but only 4 VPs are allowed.
• **Incompatible VCI Name Space**

The incompatibility of the signaling software among heterogeneous ATM switches is well known. However, the problem goes beyond incompatible software, even the valid VCI name space allowed are different among the switches. In particular, the Stinger switch only allows VCI larger than 768 to be used and the NEC uses only VCI in the range 0 to 255 (when the switch is set to operate in a common operating mode). Therefore, a VC cannot be setup between these switches without another switch in between that allows the union of the these two VCI ranges to be used. In this case, we solved the problem by routing the VCs between Scorpio and NEC over the ATML and Fore switches.

• **No VP switching**

The current software we have for controlling the Scorpio switch does not allow permanent VPs to be setup. As a result, the Scorpio switch can only be used as the originating or terminating point of a VP, and not as a VP switch.

### 3.3.3 Support Needed from the Provider

Porting to a real network makes us aware of additional support required from the public network provider before a VPG-based VPN service can be provisioned.

One important requirement is that during the service provision phase, the customer should be able to influence how the VPG is routed over the physical links. For example, the customer can specify that the VPG network should be configured such that VPGs are not routed over the same physical link. This will help to ensure that the resulting VPN is more robust with respect to failure in the provider’s network.

The second requirement is that policing of customer traffic, which is performed per VPG, is required not only at the gateway between the CPN and the VPN, but is also needed at the transit switches within the provider domain. Since VPG is a logical concept that might not be recognized by the switches, additional work is needed. One possibility is to
assign a group of VPI's to a single VPG group. For example, all the VPs in the same VPG group can have the same most significant 9 bits in the VPI field and only the last 3 bits in the VPI header varies (which will allow 8 VPs per VPG). Policing is performed per VPG by masking the last three bits of the VPI header. VP switching remains the same. Such a scheme could conceivably be implemented on existing ATM switches with only minor modifications. A related problem is to require public network providers to agree on the same masking scheme when the VPN covers more than one public network.

3.4 The High Performance Platform for Experimentation

The platform used for implementing the prototyping environment consists of a supercomputer (either the KSR-1 or SP2) located at the Cornell Theory Center (CTC) in Ithaca, New York, connected to an SGI Indigo2 at Columbia University in New York, New York. These two machines are connected by a permanent virtual circuit connection (PVC) through NYNET, an ATM network that connects several research laboratories in New York State.

![Diagram](image)

Figure 3.13: Hardware configuration of the interactive emulation platform.
The bandwidth B required by the connection is given by \( B = LM \times NO \times R \), where \( LM \) is the size of the monitoring data unit, \( NO \) is the number of objects being monitored, and \( R \) is the refresh frequency. When \( NO=500 \) objects, \( LM = 512 \) bytes, \( R=1/\text{sec} \), \( B \) is approximately 2 Mbit/sec. When the communication between the two machines is performed over Internet, both the number of objects monitored and the refresh frequency must be reduced to accommodate the lower bandwidth connection. In this case, our experience shows that, even with a reduced monitoring load, the packets carried over the Internet connection can experience delay variations of up to a few seconds. The ATM connection currently in use has been shown to be very useful for our purposes, providing higher throughput with lower delay jitter.

The simulation kernel was first implemented on the KSR-1 and then ported to the IBM SP2. These implementations differ, in particular with respect to realizing message passing and referencing of objects. For the point of implementing a simulation kernel, message passing is important because the programming model of simulation is one of objects (or logical processes) exchanging messages. Therefore, while message passing is the "natural" programming paradigm on the SP2, we have to "imitate" message passing on the KSR-1 using shared memory.

In the remainder of this chapter, we will describe some of the knowledge we have gained through using these supercomputers, in particular the performance of simulating message passing on the KSR-1 and the porting process.

### 3.4.1 Building an Emulation Platform on the KSR-1

The KSR-1 is a shared memory parallel machine with hardware support for maintaining cache consistency. Each node has a 40-MFlops processor and 32 Mbytes of local cache memory. Processors are organized in ring hierarchy. Ring 0 has 32 processors and ring 1 has 34 ring 0s (Figure 3.14). The memory access time follows the same hierarchy. Memory access takes less than 1ms within the local processor, 8.75ms within the same ring 0, and
30ms within the same ring 1. The operating system is MACH, and the OSF-1 pthread package is provided to exploit the shared memory architecture of the machine.

![Diagram](image)

Figure 3.14: Hardware configuration of KSR-1

### 3.4.2 Benchmark Measurement on the KSR-1

As the KSR-1 was our initial experimental platform, we spent substantial time on the study of emulation performance of the KSR-1. The primary goal of this study was to estimate the performance of emulating large networks on a KSR, to identify major performance bottlenecks in an emulation system, and to suggest a system structure for emulation that reduces the impact of these bottlenecks. We emulated network mechanisms as (operating system)
threads and implemented a message passing system to support asynchronous communication among mechanisms. Emulating large broadband networks thus involves executing a large number (hundreds) of threads of control, which cooperate in an asynchronous manner, using a message passing scheme for inter-thread communication. Operating system threads on the KSR-1 is available to the programmer as *pthread*, a POSIX threads standard (P1003.4a).

We did two kinds of performance experiments. First, we measured the performance of implementations of the bounded buffer paradigm [AND87]. In the bounded buffer problem (also known as the producer-consumer problem), two processes shared a common, fixed (and bounded) size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. Potential contention arises when both the consumer and producer want to access the buffer at the same time, and when the buffer is either full or empty. We chose to study this problem because of the way it captures the programming model of the message passing scheme used. Secondly, we studied the performance of a traffic control system that runs on top of our message passing scheme.

In our implementation of the bounded buffer paradigm, we found that the numbers of producers and consumers, as well as the ratio of consumers/producers, are the parameters that show the most significant impact on the performance. Having one producer and one consumer per buffer gives the best performance. Increasing the number of producer/consumer pairs results in a sharp drop in performance, roughly with $O(1/n)$, with system size $n$. Also, the farther the ratio producer/consumer is from 1, the worse the performance gets.

We also found that synchronizing multiple pthreads, using mutual exclusion locks (or mutex locks) and condition variables [BOY93], gets prohibitively expensive, when the number of threads waiting on the same condition variable increases. We therefore conclude
that the number of threads that access the same critical region should be kept to a minimum, in order to prevent a drastic decline in system performance.

In general, the performance depends on the pattern of interaction among nodes of the system. Take the example of two systems with extremely different communication patterns: a point-to-point communication pattern and a point-to-multi point communication pattern. If we express the performance of these systems as the number of messages per second to be exchanged between nodes, we get the following peak performance on a 64 processor set: 980,000 messages/sec for the former case and approximately 160,000 messages/sec for the latter. As mentioned above, these systems exhibit extremely different communication patterns. For a typical subsystem of a broadband network, we expect the communication pattern to be a mixture—broadcasts among subsets of nodes as well as one-to-one interaction—and, therefore, we estimate its peak performance to be in between the above given figures.

3.4.3 Experiences with Emulation of a Traffic Control System

In this section, we describe the emulation of a simple traffic control system (TCS). The TCS is modeled as a large number of cooperating network mechanisms, running in parallel, and interacting asynchronously by exchanging messages. In the emulation system, network mechanisms are implemented as C++ objects, and objects are mapped onto pthreads. We study performance bottlenecks in the emulation system, which occur when the system size increases. We believe that these bottlenecks are typical for systems that have similar interaction pattern.

3.4.3.1 Emulation of Traffic Control System

In our emulation of the TCS, there are three types of mechanisms, namely, call traffic generator, connection manager and, link controller. The call traffic generator provides traffic load to the system. It generates two types of requests: one type to set up a call and the other to delete a call. It is implemented as a C++ object and is mapped onto a pthread. The con-
nection manager handles call setup and delete requests. It is implemented as an object and is mapped onto two pthreads, the call-service pthread and the call-ack pthread. The link controller regulates access to resources of a link. It is implemented as an object and is mapped onto a pthread.

Figure 3.15 shows the communication pattern in the TCS. One call traffic generator communicates with exactly one connection manager. A connection manager can communicate with any link controller and vice versa.

![Diagram of communication pattern]

**Figure 3.15:** Communication pattern among network mechanisms

In order to illustrate how the TCS works, we give a step-by-step explanation of how a call setup operation is done (see Figure 3.16). Deleting a call is performed in a similar way.

- **Step 1** - The call traffic generator generates a request and sends it to the connection manager. The source address of the request is the address of the connection manager that processes this request and the destination address is randomly selected.

- **Step 2** - This message is read by the call-service pthread of the connection manager. For simplicity, a random route is generated by this pthread. This route determines the list of link controllers participating in setting up this call.
- **Step 3** - The call-service pthread sends a request to each link controller in the list.

- **Step 4** - Link controllers receive these requests and decide whether they can accept or reject them.

