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.

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.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
ON THE FOUNDATIONS OF NETWORK PROGRAMMABILITY

Mahesan Nandikesan

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

Columbia University
2002
Abstract

ON THE FOUNDATIONS OF NETWORK PROGRAMMABILITY

Mahesan Nandikesan

In recent years, programmability in telecommunication networks has been recognized as a powerful paradigm for realizing network services. Yet, no systematic foundation for capturing network programmability has been laid. The present thesis attempts to fill this void.

The essence of network programmability is the prevalence of programming interfaces for altering the network state to create and remove communication channels with quality of service. The thesis lays the foundations of network programmability in terms of (i) a core network resource model, (ii) a network-element service model, (iii) topology abstraction, (iv) a network service model, (v) APIs for accessing these models, and (vi) APIs for signaling.

The thesis identifies communication channels (modeled as graphs) and their manipulation as the essence of network control. It presents concrete resource and service models and APIs for building communication graphs, applicable to connection-oriented and connectionless networks, packet-switched, circuit-switched and wavelength-switched networks. This is the first time such generality has been achieved. Yet, the models and APIs are very compact. The thesis also presents peer-to-peer APIs for
network resource discovery. It introduces the concept of *minimality* of an API, and shows that the APIs presented are minimal.

The thesis identifies two *fundamental capabilities* of a network, which are *necessary and sufficient* to realize a network control infrastructure. A third fundamental capability is shown to help performance optimization.

The benefits of the foundations are (i) conceptual clarity resulting from the identification of network control as graph manipulation, (ii) programming simplicity, and (iii) separation of control algorithms from the network state. Of these, the first two are original contributions of this thesis. The following are demonstrated: The APIs resulting from the foundations greatly simplify the task of writing software for network resource discovery. The source code is much smaller, clearer and reflects high-level concepts much more clearly. A simple RPC-like ORB is adequate for accessing these APIs remotely – *not even a name service is necessary*. The performance of a system implemented with these APIs is only slightly lower than that of one written directly in terms of protocols.
# Table of Contents

1 Introduction ................................................................. 1
   1.1 State of the Art ...................................................... 1
   1.2 Review of Relevant Work ........................................... 4
      1.2.1 BIB .......................................................... 4
      1.2.2 GSMP ....................................................... 5
      1.2.3 GMPLS ...................................................... 7
      1.2.4 IEEE PIN ................................................... 8
      1.2.5 Programmable Network Architectures ..................... 10
      1.2.6 Active Networks .......................................... 12
      1.2.7 Miscellaneous ........................................... 13
   1.3 Structure of the Foundations ...................................... 14
   1.4 Benefits of the Foundations ..................................... 18
   1.5 Thesis Outline and Main Contributions ....................... 19

2 Modeling Telecommunication Networks .................................. 22
   2.1 Preamble ..................................................................... 22
      2.1.1 Transport ..................................................... 24
      2.1.2 Signaling ..................................................... 24
   2.2 The Core Network Resource Model ............................... 26
2.2.1 Preliminary Concepts ........................................ 28
2.2.2 A Generic Network Resource Model ....................... 32
2.2.3 Extension for Quality of Service .......................... 41
2.3 The Network-Element Services ............................... 44
   2.3.1 Segments ................................................. 44
   2.3.2 Virtual Links ............................................ 47
2.4 The Network Services ....................................... 48
   2.4.1 Topology ................................................ 48
   2.4.2 Network Communication Graphs ......................... 49
   2.4.3 Graph Embedding ...................................... 51
2.5 Scaling ...................................................... 53
   2.5.1 Domain Resource Model ................................ 54
   2.5.2 Domain Service Model ................................. 56
2.6 Note on the Development of the Model .................... 59
2.7 Generality of the Model .................................. 60
   2.7.1 ATM Networks ......................................... 61
   2.7.2 Switched Ethernet Networks .......................... 62
   2.7.3 Classical IP Networks ................................ 62
   2.7.4 RSVP-Enabled IP Networks ........................... 63
   2.7.5 GMPLS-Enabled IP Networks ......................... 64
   2.7.6 DiffServ-Enabled IP Networks ....................... 65
   2.7.7 Frame Relay Networks ................................ 66
2.7.8 SONET Networks .............................................. 66
2.7.9 Wavelength-switched Networks .......................... 67
2.7.10 Fiber-switched Networks ................................. 67
2.7.11 Hybrid Networks ............................................ 68
2.8 Concluding Remarks ........................................... 70
2.A Appendix ....................................................... 71
2.A.1 Datapath Protocol Translation ............................ 71
2.A.2 Extension to the Generic Network Resource Model for Exploiting Multiplexing Gain ......................... 77
2.A.3 Networking Capacity Graph .............................. 80

3 APIs for Building Graphs and Minimality .................. 83
3.1 API Framework for Building Graphs ...................... 83
3.2 The APIs ......................................................... 86
3.2.1 API for Controlling the Core Network Resources .... 86
3.2.2 API for Creating Network-Element Services .......... 90
3.2.3 API for Reading the Network Resource Configuration 91
3.2.4 API for Creating Network Services .................... 95
3.3 Graph Building ................................................. 97
3.4 Minimality ...................................................... 100
3.4.1 Basic Definition ........................................... 100
3.4.2 Minimality of the Core Network Resource API ....... 106
5 Benefits of the Foundations

5.1 Introduction ........................................... 154
5.2 Conceptual Clarity .................................... 154
  5.2.1 State-Space Formulation ......................... 155
  5.2.2 Fundamental Capabilities ....................... 156
5.3 Simplicity of Programming .......................... 163
  5.3.1 General Discussion of ORBs and Protocols ...... 163
  5.3.2 State Space and Source Code Comparison ........ 168
  5.3.3 Performance Comparison ....................... 176
5.4 Concluding Remarks .................................. 191
5.A Appendix .............................................. 194
  5.A.1 The Simple ORB as a Strip-Down Version of CORBA .. 194
  5.A.2 Messages Formats ................................ 196

6 Conclusion .............................................. 208

6.1 Thesis Summary ...................................... 208
6.2 Directions for further Research .................... 210

A Estimating the Schedulable Region .................. 212

A.1 Motivation ........................................... 212
A.2 Aggregate MPEG streams ............................ 217
A.3 Measurement Techniques ............................. 225
A.4 Practical Considerations ............................ 228


<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.5</td>
<td>Estimating the Schedulable Region</td>
<td>229</td>
</tr>
<tr>
<td>A.5.1</td>
<td>Single Traffic Class</td>
<td>230</td>
</tr>
<tr>
<td>A.5.2</td>
<td>Multiple Traffic Classes</td>
<td>232</td>
</tr>
<tr>
<td>A.6</td>
<td>Numerical Results</td>
<td>239</td>
</tr>
<tr>
<td>A.7</td>
<td>Concluding Remarks</td>
<td>243</td>
</tr>
</tbody>
</table>

References 245
List of Figures

1.1 The hierarchy of APIs, shown by T-shaped symbols. .................. 17

2.1 Abstractions leading to the modeling of communication channels as
graphs. .................................................. 27

2.2 A generic label switch. ......................................... 27

2.3 A multiplexer. .................................................. 36

2.4 A QoS-enabled label switch. ..................................... 44

2.5 A segment ..................................................... 45

2.6 Virtual links connecting with a segment. ......................... 48

2.7 A communication channel modeled as a graph. .................. 50

2.8 Graph embedding. G is the entire graph shown and G' is the sub-graph
indicated via the oval. ........................................ 52

2.9 Graph embedding ................................................ 53

2.10 Domains ....................................................... 56

2.11 Links of the domains A_{11}, A_{12} and A_{13}. ....................... 57

2.12 Links of the domains A_{21} and A_{22}. .......................... 57

2.13 Links of the root domain. .................................... 58

2.14 A hybrid network ............................................. 69
2.15 A networking capacity graph for a network with four nodes and six links. For clarity, the links are shown unidirectional. The schedulable regions are shown as tents. ............................... 81

3.1 The API framework for building graphs. ......................... 85
3.2 C API for manipulating the forwarding table. .................... 88
3.3 C API for manipulating the policer table. ....................... 89
3.4 C API for reading the label weights for TDM and WDM ports. . 89
3.5 IDL definitions of ports, labels, traffic and QoS parameters. . 92
3.6 IDL API for manipulating segments. ............................. 93
3.7 Amendments to the segment-manipulation API for supporting service classes. .................................................. 94
3.8 IDL API for reading the network resource configuration. ....... 96
3.9 IDL API for manipulating network communication graphs. ...... 97
3.10 Illustration of Proposition 3.1. .................................. 106
3.11 Illustration of Proposition 3.2. .................................. 113
3.12 The API framework after the domain decomposition. .......... 120
3.13 API exposed by the domain port mapping algorithms. ........ 121
3.14 (a) Model for a multiplexer. (b) Decomposition of scheduling policies.

A two-level decomposition, consisting of sub-scheduling policies $f$ and $g$ composed via a third policy $h$, is shown. ......................... 122
3.15 C API extension for synchronization between a label switch and its switch control processor. ................................................. 129

4.1 The interfaces for obtaining the link states. ......................... 138

4.2 The API between objects representing neighboring nodes. ........ 140

4.3 The IDL structures for database synchronization and flooding. .. 141

4.4 The IDL interface for database synchronization and flooding. .... 142

4.5 APIs for scaling the resource discovery algorithm. .................. 143

4.6 The APIs for the multidomain extension. ......................... 144

4.7 Illustration of Conjecture 4.1. ....................................... 145

4.8 Meta segments. ....................................................... 148

4.9 Illustration of a possible race condition. .......................... 153

5.1 Client-side C++ code for the ORB-based example. ................. 170

5.2 Server-side C++ code for the ORB-based example. .................. 171

5.3 Client-side C++ code outline for the protocol example. .......... 172

5.4 Server-side C++ code outline for the protocol example. .......... 173

5.5 Common code for both the client and the server in the protocol example. 174

5.6 Sequence 1 for the protocol-based solution. ........................ 179

5.7 The sequence of messages exchanged by two neighbors when one of them (the client) invokes getLSASummaryList on the other (the server). 180
5.8 Sequence 1 for the ORB-based solution. The following notation is used:

'Request $x$' refers to the GIOP request generated by neighbor $x$ ($x = A$, B). 'Reply $xi$' refers to the $i$-th TCP fragment generated by the GIOP reply to node $x$'s request. 'ACK $xi$' refers to the acknowledgement generated by the receipt of Reply $xi$. .......................... 181

5.9 Sequence 2 for the protocol-based (top) and ORB-based solutions (bottom). ......................................................... 182

5.10 Sequence 3 for the protocol-based (top) and ORB-based solutions (bottom). ......................................................... 183

5.11 The stripped-down CORBA module. .............................. 197

5.12 The stripped-down PortableServer module. (continued in Figure 5.13). 198

5.13 The stripped-down PortableServer module. (continued from Figure 5.12). ......................................................... 199

5.14 The structure LinkInfo in protocol format. This encoding is common to both the protocol-based solution and the ORB-based solution, except that the latter has an additional four bytes indicating the dimensionality of the operating point. ........................................ 200

5.15 The LSAHeader and LSA structures in protocol format. These encodings are common to both the protocol-based solution and the ORB-based solution. ......................................................... 201

5.16 The database description packet of the protocol-based solution. ... 202
5.17 The link-state request, reply and acknowledgement packets of the protocol-based solution, shown in order from top to bottom. 203

5.18 GIOP message header (top), request header (middle) and reply header (bottom). 204

5.19 GIOP request and reply messages for getLSASummaryList. The encoding of LSA headers is as given in Figure 5.15. 205

5.20 GIOP request and reply messages for getLSAs. The encoding of LSAs is as given in Figure 5.15. 206

5.21 GIOP request and reply messages for recvLSAs. 207

6.1 Summary of the APIs and the propositions that demonstrate their minimality. 209

A.1 Sample paths of the number of bytes arriving in three aggregate video streams. Graphs (a), (b), and (c) correspond to the streams consisting of the aggregation of 1, 19, and 57 MPEG-1 streams, respectively. Each measurement sample was taken over a period of 30ms. 220

A.2 Histograms of the sample paths shown in the Figure A.1, and also those of the sample path with measurement window \( \tau = 3\mu s, 300\mu s, 3ms \). The abscissa is scaled by 30ms/\( \tau \). The legend gives the value of \( \tau \) for each curve. 221

A.3 Histograms of the sample paths shown in the Figure A.1, but restricted to the first 9000 samples and taken with a measurement window \( \tau = 3ms. 222 \)
A.4 Quantile-quantile plots of the standard Gaussian distribution with respect to the distributions shown in Figure A.2 for \( \tau = 3 \text{ms} \). 223

A.5 Quantile-quantile plots of the distributions shown in Figure A.2 with respect to the distributions shown in Figure A.3, for \( \tau = 3 \text{ms} \). 224

A.6 Real-time estimates of \( \mu \) and \( \sigma \). 240

A.7 Estimates of the one-dimensional schedulable region for lines rates of (a) 155 Mb/s and (b) 622 Mb/s. Figure (c) shows the sample path of the operating point \( n(t) \). 241

A.8 Two-dimensional schedulable region with traffic parameters \( \mu_I = 3 \) Mb/s, \( \sigma_I = 2.1 \text{Mb/s}, \mu_{II} = 0.58 \text{Mb/s}, \sigma_{II} = 0.6 \text{Mb/s} \) and line rate (a) 155 Mb/s and (b) 622 Mb/s. In each graph, the upper curve corresponds to Equation (A.12) and the lower curve corresponds to Equation (A.11). 242
List of Tables

5.1 Comparison of the ORB-based and protocol-based solutions. .......... 176

5.2 Messages (packets) resulting from the protocol-based and ORB-based solutions. The column “PBS Packet” denotes the packets generated by the protocol-based solution, and “GIOP Packet” denotes the packets generated by the ORB-based solution. ......................... 178

5.3 GIOP marshaling of data types commonly used for network control. The quantities shown in parenthesis are bytes needed for padding that is sometimes necessary to satisfy GIOP marshaling rules. The padding bytes are shown under the assumption that only the data types listed in this table will be used. ................................. 187

5.4 Comparison of message lengths (in bytes) under the protocol-based solution (PBS) and the ORB-based solution (OBS). .................... 189

5.5 Performance comparison for the three sequences. The latency for the protocol-based solution is shown as PBS and the latency for the ORB-based solution is shown as OBS. The latencies, shown in microseconds, are valid to two significant figures. ................................. 192
Acknowledgements

My stay at Columbia for my doctoral studies has been a period of immense personal and professional growth.

First and foremost, I would like to thank my thesis advisor, Prof. Aurel A. Lazar. Working with Aurel has been a very rewarding experience. I learned a lot from his vision, enthusiasm, and penchant for abstract unification of concepts. His presence is always inspiring.

Thanks to Constantin Adam, with whom much of the early work that served as preparatory material for this dissertation was conducted. Thanks also to Dr. Giovanni Pacifici and Dr. Cheul Shim for two early projects from which I learned a lot. I also thank the members of the COMET Group, both present and past, for many valuable discussions and friendships.

Thanks to Prof. Andrew Campbell, Prof. Predrag Jelenković, Dr. Giovanni Pacifici and Dr. Nikos Anerousis for serving on my dissertation committee.

Finally, but most importantly, I thank my mother and father, who supported me throughout my education. Their encouragement and motivation resulted in my pursuing higher studies. Sadly, my father did not have the pleasure of seeing me complete my doctoral studies, due to his untimely death.
Dedicated to

my mother and the memory of my father.
1

INTRODUCTION

1.1 STATE OF THE ART

This thesis focuses on answering in part the question "What are the foundations of network programmability?" Before embarking on this quest, we review the state of the art in network programmability prior to the present work.

The term 'network programmability' is widely used with differing meanings in the literature but with no formal definition. In this thesis, the term 'network programmability' refers to the prevalence of open programming interfaces for altering the state of the network elements (switches and routers) for the purpose of creating and removing communication channels with quality of service. This is true even in connectionless networks, as will be explained in Section 2.1.2. A more precise definition will be offered later. Access to control the state of the network hardware permits network control algorithms to create and remove communication channels on demand. Different control algorithms could be executed on the same network, possibly even simultaneously. Thus, network programmability takes the approach of control theory and game theory. A network that supports network programmability will be referred
to as a ‘programmable network’.

Unlike computing systems, telecommunication networks, for much of their history, exposed few programming interfaces. Since the advent of stored program control in telephone switches and Internet routers, programmability entered the field of telecommunications in a limited way in the form of peer-to-peer protocols (which may be viewed as low-level interfaces). Such protocols focus on distributed algorithms rather than on providing open access to the hardware states. Thus, they do not permit new algorithms to be written by anyone except by the hardware vendors. This is similar to the mainframe model of computing. While the hardware and software are tightly interwoven in a mainframe computer, they are fully separated via open programming interfaces in a personal computer. The personal computer revolution has amply demonstrated the benefits of such separation.

Traditionally, telecommunications engineering has been based on protocols, with great emphasis on bit-streams and filtering of the bit-streams (the signal) from noise. There is great inertia against the use of higher-level abstractions such as functional programming interfaces. Even though programming platforms have become highly sophisticated, much of the telecommunications industry is still based on low-level protocol abstractions. This inertia is quite large, given the fact that the abstractions used by computer scientists in the last few decades has grown very powerful and sophisticated.

However, in recent times, programmability in telecommunication networks is being recognized as a powerful paradigm for dynamically creating, deploying and managing
network services and architectures. Technical conferences such as the IEEE OpenArch [35] and OpenSig [80], and industry forums such as the Multiservice Switching Forum [62] and the Softswitch Consortium [42] have been created for the furtherance of network programmability in one form or another. Independent organizations such as the IETF and IEEE are focusing on a basic aspect of network programmability, namely the open control of network elements. The IETF has created working groups for GSMP [38] to address open switch control, and MEGACO [39] to address media gateway control (in collaboration with the ITU). The IEEE has created a working group for the open control of ATM switches and IP routers [36]. The article [12] by a group of contributors to the Multiservice Switching Forum summarizes these developments. A related, but significantly different initiative, namely that of active networks, has also gained popularity [1].

Yet, unlike many other branches of computer science, the study of telecommunication network control lacks the basic foundations for the systematic development of programming interfaces. But, it has grown rich in commercially-desirable features ranging from redundancy and high performance to billing and security. Thus, telecommunication networks at once appear to be all-encompassing creations that defy taming.

The rest of this chapter is organized as follows: Section 1.2 reviews relevant work from industry initiatives and the literature. Section 1.3 summarizes the framework of the foundations. Section 1.4 summarizes the benefits of laying the foundations. Section 1.5 outlines the rest of this thesis and the main contributions thereof.
1.2 Review of Relevant Work

1.2.1 BIB

The Binding Interface Base (BIB) [51, 52] comprises of a set of resource and quality of service abstractions for capturing the resources of a network element and those of an end-system. The abstractions are represented as IDL interfaces [67] and classified into an inheritance tree. These interfaces provide methods for reading and manipulating the variables of the abstractions. The purpose of the BIB is to provide open interfaces that could be used by distributed algorithms such as those of connection control. The BIB was developed as the counterpart of the Management Information Base (MIB), which is used for management operations. The term ‘BIB’, though generally applicable to the present thesis as well, shall be reserved to reference the original papers [51, 52]. The present thesis grew out of the work on the BIB, and in many ways is a continuation of that work.

The BIB as presented in [51, 52] and later in the IEEE PIN [36] was developed for ATM networks only. It captures multiplexers based on a set of quality of service abstractions, but not based on resource abstractions. As a result, it captures ATM networks at a higher level, thus narrowing its scope. The BIB uses the schedulable region as the fundamental abstraction of a multiplexer. The evaluation of the schedulable region is left implicit. An extension to the BIB for use in wireless ATM networks was given in [6].

The BIB papers list a number of interfaces and methods. Though a vast improve-
ment compared to the past, these methods and interfaces are still bulky and hence lack the concision and crispiness we seek. A part of this is due to the fact that the BIB covers end-system control (such as the control of transport protocol engines) in addition to network element control. Yet, even the subset of methods presented in the BIB to capture network element control is large.

The focus of the BIB is on flexibility and empowering the end-systems, similar to the Internet and in contrast to the telephone network. But, the BIB does not address issues related to programming simplicity. The BIB views the use of CORBA [67] as a fundamental deviation from the traditional protocol approach, whereas, as will be seen later, the present thesis views CORBA (or whatever object request broker that is used) as a wrapper for the traditional protocol approach.

1.2.2 GSMP

The General Switch Management Protocol (GSMP) was developed as a protocol to externally control an ATM switch for the purpose of using ATM virtual circuits to transport connectionless IP flows [64]. Subsequently, GSMP was taken up at the IETF for standardization, and is currently in its late stages [38, 22]. The version being standardized by the IETF has evolved significantly from the original form of GSMP. GSMP at present is a protocol to externally control any label switch (which is a generalization of the concept of an ATM switch). Moreover, it has features to control a particular partition of a label switch and also to address quality of service parameters for call admission control under a specific list of quality of service models.
GSMP is a protocol, not a functional API. GSMP could be used for a possible implementation of a functional API, but other methods such as memory-mapped I/O and buses could also be used. In fact, most switches and routers today use the latter class of methods since that eliminates distributed processing between the switch and its controller.

GSMP is largely based on a series of service models. Thus, GSMP attempts to unify many existing service models by attempting to achieve their union (in the sense of union in C++). It does not attempt to present an underlying resource model that captures all of these various service models.

As a result of being based on service models, GSMP includes a number of operations that could be derived from a resource model. These operations could be eliminated if a resource model is present. GSMP includes operations such as reservations, switch partition handling and service class instantiations which could be derived from a resource model. Moreover, GSMP, being a standardization effort, does not focus on minimality and hence does not focus on capturing the bare essence of a label switch.

A resource model (called an ‘abstract resource model (ARM)’ in GSMP terminology) was introduced into GSMP by the author and colleagues as an extension. This resource model is based on the present thesis, and was originally presented in [2]. The inventors of GSMP also presented a resource model [63] subsequent to that presented in [2].
1.2.3 GMPLS

Generalized Multi-Protocol Label Switching (GMPLS)\(^1\) unifies various transport technologies by introducing the notion of a label [24]. A label is an identifier of an equivalence class of packets on a link, similar to how the VPI and VCI in ATM are identifiers of cells belonging to a virtual circuit. Whereas ATM has two levels of labels (VPI and VCI), GMPLS permits an arbitrary number of levels.

Viewed from a different angle, GMPLS is an architectural framework for creating virtual paths (known as label-switched paths) below the IP layer for the purpose of transporting IP packets. By the creation of label-switched paths, GMPLS brings greater control over the routes taken by IP packets. Moreover, the use of virtual paths reduces the complexity of forwarding packets in the network (since IP packet forwarding decisions are more complex than label switching). GMPLS is also used for creating virtual private networks (VPN).

The primary protocols of GMPLS are the Label-Distribution Protocol (LDP) and the Resource Reservation Protocol (RSVP), and variations and enhancements thereof (such as CR-LDP). LDP is used to distribute label information around the network so that the routers (called label-switched routers in GMPLS) can create label-switched paths. To support quality of service in label-switched paths, RSVP and variations on it can used.

GMPLS strives to enhance the connectionless packet forwarding of IP by extending

\(^1\)GMPLS is a generalization of MPLS. The review of this section applies to MPLS as well.
the latter's capabilities and performance. But, GMPLS lacks an explicit network resource model. Moreover, GMPLS makes no attempt to hierarchically structure its system of concepts. Due to the resultant vagueness, the boundaries of the implicit model of GMPLS and its hierarchical layers are unclear. Progress in knowledge is made by the development of precise and concise models, of which there is a void in the area of network control.

GMPLS does not capture quality of service, but relegates such issues to signaling protocols such as RSVP [41]. But RSVP does not provide a network resource model either. Hence, there is a void in terms of a general resource model for capturing quality of service.

GMPLS is based on protocols rather than functional APIs. Moreover, IP routing and the label distribution protocols of GMPLS are vertically integrated.

In summary, GMPLS attempts to create a common control plane, but does not address the issue of a common network resource model. There is a void in terms of such a common resource model. Once a common resource model is put forth, any control plane and quality of service model can be built on top of it. Even a connectionless network such as the Internet could be built on top of such a resource model. Herein lies the benefit of seeking such a resource model.

1.2.4 IEEE PIN

The IEEE working group on Programming Interfaces for Networks (PIN) [36] is focusing on developing open programming interfaces for controlling ATM switches and
IP routers.

The ATM sub-working group is in the process of standardizing a set of programming interfaces which grew out of the work on the Binding Interface Base and qGSMP [2]. The author of the present thesis is one of the primary contributors to the ATM sub-working group. The part on switch modeling in the present thesis is a continuation of the work of the author in the ATM sub-working group, but generalized to arbitrary label switches.

The IP working group is focusing on a set of programming interfaces for controlling IP routers. The interfaces are divided into a set of three layers. The lowest layer is an abstraction for developing resource and service abstractions. The middle layer abstracts router resources and the upper layer abstracts router services. The resource abstractions are similar to those found in the ATM working group, but adapted for IP routers.

The work of the IP working group has resulted in an initiative known as Forwarding and Control Element Separation (ForCES) [37], which has recently attained the working group status in the IETF. The goal of ForCES is to separate control functions such as the setting of routes in the routing table from IP packet forwarding functions. Moreover, ForCES considers the control of a cluster of routers. The initiative is still in its infancy.
1.2.5 Programmable Network Architectures

In this section, we discuss five programmable network architectures – xbind [52], Mobiware [8], Tempest [77], Genesis [16], and Sphere [73]. Active networks will be discussed separately in the next section.

*xbind* is a programmable network architecture built upon the BIB for controlling ATM networks. It is designed around a reference model known as the eXtended Reference Model (XRM), which has five planes – transport plane, connection management plane, resource management plane, management plane, and a network telebase. The separation between these planes is based on timescales. Orthogonal to these five planes, xbind also divides network control into three layers known as RGB. The R-layer (the “broadband network”) represents algorithms in data transport. The G-layer (the “multimedia network”) is obtained from the R-layer via a set of quality of service (QoS) abstractions. It represents algorithms in network control. The B-layer (the “services and applications network”) is obtained from the G-layer via a set of service abstractions. It represents application-level algorithms.

*Mobiware* is a programmable architecture that extends the xbind model of programmability to mobile networks. It allows for the programming of the control plane as well as the transport plane (similar to active networks). Mobiware supports programmable handoff and multiple QoS adaptation styles.

*Tempest* is a network control and management architecture which provides for the creation of virtual networks by logically partitioning ATM switches. The framework
allows different control algorithms to execute simultaneously on different partitions of an ATM switch. It provides two sets of control APIs, one for switch control and one for logically partitioning an ATM switch. It also provides a network builder that creates virtual networks. This work is extended in [44].

*Genesis* is an architecture for partitioning a router for creating virtual networks. It is the counterpart of Tempest in IP networks. Under this architecture, virtual networks can be created recursively. The Genesis life-cycle consists of the following phases: profiling, spawning, management, and architecting. Profiling refers to the specification of the virtual network that is to be created (spawned). A virtual network is specified in terms of its topology, resource requirements, user membership and security specifications. Routing protocols, QoS mechanisms and transport protocols can also be specified while profiling. The spawning phase creates the virtual network specified in the profiling phase. Once a virtual network has been spawned, its resources are managed (reallocated) dynamically via a renegotiation process. In this way, resources can be redistributed between different virtual networks. This reallocation takes place in a slow times scale (of the order of minutes or longer). Architecting refers to the analysis of the pros and cons of a virtual network’s design space.

*Sphere* is a binding model and middleware that provides building blocks from which routing protocols can be composed. Sphere’s binding model consists of the following: A database of the routing state; an update object that updates the database when new routing information is received; a set of algorithms that operate on the database to compute the forwarding table; an event generator that initiates routing
information transmission, database update, or path calculation; distributors which disseminate routing information; and packet processors which process packets before they are transmitted into, or arrive from, the network. Sphere introduces standard protocol-independent interfaces between these components, thus enabling code reuse and easing the development of new routing protocols.

1.2.6 Active Networks

Active networks [1] provide an architecture that includes a program execution environment analogous to the Java Virtual Machine [60]. These environments permit the upload of end-user programs for remote execution for arbitrary purposes subject to certain security and resource-usage restrictions. These purposes include the creation and removal of communication channels; forwarding, scheduling, and shaping of packets in the communication channels; accounting and billing. The programs could be uploaded as frequently as once per packet transmitted into the network.

In contrast, programmable networks as viewed in this thesis provide interfaces for accessing views of the network state for the sole purpose of creating and removing communication channels. The interfaces may be accessed remotely using an RPC mechanism, but programmability does not permit uploading of programs.

At a certain level of abstraction, one could argue that the programmable approach does provide an extremely simple execution environment in which the requests to the programming interfaces are the ‘programs’ which are uploaded. However, as phrased above, the above execution environment is extremely simple, providing for
only a handful of very restricted operations. The programmable approach focuses on providing interfaces (a trivial execution environment) for accessing the network state so that network operators could choose their control algorithms, whereas the active networks approach focuses on providing a general execution environment which could satisfy hitherto unimagined needs of users.

From another perspective, active networks empower end-users at the cost of network complexity, whereas programmable networks empower network programmers by promoting program simplicity.

1.2.7 Miscellaneous

In addition to the efforts of the IEEE PIN IP working group, a number of papers on programmable routers have appeared [14, 47, 76, 26]. These works attempt to program the data path of communication channels in addition to the creation and removal of these channels. They are similar to active networks, except that their programs are plugged in instead of being uploaded via IP packets. In the case of [76], the plugins are in hardware.

The paper [75] provides a concise API for manipulating a router’s forwarding table and its packet classifiers and schedulers. This API is similar to the work presented for ATM in [2, 36].
1.3 Structure of the Foundations

We begin this section by defining network control.

**Definition 1.1.** Network control is the task of creating and removing communication channels across the network on demand. Network signaling (also called signaling) is the task of exchanging messages between the distributed components of the network for the purpose of realizing network control.

Thus, communication channels, which are modeled in this thesis as graphs, are a key abstraction in network control. We focus on developing a framework that models network control as graph manipulation.

In order to lay the foundations of network programmability, we begin by identifying the basic abstractions that lead up to the abstraction of graphs. The most fundamental of these abstractions are the core network resources for data transport. These core network resources are the communication channel name-space (also known as label space), bandwidth, and buffer space. From these fundamental abstractions, two further abstractions can be derived. They are the network-element services and the network topology. The former is an abstraction of communication channels across a network element. The latter is an abstraction of the connectivity between network elements. These yield the abstraction of communication channels (network services) across the network.

The fundamental nature of these abstractions is seen as follows. As per definition, providing network services on demand is the goal of network control. The
network-element services are the building blocks from which network services are built, thus making network-element services vital abstractions for realizing network services. That the network topology is a vital abstraction follows from the following reasoning. A network transports data by routing the latter through one or more network elements. Such transport takes place via the links connecting the network elements. A sequence of network elements to route the data through can be found only if the network topology is known. That the core network resource model is a vital abstraction is seen as follows. Segments can be created only if they pass an admission control test to determine the availability of the core network resources. Hence, if we do not have a core network resource model, we cannot provide a model on which admission control tests can be devised.

Based on the above observations, we lay the foundations of network programmability according to the following structure.

**Definition 1.2.** Network programmability *is an approach to network control that entails*

1. A core network resource model,

2. A network-element service model,

3. Topology abstraction,

4. A network service model,

and APIs for
1. Control of the core network resources,

2. Creation and teardown of network-element services,

3. Reading the topology,

4. Creation and teardown of network services,

5. Message exchanges used for network signaling.

To enhance the performance of network control, the topology abstraction is typically augmented with some additional information such as the bandwidths of the links or their schedulable regions (defined in Appendix A). We shall use the term ‘network resource configuration’ to refer to the network topology and its augmentations.

Figure 1.1 depicts the system of APIs that result from the foundations given above. The network resource configuration API is for reading the topology and any additional information such as the link bandwidths or schedulable regions. The resource discovery APIs fall under the category of signaling APIs in that they are intended for message exchanges between distributed components for discovering the topology, schedulable regions, etc. All the other APIs shown in the figure are exactly as given in the definition. One of the main goals of this thesis is to demonstrate the relationships among these APIs.

This thesis attempts to develop the foundations of network programmability with conceptual understanding and simplicity as the primary goals. The foundations presented are very general in that connection-oriented and connectionless networks are
Figure 1.1: The hierarchy of APIs, shown by T-shaped symbols.

captured; packet-switched, circuit-switched, wavelength-switched, and fiber-switched networks are captured; quality of service is captured; virtually all networking technologies are covered, with the exception of multiple-access networks, e.g., CSMA/CD and mobile wireless networks. Multiple access networks differ from point-to-point networks in many ways. The present thesis does not address multiple access networks.
1.4 Benefits of the Foundations

The benefits of laying a set of foundations for network programmability are manifold.

First, by modeling communication channels as graphs and network control as graph manipulation, the foundations yield a conceptual framework of network control that is crisp, concise and hence easier to understand. A programming interface ideally must concisely and accurately capture the capabilities of the system it represents. Such a programming interface reflects a thorough understanding of the system on the part of the designer. The foundations provide such a set of interfaces. In this thesis, concision is demonstrated via the concept of minimality, which will be defined in a later chapter.

Second, the foundations yield a programming methodology that substantially simplifies the tasks of program conception and writing. By providing (i) APIs that capture the bare essence of the network resources, and (ii) APIs that reflect the bare essence of the resource discovery and graph building algorithms, the foundations make the software realizations of these algorithms follow the high-level logic closely. Consequently, the software is significantly shorter and much more readable, as will be shown in this thesis.

Third, since the foundations provide interfaces for accessing the core network resources, they permit the hardware (core network resources) to be separated from the software. This separation fosters competition in network control software products. In particular, these interfaces permit the writing of a variety of graph-building and
resource discovery algorithms, whereas the traditional network signaling protocols are algorithm-specific.

