Providing Resilient Quality of Service Connections in Provider-Based Virtual Private Networks

A thesis submitted in fulfilment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF NEW SOUTH WALES

by

Gustav Filip Rosenbaum
Master of Science
Royal Institute of Technology, 1997

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NOVEMBER 2005
Abstract

This thesis focuses on efficient provisioning of resilient Virtual Private Network (VPN) services. It first confirms the intuition that network resources can be more efficiently utilized when resilience mechanisms are implemented by a network provider in the physical network than by its VPN customers in their VPNs. Next, a Multi-protocol Label Switching-based programmable VPN architecture is presented that delivers virtual links as resilient quality of service (QoS) connections and virtual sites. Virtual sites allow customers to implement functionality like customized routing and content adaptation “in the cloud”, as opposed to the current network model where all functionality is implemented at the network edge.

To provision a resilient QoS connection, two paths need to be computed from the ingress to the egress nodes, such that both paths meet the given QoS constraints. Two different frameworks have been proposed in the literature to compute resilient QoS connections when the QoS constraints are bandwidth and end-to-end delay. They both use a preprocessing step whereby either all links with less residual capacity than the given bandwidth constraint are pruned, or the given end-to-end delay is converted to an effective bandwidth. The frameworks thus reduce the problem to one with only a single constraint. We argue in this thesis that these frameworks individually lead to poor network utilization and propose a new framework where both constraints are considered simultaneously. Our framework exploits the dependency between end-to-end delay, provisioned bandwidth and chosen path through using the provisioned bandwidth as a variable. Here, two link-disjoint paths are computed together with their respective minimum bandwidths such that both the bandwidth and end-to-end delay constraints are satisfied.
Given our framework we first propose a new generic algorithm that decomposes
the problem into subproblems where known algorithms can be applied. Then we
propose two new linear programming (LP) formulations that return the two paths
and their respective bandwidths such that they have the minimum combined cost. To
make our framework applicable in a production environment, we develop two new
algorithms with low run times that achieve even higher network performance than
their LP formulation counterpart. These algorithms systematically use an algorithm
that computes non-resilient QoS connections. As no algorithm for computing non-
resilient QoS connections with sufficiently low run time has been proposed in the
current literature we develop two new algorithms and their respective heuristics with
a run time comparable to Dijkstra’s shortest-path algorithm. Our simulations show
that exploiting the dependency between end-to-end delay, provisioned bandwidth and
chosen path can significantly improve the network performance.
Statement of Originality

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.

Signed

Gustav Filip Rosenbaum
7 November, 2005
Acknowledgments

I would like to thank my supervisor Prof. Sanjay Jha and co-supervisor Dr. Chun-Tung Chou for their invaluable input, guidance, support and discussions in regard to my PhD Candidature.

I wish to convey my appreciation to all the other members of the Network Research Lab for encouraging me to go on during hard times and for making me smile and feel good every day. Special thanks to William Lau, Albert Chung and Alfandika Nyandoro (Alfa) for their technical input throughout our many discussions.

I am also grateful to the Co-operative Research Center for Smart Internet Technology (http://www.smartinternet.com.au) who funded and set the stage for this work.

I would also like to thank all my friends and my family for their patience with and understanding of my absence over the past few years. Finally, I thank Emilia for her motivation and energy boost that made me take on that hard last mile with a smile.

This work was edited in accordance with parts D, language and illustrations, and E, completeness and consistency, of the Australian Standards for Editing Practice by Dr Bruce Howarth, who has taught in the IT field.
## Contents

### 1 Introduction

1.1 Overview .................................................. 3  
1.2 Contributions ........................................... 6  
1.3 Publications ............................................. 7  
  1.3.1 Publications Covered in this Thesis ............... 8  
  1.3.2 Other Publications ................................. 8  

### 2 Literature Review

2.1 MPLS Overview ........................................... 11  
  2.1.1 Traditional IP Forwarding ......................... 11  
  2.1.2 Introduction to MPLS ............................... 12  
  2.1.3 MPLS at a Glance .................................. 13  
  2.1.4 Traffic Engineering and Quality of Service ....... 16  
  2.1.5 VPN Services in MPLS Networks ................. 20  
  2.1.6 Programmable Virtual Private Networks ......... 22  
2.2 Introduction to Resilience Provisioning ............... 22  
  2.2.1 Resilience Provisioning in MPLS Networks ........ 27  
2.3 Providing Resilient QoS Connections ................... 28  
  2.3.1 Computing Non-Resilient QoS connections ......... 30
4.2 Problem Definition ........................................... 64
  4.2.1 Customer Implementation Scenario ................. 67
  4.2.2 Provider Implementation Scenario .................. 67
4.3 Experiments .................................................. 69
  4.3.1 Results .................................................. 71
  4.3.2 Resilient Link Price Analysis ........................ 74
4.4 Conclusion .................................................. 76

5 Resilience Differentiation in Programmable Virtual Private Networks 77
  5.1 Introduction ............................................... 78
  5.2 Resilience Issues in Programmable Virtual Private Networks .... 79
    5.2.1 Split End-to-End Resilience Provisioning ............. 81
  5.3 Resilience Differentiation in the PVN Architecture ........... 83
    5.3.1 The PVN Architecture ................................. 83
    5.3.2 MPLS-Based Recovery ................................ 84
    5.3.3 Resilience Extensions ................................. 85
  5.4 Guideline Conformance ................................... 88
  5.5 Implementing VPN Services Using PVNs .................... 90
  5.6 Conclusion ................................................ 91

6 Dynamic Routing of Resilient QoS Connections 92
  6.1 Introduction .............................................. 93
  6.2 Problem Formulation ..................................... 95
  6.3 Decomposed Computation .................................. 98
  6.4 Combined Computation .................................... 101
    6.4.1 1+1 Recovery ....................................... 102
6.4.2 1:1 Recovery ........................................ 105
6.5 Simulation Results .................................... 111
6.6 Conclusion ............................................. 116

7 Efficient Algorithms for Computing Resilient QoS Connections 117
  7.1 Introduction ........................................... 118
  7.2 Sketch of an Algorithm for Computing Resilient QoS Connections  . 120
  7.3 Computing Non-Resilient QoS Connections ....................... 122
    7.3.1 Path Candidate Set ............................... 129
    7.3.2 The Dynamic Bandwidth Shortest-Path Algorithm ......... 132
    7.3.3 DBSP Complexity ..................................... 138
    7.3.4 DBSP Complexity Reduction ....................... 142
    7.3.5 DBSP Performance ................................. 145
  7.4 Dynamic Disjoint Shortest Paths ........................... 151
    7.4.1 Analysis ............................................ 155
    7.4.2 DDSP Performance .................................. 157
  7.5 Conclusion ............................................. 159

8 Conclusion 163
  8.1 Resilience Implementation Scenarios ......................... 164
  8.2 Resilience-Differentiated Programmable VPN Architecture ...... 165
  8.3 Resilient QoS Connection Computation ....................... 165
  8.4 Future Work ............................................ 167
    8.4.1 Resilience Integration in VPN Provisioning Models ...... 167
    8.4.2 Programmable Virtual Private Networks ................ 168
    8.4.3 Computing Resilient QoS Connections .................. 169
List of Figures

2.1 Illustration of label swapping in an MPLS network. ............ 14
2.2 MPLS header. ............................................. 15
2.3 Illustration of an LSP tunnel. ................................ 16
2.4 LSR with multiple queues. ..................................... 18
2.5 Token bucket shaping. ......................................... 19
2.6 Recovery scopes. ............................................. 25
2.7 Generalization of the single-link failure model. ............... 30
3.1 VPN QoS specifications. ....................................... 42
3.2 Path provisioning models. ..................................... 44
3.3 VPN solution scalability relations. .............................. 49
3.4 Customized VPN tree. ......................................... 52
3.5 MCI backbone. .................................................. 55
3.6 PE node state information dependency on number of sites. .... 57
3.7 P node state information dependency on number of sites. ..... 58
3.8 PE node state information dependency on number of VPNs. ... 59
3.9 P node state information dependency on number of VPNs. ... 60
4.1 Mappings of resilient demands onto a VPN topology in the CI and PI scenarios. ................................. 66
4.2 Canadian metropolitan network. ........................................ 70
4.3 Accepted demands in Network A with 15 overlay points. .... 72
4.4 Full links in Network A with 15 overlay points. ............... 72
4.5 Accepted demands in Network B with 30 overlay points. .... 73
4.6 Full links in Network B with 30 overlay points. ............... 73
4.7 $K$ in Network A with 15 overlay points and $f \in \{1.0, 1.3, 1.6\}$. 75
4.8 $K$ in Network B with 30 overlay points and $f \in \{1.0, 1.3, 1.6\}$. 76
5.1 Extended PVN node architecture with two active processors. 86
5.2 MPLS tunnels and VNC controlled LSPs. ......................... 89
5.3 ELIB and LIB configurations. ..................................... 89
6.1 Network representing a typical ISP network. .................... 112
6.2 Dynamic vs. static bandwidth allocation in a 1+1 recovery context, $\lambda = 55$. ...................................................... 113
6.3 Dynamic vs. static bandwidth allocation in a 1:1 recovery context, $\lambda = 85$. ...................................................... 114
7.1 Illustration of the top layer algorithm. ......................... 123
7.2 Intuition behind overtaking. ....................................... 128
7.3 Illustration of the need for history lists. ....................... 134
7.4 PCS size in a small, medium and large network using the MinCost path selection criterion. .................................. 146
7.5 Dirty iterations in a small, medium and large network using the MinCost path selection criterion. .......................... 146
7.6 Dirty nodes in each iteration in a small, medium and large network using the MinCost path selection criterion. ........ 147
7.7 PCS size in a small, medium and large network using the Widest-Shortest path selection criterion. ................... 147
7.8 Dirty iterations in a small, medium and large network using the Widest-Shortest path selection criterion. .......................... 148

7.9 Dirty nodes in each iteration in a small, medium and large network using the WidestShortest path selection criterion. ............... 148

7.10 Rejection rates as a function of end-to-end delay in the small network. 152

7.11 Rejection rates as a function of end-to-end delay in the medium network. ................................................................. 152

7.12 Rejection rates as a function of end-to-end delay in the large network. 153

7.13 Rejection rates as a function of arrival rate in the medium network. 153

7.14 Rejection rates as a function of end-to-end delay in the small network. 160

7.15 Rejection rates as a function of end-to-end delay in the medium network. ................................................................. 160

7.16 Rejection rates as a function of end-to-end delay in the large network. 161

7.17 Rejection rates as a function of arrival rate in the medium network. 161
List of Tables

3.1 VPN Customer preferences and VPN solutions. ........................ 51

5.1 Resilience provisioning models in programmable VPNs. ............ 80

5.2 Example set of resilience classes. ......................................... 82

7.1 Functions defined on a PC $x$ at node $u$. ............................... 126

7.2 Arrival Rates ($\lambda$). .................................................. 150

7.3 Arrival Rates ($\lambda$). .................................................. 159
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>Autonomous System</td>
</tr>
<tr>
<td>ATM</td>
<td>Asynchronous Transfer Mode</td>
</tr>
<tr>
<td>ATMARP</td>
<td>ATM Address Resolution Protocol</td>
</tr>
<tr>
<td>BA</td>
<td>Behavioral Aggregate</td>
</tr>
<tr>
<td>BGP</td>
<td>Border Gateway Protocol</td>
</tr>
<tr>
<td>CE</td>
<td>Customer Edge</td>
</tr>
<tr>
<td>CR-LDP</td>
<td>Constraint Routed LDP</td>
</tr>
<tr>
<td>DiffServ</td>
<td>Differentiated Services</td>
</tr>
<tr>
<td>DLCI</td>
<td>Data Link Connection Identifier</td>
</tr>
<tr>
<td>DSCP</td>
<td>DiffServ Code Point</td>
</tr>
<tr>
<td>FEC</td>
<td>Forwarding Equivalence Class</td>
</tr>
<tr>
<td>FIB</td>
<td>Forwarding Information Base</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FTN</td>
<td>FEC-to-NHLFE</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task-Force</td>
</tr>
<tr>
<td>IS-IS</td>
<td>Intermediate System to Intermediate System</td>
</tr>
<tr>
<td>IGP</td>
<td>Interior Gateway Protocol</td>
</tr>
<tr>
<td>IntServ</td>
<td>Integrated Services</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol Version 6</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LER</td>
<td>Label Switching Edge Router</td>
</tr>
<tr>
<td>LDP</td>
<td>Label Distribution Protocol</td>
</tr>
<tr>
<td>L2</td>
<td>OSI Layer 2</td>
</tr>
<tr>
<td>L3</td>
<td>OSI Layer 3</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>LIFO</td>
<td>Last In First Out</td>
</tr>
<tr>
<td>LP</td>
<td>Linear Programming</td>
</tr>
<tr>
<td>LR</td>
<td>Latency Rate</td>
</tr>
<tr>
<td>LSP</td>
<td>Label Switched Path</td>
</tr>
<tr>
<td>LSR</td>
<td>Label Switching Router</td>
</tr>
<tr>
<td>MPLS</td>
<td>Multi-Protocol Label Switching</td>
</tr>
<tr>
<td>MCP</td>
<td>Multi-Constraint Path</td>
</tr>
<tr>
<td>NHLFE</td>
<td>Next Hop Label Forwarding Entry</td>
</tr>
<tr>
<td>NHRP</td>
<td>Next Hop Resolution Protocol</td>
</tr>
<tr>
<td>OSI</td>
<td>Open System Interconnection</td>
</tr>
<tr>
<td>OSPF</td>
<td>Open Shortest Path First</td>
</tr>
<tr>
<td>PHB</td>
<td>Per Hop Behavior</td>
</tr>
<tr>
<td>PE</td>
<td>Provider Edge</td>
</tr>
<tr>
<td>PIM</td>
<td>Protocol Independent Multicast</td>
</tr>
<tr>
<td>PPP</td>
<td>Point-to-Point Protocol</td>
</tr>
<tr>
<td>PVC</td>
<td>Permanent Virtual Circuit</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RFC</td>
<td>Request For Comment</td>
</tr>
<tr>
<td>RIB</td>
<td>Routing Information Base</td>
</tr>
<tr>
<td>RIP</td>
<td>Routing Information Protocol</td>
</tr>
<tr>
<td>RSP</td>
<td>Restricted Shortest Path</td>
</tr>
<tr>
<td>RSpec</td>
<td>Service Request Specification</td>
</tr>
<tr>
<td>RSVP</td>
<td>Resource Reservation Protocol</td>
</tr>
<tr>
<td>SDH</td>
<td>Synchronous Digital Hierarchy</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SONET</td>
<td>Synchronous Optical Network</td>
</tr>
<tr>
<td>TSpec</td>
<td>Traffic Specification</td>
</tr>
<tr>
<td>TTL</td>
<td>Time To Live</td>
</tr>
<tr>
<td>VCI</td>
<td>Virtual Circuit Identifier</td>
</tr>
<tr>
<td>VPI</td>
<td>Virtual Path Identifier</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Router</td>
</tr>
</tbody>
</table>
List of Abbreviations

WAN Wide Area Network
Chapter 1

Introduction

There is a growing interest among network providers in delivering resilient quality of service (QoS) connections. A resilient QoS connection is basically a logical unidirectional link connecting two customer sites in a virtual private network (VPN). As distinct from regular virtual links, a resilient QoS connection delivers a guaranteed level of QoS, such as a specific bandwidth and an upper end-to-end delay bound. A resilient QoS connection can also survive physical failures in the underlying network, such that connectivity is restored within a guaranteed amount of time such as 2 seconds or 200 milliseconds. These properties are becoming increasingly important for new applications like e-commerce, real-time services like voice over IP (VoIP) and streaming services such as video on demand.