- **Step 5** - Link controllers send replies to the connection manager.

- **Step 6** - These replies are received by the call-ack pthread of the connection manager. When replies from all link controllers are received, the call-ack thread decides whether the call can be accepted or rejected. If any of the replies are negative, the call is rejected. Otherwise, it is accepted.

- **Step 7** - A reply to the original request is sent to the call traffic generator.

![Diagram of call setup](image)

Figure 3.16: Steps of a call setup

In this implementation, many requests are processed simultaneously in the system. The call traffic generator sends new requests to the connection manager before earlier requests are answered. Also, many call traffic generators run concurrently. The connection manager keeps track of the states of many on-going requests in a local data structure. For each message it receives from the call traffic generator or the link controllers, it updates the state of the appropriate call in the data structure.
We measure the performance of the emulation system in terms of the number of call requests (setup and delete) completed per second. This is a measure of the combined performance of computational (traffic generation, route computation, updating of local data structures, etc.) and communication (between threads) tasks. In addition, note that the emulation system is very communication intensive. Therefore, the performance measured here is more relevant for communication intensive applications like the signaling system of a broadband network than for computational intensive applications like numerical analysis.

A number of parameters are kept constant in the following performance measurements. All experiments run on the 64 processor set. The number of hops of the routes is uniformly distributed between 1 and 5. In all experiments, a network node, which is made up of one traffic generator, one connection manager and one link controller, is always mapped onto the same processor.

We identified three performance bottlenecks in the emulation system as we increase the system size. These are caused by performance degradation due to serialization of KSR system calls, synchronization among mechanisms, and extensive context switching as a result of the fine grain application we ran. In the following, we discuss these performance bottlenecks, propose a solution for each of them and verify its effect through measurements.

3.4.3.2 Serialization of KSR System Calls

Some system calls in the KSR operating systems are serialized: “malloc()” and “free()” are two of them. For systems that perform many memory allocation calls, mechanisms spend an increasing amount of time waiting to acquire the lock that protects the global data structure, as the system size (and the number of mechanisms) gets large.

We verified this hypothesis through two measurements. We ran our traffic control system with two different libraries. The first library uses “malloc()” and “free()” with no
consideration for their cost due to serialization. The second library uses an application-level memory management scheme that attempts to minimize these calls. In order to observe the effect of serialization, only mechanisms within the same network node interact. This is done by modifying the routing policy such that it generates only the address of the link controller in the same node. Since interacting mechanisms run on the same processor, the impact of remote memory access, contention, and thread scheduling is minimized.

Figure 3.17 shows the difference in performance between using and not using application-level memory management. When application-level memory management is not used, the throughput is low for a small system size and remains at that level as the system size increases. With application-level memory management, the throughput is significantly higher for systems of up to 128 network nodes. The application-level memory management is used in all measurements discussed later.

![Graph showing impact of serialization of memory management system calls](image)

**Figure 3.17: Impact of serialization of memory management system calls**

### 3.4.3.3 Threads Synchronization

Figure 3.18 shows two ways of realizing object interaction through exchanging mes-
sages. In Figure 3.18(a), a mechanism writes directly to the input message buffer of another mechanism. In the Figure 3.18(b), a communication kernel is used for the sole purpose of routing messages (similar to the simulation kernel in Section 3.2.1). Two buffers are associated with a mechanism $M_i$: an output buffer $O_i$, which holds messages sent by $M_i$, and an input buffer $I_i$, which contains messages to be received by $M_i$. The communication kernel reads the output buffers of the mechanisms and delivers the messages to the appropriate input buffers. It is implemented as threads $C_1$,..., $C_n$, which perform identical tasks by serving the output buffers in a round-robin fashion.

The communication kernel is implemented in such a way that a thread $C_i$ avoids waiting for accessing a mutex of a buffer. It tests a buffer for full or empty before acquiring the lock to it. Also, it invokes a `mutex_try_lock` call (instead of `mutex_lock`), which returns immediately, regardless of whether the lock is acquired or not. If a message cannot be written immediately to the appropriate buffer, it is saved in an internal buffer $B_i$. Each kernel thread periodically clears its internal buffer in a new attempt to deliver such messages.

The processor set that runs the emulation is partitioned into a subset that runs the mechanisms $M_i$ and a subset that runs the communication threads $C_k$. Each CPU allocated for communication runs exactly one communication thread.
As observed in the previous section, thread synchronization overhead increases with the number of threads accessing a buffer. The use of the communication kernel provides one way of reducing expensive thread synchronization by reducing the number of threads accessing any of the input or output buffer. In addition, processor resource can be explicitly partitioned into two sets, one for computation and one for communication.

Figure 3.19 shows the performance difference between using and not using a communication kernel. In this experiment, 4 processors are dedicated for communication. The result is somewhat disappointing and clearly illustrates one of the biggest pit-fall in parallel processing. If the system is not configured correctly, using more resources can cause the performance to collapse! In this case, the collapse is much more dramatic than expected. The sharp drop in performance can be attributed to two factors: contention among increasing number of mechanisms and, non-uniform memory access time which increases sharply when the nodes in difference rings communicate.

For the system that does not use a communication kernel, the performance peaks at 16 nodes and drops very rapidly when the system size exceeds 32 nodes. On the other hand,
the system that uses a communication kernel (using 4 processors) provides a lower throughput but exhibits stable performance up to 168 nodes. In addition, if the system size is larger than 64 nodes, the throughput of the system that runs on a communication kernel is always better than the throughput of the system that does not.

![Graph showing impact of communication kernel on throughput](image)

**Figure 3.19: Impact of using communication kernel for emulation**

In the next two experiments, we investigate how performance of the TCS is influenced by the amount of resources allocated to the communication kernel. In the first experiment, we fix the size of the system and vary the number of communication processors. In the second experiment, we fix the number of communication processors and vary the system size.

In the first experiment, the size of the system is 64 nodes and the number of processors allocated to communication is increased from 1 to 32. Figure 3.20 shows that the throughput of the system increases almost linearly with the system size. From this measurement, we see that the characteristics of the emulation system is such that more resources (up to 32 processors) can be allocated to communication (and less to computation) to achieve better overall system performance.
Figure 3.20: Impact of increasing communication resources (constant system size)

Figure 3.21 shows the result for the second experiment. We run the TCS with three different numbers of communication processors: 4, 8, and 16. The throughput of the system improves significantly when the number of communication processors increases from 4 to 8. However, when increasing the number of communication processors from 8 to 16, the performance gain is significant only up to 32 nodes. We attribute this drop in performance gain to two factors. First, as the number of communication threads increases, the frequency of them trying to access locks that have already been acquired increases. Hence, on the average, a communication thread has to perform more tries to acquire a lock before getting access to one. Second, more resources (8 CPU) are dedicated to communication and less to computation. Therefore, for large systems, the gain in communication performance is partially offset by the loss in computational performance.
3.4.3.4 Context Switching Overhead

The overhead of context switching can be significant, if the amount of work done by a thread between context switches is small. In our application, this happens when a consumer, waiting on an empty buffer, wakes up and finds only one message (or very few messages) in the buffer. As the time it takes a thread to service a message is short in our application (of the same order of magnitude as a context switch), the thread finishes its work in a relatively short time, and waits on the empty buffer, causing another context switch.

We influence thread scheduling, by using conditional variables in a novel way, so that the amount of work done by a thread between context switches is increased. When a consumer waits on a conditional variable, instead of waking it up when there is one message in the buffer, it is woken up only if there are x (x>1) messages in the buffer (in implementation terms, we perform a “cond_broadcast()” call only if the number of messages in the buffer is equal to x). The same strategy applies to a producer. As a result, when a consumer or a producer is put in the run queue, there are at least x messages in the buffer.
Figure 3.22 shows the performance gain by influencing thread scheduling in such a way. The solid line graph shows throughput for $x = 10$, and the dotted line graph shows throughput for $x = 1$. The gain in performance is significant for system size of 64 nodes or more. In this experiment, the system uses application-level memory management and runs on a 4 processor communication kernel.

![Graph showing impact of influencing thread scheduling](image)

Figure 3.22: Impact of influencing thread scheduling

### 3.5 Porting the Emulation Platform from KSR-1 to SP-2

The IBM SP2 has a distributed memory architecture. Each node consists of a 266-MFlops POWER2 processor, which runs its own copy of the AIX operating system, and has between 64 and 2048 Mbytes of memory. The parallel programming environment we used is based on the Message Passing Interface (MPI) [MPI94] package. As opposed to the KSR-1, communication among processes on different processors is possible only through message passing. The benchmark for interprocess communication is therefore not dominated by memory access time and synchronization, but by the time delay to send messages from
one processor to another. For the MPI implementation used, sending a 200-byte message from one node to another takes about 50ms. Since communication on the SP-2 is off-loaded to a communication processor, communication and computation can often overlap. Note that we are interested in high message delivery rate and small delay for small messages. More details will be presented on this topic in Chapter 4.