To summarize, the benefits of network programmability are

- Conceptual clarity
- Simplicity of programming
- Separation of network state from control algorithms

Of these benefits, only the last one has been recognized by the networking community. The first two items, namely the conceptual clarity and the simplicity of programming, are original contributions of this thesis.

1.5 Thesis Outline and Main Contributions

This thesis is organized as follows: Chapter 2 presents the core network resource model, network-element service model and the network service model. Chapter 3 presents programming interfaces of categories 1 – 4 in Definition 1.2, and introduces the concept of minimality of an API. Chapter 4 presents programming interfaces of category 5 for topology and other resource discovery. Chapter 5 discusses the benefits of the foundations of network programmability. The main part of the thesis concludes with Chapter 6, which summarizes the thesis and identifies areas for further research. The above is followed by an appendix which presents a simple and fast algorithm for estimating the schedulable region.
Parts of the present thesis have appeared in [2, 5, 3, 4, 54, 56, 55]. Other research conducted as part of the author's doctoral studies can be found in [72, 53, 57].

The main contributions of this thesis are as follows.

- A conceptual framework for laying the foundations of network programmability is introduced. (Section 1.3)

- A general network resource model that captures connection-oriented, connectionless, packet-switched, circuit-switched, wavelength-switched and fiber-switched networks, and quality of service is presented. A wide variety of quality of service models can be realized on top of this network resource model. IP routing protocols can also be realized on top of this model. (Sections 2.2, 2.7, 2.A.1, 2.A.2)

- Unification of connection-oriented and connectionless networking paradigms as a result of the above resource model.

- A network-element service model and a network service model. (Sections 2.3 and 2.4)

- A set of application programming interfaces (APIs) for use in communication graph building, applicable to a wide range of networks. These APIs can be used to realize arbitrary graph-building algorithms. (Sections 3.2, 3.A.1 and 3.A.2)

- Introduction and definition of the concept of minimality of an API. Demonstration that the APIs referred to above are minimal. (Sections 3.4 and 4.4)
• APIs for distributed interactions in resource discovery. (Section 4.3)

• Identification of three fundamental capabilities requisite of networks. Two of them are shown to be necessary and sufficient for resource discovery and graph building. The third is shown to be useful for performance optimization of the resource discovery algorithms. A fourth fundamental capability is identified as being possibly necessary for handling redundancy, but is left for future work. (Section 5.2.2)

• Demonstration that the programmable approach substantially simplifies software conceptualization and writing. A thin RPC(Remote Procedure Call)-like object request broker (ORB), viewed simply as an abstraction or wrapper for a protocol, is shown to be adequate for network programmability. (Section 5.3)

• An extremely simple hardware-implementable algorithm for estimating the schedulable region. The schedulable region is shown to be well approximated by a hyperplaner region. The algorithm is a proof-of-concept of the resource model of a multiplexer. (Appendix A)
2

MODELING TELECOMMUNICATION NETWORKS

2.1 PREAMBLE

In order to lay the foundations of network programmability, we begin with the description of a network model. In this first section, we discuss the subject being modeled. The next section presents the model.

Although telecommunication networks come in different flavors such as IP, ATM and optical networks, they all share a common thread, namely that of providing communication channels. The basic goal of a telecommunication network is to create and remove communication channels on demand. Thus, communication channels, modeled here simply as graphs, are the key abstractions of telecommunication networks. Even a connectionless network such as IP has the concept of graph building, although it may be created asynchronously with respect to the transport of data. The control of telecommunication networks that we will focus on in this thesis revolves around the creation and removal of graphs.

Broadly speaking, a telecommunication network consists of a collection of switches,
routers and end-stations interconnected by communication links. (The network could be wired or fixed wireless but we will not consider mobile networks.) The purpose of such a network is to provide means by which the end-stations could exchange information\textsuperscript{1} with each other. The most straightforward way to deliver traffic is to build direct communication links between all pairs of end-stations. However, this approach is too costly and cumbersome as is well-known. Hence, switches and routers are necessary components of networks.

For a network to be of use, it must provide means by which end-stations could request the transfer of data they desire to one or more other end-stations of their choice. Moreover, the network must guarantee the reliability and timeliness of the delivery of data\textsuperscript{2}. The guarantee could be deterministic or probabilistic.

From the viewpoint of the model to be presented in this chapter, routers and switches are placed under a common abstraction. Hence, a distinction between switches and routers will not be made. To make this clear, we shall call switches and routers generically as 'label switches' as in MPLS. The term 'network element' will also be used to refer to them. But, it should be kept in mind that from the viewpoint of the network resource model and its APIs, there is absolutely no difference between a switch and a router.

\textsuperscript{1}We shall use the terms 'information', 'data', and 'traffic' interchangeably. They are interpreted as a sequence of bits. No other significance is attached to these terms.

\textsuperscript{2}In some networks such as the Internet, these guarantees are kept very vague. However, there is still an implicit guarantee. For example, one expects that a \textit{typical} Web page will not take hours to download.
2.1.1 Transport

The resources for transporting data are label switches and links. They carry traffic from one end-station to another. The carriage of traffic requires bandwidth, a vital network resource. It also requires means for identification of traffic so that the latter could be delivered to the intended destination(s). Such identification requires labels to be associated with the traffic. A label as considered here is any identifier of traffic for the purpose of data delivery. This definition is more general than the one envisioned in GMPLS. For example, an IP address could be considered a label. Thus, labels also play a vital role in the delivery of network traffic. In packet-switched networks, buffering of packets at label switches has been identified as a means for reducing bandwidth usage. In summary, the resources for transport may be broadly categorized as

- Label space
- Bandwidth
- Buffer space

2.1.2 Signaling

Delivery of traffic in a network takes place via communication channels. A communication channel is modeled as a connected graph with adequate bandwidth, label space and buffer space resources that deliver a specific set of traffic streams from a source
or set of sources to a destination or set of destinations. Traffic will be delivered by the network only upon the creation of an appropriate communication channel. This is true irrespective of whether the network is connection-oriented or connectionless.

The difference between connection-oriented and connectionless networks comes about from the entity that triggers the graph creation. In the case of connection-oriented networks, the graph creation is requested by some entity other than the network control system. Typically, it is either a source or a destination vertex of the graph. In connectionless networks, the graph creation is initiated by the network control system (which is known as the routing system in IP). Unless the IP routing system (OSPF, BGP, or RIP) creates graphs, IP packets will not be delivered. Thus, graphs in connection-oriented networks are typically created “synchronously” (on demand) with respect to the sources of traffic, whereas graphs in connectionless networks are created “asynchronously” by the routing system.

Another difference between connection-oriented networks and connectionless networks arises from the fact that resources can be reserved on a per-traffic-stream basis in connection-oriented networks but they can be reserved only for certain aggregate traffic streams in connectionless networks. As no resources are reserved for individual streams in connectionless networks, traffic may encounter poor delivery in the form of large delays or loss of data. Applications that do not require low delays (compared to the time taken by information signals such as light or electricity to propagate through the network) may find certain amount of data loss tolerable since they can recover from the loss by retransmission of the data. Most of the traffic on the Internet today
is of this type, e.g., Web downloads and e-mail. Other traffic may not tolerate data delay and loss, the prime example being real-time video traffic. Such applications require graphs to be created on a per-stream basis.

The rest of this chapter is organized as follows. Section 2.2 presents the core network resource model, Section 2.3 presents the network-element service model, and Section 2.4 presents the network service model. Section 2.5 gives an abstraction for supporting scaling. Section 2.6 makes an observation about the development of the model. Section 2.7 illustrates the applicability of the model to a number of networking technologies. The chapter concludes with some remarks in Section 2.8.

2.2 THE CORE NETWORK RESOURCE MODEL

In this section, we present the core network resource model and abstractions thereon, leading to the modeling of communication channels as graphs (Figure 2.1). The first step is to define the resource model. The second step is to define abstractions known as segments and virtual links. The third step is the modeling of communication channels as graphs of segments and virtual links. Segments are the network-element services, and the communication channels, modeled as graphs, are the network services. The concept of modeling communication channels as graphs in the context of a telecommunication network is taken from [10].

Before presenting the resource model, we give a brief outline. In this model, the network is considered as a collection of label switches and end-stations interconnected
Figure 2.1: Abstractions leading to the modeling of communication channels as graphs.

Figure 2.2: A generic label switch.

by point-to-point links. A label switch is modeled as shown in Figure 2.2. It consists of input and output ports, a forwarding fabric that forwards data from input ports to output ports, and multiplexers that multiplex the data streams as they depart an output port. The forwarding table contains entries that instruct the forwarding fabric where to forward data arriving at the input ports.

The model is developed bottom-up, based on logical constructs. Thus, physically different devices may appear to be logically identical. Some of the terminology is borrowed from MPLS [40] and the ITU document on ATM definitions [43]. But, in general, the terms defined have been assigned generalized meanings to capture their
common essence. In particular, the term 'label' is used in a generalized sense, more
general than what is conceived in GMPLS [24]. For example, an IP address may also
be considered a label, as will be discussed later in the section on examples.

The model is presented as a series of definitions. The text in between definitions is
largely explanatory. It demonstrates how commonly-used technologies are captured
by the model. We do not define these technologies, e.g., circuit-switching, time-
division multiplexing (TDM) and wavelength-division multiplexing (WDM).

This section is divided into three parts. The first part defines some basic con-
cepts, the second part models the basic support necessary to transport data, and the
third part models the additional support necessary for guaranteeing quality of service
under asynchronous multiplexing\(^3\). The appendix given as Section 2.A.2 models an
extension for exploiting multiplexing gain under asynchronous multiplexing.

2.2.1 Preliminary Concepts

**Definition 2.1.** A label is a finite sequence of bits. Two labels are identical if and
only if they consist of the same sequence of bits.

**Definition 2.2.** A label vector is a sequence of zero or more labels. The number
of labels in a label vector will be called its dimension. The following operations are
defined on label vectors

\(^3\) For the lack of a better term, this term will be used to contrast with the other
two multiplexing technologies – time-division multiplexing and wavelength-division
multiplexing. The term refers to IP, ATM, frame relay, etc. We prefer not to use the
term ‘statistical multiplexing’ since we do not want to imply stochasticity.
• Two label vectors $X = < x^1, \ldots, x^m >$ and $Y = < y^1, \ldots, y^n >$ are identical if and only if $m = n$ and $x^i = y^i$ for $i = 1, \ldots, n$.

• The label vector $X$ includes label vector $Y$ if and only if $m \geq n$ and $x^i = y^i$ for $i = 1, \ldots, n$.

• The label vector $Y$ is a $k$-fold contraction of $X$ in case $n = m - k$, $k$ is a positive integer, and $x^{i+k} = y^i$ for all $i = 1, \ldots, n$.

• The label vector $Z = < z^1, \ldots, z^q >$ is a concatenation of $X$ with $Y$ in case $q = m + n$, $z^i = x^i$ for $i = 1, \ldots, m$ and $z^{i+m} = y^i$ for $i = 1, \ldots, n$.

**Definition 2.3.** A packet is a finite sequence of bytes. A label vector is associated with each packet. Some or all of the labels in this vector may be contained in the packet itself. The rest of the labels are represented by means external to the packet. The part of the packet that excludes the label vector is called the payload. Two packets are said to be identical if they consist of the exact same sequence of bytes.

In the context of packet-switching, the label vector of a packet is contained in the packet. In the context of circuit-switching, the label vector of a packet is taken to be the time-slot (channel in the time-domain) or hierarchy of time-slots occupied by the packet. In the context of wavelength-switching, each wavelength is considered a label. Hybrid cases will be illustrated in Section 2.7.11.

In the context of wavelength-switching and fiber (or port) switching, data streams are usually not partitioned into packets. However, for uniformity of description of the
ensuing model, we shall consider data streams in the context of wavelength-switching and fiber switching to be arbitrarily partitioned into packets. In particular, a packet could be taken to a byte. Such a partitioning does not affect the content of the model, but makes the presentation more concise by avoiding repetition.

We now introduce the term ‘interconnection device’ to capture any device that can receive packets and forward them to another device. Packets flow through a series of interconnection devices, starting from a source to a sink, both of which are general-purpose processors, as will be defined below. A network will then be defined as a certain collection of interconnection devices and processors, interconnected in a certain way. An interconnection device is very basic and hence described with appeal to intuition using terms such as “arrive” and “depart”.

An interconnection device is a logical construct which has one or more input terminals on which packets may arrive and one or more output terminals on which packets may depart. At any given instant, at most one packet may arrive at any given input terminal and at most one packet may depart from any given output terminal. If an output terminal of one interconnection device is connected to the input terminal of another interconnection device, then any packet departing the aforementioned output terminal will arrive immediately at the aforementioned input terminal. An input terminal may be connected to at most one output terminal, and vice versa.

A packet that departs from an output terminal must correspond one-to-one to a packet that previously arrived on an input terminal of the same interconnection
device. The departing packet corresponding to an arrived packet may differ from the latter only in the label vector; the payloads must be identical.

Interconnection devices may place an upper bound on the number of bytes that may arrive per unit time at any given input terminal. This upper bound is called the capacity of the given input terminal. Interconnection devices also place an upper bound on the number of bytes that may depart per unit time from any given output terminal. This upper bound is called the capacity of the given output terminal.

A packet-arrival at an input terminal of an interconnection device will also be referred to as a packet-arrival at the interconnection device. Similarly, a packet-departure from an output terminal of an interconnection device will also be referred to as a packet-departure from the interconnection device.

A multicast interconnection device is an interconnection device, except that departing packets need not correspond one-to-one with previously arrived packets. Instead, up to one departing packet per output terminal may correspond to a previously arrived packet.

The packet departing on an output terminal \( p \) corresponding to an arrived packet \( A \) will be denoted \( A_p \). If no packet departs output terminal \( p \) corresponding to the arrived packet \( A \), then \( A_p \) is said to be null.

Fragmentation and reassembly are not captured by the above description. These concepts affect graph building only via the second-order corrections to bandwidth usage computations for call admission control. In order to keep the main presentation simple, discussion of fragmentation and reassembly is deferred to the appendix of the
present chapter (Section 2.A.1).

2.2.2 A Generic Network Resource Model

In this subsection, we present a number of definitions leading up to the definition of generic label switches and networks. These networks are capable of transporting data but not necessarily guaranteeing QoS. Extensions for guaranteeing QoS will be provided in the next subsection.

Definition 2.4. A port is an interconnection device with one input terminal, one output terminal, and the following properties. The capacities of the input and output terminals are identical. Whenever a packet arrives at the input terminal, an identical packet departs from the output terminal instantaneously.

In the following, we shall use the term ‘packet framing technology’ to identify framing technologies, e.g., Ethernet, ATM, SONET, IP, WDM. Each framing technology is given a code to identify it.

Definition 2.5. The type vector of a port is a vector of packet-framing-technology identifiers. A label vector $X$ is said to match a type vector $T$ if the dimension of the former does not exceed that of the latter, and the $i$-th element of $X$ is a valid label for the packet framing technology given by the $i$-th element of $T$ ($i = 1, \ldots, \text{dimension}(X)$).

A particular type of interconnection device, known as a forwarding fabric, has an associated table known as the forwarding table. The behavior of the forwarding fabric
depends on the content of its forwarding table. Thus, we first define the forwarding table, and then the forwarding fabric. Since we have not yet defined a forwarding fabric, we shall refer to it generically as an interconnection device.

**Definition 2.6.** A *forwarding table* of an interconnection device is a table of entries of the form

- \((p_1, X_1) \rightarrow (p_2, X_2, ID)\)

where \(p_1\) is an input terminal of the interconnection device, \(p_2\) is an output terminal of the interconnection device, \(X_1\) and \(X_2\) are label vectors, and \(ID\) is an identifier that is passed on with packets for interpretation by other interconnection devices (which are multiplexers defined in Definition 2.9). Entries in the table may not differ only in the buffer \(ID\).

The pair \((p_1, X_1)\) and the triple \((p_2, X_2, ID)\) will be called the input part and output part, respectively, of the forwarding table entry. The label vectors \(X_1\) and \(X_2\) will be called the input label vector and output label vector, respectively.

**Definition 2.7.** If \(p\) is an input terminal of an interconnection device and \(X\) is a label vector, then

- An entry \((p_1, X_1) \rightarrow (p_2, X_2, ID)\) in the forwarding table is said to match the pair \((p, X)\) in case \(p = p_1\) and \(X\) includes \(X_1\).

- The collection of all entries in the forwarding table that match the pair \((p, X)\) is called the matching set of \((p, X)\).
• Denote by $\hat{X}(p, X)$ the input label vector with the largest dimension in the matching set of $(p, X)$. The longest matching set is the subset of the matching set obtained by taking only those entries whose input label vector is identical to $\hat{X}(p, X)$.

**Definition 2.8.** A forwarding fabric is a multicast interconnection device along with a forwarding table, and satisfies the following.

• The number of input terminals equals the number of output terminals.

• If a packet with label vector $X$ arrives on an input terminal $p$, then corresponding to each entry $(p_1, X_1) \rightarrow (p_2, X_2, ID)$ in the longest matching set of $(p, X)$, a packet with identical payload will depart on output terminal $p_2$. For each such entry, the label vector of the departing packet is the concatenation of $X_2$ with the $m$-fold contraction of $X$, where $m$ is the dimension of $X_1$. The ID is passed along with the packet for interpretation by a multiplexer (Definition 2.9).

The above model of the forwarding fabric is based on the longest-prefix-match paradigm. This paradigm was chosen for the sake of generality. However, individual forwarding technologies may use special cases of it, such as the exact-match paradigm. The model does not place any restrictions on the use of special cases.

The above model does not capture fragmentation and reassembly of packets. An extension that captures this is given in the appendix to this chapter (Section 2.A.1).
**Definition 2.9.** A multiplexer is an interconnection device with one input terminal, one output terminal, a set of first-in-first-out buffers, a buffer manager, a scheduler, and the following properties (Figure 2.3).

- Whenever a packet arrives at the multiplexer, the buffer manager determines whether to discard the packet by executing a buffer management policy. If the packet is not discarded, then it is put at the back of the buffer corresponding to the ID that was assigned to the packet by the forwarding fabric (see Definition 2.8).

- The scheduler schedules one packet at a time from the set of buffers to depart the multiplexer.

In the context of time-division multiplexing, the multiplexer models a time-slot interchanger. Each buffer corresponds to a time slot. The time-slots are scheduled in round-robin or weighted round-robin fashion. Weighted round-robin is used if multiple time-slots are concatenated into one, as in STS-3c.

In the context of wavelength-division multiplexing, the multiplexer symbolically represents wavelength-division multiplexing. Each buffer symbolically represents a wavelength. The weight assigned to a buffer is proportional to the bandwidth associated with the corresponding wavelength. The buffer manager and scheduler are void.

**Definition 2.10.** A generic label switch is a multicast interconnection device consisting of (i) a set of ports (called input ports), (ii) a second set of ports (called output
ports), (iii) a forwarding fabric, (iv) a set of multiplexers, and the following properties.

- The cardinalities of the sets (i), (ii), and (iv) must be identical, and they must equal the number of input terminals on the forwarding fabric.

- The interconnection devices in the sets (i)–(iv) are interconnected as shown in Figure 2.2 (which shows an example with three input and output ports).

- For every forwarding table entry \((p_1, X_1) \rightarrow (p_2, X_2, ID)\), \(X_1\) must match the type vector of the input port to which the input terminal \(p_1\) is attached, and \(X_2\) must match the type vector of the output port that connects to output terminal \(p_2\) (via a multiplexer).

- The multiplexer connected to an output port must be capable of multiplexing packets whose framing technology is given by the first element of the output port's type vector. If this framing technology requires asynchronous multiplexing, then the multiplexer is constrained to have only one buffer. In this case, the buffer
management policy discards an arrived packet if and only if the buffer is full.

- The capacity of the input terminal of the multiplexer must equal that of the output terminal of the forwarding fabric that it connects to. The capacity of the output terminal of the multiplexer must equal that of the port it connects to. The capacity of an input terminal of the forwarding fabric must equal that of the input port it connects to.

A generic label switch is capable of executing the following requests from an entity known as a switch control processor, which is defined later.

- Retrieve the identities of the input and output ports, and their capacities, port type vectors, and label space ranges;

- Write and remove entries in the forwarding table.

- Clear all entries in the forwarding table.

- Read the weights of the buffers of any multiplexers that are based on time-division multiplexing or wavelength-division multiplexing.

The ID of the buffer for asynchronous multiplexed ports is taken to be well-known. The label space range of a port refers to the collection of label vectors that are supported by the port. For example, an ATM port might support only VPIs in the range 0–5 and VCIs in the range 0–1023.

The definition above declares one-to-one mappings between (i) input ports and input terminals of the forwarding fabric, (ii) output terminals of the forwarding fabric
and the multiplexers, and (iii) the multiplexers and output ports. This correspondence will often be used to identify elements of the above three sets. Moreover, since the number of input ports equals the number of output ports, it is customary to pair every output port with an input port. Both ports of such a pair will be given the same identifier, distinguished only by the fact that one is an input port and the other is an output port. A pair of such ports will be called an input-output port pair.

The model presented above was restricted to output buffering, due to its simplicity and wide commercial use. Input buffering is much more complex to model in a general way, and is not as well-understood as output buffering [49, 68].

A generic label switch is depicted in Figure 2.2. Subsequent sections of this chapter will introduce extensions to the generic label switch. We shall use the term 'label switch' to refer to a genetic label switch and its extensions.

**Definition 2.11.** A **communication channel** is a multicast interconnection device with one input terminal, one or more output terminals, and the following property. If a packet $A$ arrives at the communication channel before another packet $B$, and neither $A_p$ nor $B_p$ is null, then $A_p$ will depart before $B_p$, where $p$ is any output terminal of the communication channel. A **unicast communication channel** is a communication channel with only one output terminal. A **multicast communication channel** is a communication channel with more than one output terminal.

**Definition 2.12.** A **link** is a unicast communication channel with the following properties. Whenever a packet arrives at the input terminal, an identical packet departs
from the output terminal after a time-lag, which is fixed for each link. Sometimes, a packet may not depart; the probability of that event is known as the loss rate of the link. A link may connect only to input and output ports. If the input and output terminals of a link are connected to ports $p_1$ and $p_2$, then the first $n$ components of the port-type vectors of $p_1$ and $p_2$ must be identical, where $n$ is the minimum of the dimensions of the two port-type vectors. Moreover, the capacity of the two ports must be identical.

**Definition 2.13.** An **end-station** is a logical construct with one input port and one output port and can execute programs written in a general-purpose programming language. The input port of the end-station may be connected to the output terminal of a link, and the output port of the end-station may be connected to the input terminal of a link. The content of any packet that arrives on the input port of an end-station is readable by the program executing inside the end-station. The program may send packets on the output port of the end-station. Such packets will arrive instantaneously at the input terminal of the link that connects to the output port of the end-station.

Denote label switches and end-stations generically by the term *node*. A link is said to uni-directionally connect node $A$ to node $B$ if the input terminal of the link connects to the output terminal of an output port of $A$, and the output terminal of the link connects to the input terminal of an input port of $B$. In such a case, the link is said to connect the aforesaid output port to the aforesaid input port. Two nodes $A$ and $B$ are said to be connected if there is a pair of links $\{l_1, l_2\}$, an input-output port
pair $p_A$ of $A$, and an input-output port pair $p_B$ of $B$ such that the link $l_1$ connects the output port $p_A$ to the input port $p_B$, and the link $l_2$ connects the output port $p_B$ to the input port $p_A$. A pair of links that connects two nodes is called a link-pair. Two connected nodes will be called neighbors.

**Definition 2.14.** A switch control processor (SCP) is an end-station which has means of requesting one label switch to perform any of the instructions in the latter’s repertoire (see Definitions 2.10, 2.20, and 2.29).

A label switch that receives requests from a switch control processor is said to be controlled by that switch control processor. Two switch control processors are said to be neighbors if they each control one of a pair of label switches that are neighbors.

**Definition 2.15.** A network is a collection of label switches, switch control processors, end-stations and links such that

- Every label switch is controlled by a neighboring switch control processor;
- Every link belongs to a link-pair.

A label switch’s port that connects to the label switch’s switch control processor is called the control port of the label switch. This terminology will be used even if the switch control processor uses other means of controlling its label switch, e.g., a bus. Note that even in this case, the switch control processor is connected to the label switch on its control port. The control port is used for communicating with other SCPs (discussed in Chapter 4).
2.2.3 Extension for Quality of Service

The extensions presented in this subsection are interesting only for asynchronous multiplexing. For other multiplexing technologies such as TDM and WDM, the extensions are superfluous.

Consider the arrival of packets at the input terminal of an interconnection device. They form an arrival process [17]. These arrival processes can be split into a number of sub-arrival processes by filtering packets based on their label vectors.

**Definition 2.16.** The sub-arrival process corresponding to a specific label vector on a particular input terminal of a (multicast) interconnection device will be called the flow corresponding to that label vector on that terminal.

**Definition 2.17.** The peak rate of a flow is the least upper bound of the number bytes that may arrive in the flow during any interval of some specified short duration. If packets are of fixed length, then the peak rate may be defined as the reciprocal of the smallest inter-arrival time [17] between packets of that flow. The average rate of a flow is the average number of bytes arriving per unit time in that flow. The maximum burst size of a flow is

\[ \inf\{r > 0|\forall t > 0, A_t \leq st + r\}, \]

where \( A_t \) is the number of bytes arriving in the flow during the time interval \([0,t]\), and \( s \) is the average rate of the flow. The stochastic process \( \{A_t\} \) is called the arrival process of the flow.
Definition 2.18. A traffic contract is a set of upper bounds on some specified statistics of the arrival process of a flow. A traffic contract must contain an upper bound on the peak rate.

A traffic contract $t^1$ is said to be tighter than $t^2$ in case any flow that satisfies the latter also satisfies the former. Then, the latter is said to be weaker than the former. The minimum of the traffic contracts $t^1, t^2, \ldots, t^n$ is a traffic contract $t$ such that there is no other traffic contract that is tighter than $t$ but weaker than all of $t^1, t^2, \ldots, t^n$. Since traffic contracts are specified in terms of upper bounds, such a minimum always exists and is unique.

Definition 2.19. A policer is an interconnection device with one input terminal, one output terminal, and a table which consists of entries of the type

- (label vector, traffic contract)

If $(X_1, TC_1)$ and $(X_2, TC_2)$ are two distinct entries in a policing table, then $X_1$ may not be included in $X_2$, and $X_2$ may not be included in $X_1$. For every flow listed in the table, the policer maintains a running list of statistics identified in the traffic contract. Whenever a packet arrives at the input terminal, the policer checks whether the label vector $X$ of the packet is listed on the table. If not, it checks whether there is an entry in the policer table such that the label vector of this entry is included in $X$. If no such entry is found, then the policer makes an identical packet depart on its output terminal. If the check succeeds, it updates the statistics for the found entry and checks
whether the traffic contract is violated. If the contract is violated, no packet departs. Otherwise, an identical packet departs from the output terminal.

A rate shaper is a generalization of a policer, where packets may be delayed instead of being dropped when the packet stream violates the traffic contract. Rate shapers provide a more elaborate way of providing quality of service. In this thesis, we do not consider rate shapers.

**Definition 2.20.** A QoS-enabled label switch extends a generic label switch with the following two modifications. First, a QoS-enabled label switch has a set of policers whose number equals the number of input ports. The output terminal of each input port connects to the input terminal of a policer. The output terminal of each policer connects to an input terminal of the forwarding fabric. All other connections between interconnection devices are the same as given in Definition 2.10. Second, a QoS-enabled label switch can respond to the following requests in addition to those given in Definition 2.10.

- Retrieve a list of supported policing parameter types.

- Set and remove policer table entries.

- Clear all entries in the policer tables.

A QoS-enabled label switch is depicted in Figure 2.4.

**Definition 2.21.** A QoS-enabled network is a network in which all label switches are QoS-enabled.
Definition 2.22. A quality of service (QoS) constraint on a flow of packets departing a given output terminal of a communication channel is a set of bounds on certain specified statistics of the time lag between the arrival of a packet at the input terminal and the departure of the corresponding packet from the output terminal in concern. If no packet departs corresponding to a particular arrived packet then the time lag is taken to be infinite. A communication channel is said to provide quality of service if each packet flow departing the communication channel satisfies some specified quality of service constraint.

2.3 THE NETWORK-ELEMENT SERVICES

2.3.1 Segments

Given a label switch, suppose that a packet \( P \) with label vector \( X \) arrives at an input port \( p \). Then, corresponding to every entry in the longest matching set of \((p, X)\), at most one packet departs from an output port of the label switch. The ports on which these packets depart are given by the output ports in the entries of
the longest matching set. The departing packets will have payloads identical to that of \( P \). Due to the FIFO nature of buffers in the multiplexers of label switches, the condition given in Definition 2.11 holds true. Thus, a label switch may be viewed as containing a communication channel, known as a **segment**. The input terminal of the segment is \( p \). For simplicity of description, assume that no two entries in the longest matching set contain the same output terminal. Then, the output terminals of the segment are the output terminals of the entries in the longest matching set. If the cardinality of the latter set is one, then the communication channel is known as a **unicast segment**. If the cardinality is zero, then the segment is **null**. Otherwise, the communication channel is known as a **multicast segment** with \( k \) branches, where \( k \) is the cardinality of the longest matching set. Figure 2.5 depicts a multicast segment with three branches.

If the label switch is QoS-enabled and the policer connected to input port \( p \) has an entry for the label vector \( \tilde{X}(p, X) \) (see Definition 2.7), then the segment is said to be **policed**. With each branch of a segment, a set of quality of service constraints is associated. QoS constraints can be guaranteed via call admission control (which is discussed below) and policing.
The collection consisting of the input terminal and the set of output terminals of a segment is called the \textit{geometry} of the segment. The quality of service constraints associated with each branch of the segment is called the \textit{color} of the branch. The \textit{thickness} of the segment, which represents the traffic contract of the segment, will be defined shortly.

The collection of segments at a QoS-enabled label switch is said to be \textit{admissible} if the label switch can provide quality of service to all of those segments simultaneously. Consider a specific collection of segments, numbered $1, 2, \ldots, N$. One way to ensure that the collection is admissible is to ensure that for every output port $p$ of the label switch,

$$\sum_{i=1}^{N} n_{pi} c_i \leq C_p,$$  \hspace{1cm} (2.2)

where $C_p$ is the capacity of output port $p$, $c_i$ is the peak rate associated with segment $i$, and $n_{pi}$ is the number of output terminals of the segment that are associated with output port $p$ ($n_{pi} = 0 \text{ or } 1$). The quantity $c_i$ is called the \textit{thickness} of the segment $i$. Equation (2.2) represents the well-known (and most trivial) accounting rule known as \textit{peak-rate allocation}. Due to its linearity, any rule of the form of Equation (2.2) for ensuring the admissibility of a collection of segments, but possibly with different thickness definition, is called an \textit{accounting rule}. Peak-rate allocation does not exploit multiplexing gain. An accounting rule that exploits multiplexing gain can be obtained from the concept of equivalent bandwidths [33] or the more general concept of schedulable regions (Appendix A). Such an accounting rule is presented
as Appendix 2.A.2 of this chapter.

The network-element service model we use in this thesis is the model of a label switch in which the label switch is viewed as a black box capable of creating and removing segments on demand. Moreover, if the label switch is QoS-enabled, then it is capable of creating and removing policed segments.

2.3.2 Virtual Links

Whenever a packet arrives at the input terminal of a link, it departs unmodified from the output terminal, but after a time-lag. Thus, a link may be viewed as a collection of unicast communication channels, one per label vector. Such a unicast communication channel will be called a virtual link.

Suppose that the output terminal of a link \( l \) is connected to an input port \( p \) of a label switch, and \( X \) and \( Y \) are two label vectors such that \( Y \) includes \( X \). Then the input terminal of the segment represented by the longest matching set of \( (p, X) \) connects to the output terminal of the virtual link associated with label vector \( Y \) on link \( l \), in the following sense (see Figure 2.6). If a packet with label vector \( V \) that includes \( Y \) departs the output terminal of the virtual link, then that packet will arrive at the input terminal of the segment.

Similarly, suppose that \( Z \) is a label vector that includes \( X_2 \), and that the input terminal of a link \( l' \) connects to the output port \( p_2 \), where \( X_2 \) and \( p_2 \) belong to the output part of some entry in the longest matching set of \( (p, X) \). Then, the segment branch corresponding to the aforementioned forwarding table entry connects to the
Figure 2.6: Virtual links connecting with a segment.

input terminal of the virtual link associated with label vector $Z$ in the link $l'$: If the label vector $W$ of any packet departing the segment branch includes $Z$, then that packet will arrive at the input terminal of the virtual link.

From the definition above, it is clear that multiple virtual links can connect to the same terminal of a segment. Thus, we make an exception to the rule.

2.4 THE NETWORK SERVICES

2.4.1 Topology

A network may be modeled as a graph by considering the nodes of the network to be vertices and the links of the network to be edges. Denote the set of nodes in the network by $A$ and the set of links by $L$. The topology of the network is the graph $(A, L)$. The topology is not a network service, but it is presented here as the topology is the basis for defining network services. An extension to the topology for capturing
bandwidth and buffer space resources is given in Section 2.A.3.

2.4.2 Network Communication Graphs

We now define the concept of a network communication graph. Consider a list of segments and virtual links, organized in a directed rooted tree as follows. The vertices of the tree are non-null segments. The edges of the tree are virtual links. For every vertex \( v \) that is not the root, denote the incoming edge of the vertex by \( e_i(v) \). For every vertex \( v \) that is not a leaf, denote the set of outgoing edges of the vertex by \( E_o(v) \). Observe that \( v \) represents a non-null segment, \( e_i(v) \) represents a virtual link, and \( E_o(v) \) represents a set of virtual links.

A network communication graph is a tree organized as above and satisfying the following conditions.