Obviously, different services require different levels of resilience and QoS. A stockbroker typically does not require much bandwidth but is very sensitive to lost connectivity because each second of lost business may count for a substantial loss of revenue, whereas a streaming video service requires high bandwidth but can tolerate a connectivity outage of up to a few seconds. VoIP services again do not require high bandwidth; however, they are very sensitive to end-to-end delays because of their real-time nature and to connectivity losses lasting over about a second.

A typical service level agreement (SLA) specifies a minimum bandwidth and connection availability as an average over a long period of time, such as 99.5% over a month. 99.5% availability per month allows for over 3.5 hours of down time. Even
when the availability is as high as 99.9%, it allows for over 40 minutes of downtime per month. A SLA does not specify how this time is distributed over the period; in fact, it is likely that it comes in chunks disrupting the service for several minutes.

A common solution used today to overcome this availability problem is to implement what is referred to as multi-homing. Multi-homing in this context means that the VPN customer connects its VPN sites with redundant connections. To achieve redundancy a customer must either purchase two disjoint connections from one network provider or single connections from two different network providers. When one connection fails, the customer switches the traffic over to the other connection. Not only is this solution expensive for the VPN customer in terms of money, but in terms of administrative overhead and network complexity as well.

Clearly, network providers need to meet this growing demand for manageable VPN services with resilient QoS connections. However, today’s network infrastructures are not designed to deliver resilient QoS connections. VPN services are and have been a very lucrative business for network providers and nothing indicates that the growth of VPN services will slow down. On the contrary, new network technologies like Multi-protocol Label Switching (MPLS) have opened up for a whole new range of VPN services, where network layer VPN (L3 VPN) and Virtual Private LAN Service (VPLS) are the most prominent developments. Although the VPN service has gained momentum in many aspects, resilience to network failures has attracted little attention in this context.

This dissertation examines recent developments in VPN services and shows how MPLS can be used to provide a wide variety of VPN services that are resilient to network failures in one unified architecture. The architecture presented delivers resilient QoS connections and virtual VPN sites. We also present a new framework and algorithms for computing resilient QoS connections efficiently, both in terms of network performance and algorithm running time.
1.1 Overview

This dissertation examines the development of new VPN services, in particular how MPLS can be used to provide a wide range of VPN services that deliver resilient QoS connections. Chapter 2 examines the properties of MPLS, traffic engineering, QoS and resilience provisioning. Then, in Chapter 3, we provide a comprehensive overview of recent and future directions in VPN services. Chapter 4 presents a study that aims to explain the mutual benefits when resilience is implemented in the physical network by the network provider rather than by the VPN customers through multi-homing. Then we propose a new MPLS-based programmable VPN architecture that delivers resilient QoS connections and virtual sites in Chapter 5. In Chapter 6, we present a new framework and a theoretical approach to computing resilient QoS connections that achieves high network performance. Then in Chapter 7 the theoretical approach is followed up with a practical approach to effective computing of resilient QoS connections in a production environment. The remainder of this section presents a more detailed summary of each chapter.

Chapter 2 presents a critical review of the current literature related to QoS differentiation and resilience provisioning in MPLS networks. We start with an introduction to MPLS and describe its key strengths and explain why it has gained so much popularity amongst network providers. Then we highlight the properties of MPLS that have contributed to the recent growth of new VPN services. We follow up with an overview of QoS and how QoS is implemented in MPLS networks. Then we move on to provide a detailed description of different resilience provisioning models and their tradeoffs between network resource utilization, recovery times and scalability. Common resilience frameworks consider single link failures, in this chapter we show how such a framework can be generalized to include single node failures. Furthermore, this chapter provides an overview of current frameworks for computing resilient QoS connections and the strongly related problem of computing non-resilient QoS connections. At the end of this chapter we present a list of opportunities for improvement and gaps to be filled in the current body of research.
Chapter 3 provides a comprehensive overview of recent directions in provider-based VPN solutions. We look at both traditional and modern VPN solutions such as Virtual Private Wire Service (VPWS), Virtual Private LAN Service (VPLS), Virtual Router (VR) and BGP/MPLS VPN. We break the VPN solution concept down into three major building blocks, identify each block’s components, and compare them to each other in terms of scalability from the provider’s point of view and flexibility from the customer’s point of view. We then use the component comparisons to analyze the overall scalability in complete VPN solutions. Next, we relate VPN customer requirements to the various VPN solutions, resulting in an overview of suitable VPN solutions for different customer segments. Finally, we provide some promising research directions in VPN provisioning-related issues that should be investigated further.

Chapter 4 looks at how a network provider’s revenue is affected by where resilience is implemented in provider-based VPNs. As mentioned, resilience can be implemented either by the network provider in the physical network or by the VPN customers through multi-homing their VPNs. In this chapter we show that a network provider can significantly improve its revenue through implementing resilience. The customers also benefit from this implementation scenario from both cost and management perspectives.

Chapter 5 introduces a new programmable VPN architecture that provides resilient QoS connections with customizable levels of resilience and QoS. Moreover, the architecture presented allows customers to use virtual sites, which are VPN sites that run in a virtual router/server physically residing on the network provider’s premises. Virtual sites can be used by customers to implement functionality “in the cloud” such as customized routing and content adaptation. The architecture presented is built on top of legacy MPLS switches and can be deployed gradually in a backbone. The architecture introduces a new concept called a recovery engine. A recovery engine basically provides a specific level of resilience and QoS through implementing a recovery mechanism, which includes path computations and path supervision. The proposed architecture can be used to deliver a wide range of resilient VPN services simultaneously.
Chapter 6 looks at a new approach to computing resilient QoS connections that exploits the dependency between the end-to-end delay, chosen path and provisioned bandwidth. In this context we achieve resilience by computing two link-disjoint paths from the ingress to the egress node. Previous work described in the literature either converted the end-to-end delay constraints to an effective bandwidth or pruned all links not meeting the bandwidth constraint, thus reducing the problem to one with either bandwidth constraints or end-to-end constraints. In our approach, we look at the combined problem whereby we exploit the dependency between end-to-end delay, chosen path and provisioned bandwidth to compute two link-disjoint paths and their respective bandwidths, such that both paths meet the given bandwidth and end-to-end delay constraints and have the minimum combined cost. We propose three new algorithms. The first is a generic algorithm that addresses the problem through decomposing it into subproblems where known solutions can be applied. The other two are new linear programming (LP) formulations which address the combined problem. Our simulations show that the combined approach achieves higher network performance than the decomposed approach. The running time of the LP formulations presented are too high to be considered for a production environment. However, they can be used to benchmark heuristics that in turn can be used in production.

Chapter 7 also addresses the problem that is described in Chapter 6. However, the focus in this chapter is on providing efficient algorithms that can be used in a production environment. Thus, we emphasize both network performance and complexity. We develop a new, two-layered algorithm that computes two link-disjoint paths and their respective bandwidths such that they both meet the given bandwidth and end-to-end delay constraints. Moreover, the paths are subject to an optimality criteria such as either minimum combined cost or number of hops. The algorithm is two-layered in the sense that it systematically uses a shortest-path algorithm to compute shortest paths and their required minimum bandwidths such that they meet the bandwidth and end-to-end delay constraints. The current state-of-the-art algorithm in this context described in the literature has run-time complexity that is too high for our purposes and thus we developed a new efficient algorithm that not only improves on the run time but the network performance as well. The algorithms proposed in
this chapter can be used to implement path computations in the recovery engines described in Chapter 5.

Chapter 8 concludes the dissertation with a summary of the major results and identifies open research issues for future work.

1.2 Contributions

In this section we provide a list of contributions made in this dissertation and indicate in which chapter each contribution is first discussed.

1. A comprehensive survey of recent directions in VPN services including a new way of breaking down VPN solutions into building blocks, which in turn are used when examining the VPN solutions’ scalability from a network provider’s perspective and level of customer flexibility (Chapter 3).

2. Evidence that both a network provider and its customers benefit from a provider-based implementation of resilience mechanisms in the physical network, as opposed to customer-based implementations in the VPNs (Chapter 4).

3. A new MPLS-based programmable VPN architecture that provides resilient QoS connections with customizable levels of both resilience and QoS. This architecture allows VPN customers to use virtual sites and thus implement functionality “in the cloud”. Other traits are scalability both in terms of maintenance and number of VPNs, seamless gradual deployment and the use of legacy MPLS switches (Chapter 5).

4. A new framework for computing resilient QoS connections. In this MPLS-based framework two link-disjoint paths are computed together with their respective bandwidths, such that both paths meet the given QoS constraints. This approach differs from existing frameworks in that it explores the dependency between end-to-end delay, chosen path and bandwidth rather than reducing the problem to one that only contains either end-to-end delay or bandwidth (Chapter 6).
5. A new generic algorithm that operates in the new framework for computing resilient QoS connections. It basically breaks the problem down into subproblems where existing algorithms can be directly applied (Chapter 6).

6. Two new LP formulations that operate in the new framework for computing resilient QoS connections.
   - An LP formulation that computes two link-disjoint paths and their respective bandwidths such that they have minimum combined cost.
   - An LP formulation that computes two link-disjoint paths, one service and one backup path and their respective bandwidths such that they have minimum combined cost. This formulation explores backup bandwidth sharing to achieve a higher degree of network performance.

These LP formulations are provided in Chapter 6.

7. Fast new shortest-path algorithms that dynamically compute shortest paths and their bandwidths such that the returned paths meet the given QoS constraints. Here shortest refers to either minimum cost or minimum number of hops. The running times are comparable to Dijkstra’s shortest-path algorithm (Chapter 7).

8. New algorithms that operate in the new framework for computing resilient QoS connections. They systematically use our proposed shortest-path algorithms to compute two link-disjoint paths and their respective bandwidths such that they both meet the given QoS constraints. The returned paths have either the minimum combined cost or minimum number of hops. They run in \( O(V^2 \log V + EV) \) time where \( V \) is the number of nodes and \( E \) the number of edges in the network (Chapter 7).

1.3 Publications

In the authors list of the following publications my name appears both as Gustav and Filip Rosenbaum.
1.3.1 Publications Covered in this Thesis


1.3.2 Other Publications


Chapter 2

Literature Review

In this chapter we provide a critical literature review of QoS and resilience provisioning in MPLS networks. We introduce MPLS in Section 2.1, and explain the principles behind and the advantages of label switching compared to traditional IP forwarding. This section also gives an overview of traffic engineering and QoS in MPLS networks. Then we briefly describe customer and provider-based VPNs and point out some key properties that make MPLS a very attractive platform for implementing provider-based VPN services. A more comprehensive overview of provider-based VPNs is given in Chapter 3.

Section 2.2 introduces resilience provisioning in MPLS networks. Different resilience provisioning models from the literature are described, highlighting their individual tradeoffs between network resource utilization, recovery times and complexity. Section 2.3 introduces the problem of computing resilient QoS connections and the closely related problem of computing non-resilient QoS connections. This section also shows how frameworks considering single link failures can be generalized to incorporate single node failures. Finally, Section 2.4 summarizes the areas that require more research or improvement in the current literature.
2.1 MPLS Overview

MPLS integrates a label-swapping paradigm with network layer routing. A label is attached to each incoming packet at the ingress of an MPLS domain. The label is then used by downstream MPLS routers to forward packets across the MPLS domain to the egress router. The egress router strips off the label and forwards the packet using network layer routing.

MPLS is connection-oriented. All traffic is forwarded along preestablished label-switched paths (LSPs) across an MPLS domain. LSPs are similar to virtual circuits in ATM and Frame Relay data link connections. As opposed to ATM and Frame Relay, MPLS is independent of the data link layer protocol and thus MPLS can be used on top of ATM, Frame Relay, Synchronous Optical Network/Digital Hierarchy (SONET/SDH), Ethernet and Point-to-Point Protocol (PPP). In packet-based data link layer protocols such as SONET/SDH, Ethernet and PPP, the MPLS header is placed inside the link’s native format where the link layer's contents identifier specifies that it is carrying an MPLS packet. When MPLS is used on top of ATM and Frame Relay, the value of the top label is inserted in ATM’s VCI/PCI field (Davie et al., 2001) and in the Data Link Connection Identifier (DLCI) field when used on top of Frame Relay (Conta et al., 2001), which effectively results in native forwarding of MPLS packets.

2.1.1 Traditional IP Forwarding

In a traditional IP core network, an IP router performs a lookup for each arriving packet in a locally maintained table called the forwarding information base (FIB). The FIB maps IP destination prefixes, such as 129.94.0.0/16 and 129.94.1.0/24, to a list of next-hop routers to find where the packet should be sent next. An address prefix specifies a range of IP addresses, for instance the prefix 129.94.0.0/16 specifies the address range 129.94.0.0 to 129.94.255.255, which contains \(2^{(32-16)} = 65536\) addresses.

When a packet arrives at a router, the destination field in the packet’s IP header is extracted and used to find a longest-prefix match in the FIB, after which the router
sends the packet to the specified router. For example, suppose that an FIB contains two prefixes 129.94.0.0/16 and 129.94.1.0/24. When a packet arrives destined for 129.94.1.174 then the longest-prefix lookup in the FIB will return the entry given by 129.94.1.0/24, because that matches 24 rather than 16 bits of the 32-bit destination address.

Internet routers are segmented into domains, often termed Autonomous Systems (ASs). An AS is typically administrated by one organization such as an Internet Service Provider (ISP) or a network provider. Route distribution protocols are used to populate FIBs. Route distribution between routers in different domains is implemented using inter-AS routing protocols such as the Border Gateway Protocol (BGP). Route distribution within an AS is implemented through Interior Gateway Protocols (IGPs) like Open Shortest Path First (OSPF), Intermediate System to Intermediate System (IS-IS) and Routing Information Protocol (RIP).

Intra-AS protocols such as OSPF select routes based on shortest-path calculations using static link costs that do not reflect the current traffic loads. As a consequence, it is common for some links in an AS that uses traditional IP forwarding to become congested even though surrounding links are under-utilized. Congested links lead to poor network performance. In pure IP networks the hop-by-hop forwarding paradigm makes it hard to distribute the network load so that congested links are avoided. The process of controlling network load to optimize network performance is called Traffic Engineering (Awduche et al., 2002; Awduche, 1999) and is further discussed in Section 2.1.4.

2.1.2 Introduction to MPLS

Poor network utilization and the comparably slow longest prefix match in traditional IP forwarding networks motivated the industry to design new protocols for intra-AS routing and forwarding. Different vendors developed different proprietary solutions such as IP Switching (Newman et al., 1998; Lin and McKeown, 1997) from Ipsilon, Tag Switching (Rekhter et al., 1997) from Cisco Systems, Aggregate Route-based IP Switching (ARIS) (Boivie et al., 1997a) from IBM and Cell Switch Router (Boivie et al., 1997b; Nagami et al., 1997) from Toshiba. Eventually, the IETF MPLS Work-
ing group (IETF MPLS Working group,) was established in 1997 with the aim to converge the proprietary label-switching protocols into one unified standard. Today, the first generation MPLS-related standards are largely complete. Noticeable outstanding standardization efforts are an operations and management framework for MPLS applications (Allan and Nadeau, 2005), failure detection (Kompella and Swallow, 2005) and multicast protocol extensions for point-to-multi-point traffic engineering (Aggarwal et al., 2005).