The major differences between a shared memory machine and a distributed memory machine are summarized below.

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Shared Address Space.</th>
<th>Address space is not shared.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inter-Process Communica-tion</td>
<td>Shared Memory Access (Mutual Exclusion).</td>
<td>Message Passing (e.g. MPI).</td>
</tr>
</tbody>
</table>

Table 3.1: Shared Memory vs. Distributed Memory

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Shared Memory Machine (e.g. KSR1)</th>
<th>Distributed Memory Machine (e.g. SP2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object References and Location</td>
<td>Object references are pointers. Shared address space makes locating another object trivial.</td>
<td>Mapping from name to location (which processor) is needed.</td>
</tr>
</tbody>
</table>

Table 3.2: Task Involved in Porting the Emulation Platform

Since the programming model for the simulation kernel remains the same, much of the code can be reused. In fact, the MPI model is more natural for implementing the simulation model. The most significant change is in the ways objects are referenced (and named).
Object references are different between the two platforms because on the KSR-1, the programmer assumes a global address space. All objects can thus be referenced through object pointers. However, on the SP-2, all objects have to be explicitly given globally unique names. These names are written into the messages exchanged so that the simulator kernels have sufficient information to route them to the correct destinations. Therefore, a naming scheme has to be devised and all objects given unique names when the simulation system initializes. Currently, the unique name is an integer constructed out of the object class type and the integer index of an object within its class type. It is assumed that there are never more than 1024 instances of an objects per class.

3.6 Conclusion

In this chapter, we outlined a methodology for developing network control systems which allows for an evaluation of the dynamic behavior and overall system performance at an early stage of the development process. The prototyping approach has shown to be beneficial for our work. The capability of interactively changing network load patterns, together with the dynamic visualization of network state information, enables us to execute a wide range of what-if scenarios, which contribute to an understanding of the complex interactions in network architectures. Modifying a concept in our emulation environment is often possible in a short amount of time, which allows for the rapid exploration of the impact of the modified design on the functional and dynamic behavior of the system.

The issue of scalability manifest itself not only in the design of the control architecture, but also in the high performance platform used. Section 3.4 describes a number of performance bottleneck associated with the operating system of the parallel machine which does not scale with the number of processes and processors used in the simulation. In general, it is observed that if the emulated system scales well or has a high degree of parallelism in the emulation environment, the system will scale well in the target platform.
Developing a prototype of a network architecture is an important step towards building a network control system. Two control architectures have been developed using the approach outlined, a traffic control system for multimedia network [CHA95b] and the control system of a virtual private network [CHA97b].

Finally, many aspects of large networks, such as service efficiency, stability, survivability, and manageability, which include different classes of controllers interacting with one another, are far from being understood today. These issues need to be investigated in order to realize large-scale multi-service networks, and our platform can serve as a research tool to address some of them.
Chapter 4  Designing a High Performance ATM Connection Management System

4.1 Introduction

One of the most fundamental objectives in networking is the provisioning of connectivity, i.e., the establishment of a communication path between two end-points. In the case of an ATM network, which is connection-oriented, setting up a communication channel requires a connection management system that coordinates operations among a set of distributed software modules. The connection management system has to meet two requirements. First, it must be high performance and scalable in order to handle the anticipated volume of users. Second, because of the need to be able to rapidly create and deploy new services, it must be very flexible.

Unlike existing standards for connectivity services, which are essentially extensions of the User/Network Interface (UNI) and Network/Node Interface (NNI) standards designed for the plain old telephone service (POTS) networks, modern approaches to provisioning of network services (e.g., xbind [CHA96d], TINA [TIN95] and DCAN [KOB97]) exploit advances in distributed system technologies to provide more flexibility for service creation and deployment. In these approaches, signaling entities run on a general purpose distributed computing platform, and interactions are expressed in terms of high-level operations over an infrastructure that provides an open and uniform access to abstractions that model the local states of networking resources.
Given such a platform, the signaling infrastructure has to be carefully engineered so that the performance of the connection management system is capable of handling the large volume of expected user requests. In this chapter, we show how such a high performance connection management system, based on the xbind framework, can be designed and implemented.

Performance of distributed computing platforms has been extensively studied in the literature. A number of studies have been specifically performed for the underlying transport system. In [SCH96b], the performance of two popular Common Object Request Broker Architecture (CORBA) [OMG93] implementations for bulk data transfer have been studied. However, for signaling purposes, bulk data transfers are not an appropriate benchmark because signaling messages tend to be short, and the performance objective caters around providing low latency for short exchange of messages, rather than high throughput for large data sets. This observation is taken into account in [BLA96], where changes to the network processing software are proposed. These changes allow the processing of many short messages to be performed much more efficiently. On a higher level, in [OLI97], a large number of distributed platforms (implemented using C, C++, CORBA, Java, etc.) are benchmarked for exchanges of short messages. In [TAM97], the performance of various connection management algorithms are studied. Finally, [VEE95] describes a system-level design for improving the performance of a broadband signaling system.

Our work differs from most of the other works mentioned above in that we propose design enhancements on the system design level, instead of the protocol level. (Improvement in the protocol layer will independently help boosting the performance of our system). The closest work to ours is that of [VEE95]. Compared to [VEE95], we believe that we have gone much further in terms of exploiting the network as a programmable entity and have applied well-known techniques like caching, pre-fetching and aggregation aggressively to our scheme in a domain-specific way with substantial performance gains.
The main contribution of this chapter is in demonstrating how a high performance ATM connection management system can be built on a general purpose distributed platform such as CORBA. The behavior of the throughput-delay characteristics of the connection management system designed is studied with respect to two control parameters. For various load conditions, appropriate control parameter values can be identified such that the system can be dimensioned accordingly for the desired trade-off among resource utilization, call setup latency and call throughput. The system is implemented in C++ and runs on general purpose UNIX workstations. Its call throughput performance is comparable to the switching performance in the backbone networks for heavy traffic loads ($\sim 10^6$ calls/hr) and can achieve low call setup latency ($\sim$10ms) for light traffic loads. The results show that an open distributed platforms not only provide greater flexibility for service creation, but also enables the design of high performance connection management systems.

The chapter is organized as follows. Section 4.2 describes the connection management framework and presents a set of reference measurements. Section 4.3 describes the design of the xbind connection manager. Finally, Section 4.4 presents the performance results, including the behavior of the throughput-delay characteristics of the connection management system with respect to two control parameters.

### 4.2 Description of the Connection Management Framework

In this section, we present the overall connection management framework and some reference measurement results. There are three parts to the framework description. The architectural model behind xbind is explained in Section 4.2.1. The object interaction model is described in Section 4.2.2, and the connection setup model, including a description of the object interfaces, is described in Section 4.2.3. Section 4.2.4 contains a description of the experimental setup which is used in all of the experiments presented. A set of reference measurements is given in Section 4.2.5.
4.2.1 The Extended Reference Model

xbind is an implementation of the broadband kernel. The conceptual model behind the broadband kernel is the Extended Reference Model (XRM) [LAZ94]. The XRM models the communications architecture of networking and multimedia computing platforms. It consists of 3 components called the Broadband Network, the Multimedia Network and the Services and Applications Network (see Figure 4.1).

The broadband network is defined as the physical network and consists of switching and communication equipment and multimedia end-devices. Upon this physical infrastructure, resides the multimedia network whose primary function is to provide the middleware support needed for realizing services with end-to-end QOS guarantees over the physical network. This is achieved by building upon a set of QOS abstractions derived from the broadband network. This set of QOS abstractions jointly define the resource management and control space. The process of service creation calls for resource reservation and distributed state manipulation algorithms. From this perspective, the multimedia network provides a programming model that allows service behavior to be specified and executed. Service abstractions represent the states of a service created using algorithms native to the multimedia network. These abstractions are used by the services and applications network for managing and creating new services through dynamic composition and binding.
The RGB decomposition represents detailed viewpoints of the broadband network, the multimedia network and the services and applications network, respectively. The interface between the R- and G-models is a set of QOS abstractions typically structured as graphs that quantitatively represent various resources in the physical network. The G-model uses these graphs for creating service abstractions that are provided to the B-model for building more complex services. Thus, the interface between the R- and G-models and the interface between the G- and B-models provide abstractions that are similar in structure but differ in usage.

4.2.2 Object Interaction Model

The connection management systems resides on the G-model of the XRM. In this model, a set of connection managers interacts with a set of objects called the Binding Interface Base (BIB) [ADA96] in order to perform tasks like route lookup, name and QOS translation, resource reservation and switching table update. Specifically, the following tasks are performed by the connection manager:
• **Mapping of user-level QOS to network-level QOS.** QOS abstractions for network resources may be defined for each traffic class using specific cell loss and cell delay requirements. Service abstractions for customer premises equipment such as PCs and workstations are specified in terms of, e.g., frame rate and frame loss. In our framework, the QOSmapper translates the QOS specifications per frames to QOS specifications per ATM cells and vice versa.