- For every vertex \( v \) that is not the root, the input terminal of \( v \) must connect to the output terminal of \( e_i(v) \) (see Section 2.3.2);

- For every vertex \( v \) that is not a leaf, every output terminal of \( v \) must connect to the input terminal of a virtual link in the set \( E_o(v) \) (see Section 2.3.2).

The path from the root to each segment branch of each leaf along the tree is called a branch of the network communication graph. A network communication graph is said to be unicast if it has only one branch. Otherwise, it said to be multicast. When used in the context of 'graph building', the term 'graph' means a network communication graph.
The traffic contract of a network communication graph is the minimum of the traffic contracts of the constituent segments (see discussion after Definition 2.18). The thickness of a network communication graph is the minimum of the thicknesses of the branches of its constituent segments. The geometry of the network communication graph is the tree discussed above.

Observe that as traffic traverses the segments of a network communication graph, its statistical properties may get transformed due to queueing at the label switches. As a result, a flow that satisfied a traffic contract as it entered a network communication graph may not satisfy the contract as it enters some segments in the middle of the network communication graph. This situation results in exceedingly complex accounting rules for call admission control. One way to avoid such complexity is to avoid queue buildup at label switches, if QoS is to be guaranteed. In other words, the
quality of service constraints must be kept such that the delays and losses of a flow as it traverses a segment are negligible. However, we do not preclude other approaches.

The above definition of a network communication graph is based on a tree structure. The definition could be augmented to be based on a directed acyclic connected graph (of which a directed rooted tree is a particular case). However, in the present thesis, we shall limit our attention to the definition based on trees.

The network service model we use in this thesis is the model of a network in which the network is viewed as a black box capable of creating and removing network communication graphs on demand. Moreover, if the network is QoS-enabled, then it is capable of creating and removing network communication graphs with QoS, i.e., network communication graphs whose constituent segments are subject to admission control and policing.

2.4.3 Graph Embedding

Consider two network communication graphs $G$ and $G'$ such that $G'$ is a sub-graph of $G$, and the root and leaves of $G'$ are different from those of $G$. (see Figure 2.8). Let the sequence of vertices $v_1, \ldots, v_n$ represent an arbitrary path in the graph $G'$ from its root to a leaf. Let $v_0$ be the parent of $v_1$ in the graph $G$, and let $v_{n+1}$ be a child of $v_n$ in the graph $G$. Then, $v_0, v_1, \ldots, v_n, v_{n+1}$ is a path in $G$. Denote the forwarding table entry corresponding to this path in the graph $G$ at the label switch represented by vertex $v_i$ by $(p_i^1, X^i_1) \rightarrow (p_i^2, X^i_2, ID^i)$. Define $\Delta(i) = dim(X^i_2) - dim(X^i_1)$ for $i = 0, \ldots, n+1$, where $dim(.)$ denotes the dimension of a label vector. Suppose that
Figure 2.8: Graph embedding. $G$ is the entire graph shown and $G'$ is the sub-graph indicated via the oval.

the following holds true for all such paths $v_0, v_1, \ldots, v_n, v_{n+1}$.

$$
\sum_{i=0}^{k} \Delta(i) \geq \begin{cases} 
\Delta(0) > 0, & k = 1, \ldots, n; \\
0, & k = n + 1.
\end{cases}
$$

(2.3)

Then, the graph $G$ is said to be embedded in the graph $G'$. Equation (2.3) is a simple calculus on the label space for guaranteeing that whenever a packet $P$ arrives at the input terminal of the network communication graph $G'$, all of the corresponding packets that depart any of the output terminals of $G'$ will be identical to $P$ except possibly in the first $\Delta(0)$ labels. For example, if an ATM virtual channel is embedded in a virtual path, then $\Delta(0) = 1$, since $X_1^0$ represents a VCI (dimension 1) and $X_2^0$ represents a VPI-VCI pair (dimension 2) (See Section 2.7.1).

The above definition can be applied recursively. A graph can be embedded in another graph which in turn can be embedded in yet another graph, and so forth.

If a graph $G_3$ embeds two graphs $G_1$ and $G_2$, then the traffic contract of $G_3$ must be such that if $f_1$ and $f_2$ are any two flows that satisfy the traffic contracts of $G_1$ and $G_2$ respectively, then the superposition of the two flows (given by the superposition
of the corresponding arrival processes [17]) must satisfy the traffic contract of $G_3$.

Figure 2.9 illustrates the concept of embedding more intuitively as a pipe. $G$ is the graph shown on the top. $G'$ is the sub-graph inside the oval.

2.5 **SCALING**

Due to the complexity of controlling large networks, networks often need to be partitioned hierarchically as the number of label switches increases. The present section defines such a partitioning by introducing the concept of a domain. A domain may
be thought of as a 'contiguous' portion of a network that behaves like a giant label switch. Domains are interconnected by 'links' which are abstractions of the links that interconnect label switches. In this way, a network of label switches may be abstracted as a network of domains, where each domain is similar to a label switch. Moreover, a network communication graph that traverses only nodes inside a particular domain may be thought of as a 'segment' of that domain. We now begin the formal definition.

2.5.1 Domain Resource Model

Let $A = \{a_1, \ldots, a_n\}$ be the set of nodes in a network, and $\mathcal{A} = \{A_1, \ldots, A_N\}$ where each of the $A_i = \{A_{i1}, \ldots, A_{im}\}$ is a partition of the set $A$, i.e., $A$ is the disjoint union of $A_{i1}, \ldots, A_{im}$ for every $i = 1, \ldots, N$. Suppose further that the following hold true.

- If $a_r, a_s \in A_{ij}$ for some $r, s, i$ and $j$, then there is a path in the topology of the network between $a_r$ and $a_s$ such that the path does not traverse any vertices other than those representing the nodes of $A_{ij}$.

- If $1 \leq i < j \leq N$, then every element $A_{ik}$ of $A_i$ is a subset of some element $A_{ja(i,j,k)}$ in $A_j$, where $\alpha(i, j, k)$ is an integer in the range $\{1, \ldots, n_j\}$.

Then, each of the $A_{ij}$ is called a domain of the network.

Define $A_{0i} = \{a_i\}$ for $i = 1, \ldots, n$. If $0 \leq i < j \leq N$, then domain $A_{ja(i,j,k)}$ is called an ancestor of $A_{ik}$, and $A_{ik}$ is called a descendant of $A_{ja(i,j,k)}$; if $j = i + 1$, then $A_{ja(i,j,k)}$ is called the parent of $A_{ik}$, and $A_{ik}$ is called a child of $A_{ja(i,j,k)}$. It is clear
from the definition that the domains form a tree. Typically, \( A_N = \{ A \} \), and is called the root of the domain tree. \( A_i \) is called the set of \( i^{th} \)-level domains. The domains in \( A_1 \) are called the lowest-level domains. The complete collection of domains \( A \) is called the domain hierarchy. Different domain hierarchies can be defined on the same network. Thus, \( A \) is not unique.

A link \( l \) between two label switches \( a_r \) and \( a_s \) is said to belong to domain \( A_{ij} \) if the following hold true.

- \( a_r, a_s \in A_{ij} \), and

- No child of \( A_{ij} \) contains both \( a_r \) and \( a_s \).

The mappings

\[
\begin{align*}
(l, A_{ij}) &\rightarrow A_{i-1, \beta(i,r)} \\
(l, A_{ij}) &\rightarrow A_{i-1, \beta(i,s)}
\end{align*}
\]  

(2.4)

are called the child-mappings of the link, where \( A_{i-1, \beta(i,r)} \) is the child of \( A_{ij} \) to which \( a_r \) belongs, and \( A_{i-1, \beta(i,s)} \) is the child of \( A_{ij} \) to which \( a_s \) belongs. The pair \( (l, A_{i-1, \beta(i,r)}) \) is called a port of the domain \( A_{i-1, \beta(i,r)} \), and the pair \( (l, A_{i-1, \beta(i,s)}) \) is called a port of the domain \( A_{i-1, \beta(i,s)} \). The link \( l \) is said to connect the domains \( A_{i-1, \beta(i,r)} \) and \( A_{i-1, \beta(i,s)} \) on the ports \( (l, A_{i-1, \beta(i,r)}) \) and \( (l, A_{i-1, \beta(i,s)}) \).

Figure 2.10 illustrates the concept of domains. The eight label switches are partitioned into three domains \( A_{11}, A_{12} \) and \( A_{13} \), shown in broken circles. The top two domains \( A_{11} \) and \( A_{12} \) in the picture are further grouped together into another domain \( A_{21} \). The bottom domain \( A_{13} \) is grouped by itself into another domain \( A_{22} \). The links
of the domains $A_{11}, A_{12}$ and $A_{13}$ are shown in Figure 2.11, the links of the domains $A_{21}$ and $A_{22}$ are shown in Figure 2.12, and the links of the root domain are shown in Figure 2.13.

2.5.2 Domain Service Model

Consider a network communication graph $G$ whose vertices represent segments on label switches in the domain $A_{ij}$. Suppose that the input terminal of the segment represented by the root vertex and the output terminals of the segments represented by the leaf vertices of $G$ connect to links that belong to the parent domain of $A_{ij}$, i.e., these links connect the domain $A_{ij}$ to other domains. Then, $G$ is a communication channel whose terminals are the ports of the domain $A_{ij}$. We call $G$ a domain segment of domain $A_{ij}$. If the constituent segments of $G$ are policed, then the
Figure 2.11: Links of the domains $A_{11}, A_{12}$ and $A_{13}$.

Figure 2.12: Links of the domains $A_{21}$ and $A_{22}$. 
domain segment is said be policed.

The domain service model we use in this thesis is the model of a domain in which the domain is viewed as a black box capable of creating and removing domain segments on demand. Moreover, if the label switches in the domain are QoS-enabled, then the domain is capable of creating and removing policed domain segments.

The topology of a domain is the topology of the entire network but restricted to those vertices belonging to that domain, and edges connecting the aforementioned vertices. The partial topology of a domain $A_{ij}$ is the graph $(\tilde{A}_{ij}, L_{ij})$, where $\tilde{A}_{ij}$ is the set of children of the domain $A_{ij}$, and $L_{ij}$ is the set of links belonging to the domain $A_{ij}$ (see the previous subsection). The partial topology of a lowest-level domain coincides with its topology.
2.6 **Note on the Development of the Model**

The field of communication networking has grown fairly rich, but lacks the clarity of understanding found in other disciplines of computer science. The reason for this is in large part due to the fact that the fundamentals are not organized into an appropriate hierarchy. To use the terminology of object-oriented programming, the hierarchy of concepts has been flattened out. The result of this is that there is no clear picture as to what are the definitions and axioms, and what are the derived concepts. At least such a picture has not been made explicit. As a result of this, literature on communication networking tends to be unduly bulky.

The organization of this thesis is based on the hierarchy given in Definition 1.2. The following are some of the concepts and abstractions that were weeded out of the network model as they belong to higher layers. The list is clearly not exhaustive. It is included merely for illustrative purposes.

- Addressing (IP addresses, ATM addresses, *etc*)
- Signaling (graph building)
- Routing (resource discovery)
- Control channel management
- Label binding, assignment and distribution
- Forwarding equivalency classes
2.7 Generality of the Model

The model presented in this chapter is fairly general, capturing both connection-oriented and connectionless networks, whether they be packet-switched, circuit-switched or wavelength-switched. In particular, the model captures IP routed networks, MPLS networks, ATM networks, SONET networks, wavelength-switched networks, and hybrids thereof. The model is also compact. It captures only the bare minimum support necessary for guaranteeing quality of service.

Support for exploiting multiplexing gain is given as an extension to the basic model (see Section 2.A.2). Using this extension, various QoS models can be realized. This is because the multiplexer model is fairly general. QoS models such as the leaky bucket and equivalent bandwidths can be realized on top of the multiplexer model (see [30] for example). Per-stream queueing as well as aggregate-stream queueing can be accomplished as discussed in Section 3.A.2 of the next chapter. Measurement-based models such as the schedulable region can also be realized (Appendix A).

In order for a network to be adequately modeled by the present chapter, the definitions given in Section 2.2 must be satisfied, and all the major features of the network must be captured by these definitions, i.e.,

- The packets, frames or data streams of the network must be identified by a suitable label vector.

- The packet/frame/data stream forwarding algorithm used by the network must coincide with, or be a special case of, the algorithm presented in Definition 2.8.
• The definition of multiplexers must be shown to be applicable to the network.

• The input port policers must be shown to be applicable to the network in the case of asynchronous multiplexing.

In this section, we illustrate the applicability of the model to numerous types of networks.

As a general note, the model of this chapter does not capture specific flags that may be present in packet headers of the various technologies, e.g., the cell loss priority (CLP) flag in ATM cells and the time-to-live (TTL) field in IP packets. These flags, which vary from technology to technology, capture second-order phenomena of networking. They are less fundamental than the concepts presented in this chapter. A standardization body developing standards based on this thesis may wish to consider such flags, but in a work aimed at conceptual understanding, such second order effects are best omitted.

2.7.1 ATM Networks

By the term 'ATM network', we refer only to the transport-aspects of ATM. Labels in ATM networks are Virtual Channel Identifiers (VCI) and Virtual Path Identifiers (VPI). Every ATM packet (cell) contains exactly two labels – the VPI and VCI. The cell switching algorithm is identical to that presented in Definition 2.8, if the label vectors are limited to be two-dimensional. The definition of multiplexers presented in Definition 2.9 applies to ATM cells. So does policing. Thus, the model of the present
chapter captures ATM networks.

2.7.2 Switched Ethernet Networks

Ethernet switches function by forwarding packets based on the destination Ethernet address. Thus, the destination Ethernet address on an Ethernet frame could be taken as its label. The Ethernet switching algorithm is a special case of that presented in this chapter. Broadcast addresses can be handled by setting one forwarding table entry for each input and output port, resulting in $n^2$ entries for each broadcast address, where $n$ is the number of ports. The task of setting the forwarding table entries is a graph building task. Ethernet frames can be multiplexed and policed (though most Ethernet switches do not as they are not QoS-enabled). Thus, the model of the present chapter captures switched Ethernet networks.

2.7.3 Classical IP Networks

IP networks can be placed on the model of the present chapter as follows. Consider every bit of the destination IP address of an IP packet to be a label. Then, an IP packet has thirty-two labels. An IP router forwards packets based on the longest-matching-prefix algorithm. The difference between the IP forwarding algorithm and the algorithm presented in Definition 2.8 is that the latter takes into account the arrival port of the packet in addition to the label on the packet. Moreover, the IP forwarding algorithm does not permit the label to be changed upon forwarding. In other words, Definition 2.8 is more general, and the IP forwarding algorithm is a
particular case. Broadcast and multicast IP addresses can be handled in the same way adopted for Ethernet broadcast addresses (See the previous subsection).

Embedding of IP graphs is achieved by the concept of tunnels. An IP packet is placed inside a tunnel by encapsulating the IP packet inside another IP packet. Thus, in effect, the packet gains another set of labels, resulting in a total of sixty-four labels while traversing the tunnel. Hence, tunneling is similar to graph embedding, save the second order phenomena discussed at the beginning of this section.

Packet multiplexing and policing are applicable to IP just as they are to ATM.

Thus, the model of the present chapter captures classical IP networks. We will use the term ‘classical IP’ to distinguish it from enhancements resulting from RSVP, MPLS, DiffServ, etc.

2.7.4 RSVP-Enabled IP Networks

IP networks that support RSVP can forward packets on a per-flow basis. Flows are identified by the quintuple consisting of the source IP address, destination IP address, transport protocol ID (TCP or UDP), transport port number of the source, and transport port number of the destination. The destination IP address is typically an IP multicast address. An RSVP-capable router forwards packets based on the flow identification if there is a routing table entry for the flow. Packets that do not have such a routing table entry are forwarded as in classical IP. This two-tiered approach to forwarding packets can be captured by our model as follows. Consider an IP packet in an RSVP-enabled network to contain a label vector of thirty-three labels,
the first thirty-two of which are the thirty-two bits of the destination IP address,
and the thirty-third consists of the quadruple (source IP address, transport protocol
ID, source transport port number, destination transport port number). In this way,
the forwarding table entry for an RSVP flow will contain label vectors of dimension
thirty-three, whereas forwarding table entries for classical IP routing will contain label
vectors of dimension no more than thirty-two.

The multiplexers and policers are applicable to RSVP-enabled IP networks just
as they are to IP networks.

Thus, the model presented in this chapter captures RSVP-enabled IP networks.

2.7.5 GMPLS-Enabled IP Networks

GMPLS networks fit naturally into the model presented in this chapter. They are
both outgrowths from ATM technology. GMPLS already identifies label vectors (or
equivalently, label stacks). GMPLS strives to achieve the least common denominator
between all types of networks by introducing labels into the packets. This is similar
to the introduction of IP addresses as a least-common-denominator into Ethernet
and other layer-two networking technologies. In contrast, the model of the present
chapter strives to use the native labeling of the underlying technologies, as illustrated
by the other subsections in this section. However, the model accommodates GMPLS
as well. Whereas GMPLS strives to enhance classical IP, the model presented here
strives to capture networks just as they are.

A Forwarding Equivalency Class (FEC) in GMPLS is a collection of IP addresses
that get mapped to the same GMPLS label on a specific output port. This is captured by the present model by having a list of forwarding table entries of the form \((p^1_i, X^1_i) \rightarrow (p_2, X_2, ID)\) on edge routers, where \(X_2\) is the common label vector to which the FEC gets mapped to on port \(p_2\), the \(X^1_i\)'s are the IP addresses or IP address prefixes that belong to the FEC, and the \(p^1_i\) are the corresponding input ports. On other (non-edge) label switched routers, the forwarding mechanism is based on exact match of labels. The pushing and popping of labels on a stack is captured by the operations of label vector concatenation and contraction. The multiplexers and policers are applicable to GMPLS-enabled IP networks just as they are to IP networks. Thus, the model presented in this chapter captures GMPLS-enabled IP networks.

2.7.6 DiffServ-Enabled IP Networks

A DiffServ-enabled IP network differs from an IP network in that packets in the former have priority bits which are assigned at the source of the packets or at an edge router. This can be captured by the model of the present chapter by considering the set of priority bits to be a label in addition to the thirty-two labels of classical IP. The only question that remains is one of assignment of priorities to packets. DiffServ assigns priorities based on measurements of packet-stream bit-rates. Our resource model does not support measurement-based assignment of labels to packets. However, a software overlay mechanism added to the 'edge label-switches' can be used to achieve this.
2.7.7 Frame Relay Networks

Packets in a Frame Relay network are labeled with a Data Link Channel Identifier (DLCI). No more than one label is possible per packet. The forwarding algorithm of Frame Relay switches is based on exact match of the DLCI and hence is a special case of that presented in Definition 2.8. Packet multiplexing and policing are applicable to Frame Relay frames. Thus, the model presented in this chapter captures Frame Relay networks.

2.7.8 SONET Networks

In a SONET network, packets (SONET frames) are identified by their time slot in the link they are traveling. The time slot is thus the label of the frame. The algorithm for switching frames is based on matching the time slot of the incoming frames. This is a special case of the algorithm presented in Definition 2.8.

Each buffer of the multiplexers at the output ports represents a time slot. Frames are switched and placed in the buffer corresponding to their output time slot, and scheduled in round-robin or weighted round-robin fashion. This is the technique of time slot interchange (TSI).

SONET time-slots can be aggregated. In other words, when an STS-1 stream is embedded inside an STS-3 stream, the STS-1 stream is given two labels – the slot occupied by the STS-3 stream and the slot number of the STS-1 stream inside the STS-3 stream. Concatenated STS-nc \( (n = 3, 12, \ldots) \) is captured in the model as
follows. Each frame belonging to the STS-nc stream is given the label representing
the time-slot occupied by the STS-nc. The buffer assigned to this time-slot is given
a weight of \( n \). (see discussion after Definition 2.9).

SONET streams are implicitly policed by the physical capacity of the time slots,
and hence do not require the QoS-extension.

Thus, the model presented in this chapter captures SONET networks.

2.7.9 Wavelength-switched Networks

Wavelength-switched networks can be placed under the model of the present chap-
ter by considering each wavelength to be a label. Waveband-switching [74] can be
captured by considering each waveband to be a label and the wavelength inside each
waveband to be another label, thus forming label vectors of dimension two.

The multiplexer symbolically represents wavelength-division multiplexing. Each
buffer symbolically represents a wavelength. The buffer manager and scheduler are
void. Streams in WDM are implicitly policed by the capacity of the wavelengths.
Thus, the model presented in this chapter captures wavelength-division multiplexed
networks.

2.7.10 Fiber-switched Networks

Fiber-switched networks can be placed under the model of the present chapter by
taking all label vectors to be of zero dimension. In this way, streams are switched
based only on the arrival port. The multiplexer is void. Streams are implicitly policed
by the capacity of the fibers. Thus, the model presented in this chapter captures fiber-switched networks.

2.7.11 Hybrid Networks

The present model supports networks in which different ports of a label switch are of different type. For example, the model supports switches that have both IP and ATM ports. We present two examples.

Example 1: Figure 2.14 presents an example in which a graph traverses an IP link, followed by an ATM link, followed by a SONET link, followed by a wavelength link, followed by a SONET link, followed by an ATM link, followed by an IP link. In order to set up this graph, the following entries must be set in the forwarding tables.

- Switch 1: \((\text{in} \_ \text{port}, < x_1 >) \rightarrow (\text{out} \_ \text{port}, < x_2, x_1 >)\)

- Switch 2: \((\text{in} \_ \text{port}, < x_2 >) \rightarrow (\text{out} \_ \text{port}, < x_3, x_2 >)\)

- Switch 3: \((\text{in} \_ \text{port}, < x_3 >) \rightarrow (\text{out} \_ \text{port}, < x_4, x_3 >)\)

- Switch 4: \((\text{in} \_ \text{port}, < x_4, x_3 >) \rightarrow (\text{out} \_ \text{port}, < x_3 >)\)

- Switch 5: \((\text{in} \_ \text{port}, < x_3, x_2 >) \rightarrow (\text{out} \_ \text{port}, < x_2 >)\)

- Switch 6: \((\text{in} \_ \text{port}, < x_2, x_1 >) \rightarrow (\text{out} \_ \text{port}, < x_1 >)\)

where

- \(x_1 = \text{(Src IP Addr, Dest IP Addr, TCP protocol ID, Src TCP Port, Dest TCP Port)}\)
Figure 2.14: A hybrid network.

- $x_2 = \text{A VPI/VCI pair}$

- $x_3 = \text{An STS-1 slot}$

- $x_4 = \text{A wavelength}$

For simplicity of presentation, the thirty-three labels on the TCP/IP link have been shown as a single label $x_1$, though they are in reality thirty-three labels. Similarly, the VPI and VCI have been shown as single label $x_2$, but in reality they are two labels.

**Example 2:** Consider again the network in Figure 2.14. Suppose that the entries in the forwarding tables of switches 1 and 6 are replaced thus:

- **Switch 1:** (in\_port, $< x_1 >$) $\rightarrow$ (out\_port, $< x_2 >$)

- **Switch 6:** (in\_port, $< x_2 >$) $\rightarrow$ (out\_port, $< x_1 >$)

Then, the IP addresses and TCP port numbers are stripped from the packets as they traverse the ATM, SONET and wavelength links, but are reinserted at switch 6 before they arrive at the destination computer.
2.8 **Concluding Remarks**

The present chapter gave a model of data transport in communication networks from the viewpoint of graph building. It is applicable to a wide range of transport technologies. It strives to use the native labeling capabilities of the networks directly. It is not an overlay concept as conceived in MPLS. However, it can accommodate the approach of MPLS as well. The common thread running through all of these types of networks is that they switch packetsstreams based on labels, and, as a result, realize the concept of a communication graph, irrespective of whether the network is connection-oriented or not.
2.A \hspace{1em} \textbf{APPENDIX}

2.A.1 \hspace{1em} \textbf{Datapath Protocol Translation}

In this appendix, we present a modification to the core resource model for capturing packet segmentation and reassembly. The only aspect of the model that is changed is the forwarding fabric. The impact of this modification on network control is minimal, as will be shown. Yet, the notational tools necessary for this modification are somewhat involved. Hence, it is presented as an appendix in order to keep the main presentation of this chapter simple.

\textbf{Definition 2.23.} A sequence of packets $P_1, \ldots, P_n$, all with the same label vector $X$, is said to be a \textbf{segmentation} of a packet $P$ if the label vector of $P$ includes $X$, and $P$ is the concatenation of the payloads of $P_1, \ldots, P_n$. Then, $P$ is said to be the \textbf{reassembly} of $P_1, \ldots, P_n$.

To a first degree of approximation, we shall ignore any trailers and padding bytes introduced due to segmentation.

\textbf{Example 1.} $P$ is a TCP packet whose label vector consists of the TCP ports, and the source and destination IP addresses of the IP packets on to which $P$ will be segmented. $P_1, \ldots, P_n$ are the IP packets onto which the TCP packet is segmented. The label vector of each of $P_1, \ldots, P_n$ is their destination IP address (which is the same for all of them). Note that the TCP ports are contained in $P$ whereas the IP addresses are represented by means external to $P$ (in the IP packets). Thus, the
example does not violate Definition 2.3.

**Example 2.** $P$ is an IP packet. $P_1, \ldots, P_n$ are ATM cells. The label vector of $P$ consists of the destination IP address, and the VPI/VCI of the ATM cells onto which it will be mapped. The label vector of each of $P_1, \ldots, P_n$ is the VPI/VCI of the ATM cells. Again, note that the VPI/VCI of $P$ are represented by means external to $P$ (in the ATM cells).

**Definition 2.24.** A sequence of packets $P_1, \ldots, P_n$ with label vectors $X_1, \ldots, X_n$ is said to be embedded in a packet $P$ if each of the $X_i$’s includes the label vector of $P$, and the payload of $P$ is the concatenation of $P_1, \ldots, P_n$.

**Example.** $P$ is a SONET frame and $P_1, \ldots, P_n$ are ATM cells. The label vector of $P$ consists of the time-slot occupied by $P$. The label vector of each of $P_1, \ldots, P_n$ consists of the above SONET time-slot and the VPI/VCI of the ATM cells.

We define $\mathcal{P}^1, \ldots, \mathcal{P}^m$ to be a partition of a sequence $\mathcal{P} = \{P_1, \ldots, P_n\}$ if the following hold:

- Each of $\mathcal{P}^1, \ldots, \mathcal{P}^n$ is a sequence with at least one element, and their union is $\mathcal{P}$.

- If $P_i$ and $P_j$ are elements of $\mathcal{P}^k$ for some $i < j$ and some $k$, then $P_i, P_{i+1}, \ldots, P_j$ are elements of $\mathcal{P}^k$.

**Definition 2.25.** A sequence of packets $\mathcal{P} = \langle P_1, \ldots, P_n \rangle$ is said to be a one-step translation of another sequence of packets $\mathcal{P}' = \langle P'_1, \ldots, P'_m \rangle$ if one of the following holds:
• $P'$ can be partitioned into $n$ subsequences $P^1, \ldots, P^n$ such that $P_i$ is a reassembly of $P^i$ for all $i = 1, \ldots, n$.

• $P$ can be partitioned into $m$ subsequences $P^1, \ldots, P^m$ such that $P'_i$ is an embedding of $P^i$ for all $i = 1, \ldots, m$.

**Definition 2.26.** A sequence of packets $P = < P_1, \ldots, P_n >$ is said to be a translation of another sequence of packets $P' = < P'_1, \ldots, P'_m >$ if there exists a sequence of packet-sequences $P^1, \ldots, P^n$ and two integers $r, s$ such that $1 \leq r \leq s$ and

• $P^{i+1}$ is a one-step translation of $P^i$ for all $i = 0, \ldots, r$;

• $P^i$ is a one-step translation of $P^{i+1}$ for all $i = r + 1, \ldots, s$;

where $P^0 = P$ and $P^{s+1} = P'$.

**Example.** $P$ is a sequence of Ethernet packets obtained by segmentation of a sequence of IP packets; $P^1$ is the sequence of IP packets, which in turn are obtained by segmentation of a sequence of TCP packets; $P^2$ is the sequence of TCP packets; $P'$ is a sequence of ATM cells obtained by segmentation of the TCP packets; $r = 1$ and $s = 2$. In this example, a TCP/IP/Ethernet packet sequence has been translated to a TCP/ATM packet sequence.

If a sequence of packets $P$ is a translation of a sequence of packets $P'$, then it can easily be verified that $P'$ is a translation of $P$.

**Definition 2.27.** Let $P = \{P_1, P_2, \ldots\}$ and $P' = \{P'_1, P'_2, \ldots\}$ be two arrival processes of packets. $P'$ is said to be a translation of $P$ if there exists two increasing
sequences of integers $1 = m_1 < m_2 < \ldots$, and $1 = n_1 < n_2 < \ldots$ such that

- $P'_{m_i}, \ldots, P'_{m_{i+1}-1}$ is a translation of $P_{n_i}, \ldots, P_{n_{i+1}-1}$, and

- The arrival time of $P_{n_{i+1}-1}$ is no later than the arrival time of $P'_{m_i}$, for all $i = 1, 2, \ldots$.

The definition states that an arrival process is a translation of another arrival process if they can each be partitioned into sets of consecutive packets such that corresponding partitions are translations of each other, and there is a certain temporal relationship between them.

Translation of an arrival process of packets may result in a change in its traffic characteristics. In particular, the peak rate and average rate of the packet stream may change. Given the statistics of the original stream and the maximum packet size of the packet framing format, these changes can be estimated. (The maximum packet size is also known as the maximum transfer unit (MTU)). For instance, in Example 1 above, if the MTU of IP packets is 1500 bytes, and the average size of a TCP packet is much larger than 1500 bytes, then the average rate of IP packets may be computed approximately as $s[1 + 40/(1500 - 40)] \approx 1.03s$, where $s$ is the average rate of the TCP packet stream and the IP header is taken to be 40 bytes. In most cases, as in this example, the correction to the bandwidth usage is small and hence can be considered second order.

We shall use the term ‘departure process’ (instead of ‘arrival process’) to indicate a stream of packets that departs an output terminal of an interconnection device.
This is done so as to avoid confusion.

In the presence of protocol translation, the description of an interconnection device is changed as follows. An interconnection device is a logical construct as given previously, except that the departure process of packets at an output terminal must be such that the following holds.

- The arrival processes at the input terminals of the interconnection device can be partitioned into a set of sub-arrival processes such that the departure process at a given output terminal is the superposition\(^4\) of translations of a set of sub-arrival processes.

- Each sub-arrival process is superposed onto only one output terminal.

In other words, sets of departing packets map one-to-one to translations of sets of arrived packets. A multicast interconnection device is a logical construct as given above, but with the second bulleted constraint removed.

In preparation for the definition of a forwarding fabric, we present the following. Consider the relational operator that indicates whether a label vector \(X\) includes another label vector \(Y\) or not (Definition 2.2). It is easily verified that this operator places a partial ordering on any set of label vectors. Thus, given any label-vector set \(\mathcal{X}\) that includes the empty label vector (label vector of zero dimension), the elements of the set can be placed on the vertices of a rooted tree such that the root is the empty label vector, and the label vector represented by any vertex is included in the label vector.

\(^4\) See [17] for the definition of superposition of arrival processes.
vector represented by any descendent of that vertex, but not included in any other vertex. We shall refer to the above tree as the label vector tree of the label vector set.

The arrival process of packets at an input terminal $p$ of an interconnection device that has a forwarding table can be partitioned into subarrival processes as follows. Let $\mathcal{X}(p)$ be the set consisting of the input label vectors of forwarding table entries whose input terminal is $p$. Consider the label vector tree corresponding to the set $\mathcal{X}(p)$. For every label vector corresponding to a leaf of this tree, create a subarrival process by filtering packets that arrive on input terminal $p$ with label vector that includes the label vector of that leaf. Remove the packets used to create these subarrival processes, trim the tree by removing all its leaves, and repeat the above procedure until there are no vertices left except the root. In this way, corresponding to every label vector of $\mathcal{X}(p)$, there is a sub-arrival process. Denote the subarrival process corresponding to a label vector $X$ on input terminal $p$ by $\mathcal{P}(p, X)$.

Let $\mathcal{I}(p')$ be the collection of input parts of the forwarding table entries that have $p'$ as their output terminal. Now, in the presence of protocol translation, the definition of a forwarding fabric is changed as follows.

**Definition 2.28.** A **forwarding fabric** is a multicast interconnection device along with a forwarding table, and satisfies the following.

- The number of input terminals equals the number of output terminals.
• The departure process on an output terminal \( p' \) is of the form

\[
\bigoplus_{(p_1, X_1) \in \mathcal{I}(p')} T(\mathcal{P}(p_1, X_1))
\]

where \( T(\mathcal{P}(p_1, X_1)) \) is an operator that indicates a translation of the subarrival process \( \mathcal{P}(p_1, X_1) \), and the operator \( \oplus \) indicates superposition of arrival processes.

2.1.2 Extension to the Generic Network Resource Model for Exploiting Multiplexing Gain

An advantage of asynchronous multiplexing is its ability to exploit the variability in the bit-rate of traffic flows by allocating less bandwidth than would be otherwise necessary for guaranteeing quality of service. This phenomenon, known as multiplexing gain, is discussed in Appendix A. In the present subsection, we extend the generic network resource model of Section 2.2.2 so that call admission control algorithms could exploit multiplexing gain (on ports that use transport technologies based on asynchronous multiplexing).

There are two aspects to exploiting multiplexing gain, namely dimensioning the multiplexers, and retrieving measurements of packet-flows passing through the multiplexers (for use in call admission control). The latter requires a table (called the measurement table) specifying the list of flows to be monitored. We allow monitoring of aggregate flows, the motivation for which will be found in Appendix A. In order to specify an aggregate flow, we introduce the notion of an aggregate flow identifier.
An aggregate flow consists of a list of individual flows as defined in Definition 2.16. The table of aggregate flows to be monitored would then consist of a list of individual flows. In particular, the table would contain entries of the following form:

- (port, label vector, aggregate flow identifier)

The table is interpreted thus. Whenever a packet with a label vector $X$ arrives at a multiplexer connected to output port $p$, the table is searched for entries whose first elements are $p$ and whose second elements (a label vector) are included in $X$. If such entries are found, then the packet is taken to belong to the aggregate flows identified by all such entries. Otherwise, the packet is not considered for measurements.