Recent developments in gigabit IP routers have negated MPLS’s initial selling point of performance improvement. Today, MPLS’s strongest selling points are its traffic engineering properties and native support for creating hierarchies and virtual topologies.

2.1.3 MPLS at a Glance

Core nodes in an MPLS network are called Label-Switching Routers (LSRs) and edge nodes are sometimes referred to as Label-Switching Edge Routers (LERs). Simply put, when an IP packet arrives at an LER, it is classified using information like incoming interface, source or destination IP address, transport protocol or any other information conveyed in the packet headers. The classification maps packets to Forwarding Equivalence Classes (FECs). Each FEC is mapped to a Next-Hop Label Forwarding Entry (NHLFE) (Rosen et al., 2001; Nadeau et al., 2004), which basically specifies a label and an outgoing interface. The mapping is called FEC-to-NHLFLE (FTN). Once the FEC has been determined, the FTN is consulted to find the corresponding label, which subsequently is attached to the packet before it is transmitted on the indicated interface. When the downstream LSR receives the packet, its label is looked up in a Label Information Base (LIB) that maps incoming interface and label to an outgoing interface and a new label. The LSR swaps the incoming label with the outgoing label and forwards the packet to the next downstream router. Once the packet reaches the egress LER the label is “popped” and forwarded using network layer information.

This process is illustrated in Figure 2.1. An IP packet arrives at the ingress node destined for 129.94.172.174, and a lookup in the FIB shows that the packet belongs
to FEC $fecA$. Next $fecA$ is used to retrieve label $A$ from the FTN. Label $A$ is then pushed onto the packet before it is forwarded towards $LSR M$. When it is received by $LSR M$, the LIB instructs the LSR to swap label $A$ with label $C$ and forward it to $LSR N$. Similarly, $LSR N$ swaps $C$ with label $E$ and forwards the packet on to the egress LER. The instruction at the egress LER is to pop label $E$ after which the FIB is used to find out where to send the packet next.

Path $A \rightarrow C \rightarrow E$ is called a Label-Switched Path (LSP). An LSP can either be pre-provisioned or set up on demand. How the LSPs are constructed depends on whether the MPLS network is topology-driven or explicitly routed. In general, LSP setup is handled by the Label Distribution Protocol (LDP) (Andersson et al., 2001), which is a topology-driven protocol. Each LSR has an IP forwarding table and runs a standard IGP, such as OSPF. LDP uses information gathered by the IGP to distribute labels to its peers when building up LSPs for each ingress-to-egress FEC. In essence, an MPLS network that runs LDP will forward traffic along the same paths as a traditional IP network. As a consequence, a topology-driven MPLS network will inherit the same congestion problems that traditional IP networks experience.

To avoid these congestion problems, many network providers use explicit routing
The IETF MPLS working group has standardized two protocols to signal explicitly routed LSPs, also known as constraint routed LSPs. One protocol is called Constraint-Routed LDP (CR-LDP) (Jamoussi et al., 2002), which is an extension of the LDP protocol. The other adapts the Resource Reservation Protocol (RSVP) and is called RSVP-TE (Awduche et al., 2001a). Both protocols deliver the option of strict or loose route specification and specifying QoS parameters to control the queuing and scheduling behavior along the LSP. Loose route specification lets the network administrator specify some LSRs that the LSP must traverse, in contrast to the strict specification, which is used by the administrator to specify each hop from the ingress to the egress LSRs.

The MPLS header is regarded as a Last In First Out (LIFO) stack with one or more labels. Figure 2.2 shows an MPLS header, or label stack, with one label. The label value is represented by 20 bits, then there is a field termed EXP, named after its intended experimental use. Then follows a 1 bit field called Bottom of Stack (BoS) that indicates whether this label is the bottom one. The last field is a 8-bit Time to Live (TTL) indicator similar to the one found in the IP header. Label stacking refers to the process of pushing and popping multiple labels onto the label stack. Label stacking can be used to create tunnels that effectively form virtual topologies. Figure 2.3 illustrates an LSP tunnel between LSR P and LSR Q. When a packet labeled G arrives at LSR P label G is first swapped with label K before a second label A is pushed onto the label stack. The downstream LSRs operate on the top label, which is the most recently added label. Once the packet reaches LSR Q the top label is popped and the bottom label K is swapped with label L before it forwards the packet on.

Another feature of MPLS is LSP merging, which basically means that two distinct incoming LSPs to an LSR, that is two LSPs with different top labels, after label
swapping will carry the same top label when forwarded downstream. LSP merging can be used to construct forwarding sink trees, which are trees where the flow is directed towards the root or egress LER.

## 2.1.4 Traffic Engineering and Quality of Service

Traffic engineering refers to the process of optimizing network performance (Xiao et al., 2000; Awduche, 1999; Awduche et al., 1999). The major objective of traffic engineering is to simultaneously optimize network resource usage and traffic performance. Thus it has a twofold objective. Optimizing network resources typically involves load distribution to avoid congested nodes and links and optimizing traffic performance refers to optimizing traffic flows’ perceived QoS.

### 2.1.4.1 Optimizing Network Resources

As described in Section 2.1.3, using a topology-driven protocol such as LDP can result in a poorly utilized network with some links congested while surrounding links are under-utilized. Congestion occurs mainly because the shortest-path calculations based on static link cost metrics used by IGP protocols result in converging traffic flows at specific links. Many network providers therefore use explicitly routed LSPs to circumvent shortest-path convergence. Here, LSP setup is initiated at the ingress LER, where a path is computed and then signaled using a protocol like RSVP-TE or CR-LDP. It is the responsibility of the ingress LER to compute a suitable path; this can be done locally or through calling a centralized path computation server. In either
case, the path computation must have more information about the network than just link costs to avoid congestion effectively. Several proposals have been made to augment information carried in IGP protocols with dynamic link state information. Several extended IGP protocols have been standardized such as OSPF-TE (Katz et al., 2003) and IS-IS-TE (Smit and Li, 2004). Use of an extended IGP or other means to acquire link state information is vital for successful traffic engineering.

Furthermore, with explicit routing a network administrator can move one or more traffic aggregates from one LSP to another as network conditions change and thus maintain a balanced utilization of network resources. Changes in network conditions may be caused by link or node failures, changes in traffic patterns and changes to the physical topology such as addition of new LSRs and links.

2.1.4.2 Quality of Service

Quality of Service refers to the bandwidth, end-to-end delay and jitter that a traffic flow is subjected to. Bandwidth is often given in terms of a Traffic Specification (TSpec) (Wroclawski, 1997; Shenker et al., 1997), which is a 4-tuple specifying the maximum packet size, sustainable rate, peak rate and burst tolerance. End-to-end delay, also known as latency, is specified by a service Request Specification (RSpec), which provides an upper bound on the end-to-end delay. Jitter is a measure of the variance of inter-packet arrival times. A QoS specification for a flow can involve one, two or all three of these criteria. The most common QoS constraint is bandwidth followed by end-to-end delay. Many services require a combination of multiple QoS constraints such as bandwidth and end-to-end delay.

In a single-class service model, such as the best-effort service provided by today’s Internet, all traffic will have the same QoS attributes, which basically means that all packets are forwarded on a best effort basis. Because all traffic belongs to the same class, all packets receive the same queuing and scheduling treatment in the LSRs.

The basic forwarding implementation in LSRs (and in traditional IP routers) is to use a first-in-first-out (FIFO) queue per outgoing interface. Queuing introduces delays and if the traffic passing through the queue is bursty, like most Internet traffic, then
the queuing delay becomes unpredictable and introduces jitter. Packets are dropped when a queue overflows. To limit overflow, a larger queue can be used; however, doing so increases the end-to-end delay and jitter. Clearly, to satisfy different services LSRs need to split flows with different QoS requirements over multiple queues. Figure 2.4 shows an LSR with four internal queues. The outgoing packets on an interface are first classified by the classifier, typically by using the top label value or the 3-bit EXP field. Once the packet is classified, it is put in the appropriate queue. Subsequently the packet is fetched by the scheduler and transmitted to the downstream LSR. IETF has proposed two QoS models that have been adopted by the MPLS Working group; Differentiated Services (DiffServ) (Blake et al., 1998; Faucher et al., 2002) and Integrated Services (IntServ) (Shenker et al., 1997). The latter is tightly coupled with the Resource Reservation Protocol (RSVP) (Wroclawski, 1997; Awduche et al., 2001a; Awduche et al., 2001b).

In the DiffServ model, traffic is classified at the ingress LER and marked with a DiffServ Code Point (DSCP). A DSCP specifies a Behavioral Aggregate (BA), which determines how a packet is treated in the LSRs it traverses. The packet treatment is called Per-Hop Behavior (PHB) and specifies the scheduling treatment and in some cases the drop probability. The DiffServ model provides a more scalable way to handle QoS than IntServ; however, it is designed to provide differentiated services and not guaranteed services, even though these are possible. Its main goal is to differentiate traffic, giving precedence to some classes of traffic over others so that the overall perceived QoS is enhanced.
In the RSVP model, resources are allocated to individual traffic flows or flow aggregates. The aim of RSVP is to guarantee a certain level of QoS, such as bandwidth and end-to-end delay. IETF has proposed a deterministic framework for computing upper end-to-end delay bounds in packet networks (Shenker et al., 1997). In the given framework, incoming traffic is shaped at the ingress with a token bucket shaper such that flow-specific queuing delays in downstream routers are eliminated.

A token bucket shaper has two parameters $r$ and $b$, both given by the TSpec. $r$ tokens per second are placed in a bucket of depth $b$; in other words, $b$ is a measure of how many tokens fit into the bucket. Figure 2.5 shows a token bucket shaper with parameters $r$ and $b$. Incoming unshaped traffic is placed in a queue. As long as there are tokens in the bucket, the shaper will remove a token and pop the next packet from the queue and send it. If packets are dispatched faster than tokens are poured into the bucket, the bucket will eventually drain and stall the packet flow through the shaper until new tokens are available. If packets arrive at the same rate as tokens are added then a steady stream of packets will flow through the shaper with rate $r$ packets per second. If the average arrival rate of packets is higher than $r$ then the bucket will fill up. Once the bucket is full, new tokens will be dropped and hence will never be used by the shaper.
2.1.5 VPN Services in MPLS Networks

A VPN is an overlay network where nodes, or VPN sites, are connected with virtual links, hereafter called virtual connections. A VPN service requires a private layer 3 (L3) address space and traffic isolation such that no traffic leaks out of nor into the VPN. A virtual connection is often associated with a set of QoS constraints such as minimum bandwidth and end-to-end delay. The essence of providing a VPN service is to virtualize physical resources efficiently. In traditional IP networks, it is possible to create overlay networks using techniques like IP-in-IP encapsulation (Simpson, 1995; Woodburn and Mills, 1991) and IPsec (Kent and Atkinson, 1998). However, as described in Section 2.1.1 it is hard to design such overlay networks and provide QoS efficiently, because they are based on the traditional IP forwarding paradigm. Thus, this type of VPN has not been implemented by network providers. On the other hand, such VPN techniques have gained widespread popularity in customer-based VPNs, which are implemented as an overlay of a plain IP service. Provider-based VPNs typically create overlays of lower layers such as ATM, Frame Relay and in recent years MPLS. Because customer-based VPNs are implemented on top of the traditional Internet services, they inherit best-effort forwarding properties. A few attempts to improve on the best-effort service using customer-based overlay networks are Resilient Overlay Networks (RON) (Andersen et al., 2001), OverQoS (Subramanian et al., 2003), QoS-aware routing in overlay networks (QRON) (Li and Mohapatra, 2004) and Overcast (Jannoti et al., 2000). However, these approaches only improve on the best-effort service, they cannot guarantee specific levels of QoS. We do not provide any details of such overlay networks here as our interest is in providing guaranteed services.

One popular provider-based VPN service is the Virtual Private Wired Service (VPWS) also known as virtual leased line. VPWS extends Layer 2 connections to customer sites that are mapped onto virtual connections across the core network. Network providers have traditionally used ATM virtual/permanent circuits and Frame Relay connections to realize such virtual connections across the core network.

In MPLS networks, virtual connections are realized with LSPs. The large body of
standardization work to adopt the DiffServ and IntServ models makes it straightforward to implement virtual connections with QoS guarantees. Furthermore, label stacking allows the provider to tunnel traffic that belongs to different VPNs over the same LSP across the core network, using the inner label as VPN identifier. The label stacking property sets it apart from ATM and Frame Relay in terms of scalability, which makes MPLS more suitable for providing VPN services. Provider-based VPN services have evolved substantially over the past few years but it is hard to obtain a good understanding of the wide range of recent VPN service constructions and implications using existing literature. Therefore, we provide a comprehensive overview of recent directions in provider-based VPN services in Chapter 3.

Despite recent advances in provider-based VPN services, to the best of our knowledge, no effort has been made to incorporate both resilience and QoS into one VPN architecture. Today when a customer needs resilient QoS connections, there are two options in theory. One is to purchase redundant resources and multi-home the VPN. Multi-homing in this context means that the customer either buys QoS connections from two different network providers or purchases two disjoint QoS connections from one provider. The other option is to purchase resilient QoS connections. This option is not viable in practice, because today’s network providers are reluctant to offer resilient QoS connections. As a consequence, customers have no choice but to use multi-homing. This results in expensive VPNs that are complex to manage. Not only does a multi-homed VPN have twice as many virtual connections, it also requires failure detection and recovery mechanisms. The high cost and complexity potentially prevents new reliable services evolving. Ideally, a resilient VPN should not be more complex for a customer to manage than a non-resilient VPN.

Intuitively, a network provider can implement resilience more effectively than its customers. A network provider can facilitate finer grained redundant resource usage and implement backup resource-sharing strategies to further improve the network performance. Moreover, idle backup resources can be used to forward best-effort traffic when the provider implements resilience. However, no quantitative study has to the best of our knowledge been presented that shows how much more efficiently resilience can be implemented by the network provider.
2.1.6 Programmable Virtual Private Networks

Programmable virtual networks extend provider-based VPN services so that virtual sites as well as virtual links are provided. A virtual site can be used by customers to implement functionality “in the cloud”. Typical functions are customized routing to tailor the VPN topology and multi-casting. Another possible functionality is content adaptation such as recoding multimedia streams; these are useful in voice and video conferencing.

Furthermore, programmable virtual networks are promoted as a solution for fast and easy provisioning of new innovative services. The basic idea is to provide multiple programmable virtual private networks over one physical infrastructure. Each virtual network customer can install and run customized code in virtual sites to control routing and support application-specific tasks. A number of different programmable virtual network architectures have been proposed over the past few years, for example Tempest (Rooney et al., 1998), Virtual Active Network (Brunner and Stadler, 2000) and the Programmable Virtual Network architecture (Nguyen et al., 2002). However, resilience issues related to programmable virtual networks have not been addressed in the current literature.

2.2 Introduction to Resilience Provisioning

Resilience denotes how tolerant a virtual connection is to network failures such as single link or node failures. High resilience means that the virtual connection recovers connectivity quickly after a failure occurs, whereas low resilience means that the virtual connection takes a relatively long time to recover from a physical failure. No resilience basically means that the virtual connection is recovered on a best-effort basis.