• **Route Selection.** The path connecting two endpoints in the network is provided by a RouteObject. Routes are updated by independently operating router objects.

• **Resource Reservation.** Resource reservation tasks performed by a connection manager are divided into two groups: reserving system resources (buffer, bandwidth, CPU cycles, etc.), and reserving and setting of identifiers in the switch fabric for cell transport. Each switch has a server for changing the contents of the switching table. Instructions for making such changes are issued from a connection manager of the communication system, for call/connection control. The reservation of system resources should be based on abstractions that are independent of the details of the system hardware and provide QOS guarantees. For manipulating the switch identifiers, a set of primitives is used. See [LAZ96] for more details. The object regulating access to the switch resource is called the SwitchServer.

Figure 4.2 shows the object interaction model. Note that we draw a distinction between the control (or signaling) plane and the transport plane. Objects in the signaling plane are CORBA objects, and they communicate over an IP network. Due to their scalability and resilience to partial failures, IP based networks are ideal for the transport of short control messages with little or no call holding times. On the other hand, the transport of multimedia flows with long holding times is ideally supported by an ATM network. Due to the inher-
ently stateful nature, an ATM network can provide a much higher degree of predictability and is highly suited for transport of streams with QOS requirements.

![Diagram](image)

**Figure 4.2: Object Interaction Model.**

There are various ways of implementing the interaction between the objects, in particular, the interaction between the connection manager and the SwitchServer. Using the terminology (in pp. 343 of [GOS91]) for naming the various styles of name resolution, there are three interaction patterns, namely: recursive, iterative and transitive (Figure 4.3). The recursive pattern is essentially the hop-by-hop approach used in many of today’s signaling systems. A connection manager logically resides on each of the switches. It accepts, processes and then forwards a request in both forward and backward directions. The transitive approach is similar to the recursive approach except that when the last (or destination) switch is reached, the request is not propagated back to the path it came from but instead goes back directly to the first (or source) switch. The transitive approach has the advantage that it reduces the communication load but has the disadvantage that all the connection managers must be able to communicate with each other and the message needs to contain the identity of the source. In both of these approaches, the interaction is among connection
managers. A connection manager only interacts with its local SwitchServer and not the remote ones.

Figure 4.3: Interaction between ConnectionManager and SwitchServer.

In the iterative approach, a connection manager communicates iteratively with remote objects. Two variations are possible. In the first case (iterative among CMs), interaction is performed only among connection managers. Thus, when a connection manager receives a request, it communicates locally with its SwitchServer and remotely with other connection managers. This is the approach taken in [VEE95]. In the second case (iterative
among CMs and SSs), a connection manager communicates directly with all the Switch-Servers, both remotely and locally. This is the approach used in our design.

These four different approaches have both advantages and disadvantages. The recursive approach performs the most work per request but requires the smallest amount of connectivity. The iterative case has lower overhead and has the advantage that iterative calls can be performed in parallel. The transitive case has the smallest overhead per request but the operations cannot be parallelized. Thus, there may be cases where the transitive approach is better than the iterative case or vice versa, depending on the system configuration and load conditions. [TAM97] contains a performance study between the transitive and iterative approaches.

4.2.3 Model of Connection Setup

In this section, we limit the discussion to point-to-point connection setup on an ATM-based network and the underlying functions performed. Examples of point-to-point services are illustrated in Figure 4.4. As far as possible, we try not to prescribe a specific interaction model (e.g., transitive or iterative) but instead design the model so that a large class of connection management schemes can be implemented. It is envisioned that multiple connection schemes run simultaneously in the network.
A point-to-point connection is described using a pair of end-points. An end-point can reside in any devices (e.g., workstations, switches, etc.) and can represent multiple transport protocols. End-point is described by the `EndPoint` interface and code fragments written using IDL are given below:

```
struct EndPoint {
    EPFormat epformat;
    EndPointId epid;
    Direction dir;
};
union EndPointId switch (EPFormat) {
    case IP_epf: IPEndPointId ip;
    case ATM_epf: ATMEndPointId atm;
    case xbindATM_epf: xbindATMEndPoint xbind;
}
```
By allowing end-points of different EPFormat (end-point format) to be connected, we allow applications running different transport protocol to interact. The definitions for three common end-point formats are given below:

```c
struct IPEndPointId {
    char ipadd[16]; // xxx.xxx.xxx.xxx null-terminated
    short port;
};
struct ATMEndPointId {
    NSAP atmadd;
    short port;
    short VPI;
    long VCI;
};
struct xbindATMEndPoint {
    char ipadd[16]; // xxx.xxx.xxx.xxx null-terminated
    short port;
    short VPI;
    long VCI;
};
```

Notice that the format xbindATMEndPoint uses the IP addressing for naming and ATM names for connectivity. This is due to the fact that we use an IP network for signaling and an ATM network for transport.

Given a pair of end-points, the task performed by the connection manager in order to set up a point-to-point connection consists of the following steps:

1. Reservation of resources (processing, identifiers, etc.).
2. Reservation of switching table entries.

These steps directly use the APIs provided by the VirtualSwitch and VirtualLink interfaces [ADA96] and constitute the simplest building block of a connection manager for
an ATM-based network. Due to the nature of ATM switching, the output VCI/VPI pair of
the down-stream switch must match the corresponding input VCI/VPI pair of the immedia-
ate up-stream switch. As a result, steps (1) and (2) are performed sequentially.

The xbind connection manager performs two additional functions:

3. Mapping of user-level QOS which is specified per frames to network-level QOS which
   is specified per ATM and vice versa.

4. Route selection.

QOS mapping is performed by a QOSMapper object which provides the interface:

```
BIBStatus QOSmap(in QOSSpecification from,
               inout QOSSpecification to)
      raises (Reject);
```

The QOS specification is usually given for the user level, and the corresponding net-
work QOS is obtained from the QOSmapper. The reverse is also possible. The route used
is obtained by a call to a Route object with the IDL interface defined as:

```
BIBStatus getRoute(in NameString source,
              in NameString dest,
              inout RoutePath path,
              in short route_option)
      raises (Reject);
```

These two steps are usually performed before steps (1) and (2), and the usual order
of execution is (3)->(4)->(1)->(2).
The IDL interfaces for adding a connection are given as follows:

```idl
/* Minimum information is specified */
BIBStatus addConnection(
    in QOSSpecification qosSpec,
    inout EndPoint host_A,
    inout EndPoint host_B,
    out RoutePath path) raises (Reject);

/* Both network and user QOS are specified and there is
no need to access the QOSmapper */
BIBStatus addConnection_wQOS(
    in QOSSpecification qosSpecUser,
    in QOSSpecification qosSpecNet,
    inout EndPoint host_A,
    inout EndPoint host_B,
    out RoutePath path) raises (Reject);

/* Route is specified and there is no need to contact
route Object */
BIBStatus addConnection_wRoute(
    in QOSSpecification qosSpec,
    in RoutePath path,
    inout EndPoint host_A,
    inout EndPoint host_B) raises (Reject);

/* Both QOS and Route are specified */
BIBStatus addConnection_wAll(
    in QOSSpecification qosSpecUser,
    in QOSSpecification qosSpecNet,
    in RoutePath path,
    inout EndPoint host_A,
    inout EndPoint host_B) raises (Reject);
```
As an example, the sequence of calls used to set up a connection and the subsequent re-routing of the existing connection from A to D (shown as a solid line in Figure 4.5) to another route from A to C (shown as a dotted line in Figure 4.5) is given below:

\[
\text{addConnection(qos, ep\_A, ep\_D, path);} \\
\text{addConnection(qos, ep\_B, ep\_C, path);} \\
\text{removeConnection(ep\_B, ep\_D);} \\
\]

The end-points ep\_B, ep\_C and ep\_D are located at the input ports and their direction are set to "incoming"; only the direction of ep\_A is set to outgoing.

![Diagram showing re-routing from connection(A,D) to connection(A,C).](image)

Figure 4.5: Re-routing from connection(A,D) to connection(A,C).