**Definition 2.29.** A multiplexing-gain-enabled label switch is an extension of a QoS-enabled label switch. For ports based on asynchronous multiplexing, it imposes no limit on either the number of buffers in the corresponding multiplexer (except that there should be at least one) or the buffer management policy. It can collect the following statistics of the packets arriving at each buffer of every multiplexer: the total number of bytes arriving per aggregate flow and the histogram of the total number of bytes arriving per unit time per aggregate flow. It can respond to the following requests in addition to those given in Definition 2.20.

- **Read the configuration of each multiplexer, i.e., the number of buffers, buffer sizes and identifiers, buffer management policy and scheduling policy.**

- **Set parameters for the scheduling and buffer management policies for each multiplexer.**
• Set the time-unit and bin-width for updating the histograms.

• Set and remove entries in the measurement table.

• Read the statistics of aggregate flows.

• Reset statistics.

We now present an accounting rule for admission control of segments in the presence of multiplexing gain. Define the combination of a traffic contract, a QoS constraint, and a traffic type as a service class. A traffic type is a qualitative description of a traffic flow, examples of which are ‘MPEG-2 video’ and ‘MP3 audio’. Such descriptions are used for exploiting multiplexing gain by categorizing traffic (see Appendix A). Then, the schedulable region of a multiplexer is defined as given in Appendix A. As demonstrated in that appendix, the schedulable region of the multiplexer connected to output port \( p \) may be approximated by a region bounded by a hyperplane specified by \( K \) integers \( S_1(p), S_2(p), \ldots, S_K(p) \), where \( K \) is the number of service classes:

\[
\mathcal{S}(p) = \left\{ x \in \mathbb{N}^K \mid \sum_{i=1}^{K} \frac{x_i}{S_i(p)} \leq 1 \right\}.
\]  

(2.6)

Thus, comparing with Equation 2.2, we see that the number \( C_p/S_i(p) \) represents the thickness of a segment-branch assigned service class \( i \). Unlike in the case of peak-rate allocation, the thickness may be different for different branches of the segment. The thickness represents the effective quantity of bandwidth consumed by the segment branch.
The *effective bandwidth* approach to quality of service [33] may be considered as a special case of the schedulable region approach. The effective bandwidth approach specifies a particular formula for computing effective bandwidths (thicknesses), whereas the schedulable region approach leaves open the formula for computing the thicknesses. The method given in Appendix A is but one approach to computing thicknesses. The peak-rate allocation approach is another.

### 2.4.3 Networking Capacity Graph

Denote the set of nodes in a network by $A$, the set of links by $L$, and the schedulable region of the multiplexer whose output port connects to link $l$ by $S_l$. The *networking capacity graph* of the network is the triple $(A, L, S)$, where $S$ is the collection of schedulable regions corresponding to all links in $L$ (Figure 2.15). In this notation, $S_l$ is viewed as the capacity of edge $l$.

The *networking capacity graph* of a domain is the networking capacity graph of the entire network but restricted to those vertices belonging to that domain, and edges connecting the aforementioned vertices. The *partial networking capacity graph* of a domain $A_{ij}$ is the graph $(\tilde{A}_{ij}, L_{ij}, S^{ij})$, where $\tilde{A}_{ij}$ is the set of children of the domain $A_{ij}$, $L_{ij}$ is the set of links that belong to $A_{ij}$ (see Section 2.5.1), and $S^{ij}$ is the set of schedulable regions of the links in $L_{ij}$.

The networking capacity graph provides a comprehensive view of the network, both in terms of connectivity as well as capacity. The Shannon capacity of a link is inefficient for direct evaluation of quality of service in the presence of multiplexing
Figure 2.15: A networking capacity graph for a network with four nodes and six links. For clarity, the links are shown unidirectional. The schedulable regions are shown as tents. The schedulable region, however, is directly applicable for evaluating quality of service. In particular, it captures the capacity of a link in a way that also captures the quality of service requirements. This is in contrast to the Shannon capacity. Moreover, since schedulable regions tend to have hyperplanar boundaries (see Appendix A), the networking capacity graph lends itself to all the resource allocation and routing algorithms of multi-rate circuit switching [70, 29]. As such, these algorithms can be used for load balancing by distributing the call load on multiple paths. As a result, even though packet switching with variable bit rate traffic exposes a lot of nonlinearities, the resource allocation problem reduces to the problems of circuit-switching. This is in contrast to the QoS routing algorithms considered in the literature ([59] and references therein), where only greedy algorithms can be used for route computation [15]. In particular, none of the multi-rate circuit switching algorithms is applicable
to QoS routing.

The drawbacks of the networking capacity graph approach are similar to those of the schedulable region, namely quantization of traffic into a finite set of traffic classes. In fact, the networking capacity graph provides a set of virtual networks, one for each traffic class. However, these virtual networks are inter-dependent in that they share one schedulable region.

Despite their drawbacks, the schedulable region and the networking capacity graph provide great simplifications to the network resource allocation problem. In fact, they do so with little compromise in efficiency, the only compromise coming from the fact that traffic is quantized into a finite set of traffic classes. However, as explained in Appendix A, this quantization is not as limiting as it may appear at first.

The information contained in the networking capacity graph can be augmented with information on link delays and loss rates (Definition 2.12). We do not address these attributes any further, except to say that they can be measured using an algorithm similar to ‘ping’ used by the Internet. However, the delay measurements would have to be carried out using precision hardware in order to maintain accuracy. We emphasize that these delays and loss rates are those experienced by packets as they traverse links, not multiplexers.
3

APIs for Building Graphs and Minimality

3.1 API Framework for Building Graphs

The fundamental task of network control is to create communication channels, abstracted here as graphs. A communication graph consists of segments and virtual links, as defined in the previous chapter. Virtual links, however, need not be created dynamically as they are always present as long as the links of the network are present (see Section 2.3.2). Thus, the creation of graphs involves only the creation of segments. Hence, segments are the building blocks of graphs.

The task of building graphs involves two steps, namely the determination of its geometry and the creation of its segments. The graph geometry is not known \textit{a priori} for two reasons. First, end-users typically do not specify the geometry of the graphs they request. The geometry is of no interest to the typical end-user. Hence, given the root, leaves, thickness\footnote{We shall use the term ‘thickness’ loosely to refer to the traffic contract of a segment. This is because the thickness of a segment is determined largely by its traffic contract.} and color, the graph-building algorithm must determine
an appropriate graph geometry. Second, the best geometry (according to any given
criterion) of a graph to be created for a specific set of root and leaves may change
over time as network resource usage changes.

Figure 3.1 shows the API framework for building graphs. With every label switch,
a mirror object is associated in the corresponding switch control processor. A mirror
object is a software object that provides an API for the creation and removal of seg-
ments on the label switch it controls. A graph-building algorithm is a (distributed)
algorithm for creating graphs. It utilizes the API provided by the mirror objects to
create segments, and binds them together with the virtual links to create graphs.
The resource discovery algorithms are distributed algorithms for discovering the net-
work resource configuration. They expose an API for reading the network resource
configuration. The graph-building algorithms use this API for computing geometries.
For the purposes of this chapter, we shall consider the API for reading the network
resource configuration to be made available only at one location. This facilitates the
presentation of the concept of minimality introduced in this chapter. The next chap-
ter will show how this API can be replicated so that it is available locally at every
SCP.

The present chapter discusses the APIs shown in Figure 3.1 and some extensions
for scaling as the number of label switches grows. In the figure, APIs are shown as
‘T’-shaped attachments to the objects, and method invocations are shown as arrows.
The head of an arrow points to the API being invoked, and the tail of the arrow
points to the entity invoking the API.
Figure 3.1: The API framework for building graphs.

This chapter is organized as follows. Section 3.2.1 defines an API for manipulating the core network resources, Section 3.2.2 defines an API for manipulating network-element services, Section 3.2.3 defines an API for reading the network resource configuration, and Section 3.2.4 defines an API for manipulating network services. Section 3.3 illustrates the use of the APIs in graph building. Section 3.4 demonstrates that the APIs presented are minimal with respect to the criterion defined therein. The chapter concludes with Section 3.5.

*Description Language and Notation:* Although the APIs in this thesis are in IDL, they could equally well be represented in other emerging service description languages such as WSDL [79]. Whereas these languages are more flexible, IDL is easier to read, and adequate for our purposes. An interface specified in IDL could be realized using,
for example, SOAP [78]. Thus, the use of IDL does not mandate the use of CORBA.

The following IDL abbreviations will be used in the rest of this thesis.

```cpp
typedef unsigned short   UShort;
typedef unsigned long    ULong;
typedef sequence<octet>  OctetSeq;
typedef sequence<UShort> UShortSeq;
typedef sequence<ULong>  ULongSeq;
```

The first two of the above are applicable to C as well. In addition, the following abbreviation will be used in C code.

```cpp
typedef unsigned char    Byte;
```

### 3.2 The APIs

#### 3.2.1 API for Controlling the Core Network Resources

This section presents an API for manipulating the core resources in a network, namely labels, bandwidth and buffer space. The API is represented in Figure 3.1 as the API to the label switches. In practice, this API would reside in the switch control processors and would be a wrapper for the capabilities listed in Definition 2.10. The code implementing this API is akin to device drivers in computer operating systems.

The API consists of multiple sub-APIs. Figures 3.2 shows the API for manipulating a forwarding table, Figure 3.3 shows the API for manipulating a policer table and Figure 3.4 gives an extension for supporting time-division-multiplexed (TDM) and wavelength-division multiplexed (WDM) ports. The API for controlling a multiplexing-gain-enabled label switch is given in Section 3.A.2.
**Forwarding Table API.** The function `getPortConfigurations` returns an array of `PortConfiguration`, giving the capacity, port-type vector, minimum and maximum label vectors, and maximum transfer units (MTU) of each port. The number of ports is given in the return value. The minimum label vector represents the smallest value that each component of a label vector can take. For example, the minimum label vector for ATM consists of the minimum VPI and the minimum VCI. The maximum label vector is defined similarly. The functions `addForwardingTableEntry` and `removeForwardingTableEntry` permit entries to be added and removed from the forwarding table. The structure `LabelVector` represents a vector of labels.

**Policer Table API.** The function `getSupportedPolicingParameters` returns a list of policing parameter types supported by the label switch, if the label switch is QoS-enabled. The number of parameter types is given in the return value. Parameter types are given codes, e.g., peak rate could be given code 1, and maximum burst size given code 2. The functions `addPolicingTableEntry` and `removePolicingTableEntry` add and remove policing table entries, respectively. The traffic contract is given as a list of parameter types and values so that an arbitrary list of parameters could be specified.

**TDM and WDM API.** The method `getLabelWeights` returns the time slot weights (see Definition 2.10) for time-division multiplexed ports, and wavelength weights for wavelength-division multiplexed ports.

An extension to this API for handling resynchronization between a label switch and its SCP is given in Section 3.A.3 as an appendix.
typedef ULong Port;

typedef struct
{
    UShort label_length; /* in units of bits */
    Byte   *label;
} Label;

typedef struct
{
    Byte   vector_dimension;
    Label  *label_list;
} LabelVector;

typedef struct
{
    Byte   vector_dimension;
    Byte   *port_type_vector; /* e.g., ATM, Ethernet, SONET */
    Label  *min_label_vector;
    Label  *max_label_vector;
    ULong  *mtu_list;
} PortDescriptionVectors;

c typedef struct
{
    Port   port;
    ULong  capacity;
    PortDescriptionVectors port_description_vectors;
} PortConfiguration;

int getPortConfigurations(PortConfiguration *conf);
int addForwardingTableEntry(Port in_port, LabelVector in_label_vector,
    Port out_port, LabelVector out_label_vector,
    UShort buffer_ID);

int removeForwardingTableEntry(Port in_port, LabelVector in_label_vector,
    Port out_port, LabelVector out_label_vector);

Figure 3.2: C API for manipulating the forwarding table.
typedef struct
{
    unsigned short   parameter_code;
    unsigned long    parameter_value;
} Parameter;

typedef struct
{
    unsigned short   number_of_parameters;
    Parameter        *array_of_parameters;
} TrafficContract;

int  getSupportedPolicingParameters(unsigned short *parameter_code_list);
int  addPolicingTableEntry(Port port, LabelVector label_vector,
                           TrafficContract contract);
int  removePolicingTableEntry(Port port, LabelVector label_vector);

Figure 3.3: C API for manipulating the policer table.

typedef struct
{
    UShort        number_of_labels;
    Label         *label_list;
    UShort        *label_weights;
} LabelWeights;

int  getLabelWeights(Port port, LabelWeights *weights);

Figure 3.4: C API for reading the label weights for TDM and WDM ports.
3.2.2 API for Creating Network-Element Services

Figures 3.5–3.6 present IDL APIs for manipulating segments, i.e., network-element services. This API is represented in Figure 3.1 as the API to the mirror objects. The three methods of the interface SwitchInfo give the port type vector and the minimum and maximum label vector of each port. The two read-only attributes of the interface SegmentBranch give the supported traffic and quality of service parameter types. Each parameter type is given a code, e.g., peak rate may be given code 1. The supported traffic and quality of service parameters are returned as a list of codes. The method SegmentBranch::add creates a branch of a segment with the specified input and output ports, label vectors, and service contract, provided the call admission control test passes. The ServiceContract (Figure 3.5) is defined as a list of traffic and QoS parameters. Each Parameter is defined by specifying its parameter_code and parameter_value. An implementation of SegmentBranch::add may use any call admission control test but should perform at least as well as the test based on peak-rate allocation, i.e., if a call is admitted by the peak-rate allocation test, then it must be admitted by the test used by the implementation. The method SegmentBranch::remove removes the segment branch specified by the input and output ports and label vectors. If the output port is set to a default value, then the entire segment with the specified input port and label vector is removed.

Figure 3.7 gives an extension for supporting the concept of service classes. The ThicknessAndColor are presented as an IDL union so that they may be specified ei-
ther directly as a **ServiceContract** or indirectly via a service class identifier. The list of instantiated service classes is given by the read-only attribute **service_classes**.

The services provided by the segment-manipulation API are very similar to those provided by the resource-manipulation API of the previous subsection. The difference between the two comes from two perspectives. First, the resource-manipulation API becomes much larger in the case of a multiplexing-gain-enabled label switch (see Section 3.A.2), but the segment-manipulation API remains the same. Second, the segment manipulation API provides call admission control whereas the resource-manipulation API does not. Thus, the segment-manipulation API is an abstraction of the resource-manipulation API.

### 3.2.3 API for Reading the Network Resource Configuration

In this section we present an API for reading the network resource configuration, from which graph geometries could be computed using algorithms such as those found in [70]. The network resources are captured in the network topology (Section 2.4.1) and networking capacity graph (Section 2.A.3).

Figure 3.8 shows the API in IDL. The attribute **the.ncg** lists the contents of the networking capacity graph, and also the operating points of the links. The attribute **the.topology** lists the topology. The networking capacity graph and topology are represented as lists of links attached to each node. Nodes are identified by a fixed-length address. The structure **NodeData** contains the list of links attached to a node, and the structure **NodeDataIncludingState** contains the schedulable region and op-
struct Label
{
    UShort     label_length;  // in units of bits
    OctetSeq the_label;
};
typedef sequence<Label> LabelVector;
typedef ULong Port;
struct LabelMapping
{
    Port          in_port;
    LabelVector   in_label_vector;
    Port          out_port;
    LabelVector   out_label_vector;
};
typedef OctetSeq PortTypeVector;

struct Parameter
{
    unsigned short parameter_code;
    unsigned long parameter_value;
};
typedef sequence<Parameter> TrafficContract;
typedef sequence<Parameter> QoSContract;
struct ServiceContract
{
    TrafficContract    traffic_contract;
    QoSContract        qos_contract;
};

Figure 3.5: IDL definitions of ports, labels, traffic and QoS parameters.
interface SwitchInfo
{
    PortTypeVector getPortTypeVector(in Port the_port) raises(NotExist);
    LabelVector getMinLabelVector(in Port the_port) raises(NotExist);
    LabelVector getMaxLabelVector(in Port the_port) raises(NotExist);
};

interface SegmentBranch
{
    readonly attribute UShortSeq supported_traffic_parameters;
    readonly attribute UShortSeq supported_qos_parameters;

    void add(in LabelMapping label_mapping, in ServiceContract contract)
        raises(InvalidArgument, Rejected);
    void remove(in LabelMapping label_mapping)
        raises(InvalidArgument, NotExist);
};

Figure 3.6: IDL API for manipulating segments.
struct ServiceClass
{
    UShort service_class_id;
    ServiceContract contract;
};
typedef sequence<ServiceClass> ServiceClassSeq;

union ThicknessAndColor switch(short)
{
    case 0: ServiceContract service_contract;
    case 1: UShort service_class_id;
};

interface SegmentBranchWithTrafficClasses
{
    readonly attribute UShortSeq supported_traffic_parameters;
    readonly attribute UShortSeq supported_qos_parameters;
    readonly attribute ServiceClassSeq service_classes;

    void add(in LabelMapping label_mapping,
             in ThicknessAndColor thickness_color)
        raises(InvalidArgument, Rejected);
    void remove(in LabelMapping label_mapping)
        raises(InvalidArgument, NotExist);
};

Figure 3.7: Amendments to the segment-manipulation API for supporting service classes.
erating point of each link attached to a node. A schedulable region is represented by a vector of positive integers that represent the intercepts of a hyperplanar approximation to the schedulable region (see Appendix A). The operating point of the schedulable region is represented as a vector of non-negative integers.

### 3.2.4 API for Creating Network Services

The API for building network services (graphs) is shown in Figure 3.9. The `GraphBranchBuilder` API would be exposed by the graph building algorithm. The method `build` builds a branch of a graph from the `in_port` of `in_switch` to the `out_port` of `out_switch` with the specified thickness and color, provided that call admission control tests succeed along a path from `in_switch` to `out_switch`. An implementation of `build` may use any loop-free path and any call admission control test, but it must perform at least as well as the implementation that uses least-hop-count paths and peak-rate allocation. By this is meant that given a set of graphs already admitted, a new graph that is admissible according the peak-rate allocation rule along a least-hop-count path must also be admitted by the implementation of `build`. The parameters `in_label` and `out_label` specify the label vectors to be used by the graph builder at the two end-points of the graph branch. But, if they are set to a certain predefined value, then the graph builder will use a label vector of its choice and return the chosen label vectors as outputs. This is why the label vectors are specified as `inout` in the IDL interface. The method `remove` removes a graph branch whose endpoints (root and leaf) are specified in the input parameter list. If
typedef octet Address[<len>];
struct OneLinkOfANode
{
    Port    origin_port;
    Address destination_node;
    Port    destination_port;
};
struct OneLinkAndStateOfANode
{
    OneLinkOfANode link;
    ULongSeq    schedulable_region;
    ULongSeq    operating_point;
};

typedef sequence<OneLinkOfANode> AllLinksOfANode;
typedef sequence<OneLinkAndStateOfANode> AllLinksAndStateOfANode;
struct NodeData
{
    Address    node;
    AllLinksOfANode links; // All links originating from the above node.
};
struct NodeDataIncludingState
{
    Address    node;
    AllLinksAndStateOfANode links;
};
typedef sequence<NodeData> Topology;
typedef sequence<NodeDataIncludingState> NCG;
interface NetworkTopology { readonly attribute Topology the_topology; };
interface NetworkingCapacityGraph { readonly attribute NCG the_n cg; };

Figure 3.8: IDL API for reading the network resource configuration.
interface GraphBranchBuilder
{
    readonly attribute UShortSeq supported_traffic_parameters;
    readonly attribute UShortSeq supported_qos_parameters;

    void build(in Address in_switch, in ULong in_port,
                in Address out_switch, in ULong out_port,
                inout LabelVector in_label_vector,
                inout LabelVector out_label_vector,
                in ServiceContract contract)
        raises(InvalidArgument, Rejected);

    void remove(in Address in_switch, in ULong in_port,
                in Address out_switch, in ULong out_port,
                in LabelVector in_label_vector,
                in LabelVector out_label_vector)
        raises(InvalidArgument, NotExist);
};

Figure 3.9: IDL API for manipulating network communication graphs.

the address of the leaf is set to a default value, then the entire graph with the specified root, input port and label vector is removed. The two read-only attributes return the supported traffic and QoS parameters.

3.3 Graph Building

We now provide a very brief introduction to how the APIs of the previous section may be used to build graphs. An overview of various categories of graph-building algorithms may be found in [18]. Specific graph-building algorithms can be found in
The purpose of this section is to illustrate how the APIs of this chapter may be used.

We base our discussion on building graph branches. Multicast graphs are built as a sequence of graph branches. Graph branches to be built are typically specified in terms of the end-points (the root and the leaf, i.e., the source and the destination). Suppose first that the root and leaf of the graph branch to be built lie in the same lowest-level domain. The graph-building algorithm would have to find a path from the root to the leaf. This it can do by reading either the topology or the networking capacity graph of that domain and computing a path using algorithms such as those given in [70]. It requires the selection of label vectors for each link along the path. If, on a given link, the graph branch being created is not being merged with another branch of the same graph (i.e., not part of a multicast), then an unused label vector would have to be chosen on that link. Otherwise, the label vector already used on that link by the previous branch of the multicast would have to be used\(^2\). Next, it creates segments on each of the nodes along the path.

If the root and leaf of the graph branch are in different lowest-level domains, then the following approach is taken based on the APIs given in Appendix 3.A.1. For concreteness, we assume that the parent’s parent (i.e., grandparent) of the root and

\(^2\)It is the responsibility of the graph-building algorithm to maintain the label vector usage at each link. Moreover, whether a label vector is treated to have local significance only on a link or global significance is determined entirely by the graph-building algorithm. For example, an IP address has global significance whereas ATM VPI/VCI have local significance on every link. However, in principle, IP addresses could be made to have local significance and ATM VPI/VCI could be made to have global significance. The choice is up to the graph-building algorithm.
that of the leaf are identical. If this is not the case, the procedure presented below is repeated recursively up the domain hierarchy until a common ancestor is found. Denote the root by $r$, the leaf by $l$, the parent of the root by $A_r$, the parent of the leaf by $A_l$, and the common grandparent by $A$. First, a path is found from $A_r$ to $A_l$ in the partial networking capacity graph of $A$ by executing a path computation algorithm such as those given in [70]. Suppose that this path is from port $p_r$ of domain $A_r$ to $p_l$ of domain $A_l$. Port $p_r$ maps to a certain node $a(r)$ in Domain $A_r$ and port $p_l$ maps to a certain node $a(l)$ in domain $A_l$. These domain port mappings are obtained directly from the APIs presented in Appendix 3.A.1. Then, the next task is to find a path from $r$ to $a(r)$ and a path from $a(l)$ to $l$. This is achieved by reading the topologies or the networking capacity graphs of domains $A_r$ and $A_l$, and executing a path computation algorithm on them. Now the graphs along these paths can be setup as described in the previous paragraph since each of these paths is entirely within a single domain. Similarly, if the path from $A_r$ to $A_l$ in the partial networking capacity graph of $A$ traverses some intermediate domains in $A$, then a graph would have to be built in those domains by reading their domain port mappings and networking capacity graphs or topologies.

Removing a graph branch requires the graph branch to be identified. A branch can be identified by specifying the root and the leaf. Once the root and leaf are identified, the graph building algorithm must retrieve the geometry of the branch. For this purpose, it must maintain the geometry of every graph it builds\(^3\). Once

\(^3\)In practice, the geometries of the graphs are not stored in a central location, but
the geometry is retrieved from its internal records, the graph-building algorithm may invoke the method `SegmentBranch::remove` on the mirror object of each label switch through which the graph branch passes. A graph is removed by removing all of its branches.

3.4 **Minimality**

This section introduces the concept of minimality of APIs. The concept of minimality demonstrates (i) the relationship between the core network resource API and the network-element service API, and (ii) the relationship between the network-element service API, the network resource configuration API, and the network service API. The definition is presented in a framework that is independent of the foundations presented thus far. Two propositions will then illustrate the applicability of the definition to the APIs presented previously.

3.4.1 **Basic Definition**

A programming language function $f$ with $n$ input parameters $\xi_1, \ldots, \xi_n$ will be denoted $f(\xi_1, \ldots, \xi_n)$. A programming language function will be referred to simply as a function. An API is a finite collection of functions (also called methods)$^4$. If the method $f$ belongs to an API for manipulating an object $O$, it may also be denoted decentralized across the SCPs in the network. The tasks of retrieving the geometry of the graph and removing the segments of the graph occur in an interleaved fashion.

$^4$An IDL attribute will be considered as two methods, one for reading and one for writing. An IDL read-only attribute will be considered as one method for reading.
O :: f(ξ₁, ..., ξₙ) or O :: f. Here, the concept of an object is used in the same sense as in programming languages. It is possible to combine multiple parameters into one by placing them in a data structure such as a struct in C++ or IDL, or a sequence in IDL. In this section, we shall consider each of the constituent parameters of a data structure to be a separate parameter\(^5\). The result of this is that the number of input parameters to a method could be variable, depending on the values of some of the input parameters. Thus, when we write \(f(ξ₁, ..., ξₙ)\), it is understood that value of \(n\) could depend on the value of one or more of the input parameters.

We now present a series of definitions that lead to the concept of minimality. Minimality strives to capture redundancy-free APIs — redundancy in terms of parameters and in terms of methods.

**Definition 3.1.** The set of input parameters of a method \(O :: f(ξ₁, ..., ξₙ)\) on an object \(O\) is said to be redundancy-free if there exists no computer program that can realize \(O :: f\) without reading the values of each of the input parameters.

**Example.** Suppose that \(O\) is a label switch and \(O :: f\) is the method `removeForwardingTableEntry` (Figure 3.2). Then, \(ξ₁\) is `in_port`. The second and fourth parameters — `in_label_vector` and `out_label_vector` — are each a sequence of sequences (double sequence). Denote \(j^{th}\) bit of the \(k^{th}\) label in `in_label_vector`...
by $x^j_k$ where $k = 1, \ldots, r$, $j = 1, \ldots, r_k$, $r$ is the number of labels in the label vector, and $r_k$ is the number of bits in the $k^{th}$ label. Similarly, define $y^j_k$ for \texttt{out\_label\_vector} $(k = 1, \ldots, s)$, $j = 1, \ldots, s_k$). Then, the input parameter list (the $\xi_i$) consists of \texttt{in\_port}, $r$, $r_1, x^1_1, \ldots, x^r_1$, $r_2, x^1_2, \ldots, x^r_2$, $\ldots$, $x^r_r$, \texttt{out\_port}$, s$, $s_1, y^1_1, \ldots, y^s_1$, $s_2, y^1_2, \ldots, y^s_2$, $\ldots$, $y^s_s$. Now, the $\xi_i$ are redundancy-free because a segment cannot be removed unless all of these parameters are known. However, if we add one more input parameter to the method \texttt{removeForwardingTableEntry}, and this parameter specifies the buffer ID of the entry to be removed, then the parameter list is not redundancy-free. This is because the buffer ID is not needed for removing a forwarding table entry.

In the following, we shall say that an object $O_1$ contains an object $O_2$ if the internal variables of $O_1$ contain a reference, pointer, or some other means to locate and access $O_2$. Visually, we may think of $O_2$ as being “inside” $O_1$.

\textbf{Definition 3.2.} Given two objects $O_1$ and $O_2$ such that either $O_1$ is the same object as $O_2$ or $O_1$ contains $O_2$, a method $O_1 :: g$ is said to be \textbf{decomposable} into a set of methods $O_2 :: f_1, \ldots, O_2 :: f_n$ if a computer program can be written to realize $O_1 :: g$ using only

- \textit{the methods $O_2 :: f_1, \ldots, O_2 :: f_n$,}

- \textit{the input parameters of $O_1 :: g$, and}

- \textit{the internal variables of the object $O_1$,}
subject to the constraint that the computer program may not read or write the internal variables of the object $O_2$ except by invoking one or more of the methods $O_2 :: f_1, \ldots, O_2 :: f_n$. The decomposition is said to be tight if there does not exist an $i$ such that $O_1 :: g$ can be decomposed into the set of methods $\{O_2 :: f_1, \ldots, O_2 :: f_n\} - \{O_2 :: f_i\}$.

**Example.** Suppose that $O_2$ is a label switch and $O_1$ is its mirror object. Let $O_1 :: g$ be the method add of the interface `SegmentBranch` and $O_2 :: f_1, \ldots, O_2 :: f_n$ be the functions of the resource-manipulation API (Figures 3.2-3.4). Then, $O_1 :: g$ is decomposable into the methods $O_2 :: f_1, \ldots, O_2 :: f_n$ because a segment branch can be added by invoking the methods of the resource-manipulation API only. But, the decomposition is not tight because, for example, the method `removeForwardingTableEntry` is not needed for creating a segment branch. However, the decomposition is tight if the $O_2 :: f_1, \ldots, O_2 :: f_n$ consist only of the methods `addForwardingTableEntry`, `addPolicingTableEntry`, `getPortConfigurations`, and `getLabelWeights`. This is because these are the only methods that are required for creating a segment branch. The method `getPortConfigurations` is required for determining the port bandwidths (for admission control), and the method `getLabelWeights` is required to determine the bandwidth assigned to a time-slot in TDM ports or to a wavelength in WDM ports (again, for admission control).

**Definition 3.3.** A set of methods $O :: f_1, \ldots, O :: f_n$ on an object $O$ is said to be independent if the input parameters of each of the methods is redundancy-free, and
none of the methods is decomposable into the remaining methods.

Example. The resource-manipulation API is independent because none of its methods contains any redundant input parameters, and none of them can be realized in terms of the rest. However, if we add a new method `getAllForwardingTableEntries` for retrieving the content of a forwarding table, and a method `clearForwardingTable` for clearing the forwarding table, then the resource-manipulation API is no longer independent. This is because the method `clearForwardingTable` is decomposable into the methods `removeForwardingTableEntry` and `getAllForwardingTableEntries`.

**Definition 3.4.** Given two objects $O_1$ and $O_2$ such that $O_1$ contains $O_2$, a set of methods $O_2 :: f_1, \ldots, O_2 :: f_n$ is said to be a **minimal basis** of a set of methods $O_1 :: g_1, \ldots, O_1 :: g_m$, if the following conditions are satisfied.

1. The set of methods $O_2 :: f_1, \ldots, O_2 :: f_n$ is independent.

2. Each of the methods $O_1 :: g_1, \ldots, O_1 :: g_m$ is decomposable into the set of methods $O_2 :: f_1, \ldots, O_2 :: f_n$.

3. Condition 2 fails if any one of the $O_2 :: f_1, \ldots, O_2 :: f_n$ is removed.

4. Suppose that there exists an integer $i$ and a set of methods $O_2 :: h_1, \ldots, O_2 :: h_k$ such that

   - The set of methods $\{O_2 :: h_1, \ldots, O_2 :: h_k\} \cup \{O_2 :: f_1, \ldots, O_2 :: f_n\} - \{O_2 :: f_i\}$ is independent;
• $O_2 :: h_1, \ldots, O_2 :: h_k$, along with zero or more of the methods \{\(O_2 :: f_1, \ldots, O_2 :: f_n\)\} - \{\(O_2 :: f_i\)\}, form a tight decomposition of $O_2 :: f_i$.

Then, Conditions 3 holds if $O_2 :: f_i$ is replaced by $O_2 :: h_1, \ldots, O_2 :: h_k$.

5. The output parameters of the methods $O_2 :: f_1, \ldots, O_2 :: f_n$ contain only those required to determine the output parameters of $g_1, \ldots, g_m$, and their error flags.

Conditions 1, 2 and 5 are fairly straightforward. Condition 3 is subtle. It does not state that the decompositions in Condition 2 must be tight. This is because, for any given $i$, $O_1 :: g_i$ may be decomposable into a subset of the methods $O_2 :: f_1, \ldots, O_2 :: f_n$. Yet, for different values of $i$, this subset may be different. Condition 4 states that none of the $O_2 :: f_1, \ldots, O_2 :: f_n$ may contain any extraneous functionalities in addition to what is required for realizing the $O_1 :: g_1, \ldots, O_1 :: g_m$. We shall defer giving an example of this definition as the next section is devoted to this. Informally, we shall refer to the concept of a minimal basis as \textbf{minimality}.

Note that the set of methods $O_1 :: g_1, \ldots, O_1 :: g_m$ can have multiple minimal bases. Moreover, the number of methods in each of these minimal bases is not required to be the same. The reason for this is that minimizing the number of methods does not necessarily improve conceptual understanding. For example, given an API of methods $O_2 :: f_1, \ldots, O_2 :: f_n$, it is possible to replace it with a single method $O_2 :: f$ whose input parameters consist of the union of those of the $O_2 :: f_i$ and also an additional parameter indicating which of the $O_2 :: f_i$'s is to be invoked. Using this
Figure 3.10: Illustration of Proposition 3.1.

technique, all APIs could be reduced to a single method, however, at the cost of clarity.

3.4.2 Minimality of the Core Network Resource API

We now apply the above definitions to the network control APIs presented earlier. For the rest of this section, we assume that the network is QoS-enabled.

Consider a label switch $O_2$ and its mirror object $O_1$. We may consider $O_2$ to be contained in $O_1$ since access to the label switch is possible only through its mirror object. Figure 3.10 illustrates this. $O_2$ is shown as the core network resources and $O_1$ is shown as the network element services (which are provided by the mirror objects). The following proposition applies to this scenario.

Proposition 3.1. The API for manipulating the core network resources (Figures 3.2 - 3.4) is a minimal basis of the API for manipulating segments (Figure 3.6), provided
that we impose the restriction that no decomposition of the methods of the latter API may use error codes of any method of the former API for determining information about the label switch resources.