To provision a resilient virtual connection (or resilient connection for short) in an MPLS network, one service and a set of backup LSPs must be computed. We will use the term path and LSP interchangeably and thus refer to these LSPs as the service and backup paths. When a service path fails, its traffic is recovered using one of the
associated backup paths. A service path can be recovered on a link by link basis (Kodialam and Lakshman, 2001; Lee et al., 2004; Alicherry and Bhatia, 2004; Pan et al., 2005; Bremler-Barr et al., 2001), on a segment basis (Li et al., 2002; Bejerano et al., 2005) or end-to-end from the ingress to the egress LERs (Kar et al., 2002; Kodialam and Lakshman, 2003; Norden et al., 2004; Xu et al., 2004; Orda and Sprinson, 2004; Liu et al., 2005). Figure 2.6 illustrates the three different recovery scopes.

In Figure 2.6 (a), each link along the service path $SP$ is protected by a backup path. That is if link $a$ fails then the traffic is moved from the service path to $BP1$ at the ingress LER. An advantage of this approach is that failure detection is very fast because no failure notification must be sent to an upstream node. Hence, this recovery scope is often referred to as fast reroute. On the other hand, a backup path must be established for each link that the service path traverses and thus this scope potentially results in poor resource utilization. The main applicability of this recovery scope is when different service paths can be collectively treated in the same way when a failure occurs and thus share the same backup path. Therefore, a link-by-link recovery scope is suitable for providing resilience in a DiffServ QoS model.

In the segment recovery scope shown in Figure 2.6 (b), the service path is divided into segments and a backup path is established for each segment. Here, when link $b$ fails, the upstream LSR from the failure must signal the ingress LER, which in turn moves traffic from the service path to $BP1$. The segment recovery scope does not require as much network resources as the link-by-link recovery scope, but on the other hand a failure signaling protocol must be implemented. Furthermore, the failure signaling increases the recovery time. This recovery scope is also not suitable for fine-grained recovery, because the core LSRs must then maintain potentially very large amounts of state information.

The end-to-end recovery scope is shown in Figure 2.6 (c). Here, a backup path is established for each service path. Core LSRs do not need to maintain any recovery information at all in this recovery scope. This property makes the end-to-end recovery scope suitable for fine-grained recovery and provides a good platform to implement the IntServ model. However, the core LSRs may need to implement fail-
ure signaling. In the figure, a failure at link \( d \) will trigger the upstream LSR to send a notification to the ingress LER, three hops away. Thus, the recovery time can potentially be longer than for the other two recovery scopes. It is possible to implement hybrid recovery scopes in an MPLS network whereby, for instance, a link-by-link recovery scope is used to recover DiffServ connections while an end-to-end recovery scope is used for IntServ connections.

When backup paths are computed and provisioned depends on the provided level of resilience. There are two major classes of resilience, protection- and restoration-based. The term protection-based resilience is used to describe resilient connections that are protected by a precomputed and preestablished backup path that may or may not be used by preemptive flow aggregates, as long as the protected resilient connection operates over its service path. Preemptive flow aggregates typically carry best-effort traffic without any resilience guarantees. Once a failure occurs along the service path, the protected flows are moved over to the associated backup path. The preemptive flow aggregates are dropped or otherwise rerouted to favor the protected flow aggregate. Protection-based resilience can provide high resilience because the backup paths are computed and fully established before the protected service path fails.

Restoration-based resilience is used for virtual connections that carry flows with less stringent resilience requirements. Here, the backup path is not fully computed or used beforehand. Once a failure has been detected along the service path, an attempt is made to compute and establish a backup path. If a backup path is found, then the resilient traffic flows are moved to the backup path. Clearly, restoration-based resilience does not guarantee that traffic can be restored upon failure because there might be insufficient resources available to establish a backup path. Furthermore, restoration-based resilience takes more time to recover from network failures, because the backup path is not fully established before the failure occurs and thus in addition to failure detection, the disruption time includes path computation and path setup. To improve the restoration time, the backup path can be pre-computed but not fully established.
Figure 2.6 Recovery scopes.
Below is a summary of common recovery types including an indication of which resilience class they implement (Sharma et al., 2003):

- **1+1** Two paths are computed and established for each resilient connection and traffic is duplicated at the ingress LER and sent along both paths simultaneously. The egress LER then chooses packets from either path. This recovery type provides the shortest recovery time; however, it duplicates the traffic through the network. 1+1 recovery is only used to provide protection-based resilience. This recovery type is commonly used in SONET/SDH rings in the optical domain.

- **0:1** The backup path is not computed or established until the service path has failed. 0:1 recovery therefore cannot guarantee to recover traffic. Moreover, this recovery type takes a long time to recover traffic because the recovery procedure includes backup path computation and provisioning. The 0:1 recovery type is used to provide restoration-based resilience.

- **1:1** Two paths are computed and established for each virtual connection, a service and a backup path. In contrast to the 1+1 recovery type, the ingress LER only sends traffic on the service path during normal operation. Traffic is moved to the backup path only after the service path fails. Hence, the backup path can be used to forward preemptive traffic to reach a higher level of network performance. 1:1 recovery is used for protection-based resilience.

- **1:N** One backup path is established and is used to recover \( N \) service paths. As in the 1:1 recovery type, the backup path can be used by preemptive traffic as long as all \( N \) service paths are in operation. This recovery type achieves efficient use of the network resources, but it is hard to maintain QoS constraints if several service paths fail at the same time. This recovery type can be applied in both protection- and restoration-based resilience.

- **M:N** \( M \) backup paths are used to recover \( N \) service paths. Again, a backup path can be used for preemptive traffic as long as it is not used to recover a failing service path. As in the 1:N recovery type, M:N can be used to recover traffic in both protection- and restoration-based resilience.
2.2.1 Resilience Provisioning in MPLS Networks

The IETF MPLS working group has proposed a framework for MPLS-based recovery (Sharma et al., 2003). Its goal is to provide recovery mechanisms that are sufficiently fast for the end-user service. Moreover, different end-user resilience requirements should be met as closely as possible. This goal implies that an MPLS provider should implement more than one level of resilience. MPLS recovery should also maximize network performance; again this implies that multiple resilience levels should be implemented in the network. In (Autenrieth and Kirstadter, 2002), the authors introduce a new concept they call resilience classes. A resilience class specifies a maximum recovery time along with a specific level of QoS during both normal and failure operation. The authors propose four resilience classes, ranging from very fast recovery on a time scale less than 100 milliseconds to best-effort resilience. They further propose resilience-differentiation extensions to the DiffServ and IntServ QoS models (Xiao and Ni, 1999). In the DiffServ QoS model, the authors propose using the DSCP field to indicate which of the four resilience classes each packet belongs to. In the IntServ model, they propose extensions to the RSVP-TE protocol where the resilience attributes become part of the RSpec.

Using MPLS recovery allows faster recovery times than those possible using IGP routing protocol convergence, which often takes several seconds, or BGP, which requires several minutes. Furthermore, layer 3 recovery does not easily provide bandwidth and end-to-end delay recovery because of its inherently poor traffic engineering properties.

Although layer 0 and layer 1 mechanisms can recover quickly from link failures, they probably do so in a resource-wasteful way. They are not likely to have information about the layer 3 flows’ recovery needs and thus cannot distinguish between flows that require very fast recovery and flows that can tolerate longer recovery times. For instance, a stock broker service requires very high resilience while a file transfer probably can tolerate a few seconds disruption, but recovery mechanisms in layer 0 and 1 do not have any means to distinguish those two flows from each other and thus the same recovery treatment will be applied to both of them.
As opposed to layer 0 and layer 1 recovery, MPLS-based recovery is able to maintain fine-grained recovery, because it is tightly coupled with the network layer. MPLS can also provide more efficient recovery from node failures than layer 0 and layer 1 mechanisms, because MPLS has knowledge of the layer 3 topology. Further, MPLS-based recovery is not dependent on any particular recovery mechanisms in lower layers. Thus MPLS can deliver a unified recovery framework in a network that uses several lower layer technologies.

Today, MPLS recovery and traffic engineering frameworks provide the necessary mechanisms to establish QoS paths including support for local, segment and end-to-end recovery. Failure detection and propagation mechanisms are also readily available (Huang et al., 2002) for both the DiffServ and IntServ QoS models. The issue when providing resilient QoS connections with high resilience requirements in an MPLS network is therefore the actual path computations. Typically, an algorithm that computes physically disjoint paths should achieve high network performance and have low running time so that it can be used in production.

### 2.3 Providing Resilient QoS Connections

When a customer wishes to connect two sites with a certain level of resilience and QoS, the customer generates a request $\text{Req} = (s, d, T\text{Spec}, R\text{Spec})$ that is submitted to the network provider. $s$ is interpreted as the ingress node and $d$ as the egress node, which are the two nodes that the network provider will attach to the customer sites. $T\text{Spec}$ specifies the bandwidth requirements and $R\text{Spec}$ determines the end-to-end delay bound. Moreover, the network provider wants to optimize its network performance and thus adds an optimality criterion like minimum cost or amount of network resources used. The network provider is therefore faced with the problems of computing, signaling and monitoring the resilient QoS connection. As mentioned in Section 2.2.1, signaling and monitoring functions are readily available, but computing resilient QoS connections is still an open research topic that needs to be addressed.

Computing resilient QoS connections involves finding a service path and a set of backup paths such that the QoS constraints are satisfied. In the literature, this prob-
Literature Review

Provisioning resilience in provisioning QoS connections is most commonly dealt with in an on-line routing context (Kodialam and Lakshman, 2003; Kodialam and Lakshman, 2001; Li et al., 2002; Xu et al., 2004; Alicherry and Bhatia, 2004; Norden et al., 2004; Orda and Sprinson, 2004; Bejerano et al., 2005). That is, when a request for a resilient QoS connection arrives, the routing algorithm tries to satisfy it using only information about the current network state. No assumptions are made about future arrivals nor are any previously provisioned connections rerouted. In accordance with the traffic engineering objectives, the produced paths should be selected such that as many future requests can be satisfied as possible, thus providing optimal network performance. Several optimization criteria stem from this objective, such as minimizing the amount of network resources used, provisioned bandwidth, cost or path lengths.

Algorithms that compute resilient QoS connections proposed in the literature commonly consider a single-link failure model (Kodialam and Lakshman, 2003; Kodialam and Lakshman, 2001; Li et al., 2002; Xu et al., 2004; Alicherry and Bhatia, 2004; Norden et al., 2004; Orda and Sprinson, 2004; Bejerano et al., 2005). A single-link failure model is easily extended to a single link and node failure model, as shown in Figure 2.7. In the figure, an LSR with three incoming links and four outgoing links is replaced with two LSRs. One holds the incoming links and the other holds the outgoing links. The LSR holding the incoming links is connected to the other LSR with a link that has infinite capacity. Now, a failure of the inserted link corresponds to a failure of the original node. An algorithm designed for a single-link failure model can be applied to a single link and node failure model, once the input network graph has been converted according to this simple procedure.

To understand the problem of computing resilient QoS connections, it is necessary to understand the strongly related problem of computing non-resilient QoS connections. In the next section we provide an overview of the research that addresses the problem of computing non-resilient QoS connections before we describe the problem of computing resilient QoS connections in more detail.
2.3.1 Computing Non-Resilient QoS connections

Formally, the problem of computing a non-resilient QoS connection is: given a network graph $G = \{V, E\}$, where $V$ is the number of nodes and $E$ is the number of links in the graph, and a request $\text{Req} = (s, d, T\text{Spec}, R\text{Spec})$, find a feasible path from $s$ to $d$ that meets the constraints given by $T\text{Spec}$ and $R\text{Spec}$. This problem is often coupled with an optimality criterion such as minimum cost, number of hops, end-to-end delay or total reserved bandwidth.

The general problem of finding a feasible path given multiple constraints is called the Multi-Constraint Path (MCP) problem and is known to be NP-complete (Jaffe, 1984; Garey and Johnson, 1979). The related but restricted general problem of finding a feasible path given two constraints that satisfies one constraint while being optimal on the other is called the Restricted Shortest Path (RSP) problem, and it too has been shown to be NP-complete (Ahuja et al., 1993).

QoS constraints can be divided into two classes, bottleneck and additive constraints. For instance, bandwidth is a bottleneck constraint and end-to-end delay and jitter are additive constraints. One way to handle a problem with both bottleneck and additive constraints is to deal with the bottleneck constraints in a preprocessing step whereby all links in the network that do not have enough residual capacity are pruned, effectively reducing the problem to one with only additive constraints. As a consequence, when the given QoS constraints are bandwidth and end-to-end delay the preprocess-
ing method can be applied to reduce the problem to one that only considers a single constraint, that is end-to-end delay, and thus the problem can be solved in polynomial time. This method has been applied in numerous publications (Wang and Crowcroft, 1995; Pornavalai et al., 1998; Neve and Mieghem, 1998; Korkmaz and Krunz, 2001; Lui et al., 2004; Banerjee and Sidhu, 2002).

In a packet-switched network, QoS constraints are related to each other in ways determined by the packet schedulers used in the LSRs, as mentioned in Section 2.1.4.2. Here, the end-to-end delay is a function of the provisioned bandwidth, chosen path and burstiness of the source. By exploiting this dependency, the problem of finding a feasible path given bandwidth and end-to-end delay constraints becomes polynomial. Stiliadis and Varma provide a framework in which this dependency is exploited (Stiliadis and Varma, 1998). In their framework, the network nodes are called Latency Rate (LR) servers. An LR server implements a work-conserving packet scheduler that provides bandwidth guarantees. Schedulers like Weighted Fair Queuing (WFQ), also called Packet-level Generalized Processor Sharing (PGPS), Virtual Clock, Self Clocked Fair Queuing (SCFQ), Weighted Round Robin and Deficit Round Robin are all work-conservative and provide bandwidth guarantees. In a network of LR servers, the end-to-end delay is a function of the provisioned bandwidth, chosen path and the burstiness of the flow at the source. The upper bound of the end-to-end delay \(D_m\) on a connection specified by a TSpec \((t, r, b, M)\) in a network of LR servers is given by:

\[
D_m = \frac{(t - R)}{(t - r)} \cdot \frac{b}{R} + \sum_{(i,j) \in P} \left( \frac{M}{R} + \frac{M_{ij}^m}{C_{ij}} + \text{prop}_{ij} \right) \tag{2.1}
\]

where \(t\) is the peak rate, \(r\) the sustainable rate, \(b\) is the burst tolerance and \(M\) is the maximum packet size on the requested connection as specified by the TSpec. \(C_{ij}\) and \(\text{prop}_{ij}\) are respectively the capacity and the propagation delay of link \((i, j)\). \(M_{ij}^m\) is the maximum packet size of all LSPs that use link \((i, j)\). \(R\) is the minimum of the allocated bandwidths associated with the LSP in the traversed nodes. In other words \(R = \min(R_1, R_2, ..., R_{|P|})\) where \(r \leq R_i < t\) is the allocated bandwidth for the LSP in node \(i\) and \(|P|\) is the length of \(P\).

In (Pornavalai et al., 1998) the authors present an algorithm that finds a shortest (number of hops) end-to-end delay path that meets the given bandwidth and end-
to-end delay constraints by exploiting the dependency between the end-to-end delay and provisioned bandwidth as expressed in Eq. (2.1). In this approach, \( R \) is first assigned to \( r \) and then all links in the network with less residual capacity than \( R \) are pruned. The proposed algorithm’s run time is comparable to the Bellman-Ford shortest-path algorithm, which is \( O(|E||V|) \) where \(|E|\) is the number of links and \(|V|\) is the number of nodes in the network (Cormen et al., 1990). Ma and Steenkiste presented similar algorithms (Ma and Steenkiste, 1997) with the same run time bound that compute shortest paths based on several different optimality criteria, such as:

- **Widest Shortest** - select a feasible path with the minimum hop count; if more than one such path exists, then choose the one with the most residual capacity. If several such paths exist, one is randomly selected.