Based on these sets of interfaces, extension to the behavior of the connection manager is possible. For example, support for multiple route options is possible by writing the code that implements the logic for re-routing around the interface:

```plaintext
BIBStatus addConnection_wRoute(
    in QOSSpecification qosSpec, 
    in RoutePath path, 
    inout Endpoint host\_A, 
    inout Endpoint host\_B) raises (Reject); 
```
4.2.4 Experimental Configuration

Figure 4.6 shows the experimental setup for all the measurements. The network consists of two workstations, both HP9000/700 series with 120Mbyte of RAM, connected to an ATML Virata switch. A SwitchServer object runs on each of them. A third HP workstation serves as the controller for the ATML switch; the SwitchServer controlling the switch runs on top of it. This workstation communicates with the switch using the qGSMP protocol [QGS97]. All workstations communicate through an 10 Mb/s ethernet LAN. The code is written in C++, and the CORBA implementation used is Orbix, from Iona Technologies. The ConnectionManager, RouteObject and QOSmapper reside on the same machine, an Ultra2. The client application resides on a SUN Sparc20.

The order of executions during call setup requests is as described in the previous section. The client initiates a request to the connection manager (1), followed by QOS mapping (2), route selection (3), and finally resource reservation (4). At the end of the execution, an end-to-end ATM connection is set up from the source workstation to the destination workstation through the ATM switch.
4.2.5 Reference Measurements

Before proceeding with the design of the xbind connection manager, a set of reference measurements are performed. This set of measurements helps us to achieve a better understanding of the behavior of the underlying distributed platform and serves as a set of reference performance figures to be compared with later. In all measurements, the latency of the call establishment is taken to be the period it takes to complete a call of addConnection() by the client program to the ConnectionManager. 1,000 of such calls are performed. The call setup time of a sequential implementation where the connection manager iteratively calls the three SwitchServer (twice) is measured by sending only one add or delete request to the connection manager at one time (in order to avoid different calls interfering with one another). The minimum delay measured is 20.0ms, the average is 23.4ms...
and the maximum is 63.3ms. The 5% and 95% quantile are 20.5ms and 31.0ms, respectively.

Published connection establishment latency for point-to-point call establishment for 1 hop using UNI signalling [BAT96] has the following performance: minimum latency is 48.99 ms, average latency 53.20ms and maximum latency is 67.60ms.

The result shows that a straightforward implementation of our connection setup model on an Orbix platform provides reasonably good performance. The downside in this environment is that, since the operating system is not real-time, there exists a small probability that during a measurement interval, the client process is swapped out while a connection is in progress causing significant degradation in the worst case performance. This is particularly true for the threaded implementation, as we will see later.

The throughput-delay characteristics are also investigated. Calls generated using a Poisson model are sent to the connection manager. Successfully call setups are released with a very small holding time of 0.5 seconds. A small holding time is used so that the total number of active VCs remains small even for a very high traffic load. The throughput is measured in terms of call setup. However, since both call setup and release are performed, the number of call operations (add and delete) is twice the throughput measured.

The results are shown in Figure 4.7. (The measurement period is 1,000 call setups per client). The number of clients communicating with the connection manager is increased from one to three. The throughput measured is taken to be the total throughput of all connection managers, and the delay is calculated as the average. Results show that due to the sequential nature of the connection manager, when the number of clients increases, additional clients end up waiting for other requests to be processed, causing delay to increase dramatically for higher throughput.
Profiling the behavior of the sequential connection manager shows that 12% of the program time is spent executing user code and 11% on executing system code. Most of the time is spent waiting or idling (30% on system wait and 43% idle). It is therefore clear that the use of threads has the potential of improving the performance. By executing each connection setup using a single thread, we can overlap communication and computation of dif-
ferent connection setup requests. While one connection is waiting for a reply, another request can be processed.

Changing a sequential implementation to a multi-threaded implementation requires very little change in terms of the interface definition and interaction with the ORB. This can very easily be done using the ThreadFilter object available in Orbix. The most important changes come from reorganizing the code such that there are very limited shared states among different invocations of the addConnection() and removeConnection() calls. If states need to be shared, for example, when a connection is successfully added, the shared states are protected with mutexes. A thread-pool model is used in the implementation [SCH96a]. By default, 32 threads are created when the ConnectionManager is started and these threads are dispatched to process requests when they arrive.

Measurements show that the multi-threaded version is slightly less efficient than the sequential implementation for a single client (Figure 4.7b). This is due to the overhead incurred in the locking of shared states and the context switching between different threads. The minimum latency for 1,000 calls is 22.5ms, average latency is 24.9ms and the maximum latency is 213ms. The 5% and 95% quantile are 23.1ms and 35.1ms, respectively. Notice that the worst case call setup time is much larger than in the sequential case. The gain in using threads can be seen only when there are multiple clients. In Figure 4.7(b), the result shows that throughput can be increased much more than in the sequential case as the number of clients increases, because the connection manager is able to keep itself busy by processing more than one request simultaneously.

4.3 The Design of the Connection Manager

In a distributed environment, a well designed connection manager has to consider the following:
- the vast majority of the remote operations during connection setup have small arguments
- remote calls contribute the bulk of the latency in call processing
- most computations are executed in the communication layer

Based on these observations, we designed the `xbind` connection manager with the following features, the main objectives being to reduce the number of remote invocations and to hide the latency of remote invocations as much as possible.

Reducing Remote Invocations

- *Caching of Network States* - store or pre-fetch resources so as to minimize the number of remote procedure calls.
- *Aggregate Access to Node Server Objects* - aggregate access requests to remote objects as much as possible.

Hiding the Latency

- *Overlap communication and computation through parallelization* - design the system to run with a maximum amount of parallelization so that processors can be kept busy as much as possible.

### 4.3.1 Caching of Network States

As shown in Figure 4.6, the connection manager performs remote invocations on three classes of objects: the QOSMapper, the RouteObject and the SwitchServer object. For each of these object classes, part or all of its states are stored or cached in the connection manager.

Caching has been used as a technique for speeding up remote invocations or hiding the latency, in the classical design of computer systems [HEN90] to the design of the world
wide web servers [MIC97]. The performance of the connection manager can be improved by using caching techniques. The five types of network states stored or cached are:

- **QOSMapping**
- **Route**
- Output (or input) Switching Identifiers
- Bandwidth and buffer resources
- Existing connection states

### 4.3.1.1 QOSMapping
Since the mapping performed by the QOSMapper is relatively static and is unlikely to change at all during the lifetime of the connection manager, the QOSMapper can be created in the same address space as the connection manager. If this is the case, accessing the QOSMapper from the ConnectionManager becomes a local invocation rather than a remote invocation. In the implementation, the collocated feature in Orbix is used to place the QOSMapper in the same address space as the ConnectionManager. In this way, there is almost no change to the rest of the code.

### 4.3.1.2 Route
During repeated call setups, patterns of call requests emerge that have the same source-destination pair. These patterns might include alternate routes between the same source-destination pairs. For calls belonging to these patterns, it is not necessary to have the connection manager contact the RouteObject each time a setup request is received. Instead, when the connection manager receives a route for a specific source-destination pair from the RouteObject, it will cache the route (or alternate routes) with a time-stamp. Statistics of route selection for the alternate routes may also be included. By defining a variable time-out period for route invalidation of, say, T seconds, the connection manager can re-use the "cached" route if it is less than T seconds old. If not, a new route will be requested. If the
expected call throughput is 100 calls/s, and the probability of a source-destination (SD) pair appearing in a call request is 0.01, then by setting T=10s, there will be 1 update per 10 seconds per SD pair, instead of 100 * 0.01 * 10 = 10 updates in 10sec per SD pair -- an improvement by a factor of 10. Updates are performed on demand. Therefore, if a route "times-out", there is no extra update.

4.3.1.3 Output (or Input) Switching Identifiers
ATM-based broadband networks employ cell-based transport techniques. Each ATM cell contains a header with routing information and payload. The two key information fields for switching purposes are the virtual channel identifier (VCI) and the virtual path identifier (VPI). In order to ensure the correct delivery of ATM cells, given the input port at which an ATM cell arrives and the input VCI/VPI field in its header, the cell must be routed to the correct output port, and the VCI/VPI header of the outgoing cell must have the appropriate value substituted. The information on how this mapping should be performed is stored in the cell routing table. The process of setting up this information is illustrated in Figure 4.8.

2. commit channel with input vpi/vci = A, out vpi/vci = B
2. set input vpi/vci = B

1. obtain for output port vpi/vci (A)
1. obtain for output port vpi/vci (B)

Caching switching name spaces
- perform step (1) in advance and cache the output VCIs.
- using these pre-fetched VCIs allows the connection setup phase to be reduced to a single phase.

Figure 4.8: Setting up the switching tables in an ATM network
This process requires two phases. In phase one, an output VPI/VCI pair is obtained from the output port of each of the ATM switches located in the path of the call. In the second phase, the output VPI/VCI pair of the upstream switch is mapped into the output VPI/VCI pair of the downstream switch and, thereby, the channel is committed. In Section 4.2.3, this two phase operation is indicated as steps (1) and (2). Note that, although in our description, the output VPI/VCI of the appropriate port of each intermediate switch is reserved first and then followed by the input VPI/VCI, the reverse (input followed by output) is also possible. However, it is very important to note that if there are multiple copies of connection managers running, in order to avoid contention, all connection managers must agree on which portion of the name space should be cached in advance. Therefore, either only the input port VPI/VCI pairs or the output port VPI/VCI pairs, but not both at the same time. Note that there is no restriction or coordination needed with respect to exactly what VPI/VCI ranges can be used.