**Condition 1.** The methods of the resource-manipulation API contain the bare minimum input parameters necessary to identify forwarding and policer table entries. All the other parameters are output parameters. Therefore, there is no redundancy in the parameter-lists of any of these methods. From the semantics of the methods, it is also clear that none of these methods can be realized in terms of the remaining methods of the API. Hence, the methods of the resource-manipulation API are independent. Thus, Condition 1 of Definition 3.4 is satisfied.

**Condition 2.** The methods `SegmentBranch::add` and `SegmentBranch::remove` can be decomposed into the methods for manipulating resources. Indeed, the creation of a segment requires that the segment pass call admission control tests based on some accounting rule and that the forwarding table and policer table be set with appropriate entries as given in the definition of a segment (see the previous chapter). Call admission control could be carried out based on the peak rate (available from the input parameter list to `SegmentBranch::add`) and the capacity of the ports (available from the method `getPortConfigurations`). The forwarding table and policer table entries could be set using the methods `addForwardingTableEntry` and `addPolicingTableEntry`. If a port is time-division multiplexed, then the identifiers and weights of the time-slots could be read from the method `getLabelWeights`. If a
port is wavelength-division multiplexed, then the identifiers and weights of the wavelengths could be read from the method `getLabelWeights`. The time-slot identifiers and wavelength identifiers will be used for setting the forwarding table entries, and their weights will be used for determining the bandwidth of the slots/wavelengths. Segments can be removed using the methods `removeForwardingTableEntry` and `removePolicingTableEntry`. The three methods of the interface `SwitchInfo` can be realized directly by invoking the method `getPortConfigurations`. The attribute `supported_traffic_parameters` can be realized using the method `getSupportedPolicingParameters`. The attribute `supported_qos_parameters` does not require access to the internal variables of the label switch. This is because the label switch considered here is not required to be multiplexing-gain enabled. Thus, all quality of service computations are based on bandwidth alone. Condition 2 is thus satisfied.

**Condition 3.** Removal of the method `getPortConfigurations` would result in the port capacities being unknown and hence call admission control cannot be carried out. Moreover, this would entail that the three methods of the interface `SwitchInfo` cannot be realized (except by resorting to trial-and-error via the error codes of the method `addForwardingTableEntry`). Removal of either of the methods `addForwardingTableEntry`, `addPolicingTableEntry` would result in the segments not being created. Removal of either of the methods `removeForwardingTableEntry`, `removePolicingTableEntry` would result in the segments not being removed. Removal of the method `getLabelWeights` would make it impossible to know the times-
slot weights of time-division multiplexed ports and wavelength weights of WDM ports (and hence call admission control cannot be carried out). Thus, Condition 3 is satisfied.

**Condition 4.** Although Condition 4 does not state so, in the justification of the proposition we shall only consider decompositions that result in conceptually simpler functions. This notion will not be defined, but left intuitive. In particular, a method that performs "less functionality" than another will be considered conceptually simpler.

Each of the operations of creating a forwarding table entry, removing a forwarding table entry, creating a policing table entry, and removing a policing table entry is atomic, *i.e.*, they cannot be decomposed into simpler operations in the context of the model presented in the previous chapter. The remaining methods of the resource-manipulation API merely return a list of parameters from the label switch. These methods can be decomposed so that the methods of the decomposition return fewer parameters than the original. Such a decomposition does not violate Condition 4, as easily verified. The above exhausts all decompositions in terms of conceptually simpler methods. Thus, Condition 4 is satisfied for conceptually simpler decompositions.

**Condition 5.** The only methods in the API with output parameters (excluding error codes) are `getPortConfigurations`, `getSupportedPolicingParameters` and `getLabelWeights`. They return the capacity, port-type vector, MTUs, and label space range of each port, the list of supported policing parameters, and the timeslot/wavelength identifiers and weights of TDM and WDM ports. If the capacity
is removed from the output parameter list, call admission control cannot be car-
ried out. If the port-type vector is removed from the output list, then the method
SwitchInfo::getPortTypeVectors cannot be realized (except possibly by trial-and-
error using the error codes of the function addForwardingTableEntry, but this is
forbidden in the hypothesis of the proposition). If the MTUs are removed, call
admission control cannot be carried out accurately (see Section 2.A.1). If the la-
bel space range (minimum and maximum label vectors) is removed, the methods
getMinLabelVector and getMaxLabelVector of the interface SwitchInfo cannot
be realized. If the supported policing parameters are removed, then the attribute
supported_traffic_parameters in the interface SegmentBranch cannot be realized
(except by trial-and-error using the error codes of the method addPolicingTableEntry,
but this is forbidden). If the label weights are removed from the output parameter
list of getLabelWeights, then call admission control cannot be carried out for time-
division-multiplexed ports and wavelength-division multiplexed ports. If the label list
is removed, then the label weights are not associated with labels. Thus, Condition 5
is satisfied.

3.4.3 Extended Definition

To proceed further, we present the following extensions to the previous definitions.
Although these definitions subsume those presented previously, they are presented
here separately so as to keep the initial presentation simple and easier to conceptual-
ize.
Definition 3.5. Given an object $O$ that contains a set of objects $O_1, \ldots, O_N$, a method $O :: g$ is said to be decomposable into a set of methods $O_i :: f_j$ ($i = 1, \ldots, N; j = 1, \ldots, n_i$) if a computer program can be written to realize $O :: g$ using only

- the methods $O_i :: f_j$ ($i = 1, \ldots, N; j = 1, \ldots, n_i$),

- the input parameters of $O :: g$, and

- the internal variables of the object $O$,

subject to the constraint that the computer program may not read or write the internal variables of the objects $O_1, \ldots, O_N$ except by invoking one or more of the methods $O_i :: f_j$ ($i = 1, \ldots, N; j = 1, \ldots, n_i$). The decomposition is said to be tight if there does not exist a pair $i, j$ such that $O :: g$ can be decomposed into the methods of the set \{$O_1 :: f_1, \ldots, O_1 :: f_{n_1}, \ldots, O_N :: f_{n_N}$\} $\setminus$ \{$O_i :: f_j$\}.

Definition 3.6. Given an object $O$ that contains a set of objects $O_1, \ldots, O_N$, a set of methods $O_i :: f_j$ ($i = 1, \ldots, N; j = 1, \ldots, n_i$) is said to be a minimal basis of a set of methods $O :: g_1, \ldots, O :: g_m$, if all of the following conditions are satisfied.

1. The set of methods $O_i :: f_1, \ldots, O_i :: f_{n_i}$ is independent for each $i = 1, \ldots, N$.

2. Each of the methods $O :: g_1, \ldots, O :: g_m$ can be decomposed into the set of methods \{$O_i :: f_j; i = 1, \ldots, N; j = 1, \ldots, n_i$\}.

3. Condition 2 fails if any one of the $O_i :: f_j$ is removed.

4. Suppose that there exists an integer pair $i, j$ and a set of methods $O_i :: h_1, \ldots, O_i :: h_k$ such that
• The set of methods \( \{O_i :: h_1, \ldots, O_i :: h_k\} \cup \{O_i :: f_1, \ldots, O_i :: f_n\} - \{O_i :: f_j\} \) is independent;

• \( O_i :: h_1, \ldots, O_i :: h_k \), along with zero or more of the methods \( \{O_i :: f_1, \ldots, O_i :: f_n\} - \{O_i :: f_j\} \), form a tight decomposition of \( O_i :: f_j \).

Then, Conditions 3 holds if \( O_i :: f_j \) is replaced by \( O_i :: h_1, \ldots, O_i :: h_k \).

5. The output parameters of the methods \( O_i :: f_j \) (\( i = 1, \ldots, N; j = 1, \ldots, n_i \)) contain only those required to determine the output parameters of \( g_1, \ldots, g_m \), and their error flags.

### 3.4.4 Minimality of the Network-Element Service and Resource Configuration APIs

We now demonstrate the minimality of the segment manipulation API and the network resource configuration API with respect to graph building. Similar to how the label switch was considered to be contained inside its mirror object, we shall consider the graph-building algorithm to contain the mirror objects of the label switches, and the network resource configuration object (in the case of a single-domain network). The network resource configuration object is the object that exposes the API for reading the network resource configuration. In the notation of the extended definition of minimality given above, the graph-building algorithm is \( O \), the mirror objects are \( O_1, \ldots, O_{N-1} \) and the network resource configuration object is \( O_N \), where \( N - 1 \) is the number of label switches in the network. We shall consider the graph building
algorithm to be an object in that it expose an API. It provides the network services. The mirror objects provide the network-element services. Figure 3.11 illustrates this.

The following proposition applies to this scenario.

**Proposition 3.2.** The methods of the interfaces SegmentBranch, SwitchInfo, and NetworkTopology form a minimal basis of the interface GraphBranchBuilder, provided that the graph branch builder is required to be able to support any traffic and QoS parameters supported by all the mirror objects, and the exception InvalidArgument is not used to infer information about the label switch or the set of parameters supported by the interface SegmentBranch.

*Condition 1.* The input parameters to each of the methods of the interface SegmentBranch contain the bare minimum to create and remove segments as required by the interface GraphBranchBuilder. The three methods of the interface SwitchInfo need to have the port as input so that they can retrieve the port type
vector, and the minimum and maximum label vectors for that port. The other methods of the interfaces SegmentBranch, and NetworkTopology are attributes and hence do not contain any input parameters. It is also clear from the semantics of these methods that none of them can be decomposed into the rest. Thus, Condition 1 is satisfied.

**Condition 2.** The decomposition of the methods build and remove of the interface GraphBranchBuilder follows from Section 3.3. The decomposition of the read-only attributes supported_traffic_parameters and supported_qos_parameters may be achieved by taking the intersection of the corresponding parameters exposed by the interface SegmentBranch of each of the mirror objects. Thus, Condition 2 is satisfied.

**Condition 3.** From the discussion of Section 3.3 and the semantics of the methods, it is clear that removal of any of the methods add and remove of the interface SegmentBranch, or the read-only attribute NetworkTopology::the_topology will make it impossible to either build graphs or remove graphs. Removal of any of the methods of the interface SwitchInfo will make it impossible to determine what label vectors can be used for creating segments\(^6\) (except by using the exception InvalidArgument of the method SegmentBranch::add, but this is forbidden). Removal of either of the attributes supported_traffic_parameters or supported_qos_parameters of the interface SegmentBranch would make it impossible.

\(^6\)It might appear that the minimum and maximum label vectors would be sufficient to determine what label vectors can be used for building segments, thus rendering the port-type vector useless. This observation is true for most transport technologies but not for IP. In the case of IP, a specific range of label vectors (IP addresses) is reserved for multicast. This range cannot be known unless the port-type vector is known.
ble to implement the corresponding attributes of the interface `GraphBranchBuilder`, unless the only parameter supported is taken to be the peak rate (which is required to be supported by all QoS-enabled label switches). The latter case is forbidden in the proposition. Thus, Condition 3 is satisfied.

*Condition 4.* The read-only attributes `supported_traffic_parameters`, `supported_qos_parameters`, `port_type_vectors`, and `the_topology` have no functionality except to return a set of values from a database. These attributes can be decomposed so that each method of the decomposition returns a fewer number of values than the original attribute, but such a decomposition does not violate Condition 4. The method `SegmentBranch::add` can be decomposed tightly into the methods `addForwardingTableEntry`, `addPolicingTableEntry`, `getPortConfigurations`, and `getLabelWeights`. Even though these methods belong to the label switch, they can be brought up to the level of the mirror object by placing wrapper methods. This is because a label switch is considered to be contained inside its mirror object. It is easily verified that the decomposition specified above does not violate Condition 4. The method `SegmentBranch::add` could also be decomposed into a set of methods, each of which is a combination of the methods `addForwardingTableEntry`, `addPolicingTableEntry`, `getPortConfigurations`, and `getLabelWeights`. It is easily (but tediously) verified that such a decomposition does not violate Condition 4 either. Similarly, the method `SegmentBranch::remove` can be decomposed into the methods `removeForwardingTableEntry` and `removePolicingTableEntry` or a combination thereof. Again, it is easily verified that this decomposition does not violate
Condition 4. The above exhausts all decompositions in terms of conceptually simpler methods. Thus, Condition 4 is satisfied for conceptually simpler decompositions.

Condition 5. The methods SegmentBranch::add and SegmentBranch::remove do not return any values but exceptions. The topology returned by the attribute the_topology of the interface NetworkTopology is required for determining graph geometries. The port-type vectors, and the minimum and maximum label vectors returned by the methods of the interface SwitchInfo are needed for the graph-building algorithm to choose appropriate label vectors (unless it uses exceptions from SegmentBranch::add to determine the port-type vectors and label vector ranges, but this is forbidden). The two read-only attributes of the interface SegmentBranch are required to determine the traffic and QoS parameters supported on each label switch (unless the exception InvalidArgument of the method SegmentBranch::add is used to determine them, but this is forbidden). Thus, Condition 5 is satisfied.

This completes the justification of the proposition. A similar statement can be made about the interfaces SegmentBranch, SwitchInfo, NetworkTopology, and DomainPortMapper with respect to the interface GraphBranchBuilder, in the case of multiple domains. The justification is almost identical.

3.5 Concluding Remarks

In this chapter, we have presented APIs for graph-building and shown that they are minimal with respect to a set of criteria that we introduced in this chapter. The
APIs are very general, applicable to various transport technologies. As a result, any graph-building algorithm can use these APIs to build graphs. In particular, PNNI [9], RSVP [41], and LDP [40] could use these APIs. Moreover, routing a la OSPF [61] can be realized on top of these APIs (Chapter 4). Furthermore, the APIs presented for the core network resources and the network-element services are valid for a label switch partition [77] as well. This is because, a label switch partition behaves just like a label switch [77].

The definition of minimality presented herein will find applications in all branches of computer programming.
3.A Appendix

3.A.1 Scaling by Partitioning APIs

The structures NCGr and NetworkTopology of Section 3.2.3 grow very rapidly in their memory usage as the number of nodes in the network increases. Resource discovery algorithms run into scalability problems as the number of nodes and links increases, and the route computation algorithms become computationally very intensive (Chapter 4). Thus, there is a need for hierarchical partitioning of the API for reading the network resource configuration.

The previous chapter laid the groundwork by defining the abstraction of domains for partitioning networks. In this section, we similarly partition the API to reflect the partitioning of the network.

Observe that the networking capacity graph of a domain $A_{ij}$ belonging to $A_i$ (defined in Section 2.5.1) can be determined using the following data.

- The networking capacity graph of each of the child domains of $A_{ij}$.
- The partial networking capacity graph of the domain $A_{ij}$.
- The child-mappings of every link of the domain $A_{ij}$.

Thus, the API to read the networking capacity graph of a network could be partitioned into a set of APIs, one for reading the networking capacity graph of each of the child domains, one for reading the partial networking capacity graph of the
parent domain, and one for reading the child-mappings of the links of the parent
domain. This partitioning can be carried out recursively down the domain hierarchy.

Figures 3.12 shows the API framework for one level of this partitioning. The
API exposed by the resource discovery algorithms is identical to that given earlier.
The reason for this is that the partial networking capacity graph of a domain may
be represented using the same data structure as that for the networking capacity
graph. This is because domains can be assigned addresses just as with label switches,
although the addresses of domains are typically shorter in terms of the number of
bytes. The excess bytes could be set to zero, but we shall not get into such syntactic
details. Similarly, a partial topology could be expressed using the same data structure
as a topology.

We now discuss the domain port mapping algorithms. Observe from Equation (2.4)
that the child mapping of every link consists of two mappings, one corresponding to
each end of the link. The two component mappings of the child mapping are called
DomainPortMap's (Figure 3.13). The interface DomainPortMapper provides all the
DomainPortMap's for a single domain. The domain port mapping algorithms expose
the interface DomainPortMapper. To find the two DomainPortMap's of a link, one
must access the interface DomainPortMapper exposed by each of the two domains
that the link connects.

The API to build a domain segment is identical to that for building a segment of a
label switch. This is because a domain was modeled like a label switch in Section 2.5.
The API is shown as the interface DomainSegmentBranch in Figure 3.13. Now, to
build a graph that spans one or more segments, a graph-building algorithm may invoke the method \texttt{build} of the interface \texttt{DomainSegmentBranch} at every domain that the graph passes through. The only exception is at the root and leaf domains of the graph, where the interface \texttt{SegmentBranch} of the mirror objects must be invoked instead, because the graph terminates inside those domains. Section 3.3 discusses graph building in more detail.
typedef SegmentBranch DomainSegmentBranch;
struct DomainPortMap
{
    Port domain_port;
    Address child_node;
    Port child_port;
};
typedef sequence<DomainPortMap> DomainPortMapList;
interface DomainPortMapper
{
    readonly attribute DomainPortMapList port_mapping;
};

Figure 3.13: API exposed by the domain port mapping algorithms.

3.1.2 APIs for Exploiting Multiplexing Gain

This appendix discusses API extensions for multiplexing-gain-enabled label switches. The new feature of a multiplex-gain-enabled label switch is the controllability of its multiplexers, and the ability to read statistics of packets passing through its multiplexers.

Multiplexers

A multiplexer consists of a set of buffers, a buffer manager and a scheduler, as shown in Figure 3.14(a). The modeling issue related to the buffers is the allocation of memory, and the modeling issues related to buffer managers and schedulers are the representation of buffer management and scheduling policies.

Memory allocation schemes for buffers have many degrees of freedom. For example, memory could be completely partitioned between every buffer or shared se-
Figure 3.14: (a) Model for a multiplexer. (b) Decomposition of scheduling policies.

A two-level decomposition, consisting of sub-scheduling policies $f$ and $g$ composed via a third policy $h$, is shown.

Deselectively between certain groups of buffers while being partitioned between groups. Moreover, the sharing could be between buffers of the same multiplexer or different multiplexers. Some schemes may place additional restrictions, e.g., though memory is shared between buffers, no one buffer may be allowed to occupy the entire memory space. It is impossible to conceive of a general framework that captures every possible degree of freedom. We find the last example to be a sufficiently general model for use in a generic programming interface. In particular, we take the following model. The total memory space $B$ on the label switch is partitioned between groups of buffers. A group may consist of one or more buffers. Within a group $i$, all buffers share the memory $M_i$ allocated to the group. However, the occupancy $q_j$ of any buffer $j$ in group $i$ may not exceed $B_{ij}$, where $B_{ij} \leq M_i$. Some label switches may not support more than one buffer per partition.

In designing the programming interface, we restrict the number of buffer parti-
tions to be fixed. This will reduce clutter in the presentation without hiding the representation technique. Indeed, the technique for handling a variable number of partitions is a recursive application of that for handling a variable number of buffers.

Buffer management policies are typically based on queue-length thresholds, above which packets are dropped. One possible policy is that which triggers packet-discard whenever a buffer occupancy exceeds its limit specified in the memory allocation model above. In order to accommodate different buffer management policies, the programming interface must be capable of revealing any specific policy used by a label switch. One technique for doing so is to create a list of known scheduling policies and assign them a numeric code. Then, the programming interface would return the code corresponding to the policy used by the label switch. Most label switches do not allow the buffer management policy to be altered, but they may permit the parameters to be altered. The parameters are represented as a sequence of bytes encoded in a policy-dependent way (cf. the union of data types in the C++ programming language).

Scheduling policies can be similarly encoded. However, the encoding could be simplified by the following observation. Many label switches support scheduling policies that are compositions of other policies such as priority scheduling (PS) or weighted round-robin (WRR) scheduling. A composite scheduling policy applies different policies to different sets of buffers and combines the output of these scheduling policies using yet another policy [2, 30] (Figure 3.14(b)). For example, taking $f$ to be WRR and $g$ to be PS results in the upper three buffers of Figure 3.14(b) being served under
WRR and the lower three buffers being served under PS. Moreover, by taking $h$ to be WRR, the bandwidth allocation between the upper three and lower three buffers can be divided in any desired ratio. Alternatively, by taking $h$ to be PS, the upper three buffers could be given priority over the lower three buffers. The example given in Figure 3.14(b) illustrates a two-level decomposition. In general, the decomposition could be carried to multiple levels in the form of a tree.

The technique of assigning codes is applied to scheduling policies that cannot be obtained via composition of other policies. Policies that can be obtained by such composition are represented as a tree of codes. The parameters of each policy are encoded in a policy-dependent way, as with parameters for buffer management policies.

Just as with buffer management policies, label switches typically do not allow the scheduling policy to be altered, but they may permit the parameters to be altered.

The programming interface for manipulating the multiplexer should provide methods for the following operations.

- Reveal the scheduling policy tree and buffer management policy.

- Set parameters for each scheduling policy in the tree, and the parameters for the buffer management policy.

- Reveal the identifier $i$ and size $M_i$ of each partition.

- Reveal whether the number of buffers is fixed or variable.

  - If fixed, reveal the identifier of each buffer, its size and the partition it
belongs to.

- Else,

  * Reveal the maximum number of buffers that could be supported per partition.

  * Reveal whether buffers could be created dynamically on the fly.

- Create and remove buffers (if supported – see above). For each buffer created, the following should be specified, and a buffer identifier should be returned.

  - The buffer size.

  - Memory partition it belongs to.

  - The node on the scheduling policy onto which it feeds. For example, in Figure 3.14(b), the buffer at the bottom would feed to the node \( g \) whereas the buffer on the top would feed to node \( f \).

  - The scheduling policy parameters corresponding to this buffer. For example, in Figure 3.14(b) if \( g \) is PS and a new buffer is added to \( g \), the priority assigned to the new buffer must be specified.

- Remove all previously created buffers.

**Statistics Measurement**

Statistics measurement is vital for measurement-based admission control. The key to successful design of an interface for collecting statistics is to include parameters that
are simple to collect and yet rich enough to be the basis for deriving more complex
statistics such as the schedulable region (Appendix A).

Most label switches available today support only the collection of packet and
byte counts, from which the average rate of a stream could be determined. This
most basic of statistics is insufficient for use in measurement-based admission control.
Typically, at least the second moment of the number of bytes arriving per unit time
is necessary for any useful computations. However, the computation of the second
moment is considered too expensive to be carried out by the hardware, since it involves
multiplication.

We seek to have the hardware compute the histogram of byte counts per unit
time. From the histogram, moments could be computed. Suppose that the count of
bytes arriving during successive time units are $c_1, c_2, \ldots$. The histogram is an array
$h$, whose $i$-th bin gives the number of elements of the above sequence that lie in the
interval $[(i - 1)w, iw - 1]$, where $w$ is the bin width.

The computation of the histogram involves computation of byte counts per unit
time and the updating of the histogram. The former requires a timer, in addition to
the counting which is already available on most label switches. The latter requires (i)
the identification of the histogram bin that is to be updated, and (ii) incrementing
the bin value by one. Suppose that $w$ is a power of 2, i.e., $r = \log_2 w$ is an integer.
Then, using the notation of the C programming language, the histogram is updated
thus

$$h[c >> r]++$$
where \( c \) is the count of bytes measured in the time unit that elapsed just before the update. In other words, the computation of the histogram requires timing and the shift operation in addition to those operations presently used in label switches for statistics gathering. However, the shift operation is the most basic of hardware operations. Thus, it is readily available. Timing is also available in label switches, though at present they are not used for statistics gathering.

Actually, it is possible to construct the histogram in software by requiring the software to read the byte counts every time unit, \( e.g. \), every 1 ms. Then there would be no need for the hardware to maintain the histogram. But, this approach is not practical due to limitations on timing accuracy of most operating systems. Moreover, this approach would result in a large quantity of traffic (requests and replies) between the software and the hardware, thus potentially slowing down the other operations of both.

The programming interface for reading statistics should provide methods for the following operations.

- Reveal the maximum number of aggregate stream identifiers (Section 2.A.2) for which measurements could be collected.

- Set and remove entries in the measurement table (Section 2.A.2).

- Set the measurement window and bin widths.

- Retrieve measurements for specific aggregate stream identifiers.
- Reset measurement counters.

- Clear all measurement-table entries.

3.A.3 APIs for Supporting External Control of Label Switches

In this section, we extend the core network resource API to include support for the case where the label switch and its switch control processor are distributed. Or more generally, for the case where the label switch and its mirror object have independent life times. This is the case, for example, in GSMP [38]. In this case, the API presented here would be a wrapper for GSMP, or whatever other protocol is used to communicate between the label switch and it switch control processor.

The main issue concerning the scenario just addressed is that the label switch and its mirror object can be restarted independently. Hence, every mirror object must take special action to ensure that its view of its label switch’s forwarding and policing tables are identical to that actually on the label switch. For this purpose, whenever a mirror object is rebooted, it must clear the content of the forwarding and policing tables of its label switch. Moreover, if the label switch is rebooted, the mirror object must be made aware of this so that the latter may re-synchronize its tables with the former.

The API is shown in Figure 3.15. The function clearForwardingTable removes all entries in the forwarding table. It is intended to be used by the mirror object to clear the forwarding table of the label switch when the mirror object boots. In
void clearForwardingTable();
void clearPolicingTable();

typedef enum {DOWN, UP} State;
void switchState(State state);
void registerMirrorObject(void (*callback_function)());

Figure 3.15: C API extension for synchronization between a label switch and its switch control processor.

this way, any lingering entries on the label switch from any previous session can be cleared. The function clearPolicingTable removes all entries in the policing table. It is used only if the QoS extension is supported by the label switch.

The function switchState is a callback function intended to be implemented by mirror objects so that whenever a label switch changes state (is powered down or restarted), the corresponding mirror object is made aware of it. In the case of GSMP, this function would be a wrapper for the adjacency protocol [38]. A mirror object can register its callback function using the function registerMirrorObject.
4

APIs FOR DISCOVERING THE NETWORK RESOURCE CONFIGURATION

4.1 INTRODUCTION

In the previous chapter, we introduced a set of APIs for use by graph building algorithms to construct graphs. One of the APIs was developed for reading the network resource configuration. This chapter presents APIs for the discovery of the network resource configuration so that it may be made available for reading. These APIs, being based on the model of Chapter 2, are valid for any network as defined there. In particular, they are valid for the examples of networks given in Section 2.7.

Traditionally, resource discovery has been accomplished by running protocols directly [61, 9]. In contrast, this chapter presents functional APIs that are accessible remotely using procedures similar to RPC [13, 20]. This approach makes programming much simpler, while being almost as efficient as direct protocols. The next chapter substantiates these claims.

The rest of this chapter is organized as follows. Section 4.2 presents the the link-
state technology for discovering the network resource configuration, and Section 4.3 presents the APIs based on this model. Section 4.4 discusses the minimality of the APIs. The chapter concludes with Section 4.5.

4.2 THE LINK-STATE TECHNOLOGY MODEL

This section presents the essence of link-state technology [11] as applied to networks defined in Chapter 2. The first part describes the basic technology and the second part gives an extension to support multiple domains.

4.2.1 Basic Technology

Link-state technology strives to maintain a replicated database of the network topology and other information about the network. Every node¹ maintains a database containing the view of the network as seen by it. The view of the network includes the topology and optionally other parameters associated with each link of the network. The networking capacity graph is such a view. Nodes obtain this view of the network by exchanging information with their neighbors. The basic unit of information in the database of each node is called a link-state advertisement (LSA). A LSA contains information about all the links and neighbors connected to a particular node, called the advertising node of that LSA. Nodes are identified by a fixed-length address. When the databases of two nodes contain LSAs of the same advertising node,

¹ In the case of a label switch, the database would be maintained by the switch control processor of the label switch, but we shall avoid mentioning this technicality.
a sequence number in the LSA is used to determine which LSA is newer. Newer LSAs replace older LSAs during information exchanges with neighbors, unless one of the two neighbors is the advertising node of the LSA. In this latter case, the LSA provided by the advertising node is considered to be newer, and the sequence number is updated to reflect this. The algorithm for updating the sequence number is not relevant for this thesis.

Synchronization of the database of every node with that of its neighbors ensures network-wide synchronization. Synchronization of neighbors must take place whenever

- A neighbor is discovered, or

- The database of one of the neighbors changes.

In the first case, differences between the databases of the two neighboring nodes is explicitly compared by the two nodes and the difference is rectified by replacing older LSAs in either node with newer ones from the other. In the second case, the differences between the databases of the two nodes is known \textit{a priori} to the node whose database just changed. This is because, before the change, the two nodes had identical databases. Thus, the differences between the two nodes can be rectified without any explicit comparison. In link-state technology, the first case is called \textit{database synchronization} and the second case is called \textit{flooding}. The operations involved in flooding are a subset of those involved in database synchronization.

Whenever the database of a node changes, the change is propagated to all its
neighbors, and by recursion, to all other nodes that have a path to that node, much like how a pebble dropped on a still pond would cause ripples to propagate throughout the pond.

The only remaining essential feature of link-state technology is the procedure for neighbor discovery. This is usually carried out via a polling mechanism known as the hello protocol. Every node periodically sends a hello message on all its ports, indicating that it is alive. Any neighbor that receives this hello message will discover this node. The actual protocol is a little more complex, involving a hand-shake. It is described in Section 4.A.1. A node discovers the loss of a neighbor when it ceases receiving the periodic hello messages from that neighbor.

This completes the description of the essence of link-state technology. A more elaborate account may be found in [11]. We now briefly discuss the reasons for the choice of the LSA as the unit of information in database exchanges. These reasons, though not necessary for the subsequent development of this chapter, are presented as they give insight into the technology.

**Requirement 4.1.** One and only one node may initiate updates of the LSA corresponding to a given node. All other nodes may only flood the updates to their neighbors, and update their own databases based on updates received from neighbors, but they may not otherwise change the content of the LSA (except its age).

If this requirement was not satisfied, then two different nodes may initiate updates for the same LSA, causing an infinitely-long ‘ping-pong’ effect since each of the nodes
would repudiate the other’s updates. Hence, the network-wide database synchronization will never complete. In fact, this ping-pong effect is one way to detect that two nodes have the same address, because in this case the two nodes will issue updates for the same LSA.

**Requirement 4.2.** *Each update of an LSA must completely replace all previous versions of that LSA. In other words, in the presence of the $n^{th}$ update, all previous updates $1, \ldots, n - 1$ become obsolete.*

This requirement is necessary since flooding does not guarantee that updates are propagated in any particular order. Thus, if the requirement was not satisfied, the content of the database of a node would depend on the order in which the node received updates, thus potentially resulting in loss of information.

### 4.2.2 Extension for Scaling

The extension of the link-state technology to multiple domains is obtained by treating domains as giant label switches (Section 2.5). Then, for each level of the domain hierarchy, a separate instance of the link-state technology would be instantiated. The extension is similar to that adopted by PNNI. The purpose of the description here is to lay the model upon which APIs can be developed.

The extension requires two issues to be addressed. They are as follows.

- Unlike a label switch, a domain is a distributed system.

- The domain hierarchy is not known *a priori* to the resource discovery software.
The implications of the first issue are as follows. Consider the LSA database of a
domain. It must be maintained at least at some SCP in the domain. We shall call
it the SCP associated with the domain\(^2\). One SCP may be associated with multi-
ple domains. Whenever a neighboring domain is discovered, the SCP representing
the domain must be made aware of the discovery so that it may initiate database
synchronization with the newly discovered neighbor. This SCP must also be alerted
whenever a neighbor is lost so that it can update the LSA database and flood the
other neighbors. For this, the SCP must expose an API.

The implications of the second issue are manifold. First, since the domain hier-
archy is not known \textit{a priori}, the identity of the SCP associated with a domain is not
known. The remaining implications of the second issue manifest themselves when we
attempt to solve the above problem.

We solve the above problem via two steps. In the first step, we assume that
for every ancestor domain of a label switch, the SCP of the label switch knows the
identity of the SCP associated with that ancestor. In the second step, we show how
to eliminate this assumption by using a discovery algorithm.

We now assume that the assumption of the first step holds. The links between
domains cannot be discovered \textit{directly} since the domain hierarchy is not known \textit{a
priori}. Hence, label switches must first discover their neighbors. Then, the domain
to which the corresponding link belongs to must be determined. Given the discovery

\(^2\)For purposes of reliability, it could also be maintained at multiple SCPs, but we
shall not consider this case.
of a link between two label switches, the domain to which the link belongs can be
determined only if the list of ancestors of each of the two label switches is known (see
Section 2.5.1). For the SCP of a label switch to determine the list of ancestors of a
neighbor, the latter must expose an API giving the list of its ancestors.

We now describe the discovery algorithm for determining the SCPs associated
with the hierarchy of domains. First, consider the nodes in a lowest level domain.
For these, the basic link-state technology of the previous section applies, and hence
there is no need for any discovery of SCPs. Now, consider the discovery of the SCP
associated with a lowest-level domain. Since every node of this domain knows the
identity of every other node in this domain, suppose by convention that the SCP with
the smallest address is chosen as the one that will maintain the LSA database of the
domain. Then, every node in the domain can locally determine the identity of this
SCP. Thus, link-state technology can be run between the lowest-level domains. As a
result, the SCPs associated with the lowest-level domains know every other domain
with a common parent as them. Hence, using the same convention as before, they
can determine locally the SCP that will represent their parent. To summarize, the
SCP of the label switches know their parents' identities, and the parents know the
identities of their parents (the grandparents of the label switches). Thus, the SCPs
representing the label switches can determine the identities of their grandparents by
communicating with their parents. For this, the parents must expose an API giving
the list of their ancestors. In this way, the whole domain hierarchy can be discovered
progressively, thus removing the assumption made in the first step above.
4.3 THE APIs

In this section we present the APIs for flooding and database synchronization. For neighbor discovery, the hello protocol is still better than a functional API, and hence we do not provide a functional API for it. The hello protocol is but a small part of link-state technology, similar to the situation in operating systems where the bulk of the code is written in a high-level programming language while a small part is written in assembly language. The protocol is given in the appendix to this chapter in a form applicable to the network model of Chapter 2.

4.3.1 API for Obtaining the Link States

In order for a node to generate its LSAs, it must obtain information about its links. In particular, we are interested in obtaining information about link connectivity, schedulable regions and operating points. Figure 4.1 provides an API for this.