- **Shortest Widest** - select a feasible path with the maximum reservable bandwidth; if several such paths exist, then choose the one with the minimum hop count. If several such paths exists, one is randomly selected.

- **Dynamic-Alternative** - select a widest minimum-hop path. If no feasible minimum-hop path exists, find the widest path that is one hop longer. If several such paths exist then one is randomly selected.

- **Shortest-Dist(\(P, l\))** - select a path with the shortest distance according to \( \sum_{i \in P} \frac{1}{H_i} \), where \( H_i \) is the residual capacity on link \( i \). This algorithm was shown to be efficient in previous work (Ma et al., 1996).

The authors’ experiments show that the optimality criteria that favor short paths in terms of number of hops such as Widest Shortest and Shortest Distance result in higher network performance than optimality criteria that favor longer paths such as Shortest Widest, when network performance is measured in terms of blocking rate, that is the fraction of rejected requests for QoS connections.

The dependency between end-to-end delay and bandwidth can be taken one step further by using the provisioned bandwidth \( R \) as a variable. The provisioned bandwidth must be at least as high as the required sustainable rate \( r \) and it must be smaller than
the peak rate $t$ (given by Eq. (2.1)). Intuitively, if the end-to-end delay cannot be met on path $P$ when using $r$ units of bandwidth, then use Eq. (2.1) to compute a minimum required bandwidth $R = r + \delta$, $\delta \in [0, t - r)$ such that the end-to-end delay bound as given by the RSpec is met.

Now, the problem is: given a network graph with LR servers $G = \{V, E\}$ and a request $Req = (s, d, TSpec, RSpec)$, find a feasible path from $s$ to $d$ and its minimum required bandwidth $R$ such that the path meets the given constraints. Additionally, the solution is subject to an optimality criterion. Ma and Steenkiste proposed a set of algorithms that address this problem (Q. Ma and P. Steenkiste, 1998). Their algorithms iterate over the Bellman–Ford algorithm once for each available residual capacity in the network and then choose a feasible path according to an optimality criterion. They use the following optimality criteria:

- **Widest Shortest** - as above.
- **Shortest Widest** - as above.
- **Minimum Bandwidth** - select a feasible path that requires reservation of the minimum amount of bandwidth. If several such paths exist then choose the path with minimum hop count. If several such paths exist then select one at random.
- **Shortest Delay** - select a feasible path with the shortest end-to-end delay when the maximum reservable bandwidth is used. If several such paths exist then select one with the minimum hop count. Again, if several such paths exist then select one at random.

In accordance with their previous results, their simulations show that the Widest Shortest optimality criterion achieves best network performance, and perhaps surprisingly the Minimum Bandwidth optimality criterion performs worst. The reason Minimum Bandwidth optimality criterion performs worst is that it is greedier than the other criteria and thus has a higher tendency to saturate critical links. Their algorithms’ run times are bounded by $O(|E|^2 |V|)$. Though this run time is polynomial, it is too expensive for larger networks, particularly for dense networks.
2.3.2 Computing Resilient QoS connections

Formally, the problem of computing resilient QoS connections is: given a network graph $G = \{V, E\}$ and a request $Req = (s, d, TSpec, RSpec)$, find a service path and a set of backup paths such that the $TSpec$ and $RSpec$ are satisfied. In addition, the solution may be subject to an optimality criterion such as combined minimum cost, hop-count or resource usage. This problem is at least as complex as the single path QoS connection computation problem and thus is NP-complete in its most general form. In data communication networks however, the constraints are often bandwidth and end-to-end delay. Given this restriction of the general problem several approaches have been described in the literature.

An even more reduced variant of this problem is when only bottleneck QoS constraints are considered. Here, the problem basically reduces to finding a service path and a set of backup paths from $s$ to $d$ such that all links have at least $r$ residual capacity. In the end-to-end recovery scope, this problem is solvable in polynomial time (Suurballe and Tarjan, 1984) comparable to Dijkstra’s shortest-path algorithm (Cormen et al., 1990).

In the remainder of this section we consider the problem of a given bandwidth and end-to-end delay constraint and optionally an optimality criterion. The end-to-end delay can be replaced with a jitter constraint without affecting the arguments described below. A common approach described in the literature is to convert the end-to-end delay constraint to an effective bandwidth in a preprocessing step, thus reducing the problem to one with only a bottleneck constraint (Kodialam and Lakshman, 2003; Kodialam and Lakshman, 2001; Li et al., 2002; Xu et al., 2004; Alicherry and Bhatia, 2004; Norden et al., 2004). Converting end-to-end delay to an effective bandwidth basically requires the network nodes to be LR servers. As shown in Section 2.3.1 the end-to-end delay depends on the chosen path and provisioned bandwidth. When converting the end-to-end delay to an effective bandwidth we need to know the path to find an accurate effective bandwidth. However, the path is unknown in the preprocessing step. Hence, to ensure that the computed paths will not breach the end-to-end delay, a pessimistic path estimate must be made when converting the
end-to-end delay into an effective bandwidth. The pessimistic estimate possibly leads to suboptimal network performance.

Another approach used in the literature is to prune all links that do not satisfy the given bandwidth constraint in a preprocessing step, as described in Section 2.3.1. This approach effectively reduces the problem to one that only deals with an additive QoS constraint such as end-to-end delay. This approach has attracted some attention recently (Orda and Sprinson, 2004; Bejerano et al., 2005).

As mentioned in Section 2.3.1, yet another approach can be taken whereby the dependency between end-to-end delay, chosen path and provisioned bandwidth is exploited by using the provisioned bandwidth $R$ as a variable. This approach can potentially turn an infeasible path into a feasible path. If a path does not meet the end-to-end delay when $R = r$, it may meet the delay bound when $R = r + \delta$, $\delta \in [0, t - r)$ where $t$ is the given peak rate. This approach has been proven successful when routing non-resilient QoS connections (Q. Ma and P. Steenkiste, 1998). Here, the problem is: given a network of LR servers $G = \{V, E\}$ and a request $Req = (s, d, TSpec, RSpec)$, find a service path and a set of backup paths and their respective required minimum bandwidths such that the $TSpec$ and $RSpec$ are satisfied. By exploiting the described dependency, a routing algorithm can potentially provide high network performance. This approach has to the best of our knowledge not been addressed in the current literature. In particular, this approach has not been addressed in the end-to-end recovery scope for the 1+1 and 1:1 recovery types.

2.4 Summary

This chapter introduced MPLS, traffic engineering, VPNs, QoS and resilience provisioning. A number of issues related to providing a complete architecture that delivers resilient QoS connections have been pointed out. We summarize these issues below.
2.4.1 Outstanding Issues in Current Literature

- VPN services have evolved rapidly over the past few years. It is difficult to obtain a good understanding of the wide range of recent VPN service constructions and implications using the existing literature.

- Though it is intuitive that resilience can be more efficiently implemented by a network provider than by its VPN customers, no study has been presented that supports this intuition.

- Very little effort has been made in the current literature to address resilience provisioning in VPNs and programmable virtual networks. More specifically, there are no programmable virtual network architectures with native support for differentiated resilience and QoS described in the literature.

- A substantial amount of work has addressed the problem of computing resilient QoS connections where the QoS constraints are bandwidth and end-to-end delay. This work has used approaches that effectively convert the problem of one bottleneck and one additive constraint to a problem with either a single bottleneck or additive constraint.

We have identified yet another approach that has been successfully applied to compute non-resilient QoS connections. In this approach the bottleneck and additive constraints are considered simultaneously by using the provisioned bandwidth as a variable. Thus, a solution in this approach not only contains two link-disjoint paths but their respective bandwidths as well, such that the given QoS constraints are met on both paths. This approach has to the best of our knowledge not been addressed in the current literature. In particular, this approach has not been addressed in the end-to-end recovery scope for the 1+1 and 1:1 recovery types.
Chapter 3

Recent Directions in Virtual Private Network Solutions

Chapter 2 introduced customer- and provider-based VPNs. As the essence of this thesis is to improve on provider-based VPN services this chapter provides a more detailed overview of recent directions in provider-based VPN solutions (Rosenbaum et al., 2003). Provider-based VPN services have been available for decades but not widely used until recently. A combination of new VPN technologies, network technologies and business communication requirements are the driving force behind the take-up of VPN services. In this chapter we break down recent VPN solutions into three building blocks: VPN type, QoS specification model and VPN provisioning model. These building blocks are used to categorize VPN solutions in terms of scalability and flexibility and relate different VPN solutions to potential customer segments.

An introduction to recent directions in provider-based VPN solutions is given in Section 3.1 followed by an examination of provider-based VPN types in Section 3.2. Different QoS specification models are described in Section 3.3. In Section 3.4, we give an overview of VPN provisioning models. Section 3.5 outlines different VPN solutions and how they relate to each other in terms of scalability from the network provider perspective and flexibility from the customer perspective. Recent research approaches for optimizing VPN provisioning are investigated in Section 3.6.1. Conclusions are given in Section 3.8.
3.1 Introduction

The business world today has seen a major trend towards using Information Technologies (IT) to improve competitiveness. This new dependency on IT has created a demand for efficient communication between remote sites of an organization. One prominent communication technology that address this demand is the VPN service. VPN technology has been around for several decades but it is only during the past decade that it has become more feasible for a wider range of organizations. Feasibility is normally measured in terms of the benefit to cost ratio where the benefits of the VPN service have increased through the need for fast and reliable information flow, and the costs are dropping through cheaper equipment and new network technologies.

The growing number of supported VPNs increases the operational complexity and cost for a service provider. Operational elements include VPN deployment and management. Deployment involves optimizing resource usage within the network and setting up the configurations. Management involves monitoring and maintenance of VPNs. Operational complexity and cost affect the customer because it takes effort from both parties to deploy and manage a VPN. The difference between the customer and the network provider is that the customer only operates one to a few VPNs, whereas the network provider operates potentially thousands of VPNs.

Therefore, a good VPN solution minimizes the operational complexity and cost for both the network provider and its customers. A VPN solution is defined in this chapter as a composition of components from three different building blocks: the VPN type, the QoS specification and the VPN provisioning model in the provider network.

The various combinations result in different VPN solutions with unique characteristics. This chapter discusses the feasible combinations of the building-block components and the characteristics of the resulting VPN solutions.
3.2 Provider-Based VPN Types

Provider-based VPNs come in two fundamentally different types, layer two (L2) and layer three (L3) VPNs. They target different customer segments, where customer control and flexibility are traded against customer simplicity and maintainability. In provider-based VPNs, a customer site is connected via a customer edge (CE) node located at the customer premises to a provider edge (PE) node.

3.2.1 L2 VPNs

In L2 VPNs, the provider extends layer two services to the customer sites. A key property of L2 VPNs is that the provider is unaware of L3-specific VPN information. The customer and the provider do not exchange any routing information with each other, and forwarding decisions in the provider network are based solely on L2 information such as MAC address, ATM VC identifier, MPLS label and port number.

Currently, two different approaches to L2 VPNs are described in the literature, Virtual Private Wire Service (VPWS) and Virtual Private LAN Service (VPLS) (Augustyn and Serbest, 2003). The major difference between the two is that the VPWS provides a site-to-site service while the VPLS provides a multisite-to-multisite service. The VPWS approach can be regarded as a generalized version of the traditional leased line service, in which the sites are connected in a partial or full mesh. The VPLS approach emulates a LAN environment where a site automatically gains connectivity to all the other sites attached to the same emulated LAN.

The L2 technology used is determined by the provider. If a customer possesses sufficient knowledge, the required L2 equipment, can be supplied and managed by the customer to set up the CE node. If not, the customer must buy additional equipment and management services from a third party, typically the provider. Besides the additional cost, it will make the solution less flexible.

L2 VPNs give customers flexibility in terms of routing control and L3 protocol usage. However, they do require the customer to either buy and maintain VPN gateways or hire additional services from an external entity, such as the provider.
3.2.2 L3 VPNs

In L3 VPNs, the provider offers L3 connectivity, typically IP, between customer sites. The customer can optionally specify more advanced L3 topologies than simple full or partial mesh, such as intranet and extranet integration or hub-and-spoke. The VPN topology can be thought of as a collection of nodes, representing the VPN sites and virtual links that connect them. We will use virtual link and virtual connection interchangeably in the remainder of this chapter.

Forwarding in L3 VPNs is based on L3 information, such as IP addresses, hence the provider and customer edges exchange routing information. The provider edge nodes (PEs) maintain separate routing contexts, one for each directly attached VPN.

At present, two L3 VPN approaches dominate, BGP/MPLS VPN (Rosen and Rekhter, 1999) and Virtual Router (VR) (OuldBrahim et al., 2002). Both approaches concentrate the VPN functionality at the edge of the provider network and hide VPN-specific information from the provider core nodes (P nodes), to improve scalability. In the BGP/MPLS VPN approach (also referred to as 2547 after the relevant IETF RFC (Rosen and Rekhter, 1999)), a routing context is represented as a separate routing and forwarding table in the PE. Each PE node runs a single instance of a BGP variant called Multi-protocol BGP (MPBGP) (Bates et al., 2000) for VPN route distribution. The PE nodes use MPLS labels to keep VPN traffic isolated and transmit the packets across the core network in tunnels. The tunnels are not necessarily MPLS tunnels; they can be of any type, such as IPSec or GRE tunnels. If a tunnel type other than MPLS is used, the only nodes that need to know about MPLS are the PEs. Any routing protocol can run between the CE nodes and the PEs, but in practice the customer must use the routing protocol chosen by the provider.

In the VR approach, PE nodes have one VR instance running for each VPN context. A VR emulates a physical router and functions exactly like one. VRs belonging to the same VPN are connected to each other via tunnels across the core network. As in the 2547 approach, the tunnels can be of any type.

One advantage of the VR approach is that a customer using an IGP protocol, such as
OSPF, can use it directly to the other side. In the 2547 approach, on the other hand, the customer routing is terminated in the PE node where the provider’s internal BGP instance takes over. Hence, from the customer perspective, there is more extensive flexibility in the VR approach. However, the VR approach implies a stricter restriction on the number of supported VPNs per PE, compared to the 2547 approach, because each routing context runs as a separate routing instance. There is no difference between the approaches in terms of processing and maintaining routing information.

L3 VPNs are less scalable than L2 VPNs from the provider perspective. The main reason is that when the provider offers L3 connectivity, routing information must be taken care of explicitly in the provider network. It does, however, ease the customer’s operational burden compared to L2 VPNs for the same reason. Furthermore, L3 VPNs are less flexible than L2 VPNs. In L3 VPNs the L3 topologies are configured and maintained by the provider. Hence, the customer must ask the provider to make any changes, resulting in longer lead-times and likely additional costs. In L2 VPNs, the customer has total control over L3 topologies and route distribution. Thus, a L2 VPN customer can reconfigure its own L3 topology and route distribution at any time.

### 3.3 QoS Specification Model

The traditional model for specifying QoS arrangements involves drawing a set of network service requirements between ordered pairs of customer sites. Each ordered pair defines a direction and volume of a data flow, and they are together called the traffic matrices. The traffic matrices model gives the customer fine granularity of control over the traffic flows between customer sites, and the network provider can offer a stringent service guarantee for each data flow. However, maintaining the traffic matrices can become complicated as the VPN grows in size.