If the output (or input) VPI/VCI pairs are known in advance, then the entire process of committing can be performed in one single phase. When the connection managers agree on caching the same set of name spaces (say output VCI/VPI), step (1) in Figure 4.8 can be performed in advance. The connection manager obtains control over a set of available output VPI/VCI pairs that it requests in advance (i.e., it reserves or pre-fetches) from the SwitchServers. During connection setup time, the connection manager simply looks for an available VPI/VCI pair in its name space cache. If an available output VPI/VCI pair is found (a cache hit) for each switch/port on the path of the call, then the channel reservation process can be performed in a single step. If no free VPI/VCI pair is available (cache miss), the normal two-step operation is performed for all switches with cache misses.

By keeping the available VPI/VCI pairs on the connection managers instead of putting them in the SwitchServer which directly controls the switch, the resource state of the ATM network is partitioned and distributed to the higher level controllers (in this case, the
connection managers). This partitioning and distribution process can be performed in two ways.

In the first approach, the amount of VPI/VCI pairs reserved per port per switch is competitively decided among the controllers reserving the resources. Thus, each connection manager adjusts the number of entries (VCI/VPI pairs) kept in its cache depending on the call arrival and departure statistics, and how much it is willing to pay for a low latency call setup. In the second approach, the size of the partitions is controlled by a distributed algorithm that attempts to optimize the partitioning of the name spaces on the network level. The two approaches differ in that, in the first case, the allocation process is performed using the rules of a competitive game whereas in the second approach, the partitioning process is performed in a cooperative manner. Combinations of the two approaches above are also possible.

Note that caching is performed separately per output port. In Section 4.4.2, we studied a competitive algorithm that attempts to minimizes the amount of VCI kept in the connection manager.

4.3.1.4 Bandwidth and buffer resources
The caching of bandwidth and buffer resources is similar in nature to VPI/VCI name space caching and is normally performed per output port. This scheme is particularly useful for reservations of bandwidth and buffer space to a particular connection manager, and much research has been done to exploit the trade-off between signaling load and bandwidth utilization. Related work on applying this framework to Virtual Path (VP) management can be seen, for example, in [ANE96], [CHA97], [OHT92] and [ORD96]. The cooperative and competitive framework described above also applies to the partitioning of the bandwidth and buffer resources.
4.3.1.5  Existing connection states

The last caching scheme described here involves existing connections. When a connection has been successfully set up, its state is kept by the connection manager. This information is also replicated in the SwitchServers. By doing so, the connection manager can recover its prior state from the SwitchServers. By caching such information locally in the connection managers, the state of the existing connections can be accessed using local object invocations. This information is particularly useful during QOS renegotiations.

Table 1 shows an example of the information cache in the connection manager (only the capacity per port is shown).

<table>
<thead>
<tr>
<th>End-to-End Connectivity</th>
<th>Route Cached</th>
<th>VCIs Reserved</th>
<th>Bandwidth Reserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Source-Dest</td>
<td>Port</td>
<td>Port</td>
</tr>
<tr>
<td>Existing Connections</td>
<td>Pair</td>
<td>Available</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>A -&gt; B</td>
<td>Port 5 of Node A</td>
<td>B1</td>
</tr>
<tr>
<td>State of connection request x.</td>
<td>A -&gt; E -&gt; B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>A -&gt; E</td>
<td>Port 3 of Node E</td>
<td>B2</td>
</tr>
<tr>
<td>State of connection request y.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: State Caching on a Connection Manager

4.3.2  Aggregate Access to the SwitchServer Object

In a distributed object environment where the vast majority of object interactions are in the request-reply form with small arguments (<1 kilobytes), the most expensive operations are remote invocations. In order to increase the throughput of the system, multiple requests are
combined into a single remote invocation, as opposed to making multiple individual invocations, one per request. As a result, in a single invocation, the argument is a list of commands, each command corresponding to a single request. The number of requests (sum of add and delete) stored in the message buffer before they are sent out is a control parameter. We will refer to this parameter as the message threshold parameter later.

This approach has the advantage that, by delaying the delivery of requests, the total number of remote invocations performed by the connection manager decreases for a fixed load. This is especially useful for invocations to the SwitchServers. Obviously, by delaying the delivery of messages, the expected call setup time increases for low loads. For higher loads, the gain in reducing the number of remote invocations can outweigh the delay introduced by waiting. Such a trade-off is studied in Section 4.4.1.

4.3.3 Hiding Latency Through Parallelization

Due to the inherent delay incurred in accessing remote objects, a substantial amount of processor time could be spent waiting or idling. In order to increase processor utilization, a multi-threaded approach, where each request is processed by a thread, is investigated in Section 4.2.5. Use of multi-threading exploits inter-request parallelism. This is easy if the requests are independent, which is the case if we are considering only independent point-to-point connections.

Intra-request parallelizing, where parallelization is performed within each remote invocation, is also possible. This can be achieved by employing asynchronous object invocations, implemented as oneway calls in CORBA. Obviously, since these remote invocations are not independent, parallelizing at this level will not be as simple and has to take into account the dependency among different oneway calls. As an example, using the model in Section 4.2.3, the height of the computation graph is 4 per connection request with route of one or more hops. Parallelizing at this level has the advantage that not only can the latency of a single connection be shortened, the total throughput of a connection manager
also increases. However, note that the latency is smaller only when the connection manager is under-utilized. If the connection manager is heavily loaded, both latency per connection and total throughput increases due to the multiplexing among requests.

The problem in using oneway calls is that it is not the most natural programming paradigm in an RPC-based environment such as CORBA. Unlike with using threads, oneway calls require a different programming style and involves non-trivial changes to the connection manager code. Essentially, a new set of interfaces is needed for all objects involved in the connection setup process. Each two-way interface is mapped into two sets of corresponding oneway calls. For each of these interfaces, the implementation is written such that access to shared states is minimized.

![Diagram](image)

**Figure 4.9: Parallelization using oneway object call invocations**

Figure 4.9 explains, graphically, the possible gain in parallelization. In the top half of the figure, calls are invoked sequentially. Each sequential (or synchronous) object call has three stages, namely: send, wait and acknowledge. When remote invocations are made sequentially, the total waiting time is the sum of the individual waiting times. On the other hand, if the send functions are performed first before waiting for the acknowledgments, that is if call executions are parallelized, the waiting time is shortened. This is shown in the bottom half of Figure 4.9, where all send messages are performed together, followed by a wait-
ing period and, finally, the reception of the acknowledgments. If the waiting time for receiving acknowledgments is much longer than the execution time of send, parallel execution has the potential of shortening the total waiting time to the maximum of the individual waiting times. Thus, by parallelizing the execution, the processor can be kept busy for a longer period of time while minimizing the total waiting period (and thus the total latency).

4.3.4 Functional Design of the Connection Manager

All the design choices described in Section 4.3.1, Section 4.3.2 and Section 4.3.3 are integrated into the design shown in Figure 4.10. The connection manager state machine, as described in Section 4.2.3, is unaffected by the caching and aggregation schemes. Step (3) is a local invocation, and step (4) is a remote call only if the route cached times-out. Step (1) is a local call if there is a VCI available in the cache. Thus, in the majority of the call setups, only step (2) requires remote invocation. For remote invocations to the SwitchServer, the requests are not delivered immediately. Instead, the messages to be sent are put in a message aggregation module, with one module per remote object. Therefore, aggregation is performed per object. Messages in these modules are sent when either the number of messages reaches a particular threshold, which can be dynamically changed, or when a time-out occurs. The message threshold parameter takes into account both add and delete requests. Time-outs are used as a safety mechanism to ensure that messages do not remain in the queue for too long without being processed. They are needed when the traffic load is low. These time-outs can also be set dynamically.
In order to support different modes of invocation, the SwitchServer supports three classes of interfaces. At the lowest level, a set of functions implements the set of synchronous remote invocations supported (Module A). For each of these synchronous methods, a pair of asynchronous methods is defined so that the acknowledgment is separated from the invocation (Module B). This is needed to support the asynchronous mode of invocation. Finally, a generic, asynchronous method is defined (Module C). This generic method takes as argument a list that can be used to represent any of the asynchronous methods in Module B.

In the implementation, Module C is needed because message aggregation is performed at the application level. This means that the connection manager is aware of the se-
mantics and presence of the message queue. Ideally, the aggregation should be performed at the Orbix-level so that the applications do not have to be aware of the message aggregation.