The interface LinkState provides methods for reading all the link state except the connectivity status which is given by the interface LinkConnectivity. The interface LinkState is intended to be exposed by the mirror object of a label switch (see Figure 3.1), and the interface LinkConnectivity is intended to be exposed by the label switch. The reason for this is that changes in link connectivity can be detected by the label switch (hardware) much more quickly than it can be detected by the software.

The read-only attribute the_ports gives the list of ports on the label switch. The
interface Alert
{
    void neighborConnectivityChange(in ULong port);
    void srChange(in ULong port);
    void opptChange(in ULong port);
};

typedef sequence<Port> PortList;
interface LinkState
{
    readonly attribute PortList the_ports;
    ULongSeq getSchedulableRegion(in ULong port);
    ULongSeq getOperatingPoint(in ULong port);

    void registerToReceiveAlerts(in Alert receiver,
        in UShort sr_change_threshold,
        in UShort oppt_change_threshold);
};

interface LinkConnectivity
{
    boolean getConnectivityStatus(in ULong port);

    void registerToReceiveAlerts(in Alert receiver);
};

Figure 4.1: The interfaces for obtaining the link states.
methods `getSchedulableRegion` and `getOperatingPoint` return the schedulable region and operating point of the specified port. The method `getConnectivityStatus` returns `true` if the specified port is bidirectionally connected to another port (of either the same node or a different node), `false` otherwise. Both interfaces `LinkState` and `LinkConnectivity` provide means by which client objects can register to receive alerts when the link state changes by a specified threshold. In order to receive such alerts, the client object must implement the interface `Alert`. A client object registers itself by calling `registerToReceiveAlerts` with the argument `receiver` as the reference to the client object. The two thresholds in the argument list specify the minimum percentage change in the schedulable region or operating point required for an alert to be triggered\(^3\).

### 4.3.2 API for Database Synchronization and Flooding

The architectural reference for the flooding and database synchronization API is shown in Figure 4.2. The mirror objects have been omitted from the figure as they do not take part in database synchronization or flooding. The API for database synchronization and flooding is shown in Figures 4.3–4.4. It is exposed by the resource discovery objects. The method `getLSASummaryList` returns the headers of all LSAs in the resource discovery object. It is intended to be used for comparing the database of a newly discovered neighbor. The LSA header contains the address of the advertising

\(^3\)The percentage change of a schedulable region or operating point may be defined as the largest of the percentage changes in each dimension. Other definitions are also possible.
node of the LSA and a sequence number for the purpose of comparing it with another LSA with the same advertising node address. The LSA header also contains an age-field indicating how long ago this LSA was generated by the advertising node. This field is used to discard LSAs that are very old. The contents of the LSA headers are adequate to compare the database of a node with that of its neighbor, as discussed in Section 4.2.

The method getLSAs returns the list of LSAs corresponding to the specified advertising nodes in the resource discovery object. It is intended to be invoked on a newly discovered neighbor for the purpose of database synchronization. The method getLSASSummaryList permits a node to compare its database with that of its newly discovered neighbor, whereas the method getLSAs permits a node to get LSAs from the neighbor's database so that the two can be made identical.
typedef octet Address[<len>];
typedef sequence<Address> AddressSeq;

struct LinkInfo
{
    Address remote_node;
    ULong local_port;
    ULong remote_port;
    ULongSeq schedulable_region;
    ULongSeq operating_point;
};

typedef sequence<LinkInfo> LinkInfoSeq;

struct LSA
{
    Address advertising_node;
    ULong ls_sequence_number;
    ULong ls_age;
    LinkInfoSeq links;
};

struct LSAHeader
{
    Address advertising_node;
    ULong ls_sequence_number;
    ULong ls_age;
};

typedef sequence<LSA> LSAList;
typedef sequence<LSAHeader> LSASummaryList;

Figure 4.3: The IDL structures for database synchronization and flooding.
interface NeighborSynchronization
{
    LSASummaryList getLSASummaryList();
    LSAList getLSAs(in AddressSeq advertising_nodes) raises(InvParam);
    oneway void recvLSAs(in Address sender, in LSAList lsa_list);
};

Figure 4.4: The IDL interface for database synchronization and flooding.

The method recvLSAs is intended to be used by a node to instruct a neighbor to accept the specified LSAs as floodings. When a node A invokes this method on a node B, node B would take the lsa_list in the input parameter list as a flooding from the node whose address is sender⁴. The method is indicated as oneway. The semantics of this indication is taken to be 'exactly-once' invocation.

4.3.3 APIs for Scaling

The architectural framework for scaling the link-state technology is given in Figure 4.5. It consists of, in addition to the resource discovery objects associated with each label switch, a resource discovery object associated with every domain (shown

⁴Note that this API allows node A to impersonate another node by setting the sender to be the address of some other node. This problem can be solved by an ORB as follows. Suppose that the ORB is such that objects could be made to listen for requests coming via only a specified set of ports and label vectors. Moreover, different objects may listen on different ports and label vectors. Now, IDL requests coming from different neighbors will arrive on different label vectors (see meta segments in Section 4.A.1). Thus, by providing proxy front-end objects – one corresponding to each neighbor – the identity of the neighbor making the IDL invocation can be determined. Thus, the argument sender could be removed from the method recvLSAs.
Figure 4.5: APIs for scaling the resource discovery algorithm.

on the top left and top right in the figure). Each of the resource discovery objects exposes the interface NeighborSynchronization given in the previous section (shown as API 1 in Figure 4.5). This interface is used for executing the link-state technology at each level in the domain hierarchy. In addition, as discussed in Section 4.2.2, each resource discovery object exposes the interface DomainHierarchy (Figure 4.6) for reading the list of ancestors of its label switch or domain (shown as API 2 in Figure 4.5). Each resource discovery object associated with a domain exposes the interface DomainSynchronization (Figure 4.6) for alerting it to the discovery of neighboring domains (addNeighbor) and to the discovery of loss of a neighbor (removeNeighbor), shown as API 3 in Figure 4.5.
interface DomainHierarchy
{
    readonly attribute AddressSeq ancestors;
};
interface DomainSynchronization
{
    void addNeighbor(in Address my_switch_address, in ULong my_port,
                      in Address neighbor_address, in ULong neighbor_port)
        raises(InvalidParameter, NeighborAlreadyExists);
    void removeNeighbor(in Address my_switch_address, in ULong my_port)
        raises(InvalidParameter, NeighborDoesNotExist);
};

Figure 4.6: The APIs for the multidomain extension.

4.4 Minimality

In this section, we discuss the minimality of the APIs shown in Figures 4.1 and 4.4. These two APIs by themselves are not minimal with respect to the API for reading the networking capacity graph (Figure 3.8). To make a statement about minimality, we need to consider neighbor discovery as well.

Neighbor discovery is presented in the appendix to this chapter in terms of a protocol rather than a functional API. However, for the purposes of demonstrating minimality, we may consider a protocol to be a very low-level API. Indeed, nothing in the definition of minimality specifies the syntax or realization of APIs. Thus, with this view in mind, we state the following conjecture.

**Conjecture 4.1.** The collection consisting of (i) the methods of the interface NeighborSynchronization, (ii) the attribute the ports and the methods
getSchedulableRegion, getOperatingPoint of the interface LinkState, and (iii) the protocol for neighbor discovery, represents a minimal basis for the attribute of the interface NetworkingCapacityGraph.

Figure 4.7 illustrates the conjecture. The justification of this conjecture depends on the proof that link-state technology has no redundancies in its algorithm. Such a proof would take us too far afield. Hence, we omit the proof of the conjecture and leave it open for further work.

4.5 CONCLUDING REMARKS

In this chapter, we have summarized the essence of link-state technology, and from it derived a set of interfaces that can be used to realize all distributed interactions in link-state technology except the hello protocol. The hello protocol does not fall into the framework of function calls and hence the introduction of an interface for it
would (i) go counter to the idea of capturing the bare essence, and (ii) complicate its realization. This is similar to the case of operating systems which are almost entirely implemented in a high-level language except for a small part which is in assembly. The chapter also provided extensions to the interfaces for supporting scaling of the link-state technology. Finally, the chapter postulated the minimality of the above interfaces with respect to the interface for reading the network resource configuration.
4.A Appendix

4.A.1 Neighbor Discovery

In this subsection we present the protocol for neighbor discovery. The essence of it is the same as the hello protocol in OSPF and PNNI. However, the protocol presented here is valid for any network captured by the model of Chapter 2.

Meta Segments

The hello protocol involves the exchange of hello-messages between neighboring SCPs. Thus, the first step is to establish communication channels (graphs) between neighbors. This is achieved in a completely distributed fashion by creating meta segments. A meta segment is a segment that a switch control processor creates on its label switch from a specific port to the control port, and vice versa. The net effect of each switch control processor creating meta segments for all its ports is that there is a unicast graph from every switch control processor to all of its neighbors. We now make the above more precise.

Consider Figure 4.8. Suppose that the switch control processor of each label switch $X$ ($X = A$ or $B$) sets up a pair of segments with the following forwarding table entries for every port $i$ of the label switch $X$.

1. $(i, x(i)) \rightarrow (p_X, y(X,i), ID_X)$

2. $(p_X, y(X,i)) \rightarrow (i, x(i), ID(X,i))$
Figure 4.8: Meta segments.

where $p_X$ is the control port of label switch $X$, $y(X, i)$ is some label vector chosen by the switch control processor of $X$, $ID_X$ is the identifier of some buffer in the multiplexer connected to port $p_X$, and $ID(X, i)$ is the identifier of some buffer in the multiplexer connected to port $i$. The label vectors $y(X, i)$ must be chosen such that if $i \neq j$, then $y(X, i) \neq y(X, j)$. The label vector $x(i)$ is taken to be a well-known label vector, determined only by the port-type vector of port $i$, i.e., it depends only on whether the port $i$ is ATM, IP, SONET, etc.

The above construction of meta segments (which requires no interaction between switch control processors) effectively results in a unicast graph between the label switches A and B in both directions. This is because if the SCP of A sends a packet with label vector $y(A, A2)$, it will traverse the following virtual links and segments to arrive at the SCP of B.

• Virtual link corresponding to label vector $y(A, A2)$ on the link connected to the
control port of A.

- Meta segment \((A4, y(A, A2)) \rightarrow (A2, x(A2), ID(A, A2))\) on label switch A.

- Virtual link corresponding to label \(x(A2) = x(B3)\) between port A2 of label switch A and port B3 of label switch B.

- Meta segment \((B3, x(B3)) \rightarrow (B4, y(B, B3), ID_B)\) on label switch B.

- Virtual link corresponding to label vector \(y(B, B3)\) on the link connected to the control port of B.

The SCP of B can determine that the received packet is from the neighbor connected to port B3 since the \(y(B, i)\) are different for different \(i\). Similarly, any packet sent by the SCP of B with label \(y(B, B3)\) will arrive at the SCP of A, and A can determine that the packet arrived from the neighbor connected to port A2.

**The Hello Protocol**

Now that the mechanism for communication between neighbors has been demonstrated, we focus on the communication itself, which begins with the hello protocol for discovering neighbors. The hello protocol consists of exchanges of messages between neighbors. The messages are called hello messages. A hello-message consists of the following:

- \(A_s\): Address of the node sending the hello-message.
• $A_r$: Address of the node receiving this message, if known to the sender, a null value otherwise.

• $p_s$: Port on which the sender sends this hello-message.

• $p_r$: Port on which the receiver receives this message, if known to the sender, a null value otherwise.

When an SCP of a label switch sends a hello message, $A_s$ is taken to be the address of the label switch and $p_s$ is taken to be the port of the label switch through which the message leaves the label switch. This is because the SCP is considered to send the hello message on behalf of the label switch. For example, in Figure 4.8, a hello message sent by SCP A with label vector $y(A, A2)$ would leave label switch A on port $A2$. Thus, $A_s$ would be set to A and $p_s$ would be set to A2 on this hello message. Similarly, if the neighbor is a label switch, then $A_r$ is taken to be address of the neighboring label switch and $p_r$ is taken to be the port of the neighboring label switch on which it connects to the local label switch. Thus, in the above example, if $A_r$ is not null, then $A_r$ would be set to $B$ and $p_r$ would be set to $B3$.

Every node must send out a hello packet periodically on each of its ports. The period is known as the hello interval. With respect to the hello protocol, each port of a node can be in one of three states: down, one-way and neighbor-discovered (also called two-way). Whenever a node receives a hello-message $(A_s, p_s, A_r, p_r)$ on one of its ports, it must interpret the content as follows. Denote the address of the receiver node by $X$ and the port on which it receives the packet by $p$, and suppose that the
hello protocol on port \( p \) is in state ‘down’.

- If \( X = A_r \) and \( p = p_r \), then the neighbor \( A_s \) is considered to be discovered on port \( p \), and the state of port \( p \) is set to ‘two-way’.

- If \( A_r \) and \( p_r \) are null, then the neighbor \( A_s \) is considered not discovered on port \( p \). The state of port \( p \) is set to ‘one-way’.

- If neither of the above two conditions holds, then the state of the port remains ‘down’.

In addition to the above actions, label switch \( X \) must retain a copy of the values of \( A_s \) and \( p_s \) so that it can use them in future hello-messages it sends.

Once a neighbor is discovered (two-way), it remains discovered as long as the receiver node \( X \) continues to receive hello messages from the discovered neighbor at intervals not exceeding a specified value (known as the dead interval), and the received hello messages satisfy the first bulleted condition above. Otherwise, the state is set to ‘down’.

Once a port enters the state ‘one-way’, it remains in that state as long as the receiver node \( X \) continues to receive hello messages from the discovered neighbor at intervals not exceeding the dead interval, and the received hello messages satisfy the second bulleted condition above. If instead, the received message satisfies the first bulleted condition, then the state of the port is set to ‘two-way’. If no message is received for a period exceeding the dead interval, or if the received message does not
satisfy either of the first two bulleted conditions, then the state of the port is set to 'down'.

In addition, a port in either state 'one-way' or 'two-way' will be changed to state 'down' if the label switch hardware detects the connectivity on the port to be lost. The API for the notification of loss of connectivity was given in Section 4.3.1. This completes the description of the hello protocol state machine.

Race Condition

We now address a possible race condition that may occur during database synchronization, and how it can be avoided. Consider Figure 4.9. Label Switches A and B are neighbors, and so are label switches B and C. Suppose that the link between label switches A and B is just being discovered by the two label switches, and the link between label switches B and C has been discovered already and the database synchronization between B and C is complete. As a result of the neighbor discovery between A and B, label switch A will read label switch B's LSA summary list. Suppose that C floods an LSA to B after A reads B's LSA summary list. This could result in B not forwarding the LSA to A unless B has listed A in its list of flooding recipients. The following proposition shows how to avoid the race condition:

**Proposition 4.1.** If label switches flood LSA updates via every port that is not in state 'down' at the time of the flooding, then the race condition mentioned above cannot occur.
Figure 4.9: Illustration of a possible race condition.

The justification is as follows. Consider Figure 4.9 again. Note that label switch A begins database synchronization only after the port connecting A to B is two-way. If this port is considered two-way, then label switch A must have received a hello-message from label switch B with no null parameters (see Section 4.A.1). But, this could occur only if label switch B had already received a hello-message from label switch A, and hence label switch B considered the port connecting B to A to be either one-way or two-way (but not down).
5

BENEFITS OF THE FOUNDATIONS

5.1 INTRODUCTION

The previous chapters laid the foundations of network programmability. We now analyze the benefits arising from these foundations. In particular, we look at the conceptual clarity and simplicity of programming.

The rest of this chapter is organized as follows. Section 5.2 discusses the conceptual clarity resulting from the foundations, and Section 5.3 discusses the programming simplicity resulting from the foundations. The chapter concludes with some remarks in Section 5.4.

5.2 CONCEPTUAL CLARITY

Conceptual clarity arises from the bare-bones of network control that is captured by the model of Chapter 2 and the APIs of Chapter 3. The model aimed at concision by removing all extraneous components. In this section, we present two results that enhance the conceptual clarity. The first is the observation that network control can be formulated concisely as a control-theoretic or game-theoretic problem in a state
space. The second is the identification of the fundamental capabilities required of a network for realizing a network control system.

5.2.1 State-Space Formulation

Network control is the manipulation of graphs, i.e., the creation and removal of communication channels across the network. These communication channels are decomposed into segments and virtual links, as modeled in Section 2.4.2. However, as identified in Section 3.1, the virtual links are not software-controlled in our model. Thus, the only parameters under the control of the software are the segments. Hence, given a specific network topology, the collection of segments is what determines the collection of communication channels across the network.

**Definition 5.1.** The state of a communication network consists of the collection of segments on each network element.

The topology of the network is the space on which the state is defined.

Given a request to create a graph specified by its root, leaves, thickness and color, the network control system must determine a geometry for the graph, and create segments along the vertices of the geometry (Section 3.1). Given a request to remove a graph, the network control system must retrieve the graph geometry and remove the segments of the graph. Hence, network control may be viewed as a control operation on the state space consisting of the collection of segments in the network. This framework becomes game-theoretic when one realizes that requests to the network
control system are made by multiple users.

5.2.2 Fundamental Capabilities

In this section, we list two fundamental capabilities requisite of switch control processors in order to realize a network control system. A third fundamental capability is identified as being necessary to enhance the performance of a network control system. These capabilities are fundamental because they cannot be derived from any of the other capabilities (APIs) listed in this thesis.

The first two fundamental capabilities relate to the control of label switches and the transmission of data. They are grouped together as they both apply to switch control processors.

**Fundamental Capability 5.1.** A switch control processor is capable of

(i) *Invoking the resource-manipulation API shown in Figures 3.2–3.4.*

(ii) *Sending packets with any label vector that matches the port-type vector of its output terminal; it is capable of receiving packets from its input terminal and distinguishing them based on the label vector.*

The third fundamental capability relates to the receipt of instantaneous alerts from a label switch when the link connectivity of a port on the label switch changes. Although such an alert can be derived using Fundamental Capability 5.1 (by writing an algorithm such as the hello protocol), it will not be instantaneous. In order to
satisfy the requirement that the alert be instantaneous, a new fundamental capability is required.

**Fundamental Capability 5.2.** *A label switch is capable of sending an alert to its switch control processor when the link connectivity on any of its ports changes.*

We now demonstrate that Fundamental Capability 5.1 is necessary and sufficient for network control. We first make a few assumptions about the network. We assume that the capacity of every interconnection device is strictly greater than zero, and that the number of label vectors available on any port of a label switch is at least as large as the number of ports on that label switch. Moreover, we assume that a switch control processor knows the port-type vector of its own port, and that the topology of the network is quasi static, i.e., the topology does not change rapidly compared to the propagation time of a packet in the network.

**Lemma 5.1.** *A switch control processor that supports Fundamental Capability 5.1 can determine the identity of the control port of its label switch by executing an appropriately chosen algorithm.*

The control port may be determined as follows. Retrieve the identities and port-type vectors of all the input-output port pairs (Fundamental Capability 5.1(i)). Consider the ports of the label switch that have the same port-type vector as the port of the switch control processor. Order them arbitrarily as $p_1, \ldots, p_n$. This set of ports is non-empty since the control port must be a member of it. Let $X, Y_1, \ldots, Y_n$ be
distinct valid uni-dimensional label vectors for the port of the switch control processor. Create the following set of entries, one for each input-output port pair $p_i$ (Fundamental Capability 5.1(i)).

- $(p_i, X) \rightarrow (p_i, Y_i, ID_i)$.

where $ID_i$ is the ID of some buffer on port $p_i$. Transmit a packet with label vector $X$ on the output terminal of the switch control processor, and listen for a packet on the input terminal of the switch control processor for all label vectors $Y_1, \ldots, Y_n$ (Fundamental Capability 5.1(ii)). Exactly one packet should arrive on the input terminal of the switch control processor. The label vector of this packet will be $Y_j$ for some $j \in \{1, \ldots, n\}$. Then, the control port of the label switch is determined to be $p_j$.

**Lemma 5.2.** *Fundamental capability 5.1 is sufficient for switch control processors to discover the networking capacity graph of a lowest-level domain of a network, based on the peak-rate approximation to the schedulable region.*

The peak-rate approximation to the schedulable region of each port of a label switch can be computed at the respective switch control processors since the list of ports and capacities is available from Fundamental Capability 5.1(i). Hence, what remains to be shown is that the link-state technology described in the previous chapter could be executed in a network that supports the fundamental capability. Indeed, as can be easily checked, the only requirement for link-state technology to be feasible in a lowest-level domain of a network is that any two neighbors be able to communicate
with each other, and that any two nodes in the network have a path connecting them. The second condition is satisfied by the definition of a domain. That the first condition is also satisfied is seen as follows.

Due to Fundamental Capability 5.1(i) and Lemma 5.1, switch control processors can setup meta segments as discussed in Section 4.A.1. As a result of the meta segments, there is a graph from every node to each of its neighbors (see Section 4.A.1). Now, due to Fundamental Capability 5.1(ii), neighboring switch control processors can send and receive messages to and from each other, thus establishing communication with each other.

**Lemma 5.3.** *Fundamental capability 5.1 is sufficient for any two switch control processors in the same lowest-level domain to communicate with each other.*

The justification of Lemma 5.2 established that neighboring switch control processors can communicate with each other, and discover the topology of the domain. In order for a switch control processor to send a message to an SCP that is not a neighbor, it must forward the message to a neighbor, along with the address of the intended recipient SCP. The neighbor would then forward this packet one step closer to another SCP until the message reaches the intended SCP. This is of course the familiar IP packet forwarding algorithm, of which we shall say no more.

In the following two theorems, we impose the constraint that an SCP may not use error codes of a function to determine the resources on a label switch. In order to keep the statement of the theorems compact, we have extracted this constraint here.
Theorem 5.1. Fundamental capability 5.1(ii) and the two functions getPortConfigurations and addForwardingTableEntry of Fundamental capability 5.1(i) are necessary and sufficient for the switch control processors to discover the networking capacity graphs and partial networking capacity graphs based on the peak-rate approximation.

Necessity: In order for the switch control processors to discover the topology, they must be able to communicate with each other over some graph. In the absence of such communication, they do not have any means of gaining any knowledge of the topology (except via human intervention or other external mechanisms). Thus, switch control processors must be able to send and receive messages in order to discover the topology. Hence the need for Fundamental Capability 5.1(ii). The capability to send and receive messages, however, must be augmented by the capability to create graphs over which these messages are to be transported. These graphs require meta segments to be created. Hence, the need for the functions getPortConfigurations and addForwardingTableEntry. The port-type vector returned by getPortConfigurations is necessary to determine the well-known labels used for meta segments (see Section 4.A.1). The port capacities are necessary to compute the peak-rate approximation to the schedulable region.

Sufficiency: The case of a single domain was shown in Lemma 5.2. The more general case of multiple domains is shown inductively. Suppose that there is a link between two domains $A$ and $B$, and that the topology and other resources of these
two domains have been discovered by switch control processors in their respective domains. Denote the two label switches that the link connects by \( a \) and \( b \), where \( a \) is in domain \( A \) and \( b \) is in domain \( B \). The switch control processors of \( a \) and \( b \) would detect each other via the hello protocol, which is feasible via the meta segments setup up by each switch control processor on its label switch. Through the meta segments, the two switch control processors can communicate with each other. As a result, if \( a' \) and \( b' \) are any two switch control processors in domains \( A \) and \( B \) respectively, then they can communicate with each other. This is because, \( a' \) can communicate with \( a \), and \( b' \) can communicate with \( b \) (due to the inductive hypothesis), and the communication between \( a \) and \( b \) was just established. Hence, using a mechanism similar to IP packet forwarding, \( a' \) and \( b' \) can communicate with each other. Thus, any two nodes in the two domains that need to carry out database synchronization and flooding can do so.

**Theorem 5.2.** *Fundamental capability 5.1 (including (i) and (ii)) is necessary and sufficient for the switch control processors to create and remove graphs with QoS on demand.*

*Necessity:* In order to create graphs with QoS, forwarding and policing table entries must be created, and the label ranges and capacities of ports must be known. The port-type vectors also must be known, since the graph creation requires the topology to be discovered; this requires the port-type vectors to be known (see the justification of the previous theorem). In the case of TDM and WDM, label weights
must be known (for admission control). In order to remove graphs, forwarding and policing table entries must be removed. Thus, the need for Fundamental Capability 5.1(i).

In order for the switch control processors to coordinate and setup graphs, they must be able to communicate with each other (over meta segments). In the absence of such communication, they do not have any means of setting forwarding table entries on multiple label switches, which are distributed throughout the network (except via human intervention or other external mechanisms). Thus, switch control processors must be able to send and receive messages in order to create and remove graphs. Hence the need for Fundamental Capability 5.1(ii).

**Sufficiency:** The sufficiency of Fundamental capability 5.1 was outlined in Section 3.3.

The results of this section can be augmented to cover multiplexing gain, by including the extensions given in Chapter 2.

In addition to the above three fundamental capabilities, it is desirable to explore whether additional fundamental capabilities are needed for supporting redundancy in telecommunication networks, *e.g.*, redundant forwarding fabrics or links. If so, the model presented in the previous chapter may have to be augmented. We leave this task for future research.
5.3 Simplicity of Programming

In order to highlight the simplicity of programming that arises from the foundations, we make a comparative study of two realizations of network resource discovery. One realization is based on the database synchronization and flooding APIs of the preceding chapter and the other realization is based on the traditional protocol approach. The latter is an extension of OSPF [61] to run on networks with point-to-point links only, and distribute the networking capacity graph. We shall refer to it as the protocol-based solution, and the former realization as the ORB-based solution. The comparison is carried out in terms of state-space size and source-code clarity. This section also shows that the simplicity of programming achieved as a result of the foundations does not come with any significant cost in performance.

5.3.1 General Discussion of ORBs and Protocols

Before presenting the comparison of the two realizations, we give a general discussion of ORB-based solutions and traditional protocol-based solutions.

The use of middleware in the form of CORBA [67] and related distributed-object concepts has been thought to be a bulky component of programmable networks. Often, the complaint about network programmability is the large size and lack of performance of the middleware. In this section, we show that a thin RPC-like ORB is adequate for realizing a programmable network. In particular features such as dynamic types, dynamic invocation, naming services, event services and the other
CORBA services are not necessary for realizing a programmable network. In fact, this section shows that an ORB may be viewed as a wrapper for network control protocols. In other words, the ORB simply hides or abstracts the tasks performed by the network control protocol so that these tasks are not visible to the programmer. But, the net effect as seen by the bit-stream on the “wire” is the same in the protocol approach and the ORB-based approach, except for a small bandwidth overhead in the latter. Thus, we do not view the use of an ORB as a fundamental paradigm shift, but simply as a “compiler” for writing network control protocols. This is similar to the difference between C++ and Assembly language. The assembly instructions generated by a C++ compiler may be somewhat less efficient than a program directly written in assembly, but otherwise the two are very similar.

In this section, we present descriptive APIs for a simple ORB that can be used in network control. A concrete API based on the CORBA standard is given in Appendix 5.A.1. This section also demonstrates how the ORB can be used as a wrapper for protocols.

We begin by summarizing how protocols operate. When an object in a distributed system wishes to communicate with a remote object using a protocol, it goes through the following general steps.

1. Identify the location of the target object.

2. Setup a communication channel between the two objects, or use the communication channels setup asynchronously by an IP-type routing system.
3. Send a message to the remote object. This may involve message fragmentation and flow control.

   • Fragment the message if it is larger than the maximum transfer unit (MTU) of the protocol.

   • Send a few fragments (up to the window size of the flow control protocol)

   • When fragments are acknowledged by the remote object, send more fragments until all the fragments have been sent.

4. Wait for the reply from the remote object. This may involve

   • Message demultiplexing since the object may be communicating with multiple remote objects simultaneously.

   • Reassembly of fragments. Fragments must be kept in a buffer until all the fragments of the message have arrived.

5. If a reply was not received within a certain duration, then go back to step 3.

We now discuss how an RPC-like ORB can be used for object-to-object communication in network control. Step 1 is typically performed by a naming service in CORBA. But, we do not need such a naming service, as will be demonstrated now.

Observe that the fundamental means of addressing (locating) an object in a distributed system is by the following triple:

\(^1\)The concept of objects discussed here limits any given object to be confined to a single process within a single execution machine.
1. Address of the machine on which the object executes

2. The ID of the process in the above machine in which the object executes

3. A unique identifier of the object inside the above process (called the object key\(^2\)).

We shall refer to any triple specifying the above as an object identifier. The object identifier of an object may be learned in one of three ways:

- Known \textit{a priori},
- Read from a naming service, or
- Discovered through the network.

Most distributed systems rely on the first two methods, whereas most network control protocols rely on the last method. More precisely, network control protocols discover the target machine's address (using a hello protocol) but know the process ID and object key of the target object \textit{a priori}. The process ID is typically taken to be a well-known TCP port, UDP port, IP protocol number, ATM VPI/VCI, (or, in general a label vector) on which the target object listens for incoming messages. The reason for this is as follows. All objects in a network control system are associated with either a node or a domain. Thus, the object key and process ID can be determined based on the functionality of the object, instead of looking up a name service. And, as stated before, the target machine’s address is discovered by a hello protocol or the like. The ORB is not needed for this.

\(^2\)Strictly speaking, this should be called an \textit{interface instance key}, but we shall retain the common terminology used by the community.
Step 2 is adapted to the ORB-based solution as follows. In the case of a communication between objects residing on neighboring switch control processors, meta segments must be setup independently by each SCP as discussed in Section 4.A.1 of the previous chapter. The meta segments automatically form communication channels between neighbors as discussed in the previous chapter. The ORB is not needed for this. The case of communication between objects residing on non-neighboring machines will be addressed shortly.

An RPC-like ORB can accomplish steps 3–5 so that these steps are hidden from the programmer. All that we need from an ORB are the following features:

- Means for exposing object interfaces for remote access.

- Method invocation on remote object interfaces identified by an object identifier.

The above addresses steps 1–5 for communication between objects residing on neighboring machines. Using the above as a base case, communication between objects on non-neighboring machines can be derived as follows. Before communication can take place between SCPs that are not neighbors but are in the same domain, the network resource configuration must be discovered as discussed in Lemma 5.2. Then, either the graph builder can be used to setup a communication channel between the two SCPs, or IP-style message forwarding can be accomplished as illustrated in Lemma 5.3. Theorem 5.1 showed how SCPs in different domains can communicate with each other.

We have now addressed all the five steps needed for two remote objects to com-
municate with each other. Step 1 and the base case of Step 2 do not use the ORB. The recursive case of Step 2 uses the base case and Steps 1,3–5. Therefore, in effect, all uses of the ORB can be traced down to Steps 3–5. Thus, the ORB is used merely to translate API calls into protocol messages. No other ORB services are used. This completes the description of how an ORB may be viewed as a wrapper for network control protocols. A concrete example is given next.

5.3.2 State Space and Source Code Comparison

We now return to the comparison of the two realizations of link-state technology. In this subsection, we show that the ORB-based solution requires a much smaller state space and is much shorter and simpler to understand than the protocol-based solution. We begin with an overview of the differences that result in the simplifications.

Link-state technology involves several iterations in the communication between two neighbors — hello protocol, summary list exchanges, database synchronization, each of which involves flow control so that no neighbor is overwhelmed (see [9, 61]). Each iteration of the communication is asynchronous in that once a message is sent out, the process has to wait for an interrupt (or equivalent mechanism) to be notified of a reply from the neighbor. Alternatively, the process can poll for a reply.

By contrast, in the ORB-based solution, the ORB hides all asynchronous exchanges with neighbors, except for the hello-protocol. All required information can be obtained from the neighbor by invoking a remote API on the neighbor. In other words, API calls are blocking calls and hence all asynchronous operations are taken
care of by the ORB. Moreover, such operations as retransmissions and bit-error detection/correction are taken care of by the ORB and the layers below it. The result of this is that the states associated with the protocol are hidden behind the ORB. Furthermore, some states associated with program execution can also be hidden.

The protocol-based solution requires the following seven states [9]:

1. **Down**
2. **Init**
3. **2-Way**
4. **ExStart**
5. **Exchange**
6. **Loading**
7. **Full**

In contrast, in the ORB-based solution, the number of states required is three: 'down', 'one-way' and 'two-way' (corresponding to the three states 'Down', 'Init' and '2-Way' of the protocol-based solution). The reason for this is that only these three states are associated with the hello protocol. The other four states are associated with database synchronization, which is carried out via the ORB. Thus, in the ORB-based solution, these four states are partly hidden behind the ORB, and partly made implicit by being incorporated into the program counter. This is because, in a synchronous programming model, operations can be written in sequential order. No explicit state needs to be maintained as to how much of the operation has been completed. The state is given implicitly by the location of execution in the code. However, in an asynchronous programming model such as those of many protocols, the location in the code does not determine the state of the conversation between two neighbors.
LSAList_var lsa_list;
char *neighbor_ior = create_ior(machine_addr, process_id, obj_key);
try
{
    CORBA::Object_var obj = orb->string_to_object(neighbor_ior);
    Neighbor_var my_neighbor = Neighbor::narrow(obj);
    lsa_list = my_neighbor->getLSAs(advertising_nodes);
}
catch(CORBA::Exception &e)
{ /* error processing */}
delete [] neighbor_ior;

Figure 5.1: Client-side C++ code for the ORB-based example.

As a concrete example, consider the code necessary to read a list of LSAs from a
neighbor. In the case of the ORB-based solution, this can be accomplished as shown
in Figures 5.1–5.2. The notation used is based on the C++ mapping of CORBA (see
[66]). The function create_ior creates a “stringified” IOR (see [67]) from the spec-
ified object identifier (machine address, process ID, object key). Protection against
multiple threads for the server side is not shown.

In the protocol-based solution, the above is accomplished as shown in Figures 5.3–
5.5.