Recent developments in VPN technologies have introduced a new type of QoS specification model called the hose model (Duffield et al., 1999), which may be desirable to VPN customers. In this model, two hoses are given to the customer; one is used to send traffic out to the network and the other is used to receive traffic from the
Recent Directions in Virtual Private Network Solutions

Figure 3.1 VPN QoS specifications.

network. This means that the network provider offers an *aggregate* traffic service to the customer, where all traffic going towards the customer site must fit within the receiving hose’s capacity and all traffic leaving the customer site must fit within the sending hose’s capacity.

The hose specification contains the egress bandwidth (outbound traffic) and the ingress bandwidth (inbound traffic) pair for every customer site that takes part in the VPN. The major advantage of this model from the customer perspective is that specification is made simple and there is a high potential for multiplexing gain (Duffield et al., 1999) from using aggregate requirements rather than traffic matrices. The simple specification allows changes to be made easily including changes in the VPN membership. However, in this model the customer loses control of the data flow between sites and the service guarantee is on an aggregate basis, meaning a coarser service guarantee level.

Figure 3.1 illustrates an example to show the difference in the specification requirements between the hose and matrices specification models. The hose model requires three pairs, one for each customer site. The matrices model requires six pairs for full connectivity between the customer sites. Each pair specifies the QoS requirements between the destination and source sites. Adding a new site in the hose model
simply means one more pair of requirements whilst six additional pairs are needed in the matrices model, and this number grows exponentially as the number of sites increases.

From a network provider’s perspective, the traffic matrices model specifies strong constraints that restrict the network optimization that can be achieved. The hose model specifies aggregate requirements that are less constraining, thus giving the network provider more flexibility to do network optimization. Network optimization involves mechanisms to improve bandwidth efficiency, load balancing, and traffic multiplexing.

### 3.4 VPN Provisioning Model

For both L2 and L3 VPNs, paths must be provisioned across the core network to connect PEs. A path can be used as a tunnel if the context requires it. Path provisioning can be done in a variety of ways. Depending on the target environment, one provisioning model may be preferable over others. Selection of a suitable model can be of vital importance to the provider. Hence, knowledge of the provisioning models’ main characteristics is crucial when designing a VPN solution.

There are three predominant provisioning models in the current literature: the basic full-mesh model, the sink-tree model and the hierarchical model. Figure 3.2 illustrates the various provisioning models over the same physical infrastructure. In the figure, each PE node requires a path across the physical infrastructure to all other PE nodes to meet the VPN requirements. CE nodes are omitted for clarity. Each layer in the figure is explained in detail in the corresponding sections below.

Modern VPN technologies are based on connection-oriented traffic-engineered core network protocols, such as ATM and MPLS (Awduche et al., 2002; Aukia et al., 2000; Xiao et al., 2000). In the following sections, path refers to a traffic-engineered unidirectional path connecting two nodes together. A path can additionally be associated with a certain level of QoS and resilience. The number of hops that a path must take to reach the terminating node is called the path-hop count. A path’s path-hop
count is an indicator of how much state information, such as ATM VC identifiers and MPLS labels, is required along the path.

### 3.4.1 Full-Mesh Provisioning Model

The full-mesh provisioning model is the simplest provisioning model. It basically connects the VPN sites with dedicated PE-to-PE paths that reflect the VPN topology.

Each VPN site potentially requires a virtual link to all other sites in the same VPN. Each virtual link is mapped over the CE nodes’ adjacent PE nodes, resulting in full-mesh path provisioning between the PEs, as Figure 3.2 illustrates. If more than one VPN requires a virtual link over the same two PEs, their virtual links can be multiplexed onto the same path or tunnel across the core network. Multiplexing will decrease the number of provisioned paths seen by the core network and thus improve scalability.

However, this model potentially requires each PE node to be directly connected via
an explicit path to every other PE node in the provider network. Adding a new PE node thus requires provisioning of \(n - 1\) new paths across the core network, where \(n\) is the number of PEs. This property leads to the known \(O(n^2)\) problem (Hummel and Hoffman, 2002). The number of provisioned paths across the provider network therefore grows exponentially with the number of PEs. Counting the path-hops in the full-mesh example in Figure 3.2 gives 30 path-hops for each PE. There are 10 PEs, hence a total of 300 path hops are required in the core network to achieve full connectivity. To connect a new PE node to \(P_x\) requires 64 additional path hops in the core network.

3.4.2 Sink-Tree Provisioning Model

The sink-tree provisioning model aims to reduce the total amount of state information in the provider network. A sink tree is a tree structure in which there are multiple sources represented by the leaves, and one destination, the sink, represented by the root. In the sink-tree provisioning model paths leading to the same PE node are organized as a sink tree, with the PE node as the sink, over the core network, as shown in Figure 3.2. The set of PE nodes requiring connectivity to the sink constitutes the sink-tree sources. The sink-tree provisioning model enables multiplexing of traffic belonging to different VPNs onto the same sink tree.

The potential number of paths in this model is relatively large. The reason is that each PE node has a uniquely calculated sink tree, potentially with \(n - 1\) leaves. Introduction of a new PE node requires a new sink tree. Therefore, the number of paths is of the order \(O(n^2)\) where \(n\) is the number of PEs. The major difference to the full-mesh model is that the paths are coordinated in tree structures and hence reduce the total amount of state information in the core network. The sink tree illustrated in Figure 3.2 counts 14 path hops, which gives a total of 140 path hops in the core network to connect all 10 PE nodes with each other. 140 path hops is a significant improvement to the 300 path hops required in the full-mesh provisioning model. If an additional PE node is connected to \(P_x\), it requires provisioning of a new sink tree with 15 path hops plus an additional path hop to each existing sink tree resulting in 25 new path hops for a total of 165 provisioned in the core network.
3.4.3 Hierarchical Provisioning Model

The hierarchical provisioning model aims to reduce the number of paths visible in the core network. As with other models, traffic belonging to different VPNs can be multiplexed onto the same path if appropriate.

PEs are not directly connected to each other in this model. Instead, they are connected to intermediate cross-connects in the core network that in turn provide connectivity to the other PEs, either directly or via other cross-connects, effectively creating an overlay network as illustrated in Figure 3.2 layer 0. Once the overlay network is created, paths can be provisioned in the overlay network according to the full-mesh model or, to reduce the required state information, according to the sink-tree model. Use of the sink-tree model on layer 1 is shown in Figure 3.2.

The total number of provisioned paths across the core network is reduced from $O(n^2)$ to $O(n + k^2)$ where $n$ is the number of PE nodes and $k$ is the number of intermediate full-mesh-connected cross-connects. The load on intermediate cross-connects may force introduction of new cross-connects. How many cross-connects are introduced depends on the expected number of VPNs, the number of sites and the traffic load. To improve scalability, parallel cross-connect hierarchies can be used in the provider network, where each hierarchy caters for a distinct set of VPNs. If the number of cross-connects becomes large, additional hierarchy layers can be introduced.

The path hop counting is divided between two distinct layers, layer 0 and layer 1, as shown in Figure 3.2. Path hops in layer 0 are visible in the core network while path hops in layer 1 are transparent. In layer 0, each PE node is connected to the central node over two hops, which gives 20 bidirectional hops or a total of 40 path hops. In layer 1, 10 PE nodes are fully connected with sink trees counting 10 path hops each resulting in a total of 100 path hops. In total, 140 path hops are used. This is equal to the sink-tree model but a major difference is that the hierarchical provisioning model only has 40 path hops visible in the core while the sink tree exposes all of its 140 path hops, giving a core path-hop fraction of about 30%. As a result the hierarchical model scales better than the sink-tree model.
When connecting a new PE node to $P_x$, four path hops are required to connect it to the central node in layer 0 plus an additional 21 path hops in layer 1 to maintain full connectivity, because the sink-tree provisioning model is used. In total, 25 new path hops are necessary, giving a total of 165 path hops. Again, this is equal to the sink-tree model, but the fraction of core path hops reduces to 26%, which emphasises the scalability benefit of the hierarchical provisioning model.

The use of hierarchical path provisioning implies higher overhead in terms of packet headers and complexity compared to the full-mesh and sink-tree models. Shortest-path connections between PEs, from the layer 0 perspective, is prohibited because PEs are connected to each other in layer 1. Thus, the hierarchical provisioning model potentially consumes more bandwidth than the other two models. On the other hand, extracting the VPN service from the core network enables the provider to have separate management groups for VPN service and core network operations.

### 3.4.4 Traffic Multiplexing

Each provisioning model is associated with a spectrum that bounds the amount of possible traffic multiplexing. The lower bound of the spectrum is defined by the minimum traffic multiplexing that must be done as a characteristic of the provisioning model. At the other end is the maximum traffic multiplexing that is achievable in the model. Moving along the spectrum alters the tradeoff between traffic multiplexing and QoS granularity.

In the full-mesh model, the minimum traffic multiplexing occurs when each VPN has a path between the source PE node and the destination PE. The maximum traffic multiplexing is achieved by sharing one path between source PE node and destination PE node for all attached VPNs. The network provider can move along this spectrum by increasing or reducing the number of VPNs per path.

Traffic multiplexing for the sink-tree model is somewhat more complicated but the tradeoff is similar to the full-mesh model, where the minimum traffic multiplexing occurs when each sink tree is associated with only one VPN. The maximum traffic multiplexing is reached when all attached VPNs’ traffic uses the same sink tree to
reach the destination PEs.

The level of multiplexing within a sink tree varies at different depths. At the leaves (which are the PEs), traffic multiplexing is done based on the destination PE node and the VPNs associated with this sink tree. At branching points in the tree, flows from sub-trees are multiplexed together. Therefore, the highest multiplexing point is where traffic from all the leaves merges onto one path. This scenario may or may not exist depending on the tree structure.

Compared to the full mesh, the minimum traffic multiplexed in the sink-tree model on average is greater because traffic from multiple PE node sources can be multiplexed onto the same path in the tree. Note that it is possible to reduce the amount of traffic multiplexing in a sink tree by reducing the number of PE node sources. Hence, the maximum traffic multiplexing achievable would be reduced to the traffic from PE node sources included in the sink tree. This is desirable in scenarios where each sink tree is customized for a specific VPN.

The hierarchical model aims to do multiplexing at the cross-connect overlay. Minimum traffic multiplexing occurs when the overlay is visible to one VPN only. If there is only one stage of cross-connect, then multiplexing is done at three points. Where traffic from a VPN enters the PE node, packets that are forwarded towards the same cross-connect are multiplexed together. At the cross-connect, the traffic is de-multiplexed according to the path that the packets take inside the cross-connect. All packets going on the same path are multiplexed together. At the end of the cross-connect, the traffic is again de-multiplexed according to the destination PE. This differs from other models because traffic destined for different PE nodes can be multiplexed together at the first or second stages. Maximum traffic multiplexing is achieved when one overlay is shared by all VPNs. It is possible to implement multiple overlays that are associated with different sets of VPNs. This allows better load balancing and can be used to achieve different levels of QoS requirements.
### 3.5 VPN Solutions

A VPN service connects customer sites to each other across a shared infrastructure owned and administrated by a network provider. A VPN service is implemented using a VPN solution realized in the provider network. A VPN solution is a combination of components from three major building blocks: VPN type, QoS specification, and provisioning. The choice of components from each building block depends on a variety of factors, such as provider infrastructure, expected number of VPNs, expected number of VPN sites, expected traffic volume, customer requirements on flexibility and customer network expertise.

Figure 3.3 shows how different compositions of building block components relate to each other in terms of scalability in the provider network. Scalability is measured in number of PEs, number of VPNs and number of VPN sites. Solutions further away from the origin are more scalable. In each quadrant there are three spectrum markers representing the provisioning models. The markers are intended to give an idea of...
where a provisioning model is applicable in terms of benefit–cost ratio. For instance, if a VPN solution extends beyond the corresponding spectrum marker, it results in a network with disproportionally high operational and maintenance costs.

Quadrant Q1 in Figure 3.3 contains VPN solutions composed from L2 VPNs and hose specification components. The major reasons for the high scalability properties of these solutions are that routing-related operations and maintenance are kept outside the provider network. The less stringent QoS requirements given by the hose specification model also allow a high level of VPN traffic aggregation in the provider network, hence extending the spectrum markers.

The next quadrant, Q2, shows VPN solutions built using L3 VPNs and hose specification components. They are in general less scalable than the solutions found in Q1 because the provider must maintain routing contexts in its network. The VPN solutions still benefit from the looser QoS specification.

Quadrant Q3 illustrates VPN solutions composed from L2 VPNs and matrices specification components. These VPN solutions are comparably scalable because the routing-related operations are pushed out to the customer premises. However, solutions in this quadrangle do not scale as well as the ones in Q1 because of the stringent QoS specifications.

Finally, quadrant Q4 shows VPN solutions built using L3 VPNs and matrices specification components. This quadrant contains, in general, the least scalable solutions. Their L3 nature requires the provider to handle L3 operations on the customer’s behalf. Furthermore, the matrices specification leaves little room for VPN traffic aggregation in the provider network.

Traditional VPN solutions, such as leased line service over a full-mesh provisioning model, reside in quadrant Q3. Recent VPN solutions are found in quadrant Q4, where the BGP/MPLS VPN approach over a full-mesh provisioning model is the most widely used.

Table 3.1 shows where to look for VPN solutions that best satisfy different customer
Table 3.1 VPN Customer preferences and VPN solutions.

<table>
<thead>
<tr>
<th>Routing control</th>
<th>QoS granularity</th>
<th>VPN solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>high</td>
<td>Q3</td>
</tr>
<tr>
<td>high</td>
<td>low</td>
<td>Q1</td>
</tr>
<tr>
<td>low</td>
<td>high</td>
<td>Q4</td>
</tr>
<tr>
<td>low</td>
<td>low</td>
<td>Q2</td>
</tr>
</tbody>
</table>

segments. Routing control refers to the customer’s need to control L3-specific operations such as route distribution and L3 topologies. The QoS granularity indicates what levels of site-to-site QoS constraints the customer wants to specify. Finally, the VPN solution indicates the quadrant in which the VPN solution that best meets the customer’s requirements is found. High routing control requirements are met by L2 VPNs and stringent QoS specification by the matrices specification model.

Customer needs will continue to diverge, leading to changes in the customer population of the quadrants. Traditionally, customers resided in Q3, and it is only in the past decade that populations have shown up in Q4. The close relationship between development of new technologies and customer needs results in a two-way drive, where new customer needs lead to new technologies and vice versa. New customer needs along with emerging L2 VPLS technologies and the hose specification model will drive VPN solution usage towards the upper half in Figure 3.3. However, the bottom half will still play an important role for some customer segments as outlined in Table 3.1.

Because different customer segments have fundamentally different VPN service requirements they cannot be satisfied by one single VPN solution. Therefore, a provider that targets more than one customer segment is likely to use several VPN solutions.

### 3.6 Research Directions in VPNs

Currently no work has to the best of our knowledge addressed the issue of providing resilient QoS guaranteed VPN services. Providing resilience in a VPN context primarily concerns the VPN provisioning model. In this thesis we will look at how
resilience can be integrated in the full-mesh and hierarchical provisioning models in an end-to-end recovery scope. Resilience in the sink-tree provisioning model will not be explicitly addressed herein, though some ideas can possibly be borrowed and extended from our work on the full-mesh and hierarchical provisioning models.

The hose specification model defines an interesting new problem that opens up a new degree of flexibility in network optimizations for the network provider. Two recent approaches to optimizing provisioning for the hose model that have not been discussed in previous sections are the customized VPN tree and dynamic resizing. Customized VPN trees are a new provisioning model whilst dynamic resizing works with other provisioning models.