Figure 4.11: Interface design of SwitchServer
4.4 Performance Measurement of xbind Connection Manager

In this section, we present the performance results of the xbind connection manager. The experimental environment used is the same as that described in Section 4.2.5. Each point is the average of 10,000 calls. Recall that although the results are given for call setup per second, for each successful call setup, there is a corresponding delete operation. Thus, 10 call setups per second are equivalent to 20 call setup and delete operations per second.

In Figure 4.12(a), we show the throughput-delay characteristics of all connection managers described so far, namely: standard (or sequential) implementation, threaded implementation and the xbind implementation. For this particular measurement, the number of clients per connection manager is 1. For the xbind connection manager, routes cached time-out after 5 seconds, the message threshold is set to 1 (messages are sent immediately with no aggregation) and the amount of VCI cached is made so large (>200) that all arriving calls execute step (1) locally. Finally, messages that remain in the queue for more than 1 second will be flushed.

From Figure 4.12(a), it is clear that the xbind implementation out-performs the other implementations by a very significant margin. The smallest average latency is 11.0ms for a load of 1 call/s, and the system can support up to 60 calls/s with average call setup latency of about 40ms.
Figure 4.12: (a) Throughput-delay curve of all three connection managers (one client)

Figure 4.12: (b) Throughput-delay curve of xbind connection manager

Figure 4.12(b) shows the throughput-delay characteristics of the xbind connection manager when multiple clients are used. Due to the use of oneway asynchronous calls to exploit intra-request parallelism, the processor running the connection manager is kept
busy enough such that adding more clients does not extract higher throughput. In fact, when three clients are used, the latency increases dramatically due to excess load.

The above figures demonstrate the gain in the caching schemes described in Section 4.3.1. In the next section, we will study the impact of the message threshold parameter.

4.4.1 Impact of Message Threshold Parameter

The message threshold parameter, T, influences the throughput-delay characteristics of the connection manager in two different ways. When the threshold parameter increases, obviously the message stays in the aggregation module for a longer time thereby increasing the total delay. Assume that the call arrival rate is \( \lambda \), and for every add request there is a corresponding delete request, the average waiting time for a message before it is delivered is \((T-1)/\lambda\).

On the other hand, as the threshold increases, the total number of messages delivered in a single remote invocation also increases. As a result, the total number of remote invocations for a fixed message arrival rate is reduced. A reduction in the total number of remote invocations leads to a smaller communication delay between the ConnectionManager and the SwitchServer. Again, let \( \lambda \) be the call arrival rate and assume that there are 3 SwitchServers. For every add request, there is a corresponding delete request and for every send, there is an acknowledgment. The total number of remote invocations performed, \( N \), is \( 4\lambda(1 + 3/T) \).

Figure 4.13(b) is a plot that shows how \( N \) changes with increasing value of \( \lambda \) for various values of \( T \). Increasing the message threshold parameter \( T \) decreases \( N \) but the reduction rate decreases with increasing \( T \). For \( T>10 \), the reduction in \( N \) is rather insignificant. The asymptotic reduction, therefore, the smallest possible \( N \), achieved when \( T=\infty \), is shown for comparison purposes. Note that for a given connection manager, there is a bound on the maximum number of remote invocations that can be processed in a second.
\[ \lambda = \text{Call Arrival Rate} \quad T = \text{Message Threshold Parameter} \quad \text{inf} = \text{infinity} \]

Figure 4.13: Remote invocation load on the ConnectionManager

The influence of \( T \) on the performance of the connection manager is thus rather complex. It directly changes the waiting time of a message before it is delivered and by changing the total number of remote invocations performed, indirectly changes both the processing latency and maximum throughput attainable.

Without resorting to an analytical model, the overall impact of the message threshold parameter on the throughput-delay characteristics of the connection manager is evaluated experimentally. The results are shown in Figure 4.14. For each threshold parameter, the arrival rate of call setup requests to the connection manager sent from one client is increased from a low load of 1 calls/s to the maximum load whereby the system is stable.
Figure 4.14: Impact of threshold parameter on throughput-delay characteristics of xbind connection manager

A trend can be seen from the plots. For low traffic loads, a small threshold gives better average latency. However, as the traffic load increases, the total message processing load experienced by the connection manager reaches a saturation point. In order to support a higher traffic load, the threshold parameter has to be increased, thus reducing the total processing load. As a result, a lower call setup latency is traded for higher throughput. When
the threshold is set to 15, a throughput of 170 calls/s can be supported with an average latency of 85ms. As indicated by Figure 4.13(b), increasing the threshold further does not significantly reduce the processing load on the connection manager. Given an extremely heavy load, the connection manager spends most of its processing time receiving messages from and sending messages to the client applications, leaving little time to process these requests.

Although the measurement shows that there is a gradual increase in both latency and throughput as the threshold parameter is increased, from the point of view of setting the threshold parameter for good performance, it is not necessary to tune the parameter precisely. Instead, the operation of the connection manager can, in general, be classified into three regions: low traffic load (0-50 call/s), medium traffic load (50-100 call setup/s) and high traffic load (>100 call setup/s). A threshold parameter can be associated with each of these operating regions. This is illustrated in Figure 4.14(b). A threshold of 1 is used for the low traffic case, which provides low latency call setup but only up to 50 calls/s. A threshold of 4 (or 5) is used for the medium load case. For arrival rates of greater than 100 calls/s, a threshold of 10 is used. By estimating the call arrival rate, the connection manager can dynamically change its threshold parameter so as to obtain a good trade-off between latency and throughput.

4.4.2 Dimensioning the Switching Identifier Cache Size
Caching for the switching identifier is performed per output (or input) port per switch. The switching identifier for a particular port is, in general, a VPI/VCI pair. For the sake of simplicity, we will refer to it as a VCI cache.

When a call request arrives and there is a VCI available in the VCI cache, the channels on the SwitchServer can be committed in a single phase. This will be referred to as a cache hit. Otherwise, there is a cache miss and an extra remote invocation is executed to fetch a new VCI from the SwitchServer. In this section, we identify the size of the VCI
cache required to satisfy the requirement that with probability $P$, there is a VCI locally available for immediate use.

Note that from the SwitchServer's point of view, the total amount of VCIs allocated to the ConnectionManager is the sum of connections that have been set up and the VCIs cached by the ConnectionManagers.

The system is modeled in the following way. Assume that there is an infinite amount of VCIs available in the VCI cache, and initially let the variable $N_v$ be 0. When a request arrives, one VCI is removed from the VCI cache, and $N_v$ is increased by one. Processing continues and eventually, a commit channel request is sent to the SwitchServer. When the acknowledgment returns from the SwitchServer, the channel has been committed and $N_v$ is decreased by one. The variable $N_v$ can therefore be interpreted as the number of requests currently being processed and each of these requests requires a single VCI.

In the implementation, the number of VCIs is not infinite and has to be replaced. The request for a replacement VCI is piggy-backed onto the commit channel request, and a new VCI in returned to the cache when the commit channel acknowledgment returns. With such a scheme, the significance of $N_v$ is obvious. $N_v$ is also the number of VCIs needed to sustain the processing of call setup request in a single phase. As a result, it is possible to calculate the size of the VCI cache needed so that a certain percentage $P$ of the total request can be processed in a single phase, if the distribution of $N_v$, $P(N_v \leq X)$ is known,
Figure 4.15 shows two sample paths of $N_v$ with an arrival rate of 100 calls/s and a call holding time of 5 seconds. In Figure 4.15(a), the message threshold parameter is 5, and in Figure 4.15(b), the message threshold parameter is 15. The figures show that the distribution of $N_v$ is random even though the threshold itself is a constant. This is because the threshold takes into account both add and delete requests, while $N_v$ only takes into account the add requests. In addition, the figures show that by using a large message threshold parameter, more messages are buffered in the message queue, and $N_v$ tends to be larger.

In order to have an estimate of the distribution of $N_v$, we model the system in the following way. The call arrival process is modeled as a Poisson process, and the call processing process is modeled with a general distribution with average processing rate of $\mu$. In addition, we assume that there are an infinite number of servers. Each request is served by a server, and the servers have independent and identical service time distribution. In other words, the system is modeled as a $M/G/\infty$ queue (Figure 4.16).
\[ \lambda = \text{Call Arrival Rate} \quad \mu = \text{Call Processing Rate} \]

Figure 4.16: Modeling the processing on the ConnectionManager as a M/G/\infty queue

Modeling the system as a M/G/\infty queue allows the distribution of \( N_v \) to be computed in a very easy way. In fact, \( N_v \) is another Poisson process with mean \( \lambda/\mu \) (see Chapter 2 of [WOL89]). The value of \( \mu \) is the average latency for a VCI to be placed. It depends on the message threshold parameter, ConnectionManager processing time, communication delay between ConnectionManager and SwitchServer and the processing time on the SwitchServer. This is intuitively correct. A larger threshold or a larger processing and communication delay will require more VCIs to be cached for a fixed hit frequency. \( \lambda/\mu \) can be estimated in two ways, either directly, as the product of the estimated arrival rate and average latency for a VCI to be replaced, or as the average value of \( N_v \), since the distribution is Poisson.