Observe that the client side of the protocol-based solution requires the following
to be maintained as state: retransmission list and timer, received packet sequence
number, and the fact that we had sent a request message. In contrast, the ORB-based
solution on the client side requires no state, retransmission lists or retransmission
LSAList* Neighbor_i::getLSAs(const AddressSeq &advertising_nodes)
{
    unsigned int num_req_nodes = advertising_nodes.length();
    LSAList* lsa_list = new LSAList;
    lsa_list->length(num_req_nodes);

    for(unsigned int i = 0; i < num_req_nodes; i++)
    {
        Iterator it = m_lsa_list.find(advertising_nodes[i]);
        if(it == m_lsa_list.end())
        {
            delete lsa_list;
            throw InvParam(i);
        }

        (*lsa_list)[i] = it->second;
    }

    return lsa_list;
}

Figure 5.2: Server-side C++ code for the ORB-based example.
packet = prepare_lsa_request_packet();
send_packet(packet);
retransmission_queue.insert(packet);
return;

// The following is called by the receiver thread (Figure 5.5).
void process ls_update(const Packet &packet) {
    *
    * Check the packet to see which neighbor sent this packet.
    * Retrieve local records to see if we are expecting an
      update packet from this neighbor, and if so retrieve the
      sequence number associated, and remove the packet from the
      retransmission_queue.
    * Check if the sequence number on the received packet corresponds
      to the expected sequence number as given by the local records.
    * If so, send the neighbor an acknowledgement for this packet,
      and update the local records with the sequence number expected in
      the next packet to be received.
    * Otherwise, re-send a request packet.
}

Figure 5.3: Client-side C++ code outline for the protocol example.
// The following two functions are called by the receiver thread
// shown in Figure 5.5.
void process_ls_req(const Packet &packet)
{
    * Make copy of the requested LSAs.
    * Send as many LSAs as possible in one protocol data unit.
    * Keep a record of which packets were sent, and the sequence number
      associated with the packet.
}

void process_ls_ack(const Packet &packet)
{
    * Check the packet to see which neighbor sent this packet.
    * Retrieve local records to see if we are expecting an
      acknowledgement packet from this neighbor, and if so retrieve the
      sequence number associated.
    * Remove the entries corresponding to this sequence number from the
      list copy made in process_ls_req (see above).
    * Transmit a packet with as many unsent LSAs as possible in a new
      protocol data unit.
}

Figure 5.4: Server-side C++ code outline for the protocol example.
Retransmission Thread:

    while(1)
    {
        sleep_until_there_is_something_to_retransmit();
        for(int i = 0; i < retransmission_queue.length(); i++)
        {
            packet = retransmission_queue[i];
            send_packet(packet);
        }
    }

Receiver Thread:

    while(1)
    {
        sleep_until_a_packet_is_received();
        packet = read_packet();
        switch(packet->type)
        {
            case HELLO:        process_hello(packet);    break;
            case DB_DESC:      process_db_desc(packet);  break;
            case LS_REQ:       process_ls_req(packet);   break;
            case LS_UPDATE:    process_ls_update(packet); break;
            case LS_ACK:       process_ls_ack(packet);   break;
            default:           process_recv_error();     break;
        }
    }

Figure 5.5: Common code for both the client and the server in the protocol example.
timers to be maintained explicitly. On the server side, the protocol-based solution requires the initial LSA list and the packet sequence number to be maintained as state information. The ORB-based solution again requires no state, retransmission lists or retransmission timers to be maintained explicitly.

Observe further that the protocol solution requires a message-demultiplexing capability, which is accomplished by the while-loop. This loop decodes the message type and forwards it to the appropriate function. The loop is in a thread by itself so that it can listen for incoming packets. The ORB-based solution does not require such a demultiplexing component since that is taken care of by the ORB.

To summarize, asynchronicity results in a larger part of the state space entering the application level, whereas synchronicity results in many of the states being hidden by the ORB or implicitly held in the program counter. The result is that solutions using the ORB require a smaller state space.

The ORB-based solution also abstracts away all marshaling and de-marshaling tasks. Problems related to byte order and machine word-length are taken care of. This makes the code much more portable and keeps the code clear of low-level details.

The end-result of the above comparisons is that

the code based on the ORB is much clearer, shorter and represents the fundamental ideas of link-state technology very clearly.

The difference in clarity and ease of reading is so enormous that the tasks of code writing, debugging and maintenance are substantially improved. It should be em-
<table>
<thead>
<tr>
<th>Measure</th>
<th>ORB-based Solution</th>
<th>Protocol-based Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Space Size (per port)</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Source Code (main body)</td>
<td>5 pages</td>
<td>16 pages</td>
</tr>
<tr>
<td>Initialization Code</td>
<td>3 pages</td>
<td>2 pages</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the ORB-based and protocol-based solutions.

phasized that the reduction in code length is an advantage, but not as great as the improvement in clarity. Even if the code were the same size, an improvement in clarity and adherence to the high-level logic are key advantages.

Table 5.1 summarizes the differences quantitatively, based on the sample software written for both the protocol setting and the ORB setting. It is interesting to note that the initialization code takes a significantly larger percentage in the ORB-based solution than in the protocol-based solution. This we believe is a characteristic of code written with a high-level of abstraction.

5.3.3 Performance Comparison

In the present subsection, we compare the ORB-based solution and the protocol-based solution from a performance point of view. Specifically, we compare the set of messages generated and the speed of the message exchanges. Since the ORB is being used as a wrapper for protocols, such an approach is appropriate. The comparisons are based on GIOP messaging [67, 71] and omniORB 2.8 [7] – a partial implementation of CORBA 2.3.
Message Sequences

We analyze the sequence of messages that the protocol-based and ORB-based solutions generate. First, we briefly review the GIOP message formats, details of which may be found in [67] or [71]. Every time an IDL method is invoked by a client on a server that resides in a different address space, the request is translated into a GIOP request message. The response from the server to this message is returned in a GIOP reply message. Thus, every IDL method that is not oneway is translated into two GIOP messages. oneway methods are translated into a GIOP request message only.

Table 5.2 shows the list of messages generated by the protocol-based solution, and the corresponding GIOP messages generated by the ORB-based solution. The packet types of the protocol-based solution are taken from OSPF [61]. The hello message has been omitted from the comparison since both solutions use the same message. The database description message of the protocol-based solution is the equivalent of the GIOP reply message for getLSASummaryList. The link-state request message of the protocol-based solution is the equivalent of the GIOP request message for getLSAs. The link-state update message of the protocol-based solution corresponds to the GIOP reply message of getLSAs during database synchronization, but it corresponds to the GIOP request message of recvLSAs during flooding. This is why row 3 in the table is split into cases (a) and (b). The link-state ack message of the protocol-based solution corresponds to the GIOP reply message of recvLSAs during flooding. During database synchronization, the link-state ack message of the protocol-based solution corresponds
<table>
<thead>
<tr>
<th>PBS Packet</th>
<th>GIOP Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Database description</td>
<td>getLSASummaryList reply</td>
</tr>
<tr>
<td>2 Link-state request</td>
<td>getLSAs request</td>
</tr>
<tr>
<td>3 Link-state update</td>
<td>(a) getLSAs reply</td>
</tr>
<tr>
<td></td>
<td>(b) recvLSAs request</td>
</tr>
<tr>
<td>4 Link-state ack</td>
<td>recvLSAs reply</td>
</tr>
</tbody>
</table>

Table 5.2: Messages (packets) resulting from the protocol-based and ORB-based solutions. The column “PBS Packet” denotes the packets generated by the protocol-based solution, and “GIOP Packet” denotes the packets generated by the ORB-based solution.

to the TCP ack in the ORB-based solution.

There are three types of message sequences in the link-state technology, which are derived from the description in Section 4.2 of Chapter 4.

1. LSA summary list exchange

2. LSA update during database synchronization

3. LSA update for flooding

We now analyze the message sequences case-by-case. For ease of reference, they are denoted as Sequence 1, Sequence 2, and Sequence 3, respectively.

*Sequence 1.* The first sequence is that of reading a neighbor’s database summary list. Figure 5.6 shows the messages exchanged by two neighbors executing the
Figure 5.6: Sequence 1 for the protocol-based solution.
Figure 5.7: The sequence of messages exchanged by two neighbors when one of them (the client) invokes `getLSASummaryList` on the other (the server).
Figure 5.8: Sequence 1 for the ORB-based solution. The following notation is used:

‘Request $x$’ refers to the GIOP request generated by neighbor $x$ ($x = A, B$). ‘Reply $x_i$’ refers to the $i$-th TCP fragment generated by the GIOP reply to node $x$’s request.

‘ACK $x_i$’ refers to the acknowledgement generated by the receipt of Reply $x_i$. 
Figure 5.9: Sequence 2 for the protocol-based (top) and ORB-based solutions (bottom).
Figure 5.10: Sequence 3 for the protocol-based (top) and ORB-based solutions (bottom).
protocol-based solution. Of the two nodes, the one with the larger address is designated the master, and the other the slave. The master initiates database description packets describing its database. The slave responds to these packets with its own database description packets. Flow control is achieved by taking a database description packet from a neighbor as an implicit acknowledgement of the packet sent by the local node previously.

In preparation for describing the message sequence for the ORB-based solution, we describe the following sequence. Figure 5.7 shows the sequence of messages generated in the ORB-based solution when one node (the client) invokes getLSASummaryList on the other (the server). The method invocation results in a GIOP request message being sent from the client to the server. The server would respond with a GIOP reply which contains the LSA summary list. However, this list may be split over multiple TCP packets, depending on the Message Transfer Unit (MTU). For simplicity and concreteness, a TCP window size of 1 is shown in the figure.

Figure 5.8 shows Sequence 1 for the ORB-based solution. In this scenario, both neighbors invoke getLSASummaryList on each other. In the diagram, the invocation times of the two neighbors are shown to be roughly synchronized. If, this is not the case, the period from the time of the first invocation to the time of the second invocation would look like Figure 5.7. During the overlapping period, TCP acknowledgements in one direction can be piggybacked with TCP packets transporting GIOP reply messages in the other direction. Thus, the pattern of messages looks like that for the protocol-based solution, except at the beginning.
Sequence 2. Figure 5.9 demonstrates Sequence 2 by comparing the sequence of messages generated by the protocol-based and ORB-based solutions when one of the neighbors requests an LSA (or list of LSAs) from the other. Again, the TCP acknowledgement to the GIOP request has been shown piggybacked with the GIOP reply. As a result, the message sequences for the two solutions have the same number of messages.

Sequence 3. Figure 5.10 demonstrates Sequence 3 by comparing the sequence of messages generated by the protocol-based and ORB-based solutions when one of the neighbors floods an LSA to the other. Again, the message sequences for the two solutions have the same number of messages.

Message Lengths

We now proceed to compute the message lengths. For this purpose, we have provided Table 5.3 which summarizes the encoding rules for the datatypes commonly needed for network control. Variables of type octet are marshaled as one byte, with paddings of one or three bytes depending on the data type that follows it (see [67] or [71]). Variables of type short are encoded with two bytes, with an additional two bytes of padding if necessary. Variables of type long are encoded with four bytes. Sequences are encoded with four bytes giving the sequence length, followed by the encodings of the constituent elements. Arrays and structures are encoded as a list of constituent members.

A GIOP request message constitutes three parts – the GIOP message header,
GIOP request header, and the GIOP request body. The message formats are shown in Appendix 5.A.2. The GIOP message header is twelve bytes long. The GIOP request header length is twenty-four bytes plus the number of characters in the method name it represents. In this calculation, we set the object key length to be twelve bytes, a number that far exceeds the needs of a network control program, but was chosen because omniORB uses it. Since all our experiments are based on omniORB, we retain the numbers used by it. Thus, the service context length was set to zero and the requesting principal was set to 'nobody' (as used by omniORB on the Windows NT 4.0 platform). The GIOP request body consists of the input parameters of the method it represents. The encoding of these parameters is as discussed in the preceding paragraph.

A GIOP reply message also consists of three parts – the twelve byte GIOP message header, an eight-byte GIOP reply header, and the GIOP reply body. The GIOP reply body consists of the output parameters of the method it represents. The encoding of these parameters is as discussed earlier.

The message formats for the protocol-based and ORB-based solutions are shown in Appendix 5.A.2. Table 5.4 shows the lengths of these messages for various number of LSAs and links. Addresses were taken to be four bytes long. The columns labeled 1–4 correspond to the rows with the same label in Table 5.2. Under column 3, rows labeled 'OBS' contain two values in the table. They correspond to cases 3(a) and 3(b) in Table 5.2.

The main observation from Table 5.4 is that the encoding of parameters in ORB-
<table>
<thead>
<tr>
<th>IDL Data Type</th>
<th>Marshaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>octet</td>
<td>1 byte + (1 or 3 bytes)</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes + (2 bytes)</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
</tr>
<tr>
<td>sequence</td>
<td>4 bytes for length + list of elements</td>
</tr>
<tr>
<td>array</td>
<td>List of array elements</td>
</tr>
<tr>
<td>struct</td>
<td>List of members</td>
</tr>
</tbody>
</table>

Table 5.3: GIOP marshaling of data types commonly used for network control. The quantities shown in parenthesis are bytes needed for padding that is sometimes necessary to satisfy GIOP marshaling rules. The padding bytes are shown under the assumption that only the data types listed in this table will be used.

Based solution is as efficient as that in the protocol-based solution, except when the packet sizes are very small. However, in the latter case, not much bandwidth is used as the packets are small. In general, the only overhead in using GIOP consists of

- GIOP headers,
- TCP header,
- Padding bytes
- The fact that sequence lengths are always encoded as four bytes
- The fact that method names are encoded as strings rather than given numeric
codes.

The last item is conceptually trivial to rectify. Padding bytes can be minimized by grouping variables of the same data type together. The overhead due to the GIOP header and TCP header is the price paid for gaining simplicity in programming. However, these headers are fixed length and hence do not grow with packet size, as illustrated in Table 5.4. In fact, from Table 5.4, we see that the ORB-based solution's messages are comparable in length to the protocol-based solution's messages, except when the message bodies are very small.

Latency

In this section we compare the time taken by the protocol-based and ORB-based solutions to execute each of the three types of sequence of messages. The results reported here were obtained from experiments conducted using omniORB 2.8 [7] running on Pentium II 600 MHz uni-processor machines operating Windows NT 4.0. The two machines were interconnected by a switched 10 Mb/s Ethernet segment.

We conducted a series of experiments to infer the performance penalty due to the ORB. In particular, the experiments measured the time taken by each of the three sequences 1, 2, and 3 discussed earlier. The number of LSAs and the number of links per LSA were varied. The results are shown in Table 5.5. The table at the bottom shows the number of LSAs and links per LSA used in each measurement. The experiments were repeated thousands of times for better accuracy.

We see from the data that the penalty introduced by the ORB is in the order of
<table>
<thead>
<tr>
<th>LSAs×Links/LSA</th>
<th>Protocol</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1×5</td>
<td>PBS</td>
<td>24</td>
<td>12</td>
<td>228</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>40</td>
<td>71</td>
<td>264</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>311</td>
<td></td>
</tr>
<tr>
<td>1×25</td>
<td>PBS</td>
<td>24</td>
<td>12</td>
<td>1,028</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>40</td>
<td>71</td>
<td>1,144</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,191</td>
<td></td>
</tr>
<tr>
<td>10×5</td>
<td>PBS</td>
<td>132</td>
<td>48</td>
<td>2,172</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>148</td>
<td>107</td>
<td>2,388</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2,435</td>
<td></td>
</tr>
<tr>
<td>10×25</td>
<td>PBS</td>
<td>132</td>
<td>48</td>
<td>10,172</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>148</td>
<td>107</td>
<td>11,188</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11,235</td>
<td></td>
</tr>
<tr>
<td>100×5</td>
<td>PBS</td>
<td>1,212</td>
<td>408</td>
<td>21,612</td>
<td>1,208</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>1,228</td>
<td>467</td>
<td>23,628</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23,675</td>
<td></td>
</tr>
<tr>
<td>100×25</td>
<td>PBS</td>
<td>1,212</td>
<td>408</td>
<td>101,612</td>
<td>1,208</td>
</tr>
<tr>
<td></td>
<td>OBS</td>
<td>1,228</td>
<td>467</td>
<td>111,628</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>111,675</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of message lengths (in bytes) under the protocol-based solution (PBS) and the ORB-based solution (OBS).
50% when the network is very small (and hence the packet sizes are very small). It is much smaller (about 20% or less) when the network (domain) size is more typical (100 nodes or more).

A closer look at the columns 3(a) and 3(b) in Table 5.4 indicates that the GIOP reply for getLSAs is virtually the same size as the GIOP request for recvLSAs. Hence, Sequences 2 and 3 should take virtually the same time. However, in Table 5.5, the values for the protocol-based solution with respect to Sequences 2 and 3 are almost identical as expected, but the values for the ORB-based solution with respect to the two sequences are somewhat different. This is due to a peculiarity of omniORB. It was observed that the way omniORB splits the GIOP messages into TCP packets is different for getLSAs replies and recvLSAs requests, even though the two have almost identical lengths. For example, the GIOP reply for getLSAs for the row labeled “100x25” in Table 5.4 was split into 26 TCP packets whereas the GIOP request for recvLSAs was split into only 15 TCP packets, even though both are almost the same length (column 3, row ‘OBS’). This accounts for the differences between the values for Sequences 2 and 3 under the ORB-based solution in Table 5.5.

The above observation indicates that the 21% penalty indicated for sequence 2 under the ORB-based solution will be probably closer to 9% if 15 TCP packets are used instead of 26. The reason why omniORB uses 26 packets in one case and 15 in the other is not clear. However, this use of different number of packets in the two cases is clearly not intended for speed optimization.

In this age where bandwidth and processing speeds double every twelve to eighteen
months, a performance penalty of the order of less than $9\%$ (or even $21\%$) should be considered acceptable, particularly since it brings a tremendous simplification to the conceptualization and writing of software. Moreover, the above comparison is based on omniORB. Efforts such as those reported in [31] show that the performance of the ORB could be improved considerably by careful optimization, thus reducing the gap between the protocol-based and ORB-based solutions even further. A related piece of work that compares various ORBs may be found in [27].

5.4 Concluding Remarks

This chapter presented two key benefits of the foundations to network programmability – conceptual clarity and simplicity of programming. The conceptual clarity resulted in the identification of two fundamental capabilities that are necessary and sufficient for network control. It also resulted in the understanding that a thin RPC-like ORB is sufficient for realizing a programmable approach to network control. The simplicity of programming was demonstrated in terms of the reduction in state-space size and source-code length, and in terms of the source code following high-level logic very closely.

The chapter also showed that a thin RPC like-ORB does not impose a great reduction in performance compared to the traditional protocol approach. In the case of omniORB, the solution is shown to be about $20\%$ slower than the protocol solution for network domains of realistic size (100 or more nodes). However, it was explained
<table>
<thead>
<tr>
<th></th>
<th>Sequence 1</th>
<th></th>
<th></th>
<th></th>
<th>Sequence 2</th>
<th></th>
<th></th>
<th></th>
<th>Sequence 3</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PBS</td>
<td>OBS</td>
<td>%</td>
<td>PBS</td>
<td>OBS</td>
<td>%</td>
<td>PBS</td>
<td>OBS</td>
<td>%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>310</td>
<td>480</td>
<td>55 %</td>
<td>470</td>
<td>650</td>
<td>38 %</td>
<td>480</td>
<td>610</td>
<td>27 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>310</td>
<td>480</td>
<td>55 %</td>
<td>1,150</td>
<td>1,500</td>
<td>30 %</td>
<td>1,160</td>
<td>1,410</td>
<td>22 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>490</td>
<td>630</td>
<td>29 %</td>
<td>2,190</td>
<td>2,810</td>
<td>28 %</td>
<td>2,180</td>
<td>2,600</td>
<td>19 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>490</td>
<td>630</td>
<td>29 %</td>
<td>9,300</td>
<td>11,400</td>
<td>23 %</td>
<td>9,300</td>
<td>10,600</td>
<td>14 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2,260</td>
<td>2,500</td>
<td>11 %</td>
<td>19,800</td>
<td>23,800</td>
<td>20 %</td>
<td>19,500</td>
<td>21,800</td>
<td>12 %</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2,260</td>
<td>2,500</td>
<td>11 %</td>
<td>92,000</td>
<td>111,000</td>
<td>21 %</td>
<td>93,000</td>
<td>101,000</td>
<td>9 %</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>#LSAs</th>
<th>Links/LSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 5.5: Performance comparison for the three sequences. The latency for the protocol-based solution is shown as PBS and the latency for the ORB-based solution is shown as OBS. The latencies, shown in microseconds, are valid to two significant figures.
that 9% is a more realistic value. The 9% penalty will reduce even further as the
ORB is optimized. In this age where bandwidth and CPU speeds double every twelve
to eighteen months, a penalty of 9% or even 20% should be considered acceptable,
particularly when it gives rise to tremendous simplification of software development.
5.A  APPENDIX

5.A.1  The Simple ORB as a Strip-Down Version of CORBA

As identified in Section 5.3.1, an ORB for use in network control must provide means for exposing interfaces to objects for remote access, and means for invoking methods on objects identified by object identifiers. In this appendix, we present an extremely small subset of the CORBA API for realizing this.

CORBA provides the PortableServer for exposing interfaces to objects. It provides stubs, skeletons, and object references for accessing remote objects. We shall discuss these in turn.

The PortableServer of CORBA supports the registration of objects for remote access. However, it does not permit the specification of TCP port numbers on which requests may be accepted for the registered objects. Figures 5.12–5.13 show the PortableServer module with some modifications shown in underlined text. The PortableServer module contains the methods activate_object_with_id and deactivate_object for registering and unregistering objects with the object adapter. A new method set_object_service has been added to indicate to the object adapter on what TCP ports and network interface cards to listen for CORBA requests. In principle, TCP could be replaced by some other reliable transport protocol, but we shall restrict the interface to TCP as it is the only widely-used reliable transport protocol.

The method set_object_service sets the threading model and the GIOP channels on a per-object basis. If the parameter thread_per_request is set to true, then
each invocation on the object with the specified object ID will generate a new thread. Otherwise, requests to the object will be serialized. The structure Channel describes a GIOP communication channel for the Portable Object Adapter (POA) to listen for incoming requests. Each channel specifies the TCP port on which to listen for requests. Moreover, the requests are limited to those arriving on the network interface card specified by nic_addr. Thus, if there are multiple network interface cards (ports) on a computer, the handling of requests could be limited to those arriving only on a certain set of network interface cards. This feature is very useful for building security features into network control software.

CORBA provides the methods string_to_object and object_to_string, shown in Figure 5.11, for creating object references from object identifiers, and vice versa. It provides the method resolve_initial_references to obtain a reference to the PortableServer, and the interface Object for manipulating object references. The method ORB.init initializes the ORB. The method set.timeout has been added for specifying how long the ORB would wait for getting a response from a proxy object when a method is invoked on the proxy object.

CORBA provides stubs and skeletons for method invocation on objects identified by object reference. Details of these may be found in the IDL-to-C++ mapping specification [66]. Although the specification could be trimmed down considerably for use in network control, we do not present it here as it is an exercise that takes us far into the field of programming languages. The trimmings come from the fact that dynamic types, value types, dynamic invocation interfaces, and tools such as Named-
Value, NVList, Environment, Request, Context, and TypeCode are not necessary for network control.

The trimmed-down version of the CORBA interfaces presented above is a complete list of interfaces required by network control applications. It is much smaller than the version found in the minimumCORBA working document of the OMG [65], which is attempting to produce a trimmed down version of CORBA for portable computing devices.

5.A.2 Messages Formats

The messages of the protocol-based solution are given in Figures 5.14–5.17, and the messages of the ORB-based solution are given in Figures 5.18–5.21. Refer to Figures 4.3–4.4 for IDL declarations.
module CORBA
{
    typedef string ORBid;
    typedef sequence<string> arg_list;
    typedef string ObjectId;

    exception InvalidName {};

    interface Object
    {
        boolean is_nil();
        Object duplicate();
        void release();
        void set_timeout(in unsigned long timeout);
    };

    interface ORB
    {
        string object_to_string(in Object obj);
        Object string_to_object(in string str);

        void run();
        void shutdown(in boolean wait_for_completion);
        void destroy();
    };

    ORB ORB_init(inout arg_list argv, in ORBid orb_identifier);
    Object resolve_initial_references(in ObjectId identifier)
        raises (InvalidName);
};

Figure 5.11: The stripped-down CORBA module.
module PortableServer
{
    native Servant;
typedef sequence<octet> ObjectId;

    interface POAManager
    {
        exception AdapterInactive {};
        void activate() raises(AdapterInactive);
    };

    struct Channel
    {
        unsigned long nic_addr;
        unsigned long tcp_port;
    };

typedef sequence<Channel> ChannelSeq;
};

Figure 5.12: The stripped-down PortableServer module. (continued in Figure 5.13).
module PortableServer
{
    interface POA
    {
        readonly attribute POAManager the_POAManager;

        exception ObjectAlreadyActive {};
        exception ObjectNotActive {};
        exception ServantAlreadyActive {};
        exception InvalidChannel { UShortSeq inv_item_indexes; };

        void destroy(in boolean etherealize_object,
                    in boolean wait_for_completion);

        void activate_object_with_id(in ObjectId id, in Servant p_servant)
            raises(ServantAlreadyActive, ObjectAlreadyActive);

        void deactivate_object(in ObjectId oid) raises(ObjectNotActive);

        void set_object_service(in ObjectId oid, in bool thread_per_request,
                                 in ChannelSeq ch_seq)
            raises(InvalidChannel);
    };
};

Figure 5.13: The stripped-down PortableServer module. (continued from Figure 5.12).
Figure 5.14: The structure `LinkInfo` in protocol format. This encoding is common to both the protocol-based solution and the ORB-based solution, except that the latter has an additional four bytes indicating the dimensionality of the operating point.
Figure 5.15: The **LSAHeader** and LSA structures in protocol format. These encodings are common to both the protocol-based solution and the ORB-based solution.
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

+-----------------------------------------+
| Version #  | 2   | Packet Length  |
+-----------------------------------------+
| Checksum   |      | Unused         |
+-----------------------------------------+
| Sequence Number                  |
+-----------------------------------------+
| LSA Header 1                        |
+-----------------------------------------+
| ...                                  |

Figure 5.16: The database description packet of the protocol-based solution.
Figure 5.17: The link-state request, reply and acknowledgement packets of the protocol-based solution, shown in order from top to bottom.
Figure 5.18: GIOP message header (top), request header (middle) and reply header (bottom).
Figure 5.19: GIOP request and reply messages for getLSASummaryList. The encoding of LSA headers is as given in Figure 5.15.
Fig. 5.20: GIOP request and reply messages for getLSAs. The encoding of LSAs is as given in Fig. 5.15.
Figure 5.21: GIOP request and reply messages for recvLSAs.
6

CONCLUSION

6.1 THESIS SUMMARY

In this thesis, we have presented a conceptual framework for the development of programmable networks. The framework is based on the progression consisting of (i) a core network resource model, (ii) a network-element service model, (iii) topology abstraction, (iv) network service model, (v) APIs for accessing these models, and (vi) APIs for signaling.

The thesis identifies communication graphs and their manipulation as the essence of network control. It presents concrete resource and service models and APIs that are very general, capturing connection-oriented and connectionless networks, packet-switched, circuit-switched and wavelength-switched networks. This is the first time such generality has been achieved. The resource model is kept very simple, capturing the bare essence of data transport from the viewpoint of network control. In particular, abstract concepts such as the schedulable region have been shown to be derivable from the resource model.

The APIs to the models have been kept to the bare minimum. The thesis intro-
duced and presented a definition of minimality of an API, and showed that the APIs presented are minimal. Figure 6.1 summarizes the APIs and the propositions that capture their minimality.

![Diagram]

Figure 6.1: Summary of the APIs and the propositions that demonstrate their minimality.

During the course of the development, the thesis identified a set of three fundamental capabilities of switch control processors, based on which any network control
infrastructure could be built. Two of these fundamental capabilities are necessary and sufficient to build a network control infrastructure, while the third is useful for performance optimization of the network control infrastructure. To handle redundancy in networks, a fourth capability was identified as being necessary, but left for future research.

The thesis demonstrated that the APIs greatly simplify the task of writing software for network resource discovery. In particular, it was shown that the source code is much smaller, clearer and reflects high-level concepts much more clearly. A simple ORB, based only on RPC, was shown to be adequate for accessing these APIs remotely. Moreover, it was shown that the performance of a system implemented with these APIs will not be much lower than that of those written directly in terms of protocols.

6.2 DIRECTIONS FOR FURTHER RESEARCH

We identify a number of areas for further research.

- Identification of a fundamental capability (or API) that addresses redundancy in telecommunication networks.

- Expansion of the forwarding table model and API to capture (i) memory limitations that may restrict the number of forwarding table entries, and (ii) throughput limitations of the forwarding fabric that may arise from the use of multicast segments. Forwarding fabrics are typically non-blocking as far as unicast seg-
ments are concerned, but not when multicast segments are introduced.

- Definition of APIs for distributed interactions in graph building, similar to those given for resource discovery.

- Methodology and design patterns for how the APIs of this thesis may be used. The thesis briefly presented how these APIs may be used. However, a thorough study is necessary.

- Analysis of the performance of different graph building algorithms under an ORB-based solution. In particular, the effect on state space, source-code length, clarity and performance need to be studied in the context of graph-building. From the analysis presented in this thesis for resource discovery algorithms, there is great promise for simplification.

- Study of how different versions of an API could be made to be backward compatible. In protocols, this is achieved using a version field. The OMG IDL is not flexible enough to handle such versioning effectively. On the other hand the Simple Object Application Protocol (SOAP) being developed by a consortium of companies appears to show more promise [78]. However, the problem of backward compatibility is not unique to network control.
ESTIMATING THE SCHEDULABLE REGION

This appendix discusses the estimation of the multiplexing capacity known as the schedulable region [34, 58]. A very low-complexity hardware-implementable algorithm is presented. The algorithm is shown to produce reasonably good estimates rapidly. It serves as a proof-of-concept of the resource model of a multiplexer presented in the thesis.

The appendix is presented for an ATM multiplexer. However, the general concepts apply to other types of packet multiplexers as well. The issues that need to be addressed for this generalization are pacing and packet size limits. These issues are left for future study.

A.1 MOTIVATION

Estimation of the schedulable region is closely akin to the problem of call\textsuperscript{1} admission control. The main type of traffic that has been a challenge for performing call admis-
sion control is VBR video. While voice traffic has been adequately captured by the on-off model, VBR video has remained elusive, with a number of models suggested but none that is analytically tractable and still faithful to the empirical data. The main problem is that video traffic exhibits rather complex characteristics such as multiple timescales, long range dependence and subexponentiality [46]. Moreover, using such models to evaluate the queueing behavior is even more complex. As a result, many researchers pursued asymptotic techniques to evaluate the queueing behavior [46, 48, 19]. Though many theoretical results have been developed for understanding the behavior of queues, the practical use of these asymptotics for call admission control has yet to be demonstrated. The parameters involved in such asymptotic equations are difficult to be evaluated in practice by a computer program residing in a switch control processor. Moreover, video traffic streams can be highly variable. As such the parameters describing a video stream can be quite varied. These parameters, however, are typically not known a priori, particularly for live video streams. Thus, many researchers have begun the study of measurement-based admission control (MBAC) [25, 45, 32, 50, 28, 23, 21].

Many MBACs have the following two characteristics [15]: They are greedy in that they admit all calls as long as there are sufficient resources available at the time of admission. They are localized in that the admission decisions are not coordinated across switches in the network. The reason for these characteristics is that MBACs estimate resource usage instead of resource capacity. Since resource usage can change rapidly as calls arrive and depart, estimates of resource usage are not very helpful for
network-wide optimization.

In this appendix, we consider the schedulable region as the capacity of multiplexing resources [34, 58]. We shall define the schedulable region concept in three steps. First, suppose that all calls passing through the multiplexer have identical cell-level statistics. Denote by \( S \) the largest number of calls that can pass through the multiplexer while it can guarantee quality of service to all of them simultaneously. Then, the schedulable region is defined as the set \( \{0, 1, \ldots, S\} \). Now, suppose that all calls passing through the multiplexer can be categorized into a finite number of classes, known as traffic classes, such that all calls in a given traffic class have identical cell-level statistics. Then, the schedulable region is defined as the set of all possible combinations of calls of each traffic class such that the multiplexer can guarantee quality of service to all of them simultaneously. Thus, the schedulable region is a subset of \( N^K \), where \( N \) is the set of non-negative integers and \( K \) is the number of traffic classes. The dimensionality of the schedulable region is defined as \( K \). The \( K \)-dimensional vector consisting of the number of calls of each traffic class flowing through the multiplexer is called the operating point of the multiplexer.

The assumption that all calls in a given traffic class have the same statistics is rather restrictive in practice because video traffic does not exhibit any uniformity. It would be necessary to instantiate a very large number traffic classes to accommodate real video streams, making the estimation of the schedulable region very complex. Nevertheless, it is possible to group VBR video traffic into clusters. For example, the mean rates of the MPEG-1 streams used in this appendix were all in the range of
several hundred kb/s, and the variances were similarly clustered. The screen resolution of the streams were all the same. It was shown in [32]\textsuperscript{2} that such clustering does not affect the definition of one-dimensional schedulable regions. The same argument could be used to cover multi-dimensional schedulable regions.

In this appendix, we take the following approach to alleviate the restriction imposed by traffic classes. A set of traffic classes that reflects common traffic usage is selected. Any call that does not fall into one of these traffic classes is placed under a premium category and admitted based on peak-rate allocation. Other calls are admitted based on the size of the schedulable region and the current operating point of the multiplexer. Thus, the latter set of calls allow exploitation of multiplexing gain whereas the former do not. Hence, users of the premium category will be charged higher to compensate for the high resource reservation. Viewed differently, users are given a price-incentive to use one of the traffic classes. From a practical point of view, a traffic class is defined by factors such as the encoding format (MPEG-1, MPEG-2), number of pixels and colors, and video frame rate. Similar characterizations can be made for traffic classes that represent audio traffic or other content. Users can choose to use traffic that does not fall into one of the traffic classes, but they would have to pay more. A certain portion of the resources is allocated for the premium category (much like first- and business-classes in commercial airlines) and another portion is allocated for the traffic classes.