### 3.6.1 Customized VPN Tree

In this provisioning model, a shared resource tree is customized for the use of a single VPN. This shared tree is optimized to be bandwidth efficient for the VPN. Computing the optimal shared tree is known to be NP-hard (Kumar et al., 2001), so polynomial-based approximations (Kumar et al., 2001) have been developed recently that give nearly optimal solutions. An example of the benefits of using a shared tree is illustrated in Figure 3.4.

Suppose a customer’s VPN consists of three sites, A, B and D, connected to PEs. For simplicity, assume all the sites require 1 Mbps ingress bandwidth and 1 Mbps egress bandwidth. The optimal customized resource tree consists of 1 Mbps reserved
bandwidth in all links in both directions. The bandwidth saving is achieved by using the knowledge of the VPN requirements as seen at node C. Both sites A and B have maximum egress bandwidth of 1 Mbps but the maximum ingress bandwidth at site D is only 1 Mbps. Thus only 1 Mbps needs to be reserved for traffic from node C to node D. If traffic to D exceeds 1 Mbps, node C will drop excessive traffic. From this example, we can deduce that the optimal customized tree is one that uses minimal network resources to satisfy the hose specification requirements of the VPN.

Although the customized shared VPN tree solution provides high bandwidth efficiency, the computational cost for the approximation is still high. The tree is customized based on the bandwidth requirements for each site and the location of the PE nodes the sites are attached to. Therefore, if the set of PE nodes changes or the bandwidth requirement is changed, the entire tree must be recomputed to remain optimal. A research direction would be to find temporal solutions that modify the tree for the required changes in the VPN without significant penalties.

Using customized trees also restricts the traffic multiplexing that can be done by the network, as each VPN must be forwarded along unique routes dictated by the customized shared tree solution. On the other hand, the shared tree allows VPN traffic from different PE node sources to be multiplexed onto the same path, as well as traffic that is forwarded to different PE node destinations. This can potentially improve the multiplexing of traffic within a VPN (Duffield et al., 1998).

One issue that has not been considered in the research on customized trees is the possibility of starvation. In Figure 3.4, node C allows the maximum traffic of 1 Mbps to flow to node D but the maximum aggregate traffic flow into node C destined for node D is 2 Mbps. Therefore, approximately 1 Mbps will be dropped in the worst case. However, if the dropped 1 Mbps consists mostly of traffic from node B, node B sees a starvation effect. Research is required on the consequences of this problem and mechanisms to alleviate it. One potential mechanism is the use of DiffServ (Ganti et al., 2002) to differentiate the traffic and drop the relatively less significant packets first.
3.6.2 Dynamic Resizing

This provisioning model is a complementary method that works with other provisioning models. The approach is to monitor traffic flow for each link and dynamically resize the link based on predictions from the information obtained. If a full-mesh model is used then each path will have monitoring and adjustments for all links along the path. If a sink-tree model is used then each branch of the tree will have monitoring and adjustments, and similarly for each tunnel in a hierarchical model.

The advantage of this model is that bandwidth is constantly adjusted to the changes in the VPNs without the need to restructure as in the customized shared VPN tree model. However, the on-demand nature of the approach means that QoS guarantees must be looser, to fit the delay in response to any increase in traffic for the VPN. The need to monitor and adjust each VPN per link places considerable demand on computation and memory resources at the routers and the scalability in the core for this approach should be investigated. Duffield et al (Duffield et al., 1998) provide a good analysis of the benefits that can be achieved using dynamic resizing for provisioning hose specifications and the various tradeoffs involved. The study also emphasized that traffic aggregates affect these benefits.

3.7 Provisioning Model Measurements

The measurement of state information consumption aims to clarify the relation between the different provisioning models, in terms of required state information, when the number of provisioned VPNs and the number of sites varies.

3.7.1 Methodology

The program used for the measurements was written in C and uses Dijkstra’s algorithm to calculate shortest paths in terms of number of hops. If a path traverses a node, it counts as one. The end points of a path are counted once for each end. In the full-mesh provisioning model the program connects PE nodes to each other with shortest paths. In the sink-tree provisioning model, the program creates optimal
sink trees to connect PE nodes. An optimal sink tree is one in which the sources are connected along the shortest path to the sink.

In the overlay provisioning model, a set of P nodes is selected as overlay points. These P nodes are connected to each other in a full-mesh where each path in the mesh is the shortest path between two overlay points. PE nodes are then connected to the closest overlay point with shortest paths. The state information required to connect overlay points together and to connect PE nodes to the overlay points, is counted in layer 0. PE nodes are connected to each other in layer 1 with optimal sink trees. All state information required to establish the sink trees is counted in layer 1, including state information in the overlay points. As an effect, state information at layer 1 in an overlay point is considered as part of edge state information. Conceptually, overlay nodes are both P nodes and PE nodes where the P node part belongs to layer 0 and the PE node part belongs to layer 1.

No multiplexing of VPN traffic is carried out in either provisioning model. A path connecting two PE nodes is exclusively used by a single VPN.

The measurements are based on MCI’s Internet backbone. MCI is a large North American carrier with a continent-wide network. MCI’s core topology is shown in Figure 3.5. This topology is given as input to the program along with a number of VPNs and the average number of sites per VPN. The program calculates the total number of sites and distributes them randomly over the geographical area represented by the core node topology. The program associates each site with the closest P node.
and then it creates a number of PE nodes at each P node. The number of PE nodes is given by \( n + 1 \) where \( s = q \times n + r \), \( s \) is the number of sites associated with a P node and \( q \) is the maximum number of sites per PE node (given as input). The sites associated with a P node are then connected to a PE node neighboring the sites’ associated P node. Hence, different numbers of VPNs and sites generate different topologies with respect to the network edge while the core remains the same.

### 3.7.2 Results

This section presents the results from running the program with varying numbers of either average sites per VPN or total number of VPNs. Each PE node can manage up to a maximum of 100 sites. The overlay points in the overlay provisioning model are nodes 9 and 15. When varying the number of VPNs, each VPN has an average number of 20 sites. When the number of average sites per VPN is varied, the number of VPNs is 100.

Figure 3.6 shows how the amount of state information in the PE nodes depends on the total number of VPN sites. All provisioning models show an exponential growth of state information. The amount of state information grows fastest in the full-mesh model while the overlay and sink-tree models grow nearly identically. The overlay model grows slightly faster than the sink-tree model because all state information required in layer 1 is counted, which includes both the PE node and the overlay point state information. Recall that the overlay model uses optimal sink trees to connect PE nodes in layer 1, similar to the sink-tree provisioning model.

How the amount of used state information varies with the number of VPN sites in the core nodes is shown in Figure 3.7. There is a significant difference between the state information used by the provisioning models. In the full-mesh model the state information grows exponentially while the sink-tree model follows a linear growth and the overlay model has a nearly constant usage of state information. Note that an increasing number of VPN sites results in a growing number of PE nodes. In the full-mesh case, that clearly results in the expected exponential growth shown in Figure 3.7. The sink-tree model uses one sink tree per VPN and PE node so adding a new PE node results, in the worst case, in 100 new trees in the core, one for each
Recent Directions in Virtual Private Network Solutions

Figure 3.6 PE node state information dependency on number of sites.

VPN. Hence the linear growth of state information. In the overlay case the amount of state information is nearly independent of the number of VPN sites. Adding a new PE node only requires a path to the closest overlay point independently of the number of VPNs, thus the amount of state information grows slowly as seen in Figure 3.7.

Combining the results shown in Figures 3.6 and 3.7 it is evident that the full-mesh provisioning model is the most sensitive to the number of VPN sites. The sink-tree model has a linear growth of state information in the core but exponential growth in the edge nodes. The overlay provisioning model gives a core network that is nearly independent of the number of VPN sites and an edge that grows similar to the sink-tree model.

Figure 3.8 shows how the amount of state information used in the PE nodes depends on the number of provisioned VPNs. Here, the average number of sites per VPN is kept constant at 20. All three provisioning models grow linearly with the number of VPNs. The state information used by the full-mesh model grows fastest because
Figure 3.7 P node state information dependency on number of sites.

Each PE node will have on average 19 incoming and 19 outgoing paths for each VPN present at the node. PE nodes in the sink-tree model will only have one incoming and 19 outgoing paths for each represented VPN, resulting in about half the growth of the full-mesh model. The overlay provisioning model’s state information consumption grows faster than the sink-tree model. In addition to the states in the PE nodes, the states at layer 1 in the overlay points (nodes 9 and 15) are counted, resulting in a slightly steeper line than in the sink-tree case.

Looking at the consumed state information in the core nodes, shown in Figure 3.9, it is evident that the growth of state information is linear in all three cases. Again, the state consumption in the full-mesh provisioning model shows the fastest growth but it does not grow exponentially. The growing number of VPNs results in a growing number of PE nodes. Each PE node supports up to 100 VPNs and hence needs at most $100 \times (20 - 1)$ incoming and outgoing paths. The number of paths across the core is therefore linearly dependent on the number of PE nodes. The same argument applies to the sink-tree model: here each PE node needs at most 100 incoming paths and at most $100 \times (20 - 1)$ outgoing paths. In the overlay provisioning model the
state information consumed in the core is nearly constant for the same reasons as in the site variation case discussed above.

3.8 Conclusion

This chapter looked at both traditional and modern provider-based VPN solutions, and breaks them down into three major building blocks; VPN type, QoS specification and VPN provisioning model. It identified each block’s components and compared them to each other in terms of scalability from the provider’s point of view and flexibility from the customer’s point of view. The component comparisons were then used to analyze the overall scalability in complete VPN solutions. VPN customer requirements were also related to the various VPN solutions, resulting in an overview of suitable VPN solutions for different customer segments. Finally, some promising recent research directions in VPN provisioning were examined, along with related issues that should be investigated further. Of the identified research directions we will address resilience integration in the full-mesh and hierarchical VPN provi-
Figure 3.9 P node state information dependency on number of VPNs.

sioning models in this thesis. Issues related to resilience integration in the sink-tree provisioning model, customizable VPN trees and dynamic resizing are left as future work.
Chapter 4

Resilience Provisioning in Provider-Based Virtual Private Networks

Guaranteed levels of network resilience are essential for many emerging and future Internet services. As described in Chapter 2, the only overlay network model that can support stringent resilience requirements is provider-based VPNs. Hence, we provided an overview of provider-based VPNs in Chapter 3. This chapter examines where resilience is best implemented in provider-based VPNs: by the provider in the physical network or by the customers in the VPNs (Rosenbaum and Jha, 2005). We look at how the network provider revenue is affected by where resilience is implemented. The idea is to measure the revenue in the two implementation scenarios and compare the generated revenues to find out if the network provider has something to gain by implementing resilience in the physical network and if so under what circumstances. The results of our linear programming-based experiments show that a network provider can potentially double its revenue when resilience is implemented in the physical network. The customers benefit as well because the VPNs are significantly less complex and can be built at the same or lower cost than when resilience is implemented in the VPNs.

An introduction to resilience provisioning in provider based VPNs is given in 4.1. Section 4.2 describes the resilience provisioning problem and how resilience can be implemented by the provider and by the customer. Then we describe the experi-
mental setup and the experimental results in Section 4.3. Conclusions are given in Section 4.4.

4.1 Introduction

Guaranteed levels of network resilience are essential for many emerging and future Internet services. Businesses’ increased dependency on the Internet pushes some traditional services toward reduced tolerance of network failure, but the current Internet is not suitable for such services as it is a best effort network. Recent studies (Feamster et al., 2003; Iannaccone et al., 2002; Markopoulou et al., 2004) show that failures leading to unreachable destinations are still common in both the Internet and autonomous systems. Hence, special mechanisms should be implemented to protect services with high resilience requirements.

Overlay networks have been proposed to improve services’ perceived network resilience. An overlay network can be realized with or without a network provider’s direct involvement. In the latter case, the overlay network owner constructs its virtual topology using tunnels between its points of presence (PoPs), whereas in the former case, the overlay owner buys a virtual topology or virtual links between its PoPs from a network provider. The overlay network owner is in this case referred to as a customer, since it buys virtual network resources from a network provider.

Examples of overlay networks that aim to improve services’ perceived network resilience, without involving a network provider, are RON (Andersen et al., 2001) and OverQoS (Subramanian et al., 2003), QoS-aware routing in overlay networks (QRON) (Li and Mohapatra, 2004), and Overcast (Jannoti et al., 2000). These approaches offer an improved service compared to plain Internet connectivity but neither of them can offer any guaranteed levels of network availability, as they are built on top of Internet. The situation is different when the overlay network is constructed with virtual resources bought from a network provider. Here, the network service availability is specified in service level agreements (SLAs) between the network provider and its customers and hence a certain level of resilience can be achieved. Current SLAs offer guaranteed service availability expressed in terms of percentage
over a period of time, such as a month or a year. For example, a SLA specifying availability as 99.5% over a month allows for about 3.6 hours of outage time each month, but it does not specify how this time is distributed over the time period. In fact, it is likely that this time comes in chunks disrupting the network service for several minutes. Many services such as VoIP, stock broking and e-commerce services do not tolerate outages longer than a few seconds or even fractions of a second. The more dependent a business is on its IT, the less tolerant it is to lengthy outages.

Today, businesses that run services with high resilience requirements in provider based overlay networks, such as provider based VPNs, must protect themselves against network failures through redundancy. They have basically two ways to achieve a satisfactory level of redundancy, either to use multiple network providers or to use physically disjoint resources from one provider. Either way, redundant resources impose increased cost and administration of the VPN.

An alternative to building a resilient VPN with redundant resources is to build it with resilient resources for which the SLA specifies a maximum outage time, such as 0.5 seconds, in addition to the original availability guarantee. However, using resilient resources implies that the network provider implements resilience in its network. If the provider’s network is based on a traffic engineering protocol such as MPLS or ATM, the functionality can be implemented at the network edge using an end-to-end recovery scope. However, implementing resilience will increase the investment and management complexity of the physical network so why should the network provider implement resilience? Intuitively, a provider can implement resilience more efficiently than its customers since it has complete knowledge of the physical network. In this chapter we quantify the efficiency difference and interpret the results from a network provider’s revenue perspective.

There are basically two categories of provider based VPNs. The first category does not require intermediate customer controlled routing between the overlay’s PoPs. That is, a virtual link is set up between each pair of PoPs that require connectivity. Examples of such VPNs are the common “Central services” and fully meshed VPN topologies (Pepelnjak and Guichard, 2002). The second category includes VPNs that require some kind of customized routing, such as VPNs that use multi-PoP paths to
forward traffic from one PoP to another. Typical VPNs in this category are partly meshed VPNs and VPNs that implement multicast-based services (Cui et al., 2004; Zhu et al., 2004).

This chapter seeks to answer whether and under what circumstances a network provider can increase its revenue by implementing resilience to cover for single link failures and hence simplifying its customers’ networks and management complexity. To our knowledge, no literature has yet addressed the impact of customer versus provider based resilience provisioning on a network provider’s revenue and consequently on the customer’s cost of setting up resilient VPNs. This chapter also delivers a modification of a linear programming formulation of an optimal online routing strategy presented by Kodialam et al. in (Kodialam and Lakshman, 2003). The modification corrects two flaws that we encountered during our experiments. This chapter focuses on the first VPN category, in which customers do not route traffic via intermediate PoPs.