Using this model, we compare the measured distribution of \( N_v \) with the estimated distribution (using the average value of \( N_v \) measured). This is shown in Figure 4.17. The measured distribution is for a total of 100,000 call setups. Note that the region of interest is the region where \( P(N_v \leq X) \) is large, therefore where \( P(N_v \leq X) > 0.9 \). For these regions, the estimated distributions match the measured distributions rather well.
By modeling the system as a $M/G/\infty$ queue, the question of how to dimension the VCI cache for each output port can now be answered. An example using two of the plots in Figure 4.17 is given in Table 4.2. For each cache size, the estimated and measured cache hit frequencies are shown for comparison. For the medium load case, by keeping 10 VCIIs in the connection manager, there is almost no cache miss. For the high load case, 20 VCIIs are sufficient to attain a cache hit ratio of more than 0.99.
<table>
<thead>
<tr>
<th>VC/Cache Size</th>
<th>Estimated Cache Hit Frequency</th>
<th>Measured Cache Hit Frequency</th>
<th>VC/Cache Size</th>
<th>Estimated Cache Hit Frequency</th>
<th>Measured Cache Hit Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium Load (L=50, m=12, T=5)</td>
<td>0.9529</td>
<td>0.9859</td>
<td>12</td>
<td>0.9354</td>
<td>0.9081</td>
</tr>
<tr>
<td></td>
<td>0.9950</td>
<td>0.9996</td>
<td>13</td>
<td>0.9653</td>
<td>0.9429</td>
</tr>
<tr>
<td></td>
<td>0.9997</td>
<td>0.9999</td>
<td>15</td>
<td>0.9916</td>
<td>0.9771</td>
</tr>
<tr>
<td></td>
<td>0.9999</td>
<td>0.9999</td>
<td>20</td>
<td>0.9999</td>
<td>0.9935</td>
</tr>
<tr>
<td>High Load (L=100, m=62, T=15)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Difference between estimated and measured cache hit frequency

As a measure of the overhead incurred by the caching scheme, we use the ratio of $N_V$ to the average number of calls. For the medium load case, the average number of calls going through the output port is $50/0.2 = 250$ calls. The ratio of VCI cache size to average call occupancy is therefore $10/250 = 0.04$. For the high load case, the ratio is also $0.04 (20/500)$. This is a rather small price to pay for a large improvement in throughput-delay characteristics of the call processing system.

Another measure of overhead is the storage requirement. This overhead can be estimated in the following way. Let there be 128 nodes in the network, with 32 ports per node. Assume that 32 VCIs are cached for each port. (This is more than enough to support 100 calls/s with a message threshold of 15, as indicated in Table 4.2). The total number of VCIs cached is therefore $128 \times 32 \times 32 = 128K$ VPI/VCI pairs. Let the storage for each VPI/VCI pair be 16 bytes (we include the overhead to support searching and indexing). The total storage needed is therefore only $128K \times 16 = 2$ Mbyte, a small requirement given the size of the network and arrival rate considered.
4.5 Conclusion

In this chapter, we have described a CORBA-based call processing system for ATM networks which has low latency and high throughput. The components of the connection manager consist of a connection manager state machine, a message aggregation queue, a route cache module, a name (VPI/VCI) cache module and a bandwidth cache module.

The latency and throughput of call processing is improved by caching and message aggregation schemes which reduce the number of remote accesses. Parallel processing of a single call request is also used to enhance the performance of call processing. The behavior of the throughput-delay characteristics of the call processing system is studied with respect to two control parameters, namely: message aggregation threshold and size of switching identifier cache. Based on the results, appropriate control parameter values can be identified for the desired trade-off among resource utilization, call setup latency and call throughput.
Chapter 5  Conclusion and Direction for Future Research

5.1 Conclusion

In this dissertation, we have presented problems and solutions related to provisioning of ATM VPN service and connection management in ATM networks.

In Chapter 2, the deficiency of current VPN services is highlighted, in particular, the lack of customer control. As users become more sophisticated and services become more complex, the issue of controllability will be increasingly important. The proposed solution is a VPG-based VPN which gives the user control over a "virtual" ATM network, superimposed on the provider's network. A VPG-based VPN provides a very good intermediate point between the use of leased line, where there is minimum network and user interaction, and the use of VC setup, where every change in the user domain must be known to the network provider.

In order to exploit the new control capability, a layer control architecture, organized according to time scale, is used. Each control layer is modeled by a generic controller design concept which allows a large class of control objectives and control schemes to be implemented.

The concept of QOS guarantee is based on constructing Schedulable Regions over CBR VP. This approach requires a minimum guarantee from the network providers and at the same time can meet the need for better efficiency through multiplexing.
The performance of two sets of control algorithms, each consisting of a VP admission control algorithm and a VP reallocation algorithm are compared. As a reference, the performance of static allocation is also included. The results show that a more computational intensive algorithm works well for a small recomputation period (<30s). When the recomputation is larger, a simple heuristic algorithm works better. Both algorithms are better than static allocation.

In Chapter 3, an approach for developing networking software is presented. The objective is to allow better code and design re-use between the emulation platform and the target platform. Experience has shown that this approach can be very useful. A significant amount of software written for VPN control architecture on the emulation platform has been re-used on the target platform (xbind). The porting process, however, is still rather tedious. The observation is that this process is rather well defined, and a large portion of the porting could be automated by using a compiler. This compiler performs function similar to the stub generator in RPC or IDL compiler in CORBA.

The design of a high performance CORBA-based ATM connection manager is covered in Chapter 4. The components of the connection manager consists of a connection manager state machine, a message aggregation queue, a route cache module, a switching identifier (VPI/VCI) cache module and a bandwidth cache module. The latency and throughput of call processing is improved by message aggregation and the various caching schemes which reduce the number of remote accesses. Parallel processing of a single call request is also used to enhance its performance. The behavior of the throughput-delay characteristics of the call processing system is studied with respect to two control parameters, namely: message aggregation threshold and size of switching identifier cache. Based on the results, appropriate control parameter values can be identified for the desired trade-off among resource utilization, call setup latency and call throughput.
Throughout this dissertation, strong emphasis has been given to the ability to demonstrate the concepts described through prototype implementations. Both the VPN control architecture and the connection management architecture runs on the xbind platform.

5.2 Direction for Further Research

Issues related to the requirements on the VPG topology are not studied. Such requirements can be considered as an additional services provided by the providers to the customers. An important issue is how should the mapping between the VP and the VPG be performed. In general, more statistical multiplexing is possible when VPGs contains more VPs. On the other hand, if VPs are concentrated in a few VPGs, the reliability of the VPN in terms of VPG failure will be low. A good trade-off between multiplexing gain and reliability is needed.

The work on prototyping has demonstrated the feasibility and utility of the prototyping approach proposed. The prototyping process is, however, still rather tedious. Enhancing the approach further, for example by adding a RPC layer on the emulator, or automating part of the porting process by adding a compiler for translating messages to stub-code for object invocations, would be good extensions.

The tasks of provisioning of ATM VPN and design of ATM connection management systems are related to the issue of defining the boundaries among different domains (applications, valued added service provider and network provider etc.) and the distribution of intelligence and states among these domains. It should be clear that the solution to VPN provisioning assumes a more restricted interaction model, and the solution for connection management uses a much more open interaction model. Depending on user (or service) requirements, there is probably no one interaction model that will satisfy the needs of most, if not all, users. The challenge is therefore to be able to provide a unified framework where all these different interaction models can be supported in a clean and efficient way.
Switching identifiers in an ATM network, just like buffer and link bandwidth, are resources that can be partitioned and reserved. For VC service, both switching identifiers and bandwidth are reserved on demand, and for VP service, these resources are reserved in advance. Advance reservation of switching identifiers can be considered as an alternative to, or mid-way point between, VC service and VP service, since only bandwidth is reserved on demand. A new class of VPN services, where all three categories of reservation are used in combination, can be provided. Trading of both bandwidth and switching identifiers makes sense in this context. The potential for such a service seems promising, but more investigation will be needed. In addition, such a service will involve partitioning of both bandwidth and switching identifiers. How these resources should be shared or partitioned is not clear at this point.

Finally, many issues concerning state distribution and scalability are far from understood. For example, when a VC is set up, the question of what type of information needs to be recorded, say for billing purposes, arises. Furthermore, should the state be kept in the connection manager, in the switch software, or perhaps should the information not be kept track of at all? In order for the control and management system to scale, the appropriate granularity of information and where it should be stored have to be found.
References


[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley, 1995.