\textsuperscript{2}The paper listed as [32] discusses the estimation of one-dimensional schedulable regions. The differences between this paper and the present appendix will be discussed shortly.
The advantage of the schedulable-region approach is that resource usage can be optimized system-wide. Techniques for such optimization have been studied in the context of multirate circuit switching [70]. These techniques apply to the schedulable region since the schedulable region can be approximated well by a region bounded by a hyperplane as discussed later in this appendix. Moreover, the schedulable region provides a metric for comparing multiplexers. All other factors (traffic statistics, output port capacity (bandwidth), QoS requirements) being equal, the larger the schedulable region the greater the multiplexing gain.

The focus of this appendix is to develop a low-complexity algorithm that can produce good estimates of the size of the schedulable region. The algorithm presented requires two sets of measurements: one that maintains a histogram of cell counts, and one that accumulates weighted call holding times. The two measurements are taken independently, and combined once every measurement cycle, which was taken to be five minutes long. The advantage of this approach is that the former set of measurements can be accumulated very efficiently by the switches while the latter set of measurements is collected by the switch control processor at a slower timescale. This is in contrast to the algorithm presented in [32], where there is no such separation. In our scheme, the measurements needed for accurate capacity estimation can be collected over a rather short duration, without the need for storing any history of the traffic processes. This is also in contrast to [32]. Moreover, the calculations involved in the method presented in this thesis are much simpler compared to those presented in [32]. Under the sliding-window technique presented in [32], the estimation would
require the history of the traffic process to be maintained for a duration as long as the average call holding time. For the case of video traffic, the average call holding time could be rather long, given the fact that movies run into a couple of hours. Under the jumping-window technique of [32], the estimator would produce estimates at intervals of duration equal to the average call holding time. Moreover, the technique of [32] requires the average holding time of calls to be known \textit{a priori}.

The rest of this appendix is organized as follows: Section A.2 provides empirical data on aggregate video traffic and Section A.3 discusses techniques for measuring the parameters of the traffic online. Section A.4 shows how these techniques can be realized in practice. Section A.5 uses the results of the preceding two sections to develop an algorithm for online estimation of the schedulable region. Section A.6 presents numerical results from the algorithm and compares them to offline computations. The appendix concludes with Section A.7.

**A.2 Aggregate MPEG Streams**

In this section, we demonstrate the effectiveness of the Gaussian approximation to aggregate MPEG streams. The nineteen streams used for this study were collected at the University of Würzburg [69]. The streams were encoded in MPEG 1 format for a duration of approximately half an hour. The video contents encoded ranged from TV news casts to action sports. Further details of the samples can be found in [69]. In order to generate aggregate streams with more than nineteen component streams,
the individual streams were replicated and started at random points in the sequence. When a sequence reached its end, it was wrapped around. Since the starting points of the replicated sequences were chosen at different points, the replicated sources can be considered to be roughly independent.

We now consider the aggregate streams generated by one, nineteen, and fifty-seven sources. Denote the number of bytes generated by the aggregate stream with $m$ sources ($m = 1, 19, 57$) during the time interval $(0, t]$ by $A_{t,m}$. We assume that every video source transmits each video frame uniformly during a period of 30ms, i.e., $A_{t,1}$ is a piecewise linear continuous function of $t$, where each linear region is of duration 30ms. In the ATM literature, the above is referred to as pacing of cells. Pacing is a way of limiting the instantaneous bit-rate of a stream.

Define $a_{p,m}(\tau) = A_{(p+1)\tau,m} - A_{p\tau,m}$, where $p \in \mathbb{N}$ and $\tau > 0$. For every $p$, $a_{p,m}(\tau)$ is the number of bytes arriving during a period of duration $\tau$. Thus, $\tau$ will be called the measurement window.

Figure A.1 shows the sample path $\{a_{p,m}(30\text{ms}); 0 \leq p \leq 40000\}$ for $m = 1, 19, 57$. Figure A.2 shows the histograms of these sample paths. In addition, each of the graphs in Figure A.2 also shows the histogram of sample paths corresponding to measurement windows $\tau = 3\mu\text{s}, 300\mu\text{s}, 3\text{ms}$. The abscissa for each graph has been scaled by a factor of $30\text{ms}/\tau$ so that the histograms can be compared. Observe that for a given $m$, the distributions corresponding to $\tau \leq 3\text{ms}$ are almost identical, whereas the distribution corresponding to $\tau = 30\text{ms}$ is somewhat different. This deviation arises from the fact that each video source paces its cells over 30ms periods. When $\tau$
is close to the pacing period, sampling and pacing interfere with each other. However, if $\tau$ is much smaller than the pacing period, then the two phenomena occur at different timescales and hence do not interfere with each other in any significant way. Now, from a practical point of view, the larger the measurement window $\tau$, the smaller the frequency of measurement. Thus, since the measurements are identical for all $\tau \leq 3\text{ms}$, we take the measurement window to be $3\text{ms}$. Note that the measurement window is independent of $m$.

Figure A.3 shows the distribution of the sample paths $\{a_{p,m}(3\text{ms}); 0 \leq p \leq 9000\}$ for $m = 1, 19, 57$, i.e., the sample paths restricted to the first five minutes. Again, the abscissa has been scaled by $30\text{ms}/\tau$, where $\tau = 3\text{ms}$. Figure A.4 shows the quantile-quantile plots of the distributions of $\{a_{p,m}(3\text{ms}); 0 \leq p \leq 40000\}$ ($m = 1, 19, 57$), with respect to the standard normal distribution. Figure A.5 shows the quantile-quantile plots of the distributions of the sample paths $\{a_{p,m}(3\text{ms}); 0 \leq p \leq 40000\}$ with respect to those of $\{a_{p,m}(3\text{ms}); 0 \leq p \leq 9000\}$ for $m = 1, 19, 57$.

The following observations can be drawn from the above graphs. The distribution of a single source is far from being a Gaussian, but those of the two aggregate streams are approximately Gaussian, as seen from Figure A.4. In other words, the law of large numbers begins to manifest itself even when the number of streams is as low as nineteen, certainly by the time this number reaches fifty-seven.

Another observation can be made from the quantile-quantile plots shown in Figure A.5:
Figure A.1: Sample paths of the number of bytes arriving in three aggregate video streams. Graphs (a), (b), and (c) correspond to the streams consisting of the aggregation of 1, 19, and 57 MPEG-1 streams, respectively. Each measurement sample was taken over a period of 30ms.
Figure A.2: Histograms of the sample paths shown in the Figure A.1, and also those of the sample path with measurement window $\tau = 3\mu s, 300\mu s, 3$ms. The abscissa is scaled by $30ms/\tau$. The legend gives the value of $\tau$ for each curve.
Figure A.3: Histograms of the sample paths shown in the Figure A.1, but restricted to the first 9000 samples and taken with a measurement window \( \tau = 3\text{ms} \).
Figure A.4: Quantile-quantile plots of the standard Gaussian distribution with respect to the distributions shown in Figure A.2 for $\tau = 3\text{ms}$. 
Figure A.5: Quantile-quantile plots of the the distributions shown in Figure A.2 with respect to the distributions shown in Figure A.3, for $\tau = 3ms$. 
The distributions of the aggregate streams taken over a five-minute period are very close to the distributions taken over the entire half-hour period.

Thus, the distribution of the aggregate streams can be estimated by taking measurements over a relatively short time period. However, the above observation does not apply to the single-source stream, as is evident from the first graph of Figure A.5.

A.3 Measurement Techniques

The preceding section identified two main properties of aggregate MPEG video traffic, namely that the distribution is near-Gaussian and that this distribution measured over a relatively short time period is a good estimate of the distribution measured over the entire duration of the stream. Thus, given a fixed set of video streams, the mean and variance of the aggregate stream can be estimated rapidly. The next step is to consider such measurements when the set of video streams (calls) keeps changing over time. This is the topic of the present section. First, we place some conditions under which the techniques presented herein are applicable.

1. The cell arrival processes of any two calls must be statistically independent.

2. All calls must have the same mean and variance. More generally, calls can be grouped into one of a finite number of categories called traffic classes; in each category, all calls have the same mean and variance. This assumption is usually not satisfied in practice. However, it is adequate for the means and variances
of all calls of given traffic class to be clustered together. The experimental results in this appendix show that the schedulable estimation algorithms perform properly in the face of such clustering. The results presented in this section are for a single traffic class. In the case of multiple traffic classes, multiple parallel measurements have to be taken, one for each traffic class. These parallel measurements are independent of each other.

3. Call arrivals and departures occur at a timescale much slower than the minimum of the line rate (cells/second) of the multiplexer and $1/\tau$. If the call arrival rate gets very large, then the above condition can still be satisfied by considering only a fraction of the calls in the measurement process.

4. Call arrival and departure processes are independent of the cell arrival and departure processes of any single video source.

5. The probability distribution of the number of calls in the multiplexer changes slowly compared to the measurement cycle.

Suppose that the number of calls through the multiplexer is kept fixed at $n$. Define the mean and variance of the number of bytes that arrive in the aggregate stream over a period of duration $\tau$ by $\mu(n, \tau)$ and $\sigma^2(n, \tau)$. Observe that for large $n$ (nineteen or larger) the distributions of $a_{p,n}(\tau)/\tau$ are identical for all $\tau \leq 3\text{ms}$ (Section A.2) Therefore, as a result of the first two conditions, we have, for large $n$,

\[
\begin{align*}
\mu(n, \tau) &= n\mu\tau \\
\sigma^2(n, \tau) &= n\sigma^2\tau^2 \quad (\tau \leq 3\text{ms})
\end{align*}
\]

(A.1)
where $\mu$ and $\sigma$ are constants that depend on the statistics of an individual video stream. Thus, it is sufficient to estimate $\mu$ and $\sigma$.

Now suppose that the number $n(t)$ of calls through the multiplexer varies over time $t$. Denote by $A_t$ the number of bytes arriving in the aggregate stream over the period $(0, t]$. Define $a_p(\tau) = A_{(p+1)\tau} - A_{p\tau}$, where $p \in \mathbb{N}$ and $\tau > 0$. In view of Condition 4 above, we may write the following provided that $n(t)$ does not change in the interval $((p+1)\tau, p\tau]$.

$$\mathbb{P}(a_p(\tau) = x) = \sum_{m=0}^{N} \pi_m \mathbb{P}(a_{p,m}(\tau) = x), \quad (A.2)$$

where $\pi_m = \mathbb{P}(n(t) = m)$ and $N$ is an upper bound on the number sources. Due to condition 5, $\pi_m$ may be taken to be independent of $t$. Due to condition 3, the probability of $n(t)$ changing during the interval $((p+1)\tau, p\tau]$ is small. Thus, we take the above equation to be approximately true, omitting a rigorous proof. Now, by taking expectations and summing over $p$,

$$\mathbb{E}(A_t) = \sum_{m=0}^{N} \pi_m \mathbb{E}(A_{t,m}). \quad (A.3)$$

for any $t$ that is a multiple of $\tau$. However, for any $t > 0$, we can find a $\tau \leq 3$ms such that $t$ is a multiple of $\tau$. Thus, the previous equation is valid for all $t > 0$. Yet, the above derivation is heuristic due to Equation (A.2). Its use however is validated by the numerical results presented later.

Now, $\mathbb{E}(A_{t,m}) = m\mu t$. Moreover, assuming ergodicity, $\pi_m = \lim_{t \to \infty} t_m/t$ and $\lim_{t \to \infty} A_t/t = \mathbb{E}(A_1)$, where $t_m$ is the total time in the interval $(0, t]$ during which
\( n(.) \) was \( m \). Observe also that \( \mathbb{E}(A_1) = \mathbb{E}(A_t)/t \) and \( \lim_{t \to \infty} \sum_{m=0}^{N} m t_m / t = \mathbb{E}(n(\cdot)) \).

After substituting into Equation (A.3), we obtain

\[
\mu = \lim_{t \to \infty} \left( \frac{A_t}{t \mathbb{E}(n(\cdot))} \right).
\]  

(A.4)

In other words, \( A_t/t\mathbb{E}(n(\cdot)) \) is an estimate of \( \mu \).

We now proceed to estimate \( \sigma \). We have from Equation (A.2)

\[
\mathbb{E}(a_1^2(\tau)) = \sum_{x=0}^{\infty} x^2 \sum_{m=0}^{N} \pi_m \mathbb{P}(a_{1,m}(\tau) = x) = \sum_{m=0}^{N} \pi_m \mathbb{E}(a_{1,m}^2(\tau)).
\]  

(A.5)

Now, \( \mathbb{E}(a_{1,m}^2(\tau)) = m \sigma^2 \tau^2 + m^2 \mu^2 \tau^2 \) for \( \tau \leq 3ms \) (Equation (A.1)). Define \( B_t(\tau) = \sum_{p=0}^{\lfloor t/\tau \rfloor} a_p(\tau)^2/\lfloor t/\tau \rfloor \). Observe that \( \mathbb{E}(a_1^2) = \lim_{t \to \infty} B_t(\tau) \). Then, using the same limit expression as before for \( \pi_m \), we have

\[
\sigma = \lim_{t \to \infty} \sqrt{\frac{B_t(\tau)/\tau^2 - \mu^2 \mathbb{E}(n^2(\cdot))}{\mathbb{E}(n(\cdot))}}, \quad \tau \leq 3ms
\]  

(A.6)

Thus, dropping the limit from above gives an estimate of \( \sigma \). 

### A.4 Practical Considerations

The important feature of Equations (A.4) and (A.6) is that they consist of two components, one that involves measurements on \( A_t \) alone (cell level), and one that involves measurements on \( n(t) \) alone (call level). To see the usefulness of this separation, consider a measurement period of five minutes. For this period, the switching hardware can simply take cell counts on the aggregate arrival process \( A_t \), without the need to interact with the switch control processor or be even aware of \( n(t) \). Even the quantity \( B_t(\tau) \) can be evaluated based on cell counts: The switching hardware computes
a histogram of $A_{(j+1)r} - A_{jr}$. For every sample point obtained, the histogram is updated by incrementing the appropriate bin, which is determined by the value of the sample point. For example, if the bin width is 100 and the sample obtained is 723, then bin number $\lceil 723/100 \rceil = 7$ is updated. Once every measurement period (five minutes in this case), the switch control processor reads the histogram and computes the variance. The accuracy of this procedure depends on the number of bins used in the histogram, i.e., there is a trade off between accuracy and memory space. The number of histograms per multiplexer is typically small since it is equal to the number of traffic classes.

By reading the cell-level statistics from the switch hardware and maintaining the call-level statistics in software, the switch control processor can computer $\mu$ and $\sigma$ as per Equations (A.4) and (A.6).

### A.5 Estimating the Schedulable Region

Consider a multiplexer of a switch handling $K$ traffic classes. With each traffic class, associate a set of QoS constraints. For the purpose of this appendix, the QoS constraints shall take the form of loss bounds. Denote the number of calls in each traffic class $k$ by $n_k$ and denote $n = (n_1, \ldots, n_k)$. The vector $n$ is called the operating point of the multiplexer. The schedulable region $\mathcal{S}$ is defined as the set of all $n \in \mathbb{N}^K$ such that QoS can be provided to each of the $K$ traffic classes [34]. The number $K$ will also be referred to as the dimensionality of the schedulable region. We divide the
estimation procedure into two parts.

A.5.1 Single Traffic Class

In the case of a single traffic class, the schedulable region $S$ is a bounded set of consecutive non-negative integers $\{0, 1, \ldots, S\}$. This is because under a work-conserving scheduling policy, a one-dimensional schedulable region contains no holes, i.e., if $m > 0$ and $m$ is a member of the schedulable region, then $(m - 1)$ is also a member of the schedulable region. Thus, it would be sufficient to estimate $S$.

Consider a bufferless server with an arrival process consisting of the aggregation of $n$ video streams. For practical purposes, the server can be supplied with a small buffer without altering the results of this section significantly. For this arrival process, the minimum service rate required to avoid cell loss is given by the peak rate of the aggregate stream. Since the arrival rate of the aggregate stream has a Gaussian distribution, the probability that the number of bytes arriving in a period of duration $\tau$ will exceed any given number $P > 0$ is $1 - \Phi_{\mu(n, \tau), \sigma(n, \tau)}(P)$, where $\Phi_{\mu(n, \tau), \sigma(n, \tau)}$ is the cumulative probability distribution of a Gaussian random variable with mean $\mu(n, \tau)$ and variance $\sigma^2(n, \tau)$. Thus, the smallest service rate for which the cell loss probability does not exceed a given bound $\ell$ is $b(n) = \Phi_{\mu(n, \tau), \sigma(n, \tau)}^{-1}(1 - \ell)/\tau$, or,

$$b(n) = (\mu n + \alpha(\ell)\sigma\sqrt{n}), \quad (A.7)$$

where $\alpha(\ell) = \Phi_{0,1}^{-1}(1 - \ell)$. For example, if $\ell = 10^{-5}$, then $\alpha(\ell) \approx 5$. Now, the boundary $S$ of the schedulable region $S = \{0, 1, \ldots, S\}$ is the largest value of $n$ satisfying
$b(n) \leq c$, where $c$ is the service rate of the queue. Thus, in view of Equation (A.7), the boundary of the schedulable region is given by

$$S = \left[ \frac{2c \mu + \alpha(\ell)^2 \sigma^2 - \sqrt{(2c \mu + \alpha(\ell)^2 \sigma^2)^2 - 4 \mu^2 c^2}}{2 \mu^2} \right].$$  \hspace{1cm} (A.8)

In the above calculations, we took the buffer size to be close to zero. Apart from the simplification apparent from the above, the zero buffer assumption also brings in another important simplification. Without this latter simplification, the computations of the schedulable region would be exceedingly difficult, as will be explained now.

In the case where there is a near-zero buffer and sufficient bandwidth to make the cell loss ratio negligible, the input streams entering the buffer are virtually identical to the output streams leaving the buffer. This is because, in the absence of buffering, there is no distortion of cell timings. By dimensioning the buffers to be very small and confining the operating points of the multiplexers to be within the schedulable region, distortions can be avoided. In this way, the multiplexer can be operated in the so-called linear region. This is similar to the linear region of operation of a transistor. For faithful signal amplification, a transistor must operate in its linear region.

The advantage of limiting the operation of a multiplexer to its linear region comes from the following. The output of one multiplexer feeds into the input of the next. Hence, if a stream traverses many multiplexers, it may change its statistical properties along the way (due to distortion). However, if the multiplexers operate in their linear region, then the traffic does not change its statistical properties. This invariance is
very important. Its absence would necessitate the analysis of how the traffic changes its statistical properties as it traverses the network. Such an analysis has been elusive, and is possibly intractable.

Before closing this subsection, we observe that multiplexing gain can be achieved in two ways – spatially and temporally. The first way is the result of the fact that the peaks and troughs of the traffic processes of two independent streams are uncorrelated and hence the probability of peaks of two independent streams occurring simultaneously is small. Thus, the bandwidth required to support two such streams is typically less than the sum of the peak rates of the two streams. This technique, which is what we adopt in this appendix, does not require buffering. Spatial multiplexing does not distort the statistics of any of the streams. For there to be spatial multiplexing gain, at least two traffic streams must be present.

The second way – temporal multiplexing – does not require more than one stream. Temporal multiplexing is achieved by delaying (buffering) packets that arrive during peak periods and scheduling them during non-peak periods. Temporal multiplexing results in traffic distortion and causes computational complexity as discussed above.

In this appendix, we avoid temporal multiplexing.

A.5.2 Multiple Traffic Classes

In the case of multiple traffic classes, the boundary of the schedulable region takes the form of a surface in $\mathbb{N}^K$, where $N$ is the set of non-negative integers. In order to estimate this surface, we proceed as follows. Consider the bandwidth function $b(n)$
from the preceding subsection, except that now the argument \( n = (n_1, \ldots, n_K) \) is a vector. It denotes the minimum bandwidth required to support all the calls of each traffic class under a specified scheduling policy. Each traffic class is assigned its own buffer. As before, the size of the buffers is taken to be very small. We now add the suffix \( k \) to the quantities \( \mu, \ell \) and \( \sigma \) to indicate the traffic class. The measurement window \( \tau \) is kept the same for all traffic classes.

We seek to decompose the bandwidth function as follows:

\[
b(n) \leq \sum_{k=1}^{K} b(n_k e^k),
\]

where \( e^k \) is the unit vector with all zeroes except in the \( k \)th element, which is 1. Thus, \( b(n_k e^k) \) represents the bandwidth required to support only the calls of traffic class \( k \). The motivation for this inequality comes from the spatial multiplexing that occurs between traffic sources. The inequality in essence says that if two streams can be individually guaranteed QoS by providing bandwidths \( b_1 \) and \( b_2 \), respectively, then they can be jointly guaranteed QoS by providing them bandwidth not exceeding \( b_1 + b_2 \). This condition is a very natural requirement for economizing bandwidth. In fact, any scheduling policy that does not satisfy this condition is wasting bandwidth! In general, however, not every work-conserving scheduling policy satisfies this condition. In particular, the inequality is not valid for priority schedulers. In fact, the priority scheduling policy has been identified in [34] as unsuitable for providing quality of service guarantees. The condition applies to the weighted round-robin scheduling policy (provided the weights are appropriately chosen), and the Maximum Laxity Thresh-
old (MLT) scheduling policy (provided that the loss constraints are virtually zero) [34]. The decomposition is approximately true for the Magnet Real-Time Scheduling (MARS) policy, since MARS is an online approximation of MLT. We now analyze each of the above scheduling policies qualitatively with heuristic arguments.

**Priority Scheduling Policy:** Consider two buffers, and suppose that the traffic streams entering the two buffers is of variable bit rate. If the buffers are not small, then the bandwidths \(b_1\) and \(b_2\) required to serve the two streams *individually* are less than the peak rates of the respective streams. This is because by buffering some traffic, the buffers can be served at a rate between the peak cell rate and the average cell rate. Now, consider the case where the two buffers are served by a priority scheduler with service rate \(b_1 + b_2\). Then, the buffer with the higher priority preempts the other buffer. Hence, it 'sees' the entire bandwidth \(b_1 + b_2\) of the priority scheduler. This means that its queue buildup will reduce and it will take up more bandwidth during periods of high activity. As a result, the lower priority buffer gets a bandwidth smaller than \(b_2\) during high-activity periods of the high priority buffer. Hence, the lower priority buffer no longer gets adequate quality of service. However, if the buffer sizes are small and the loss constraints are virtually zero, then the phenomenon described above cannot happen in any significant way since \(b_1\) and \(b_2\) would then be close to the peak rates of the respective streams. But, in this case, the priority scheduling policy is hardly better than FIFO for providing QoS. Hence, the priority scheduling policy is unsuitable for our purposes.

**Weighted Round-Robin Scheduling Policy:** A weighted round-robin policy dis-
tributes the bandwidth of the server to the various buffers according to a specified set of weights. Inequality (A.9) is valid for the weighted round-robin (WRR) policy provided that $b(n_k e^k)$ is less than the bandwidth assigned to traffic class $k$ (for all $k$). In this case, each buffer is guaranteed its share of the bandwidth and hence QoS is guaranteed (even if the buffer sizes are large and the loss constraints are lax).

**Dynamic Weighted Round-Robin Scheduling Policy:** A variation on WRR, which we shall call Dynamic WRR, is the following. The weights of the scheduler are adjusted dynamically so that the bandwidth assigned to each buffer is always sufficient to guarantee QoS, provided that the total bandwidth of the multiplexer is adequate. The weights have to be adjusted only when the operating point of the multiplexer drifts significantly in the schedulable region. The timescale on which such a drift may occur is expected to be much slower than the call arrival rate, but we do not provide any data on it. Under the dynamic WRR scheme, inequality (A.9) is always satisfied, irrespective of the delay and loss constraints.

**Maximum Laxity Threshold Policy (MLT):** The MLT policy operates by delaying every cell as long as it can without violating the QoS constraints. Cells that cannot be delayed further are served first. This action is carried out with knowledge of both past and future cell arrivals. This is why it is an offline algorithm. As will be apparent from the discussion of the priority scheduling policy, the bandwidth decomposition we seek fails only if cells are scheduled sooner than what they would have been in an isolated buffer, or if they are scheduled while they would have been dropped in an isolated buffer. MLT does delay cells that can tolerate further delay if there
are other cells that cannot be delayed further. However, MLT does not consider loss constraints. Therefore, the bandwidth decomposition is true for MLT provided that the loss constraints are virtually zero. Now, for the algorithm presented in Section A.5.1 to be valid, the delay constraints would have to be virtually zero. But, if both the loss constraints and delay constrains are to be virtually zero, MLT is hardly better than FIFO for providing QoS. Hence, MLT is unsuitable for our purposes.

From the foregoing, we see that only three of the scheduling policies discussed are applicable for use in the schedulable region estimation technique discussed, namely FIFO, WRR and dynamic WRR.

In the case of WRR, the schedulable region is the Cartesian product of the one-dimensional schedulable regions for each traffic class, since the bandwidth of the scheduler is partitioned between the traffic classes, i.e., the schedulable region is estimated thus:

\[
\bigotimes_{k=1}^{K} \{ n_k \in \mathbb{N} | \mu_k n_k + \alpha(l_k) \sigma_k \sqrt{n_k} \leq c_k \} \tag{A.10}
\]

where \( c_k \) is the bandwidth assigned to traffic class \( k \), and \( \sum_{k=1}^{K} c_k = c \). Note that in the case of WRR, we cannot let the operating point of the multiplexer go to a region where traffic class \( k \) needs more bandwidth than \( c_k \) (for some \( k \)), even if the other traffic classes are using much less than their share of the bandwidth. This is because, as discussed before, QoS cannot be guaranteed.

In the case of dynamic WRR, we estimate the schedulable region using the bandwidth decomposition technique. Due to Equations (A.9) and (A.7), the schedulable
region may estimated as

\[ \{ n \in \mathbb{N}^K \mid \sum_{k=0}^{K} (\mu_k n_k + \alpha(\ell_k) \sigma_k \sqrt{n_k}) \leq c \}. \]  
(A.11)

Due to the bandwidth partitioning resulting from WRR, the schedulable region under WRR will be significantly smaller than that under dynamic WRR.

We now estimate the schedulable region when the scheduling policy is FIFO, i.e., when all the traffic classes are fed into a single queue. Consider the case where \( \ell_1 = \cdots = \ell_K = \ell \). Since the aggregate streams corresponding to each traffic class is a Gaussian process, the aggregation of these aggregates is also a Gaussian process with mean \( \sum_{k=1}^{K} \mu_k \tau n_k \) and variance \( \sum_{k=1}^{K} \sigma_k^2 \tau^2 n_k \). If this aggregate of aggregates is passed through a FIFO server, the schedulable region can be estimated as follows (from Equation (A.7)):

\[ S = \left\{ n \in \mathbb{N}^K \mid \sum_{k=1}^{K} \mu_k n_k + \alpha(\ell) \sqrt{\sum_{k=1}^{K} \sigma_k^2 n_k} \leq c \right\}. \]  
(A.12)

The above calculation guarantees that the loss probability of the entire traffic (consisting of all the traffic classes) is bounded by \( \ell \). It does not necessarily imply that the loss probability is guaranteed for each traffic class individually. However, due to the independence of the calls and the near-zero size of the buffer, each traffic class is guaranteed a loss probability bound of \( \ell \), provided that each call is given an equal opportunity to enter the FIFO buffer. By this is meant the following: If one cell arrives in each of two traffic classes simultaneously, then they would each have an equal chance of getting into the FIFO buffer. In other words, if two cells arrive
simultaneously, the decision as to which cell gets in first is determined by a 'coin toss'. In this way, no traffic class is given preferential treatment in this FIFO buffer.

The expressions in Equations (A.11) and (A.12) are rather complex. They do not lend themselves for easy visualization of the shape of the schedulable region. In the next section, we shall show that these expressions for the schedulable region can be well-approximated by linear expressions, resulting in a schedulable region with hyperplanar boundary.

Before concluding this section, we present a trivial upper bound and a lower bound to the schedulable region. The lower bound is obtained by replacing each call in the multiplexer with a constant bit-rate call of rate equal to the peak cell rate of the original call. The resultant schedulable region will be called the peak-rate approximation to the schedulable region. The upper bound is obtained by replacing each call in the system with a constant bit-rate call of rate equal to the average cell rate of the original call. The resultant schedulable region will be called the mean-rate approximation to the schedulable region. The validity of these bounds is self evident. The difference between the peak rate approximation and the actual schedulable region shows the amount of multiplexing gain achieved. The mean rate approximation is a theoretical limit much like the speed of light in free space.
A.6 Numerical Results

We now present numerical results to illustrate the algorithms presented earlier. Figure A.6 shows a graph of the estimates of $\mu$ and $\sigma$ for a single traffic class over time as calls arrived and departed. The graphs also show the value of these quantities as measured offline (by keeping the number of calls fixed). The duration between changes of the operating point was varied in the range 0.1 – 3 seconds. From the first graph we infer that the estimate of $\mu$ is within about 3% of the offline measured value. The estimate of $\sigma$ is less accurate with an error margin of about 25%. However, $\sigma$ has only a second order effect in our computations, as can be seen from Equation (A.7). As $n$ increases, the term involving $\sigma$ increases proportional to $\sqrt{n}$ while the term involving $\mu$ increases proportional to $n$. Hence, a degradation in accuracy of the estimate of $\sigma$ is tolerable.

Figure A.7 shows the one-dimensional estimates of the schedulable region over time for a loss constraint of $10^{-5}$. The sample path of the operating point process $\{n(t)\}$ is also shown. In addition, the figure also shows the peak-rate approximation, the mean-rate approximation, and an offline computed schedulable region. Even though the online estimates of the standard deviation are not very accurate, the schedulable region estimates are accurate to within 10%, as seen from the graph. Similar results were obtained over several simulation runs. Moreover, the graph shows that the accuracy of the estimate is about the same when the number of calls in the system is small and when the number of calls in the system is large. This is an attractive
Figure A.6: Real-time estimates of $\mu$ and $\sigma$.

Now, consider the case of two traffic classes. The mean $\mu$ and standard deviation $\sigma$ for each class is estimated independently as discussed above. Suppose that for traffic class I, $\mu_I = 3$ Mb/s and $\sigma_I = 2.1$ Mb/s (e.g., MPEG II), and for traffic class II, $\mu_{II} \approx 0.58$ Mb/s and $\sigma_{II} = 0.6$ Mb/s (the values from Figure A.6). Figure A.8 shows the two-dimensional schedulable region estimates for the above traffic parameters, with loss rates $\ell_I = \ell_{II} = 10^{-5}$. The graph on the top is for a line rate of 155 Mb/s and the graph on the bottom is for a line rate of 622 Mb/s. The upper curves in
Figure A.7: Estimates of the one-dimensional schedulable region for lines rates of (a) 155 Mb/s and (b) 622 Mb/s. Figure (c) shows the sample path of the operating point $n(t)$. 
these graphs represent the estimate based on Equation (A.12) and the lower curves represent the estimate based on Equation (A.11). No offline estimates are shown as no packet-size traces for traffic class I were available at the time of the experiments. However, the accuracy of the estimate of Equation (A.12) is the same as in the single-dimensional case since the derivation of the former is based on the latter.

Figure A.8: Two-dimensional schedulable region with traffic parameters $\mu_I = 3$ Mb/s, $\sigma_I = 2.1$ Mb/s, $\mu_{II} = 0.58$Mb/s, $\sigma_{II} = 0.6$ Mb/s and line rate (a) 155 Mb/s and (b) 622 Mb/s. In each graph, the upper curve corresponds to Equation (A.12) and the lower curve corresponds to Equation (A.11).
We observe further from these graphs that the boundary of the schedulable region can be approximated by a straight line. Estimates based on Equation (A.11) have a slightly larger curvature since they do not take into account the multiplexing gain across traffic classes, while the estimates based on Equation (A.12) do take into account this gain. The above is an empirical conclusion of the form of the two equations. The near-linearity of the equations provides a great simplification of the schedulable region concept. In fact, we observed that all the schedulable regions presented in [34] are nearly linear, except for those that have significant inter-call correlation. Since we assumed that calls are uncorrelated, there is no contradiction.

A.7 CONCLUDING REMARKS

This appendix presented a very simple algorithm for estimating the schedulable region rapidly. The algorithm, which scales linearly with the number of traffic classes, provides a clean separation between cell-level measurements and call-level measurements. The appendix showed empirically that, under certain practically appropriate conditions, the schedulable region is well-approximated by a region with hyperplanar boundary.

The algorithm presented is valid for a certain class of scheduling policies, namely those that satisfy Inequality (A.9). Scheduling policies that do not satisfy this inequality are shown to waste bandwidth and hence go counter to the idea of economizing bandwidth by statistical multiplexing. In particular, the priority scheduling
policy is shown to not satisfy this inequality but the weighted round-robin policy is shown to satisfy it. A variant of the latter, called dynamic weighted round-robin, was introduced in this appendix as an enhancement that results in larger schedulable regions.

The appendix also discussed two types of multiplexing gain, namely spatial multiplexing gain and temporal multiplexing gain. Though both can be employed simultaneously, the former was shown to be preferable as it does not distort the statistics of traffic streams as the traffic traverses a network. Distortion causes tremendous computational complexities for evaluating the schedulable region (and for call admission control in general). The two types of multiplexing gain were compared to the linear and non-linear regions of a transistor. The algorithm presented in this thesis is based entirely on spatial multiplexing gain, which may be thought of figuratively as the linear region of the multiplexer.
References


[27] MLC Systeme GmbH (Germany) and Charles University (Prague). CORBA comparison project: Final report, June 1998.


[52] A. A. Lazar, K.-S. Lim, and F. Marconcini. Realizing a foundation for programmability of ATM networks with the binding architecture. IEEE Journal on Selected Areas in Communications, 14(7), September 1996.