4.2 Problem Definition

There are basically two ways to implement resilience in VPNs, either to use redundant virtual resources or to use resilient virtual resources. In the former case, the customer implements resilience, hence we will refer to that case as the customer implementation (CI) scenario. In the latter case the network provider implements resilience and consequently we will refer to this case as provider implementation (PI) scenario. We want to find out if the provider can increase its revenue through implementing resilience, that is if the PI scenario generates more revenue than the CI scenario.

The two implementation scenarios are compared in a context consisting of a set of resilience demands (RDs) that originate from resilient VPNs, and a set of non-resilience demands (NRDs) originating from general traffic and VPNs that do not require any specific level of resilience. Furthermore, we compare the implementation scenarios in a single link failure model.
In this chapter we are concerned with VPNs in which there is no intermediate routing between PoPs. A demand is therefore mapped to a path that spans exactly one virtual link, the virtual link between the demand’s source and destination PoPs. Figure 4.1 shows how RDs are mapped to a VPN topology in the two implementation scenarios. In the CI scenario, the customer maps its resilience demands onto two virtual links, one that is used during ordinary traffic conditions and the other for use when the first link fails. A customer has basically two ways to acquire redundant resources: to use multiple network providers or to use physically disjoint resources from one network provider. In this chapter, customers use the latter method whereby two link-disjoint paths are purchased from one network provider.

In the PI scenario, the customer maps its demands to resilient virtual links, as illustrated in Figure 4.1. Because the virtual links in this scenario are resilient, the customer does not need to set up redundant links. Hence, the customer’s logical topology is much simpler than in the CI scenario. Once a demand is mapped onto the logical topology, the resulting resilient virtual links must be realized by the provider in the physical network.

Given the above context, the provider’s revenue in the CI scenario, $r_{ci}$, is the number of satisfied RDs, $D_{rd}^{ci}$, multiplied by the price of a disjoint pair of paths, $p_{dj}$, plus the number of satisfied NRDs, $D_{nrd}^{ci}$, multiplied by the price of a regular virtual link, $p_{nrd}$. In the PI scenario, the revenue is the sum of the number of satisfied RDs multiplied by the price of a resilient virtual link, $p_{rd}$, plus the number of satisfied NRDs multiplied by $p_{nrd}$. The revenue in the two scenarios can be formulated as follows:

$$r_{ci} = p_{dj} D_{rd}^{ci} + p_{nrd} D_{nrd}^{ci}$$  \hspace{1cm} (4.1)$$

$$r_{pi} = p_{rd} D_{rd}^{pi} + p_{nrd} D_{nrd}^{pi}$$  \hspace{1cm} (4.2)$$

Now, we want to find out how much a network provider should charge for resilient virtual links without losing any revenue. In other words, we want the provider revenue to be at least as high in the PI scenario as in the CI scenario, that is $r_{pi} \geq r_{ci}$. This inequality together with Equation (4.1) and Equation (4.2) gives the following
expression for $p_{rd}$:

$$p_{rd} \geq p_{dj} \frac{D_{ci}^{rd} + (D_{ci}^{nrd} - D_{pi}^{nrd}) p_{nrd}}{D_{pi}^{rd}}, \quad D_{pi}^{rd} \neq 0$$

(4.3)

The price $p_{dj}$ of two disjoint virtual links can be expressed in terms of $p_{nrd}$. Let $p_{dj} = 2f p_{nrd}$ where $f$ is a constant representing the price difference factor between disjoint virtual links and regular virtual links. Substitution of $p_{dj}$ in Equation (4.3) gives:

$$p_{rd} \geq \frac{2f D_{ci}^{rd} + D_{ci}^{nrd} - D_{pi}^{nrd}}{D_{pi}^{rd}} p_{nrd}$$

(4.4)

$$K = \frac{2f D_{ci}^{rd} + D_{ci}^{nrd} - D_{pi}^{nrd}}{D_{pi}^{rd}}$$

(4.5)

Both Equation (4.4) and Equation (4.5) are under the condition that $D_{pi}^{rd}$ is nonzero. Combining Equation (4.4) and Equation (4.5) gives $p_{rd} \geq K p_{nrd}$, hence $K$ symbolizes the minimum price of a resilient virtual link expressed in terms of the price of a regular virtual link. $K$ is a function of the number of accepted RDs and NRDs in the two different implementation scenarios. Thus, to find the minimum price, we must find the number of accepted RDs and NRDs in the two implementation scenarios.

We use an online routing strategy for virtual link allocation, where demands are satisfied one by one without any prior knowledge of future demands. The objective is
to minimize the amount of additional resources needed to satisfy the current demand, which is analogue to the shortest-path routing in a non-resilient network.

4.2.1 Customer Implementation Scenario

In the CI scenario each RD is mapped to two virtual links between the demand’s ingress and egress PoPs. Hence, the network provider must provision two physically disjoint paths across the core from the corresponding ingress and egress nodes. This routing problem can be formulated as a network flow problem, where each link has unit cost, and two units of supply are injected at the ingress node \( s \) and two units are extracted from the egress node \( d \). Suurballe et al. (Suurballe and Tarjan, 1984) proposed a minimum flow algorithm to solve this problem. We provide a brief description of this algorithm for completeness.

- Strip the network of all links with less residual capacity than the current demand’s requested bandwidth.

- Determine the shortest path tree from the source \( s \). Let the shortest path length from \( s \) to node \( i \) be denoted by \( c_i \). Set the cost of each link \((i, j)\) to \( 1 - c_j + c_i \).

- Let \( P_1 \) represent the shortest path from \( s \) to destination \( d \). Reverse all links on \( P_1 \) in the network, leaving the costs unchanged. Calculate the shortest path \( P_2 \) from \( s \) to \( d \) using the new network.

- Eliminate all links from \( P_1 \) and \( P_2 \) that appear in both \( P_1 \) and \( P_2 \). The set \( P_1 \cup P_2 \) now contains the two optimal link-disjoint paths.

4.2.2 Provider Implementation Scenario

In the PI scenario, RDs are mapped to resilient virtual links that the provider must realize across the physical network. Hence, the provider must establish a service path and backup path between the demand’s ingress and egress nodes. The provider has, in contrast to the CI scenario, full knowledge and control over both the service path and the backup path and can therefore coordinate the resources used. Thus it is possible to share backup capacity between two link-disjoint service paths when
their backup paths traverse the same links. The reason is that at most one of the link-disjoint service paths will fail at any one time since we consider only single-link failures in this chapter. If two link-disjoint service paths have \( a \) and \( b \) units of reserved bandwidth respectively, then it is sufficient to reserve \( \max(a, b) \) on links that are shared by their backup paths.

Several heuristic algorithms that exploit backup sharing have been proposed in the literature (Lau and Jha, 2004a; Li et al., 2003b). In this chapter we want to find the optimal solution for each demand. Hence, we used an optimal linear integer program formulation proposed by Kodialam et al. (Kodialam and Lakshman, 2003) that they refer to as Complete Information Scenario (CIS). However, during our experiments we encountered two flaws in that formulation.

The first flaw is the way the service path is calculated. The formulation does not take the residual link capacity into account when calculating the cost of setting up the service path. The objective function in Equation (4.6) assumes that all links have \( b \) units of residual bandwidth. Consequently if a link does not have enough residual bandwidth to accommodate the service path then the formulation produces a non-feasible solution. Therefore, the formulation must be modified to take residual bandwidth availability into account. We propose a new objective function in Equation (4.7) that depends on \( \alpha_{ij} \), which is defined in Equation (4.8). It states that the cost of using link \((i, j)\) in the service path is \( b \) if there is enough residual capacity, otherwise the cost is set to infinity.

The second flaw in CIS results in unnecessary loops in the backup path. A backup path can meet all constraints in the original formulation and still form unnecessary loops that reserve bandwidth resources indirectly. Although the loop component incurs zero cost through bandwidth sharing, it increases the bandwidth dependencies between links and indirectly reduces the available backup bandwidth for future demands. Hence, the solution does not fully comply with the objectives of the strategy. The loop can form since it satisfies the flow conservation constraints specified in equations (4), (5) and (6) on page 405 in (Kodialam and Lakshman, 2003) without increasing the objective function, because the loop link cost is zero (expressed implicitly by \( z \)). We propose a set of additional constraints, Equation (4.9), Equation
(4.10) and Equation (4.11) that together prevent loops in the backup path.

\[
\min \left( \sum_{(i,j) \in E} b \cdot x_{ij} + z_{ij} \right) \tag{4.6}
\]

\[
\min \left( \sum_{(i,j) \in E} \alpha_{ij} \cdot x_{ij} + z_{ij} \right) \tag{4.7}
\]

\[
\alpha_{ij} = \begin{cases} 
  b & \text{if } R_{ij} \geq b, \\
  \infty & \text{otherwise}.
\end{cases} \tag{4.8}
\]

\[
\sum_{j} y_{ji} \leq 1, \quad \forall i \neq \{s\} \tag{4.9}
\]

\[
\sum_{j} y_{js} = 0 \tag{4.10}
\]

\[
\sum_{j} y_{dj} = 0 \tag{4.11}
\]

In the constraints, \( y_{ji} \) is a binary decision variable that indicates whether link \((j, i)\) is used in the backup path. Constraint Equation (4.9) allows the use of at most one incoming link for each node. None of the incoming links can be used in the source node Equation (4.10) and Equation (4.11) prevents the use of outgoing links from the destination node. Similar constraints must be introduced on the service path if demands are allowed to have zero bandwidth requirement.

### 4.3 Experiments

The aim of the experiments is to find out how many RDs and NRDs are accepted in the CI and PI scenarios respectively. This information can then be used to deduce how much a network provider must charge for a resilient virtual link to avoid any loss of revenue. The proposed algorithms in Section 4.2 are used to route RDs in the two implementation scenarios and Dijkstra’s shortest path algorithm is used to route NRDs. The algorithms are implemented with C++ and CPLEX.
The context in which the two implementations operate consists of two demand sets and a physical network. The first set contains RDs ranging from 0 to 5200 demands that are randomly distributed across the overlay points. The second set contains 0, 900 or 1700 NRDs depending on the experiment number; they are randomly distributed across all nodes in the physical network. All demands request one bandwidth unit. The reason for choosing one unit is to improve the fairness when the network approaches saturation. If the bandwidths are chosen randomly between two values, low-end demands will have a much higher probability of being accepted than high-end demands.

Two different physical networks were used, the first network, Network A, is a 25-node Canadian metropolitan network, shown in Figure 4.2 with 108 unidirectional links. W. Grover used this network in his studies of self-healing networks (Grover, 1987). The second network, Network B, is a 50-node network with 200 unidirectional links, which is an AS generated with the topology generator BRITE (Medina et al., 2001). The link capacities were chosen to be 70 units in Network A and 40 units in Network B so that the networks become saturated after about 2000 RDs. The experiments were repeated for 5, 15 and 25 overlay points in Network A and for 10, 30 and 50 overlay points in Network B. The routing algorithms first try to satisfy as many RDs as possible and then as many NRDs as possible. The results are plotted as a function of the number of RDs. Throughout, results relating to measurements in the CI scenario are plotted with dashed lines while solid lines represent measurements in the PI scenario.
4.3.1 Results

For the CI scenario in Network A, Figure 4.3 shows that the acceptance rate is 100% until about 800 RDs when no NRDs are present. Then the rate decreases quickly and levels out with steady slow growth of almost 5% from about 1600 RDs onwards. The acceptance rate for the set with 900 NRDs decreases rapidly from 100% at around 400 RDs to about 20% at 2000 RDs. Network saturation is represented as full links in Figure 4.4, and it is apparent that the decline in acceptance rates coincides with network saturation. Full links start to appear around 400 RDs and increase rapidly thereafter until 1200 RDs, after which the increase slows down but still shows steady growth. As shown in Figure 4.3 and Figure 4.4, the set with 1700 NRDs saturates some links even though no RDs are present.

For the PI scenario, shown in Figure 4.3, the 100% acceptance rate of RDs is maintained until about 1600 RDs, and reduces to 20% after 2400 RDs. This is a significant improvement compared to the CI scenario and at 3200 RDs the system has accepted about twice as many RDs as in the CI scenario. The reason behind this significant improvement can be understood by considering the network saturation shown in Figure 4.4. The bandwidth sharing exploited in the PI scenario keeps the network unsaturated until about 1200 RDs, while the CI scenario starts to saturate around 400 RDs. Note that the number of saturated links in the PI scenario passes the CI scenario at 3600 RDs, and at 5200 RDs the PI scenario has saturated about 20% more links than the CI scenario. This is because the algorithm used in the PI scenario continues to satisfy RDs and therefore continues to exhaust the network resources. Recall that the CI scenario has an acceptance rate less than 5% after 1600 RDs while the PI scenario maintains an acceptance rate of about 20%. It is also worth noticing that the number of accepted NRDs becomes higher in the CI scenario than in the PI scenario after 3600 RDs. The experiments using Network B were carried out in exactly the same way as those using Network A. The corresponding graphs are shown in Figs. 4.5 and 4.6 with the same proportion of overlay points (3/5). One difference that can be noted between the two sets of graphs is that the curve trends in Figure 4.5 show almost no growth after they have leveled out at about 2400 RDs, as opposed to the trends seen in Figure 4.3. This is particularly striking in the CI scenario.
Accepted Demands

Number of RDs
No NRDs, CI
No NRDs, PI
900  NRDs, CI
900  NRDs, PI
1700 NRDs, CI
1700 NRDs, PI

Figure 4.3 Accepted demands in Network A with 15 overlay points.

Full links

Number of RDs
No NRDs, CI
No NRDs, PI
900  NRDs, CI
900  NRDs, PI
1700 NRDs, CI
1700 NRDs, PI

Figure 4.4 Full links in Network A with 15 overlay points.
Figure 4.5 Accepted demands in Network B with 30 overlay points.

Figure 4.6 Full links in Network B with 30 overlay points.
Just as in Network A, more than twice as many demands are accepted in the PI scenario as in the CI scenario, at around 1800 RDs. In the CI scenario the growth of accepted demands is approximately zero even though about 80 links (or 40%) still have spare capacity. This indicates that the network has been partitioned into small isolated islands.

4.3.2 Resilient Link Price Analysis

Figure 4.7 shows how $K$ varies with the number of RDs in Network A with 15 overlay points. $K$ is calculated according to (4.5) given in Section 4.2 with $f \in \{1.0, 1.3, 1.6\}$. Figure 4.8 shows the corresponding calculations for Network B with 30 overlay points. The same set of experiments were carried out with 5 and 25 overlay points in Network A and 10 and 50 overlay points in Network B. The results show that $K$ stays within 3% of the values measured for 15 and 30 overlay points in the respective network. Hence, $K$ can be considered as independent of the size of the VPNs.

However, $K$ depends on the chosen $f$-value. Figure 4.7 and 4.8 show that $K$ actually drops below $f$ in each case in both networks if there are enough RDs. It indicates that it is more than twice as resource efficient to provision one resilient virtual link between two overlay points than it is to provision a pair of disjoint virtual links between the same points. One implication is that a network provider that offers a resilient virtual link at the same price as two disjoint virtual links more than doubles its revenue, while giving the customer a resilient VPN that is easier to operate and maintain.

Looking at the plot for “$f = 1.0$ No NRDs” in Figure 4.7, it shows that $K$ does not drop until about 800 RDs, which reflects that equally many demands are satisfied in the two implementation scenarios until that point. It is interesting to note that $K$ actually drops below 1, which happens in Figure 4.7 and 4.8 at 3200 and 1800 RDs respectively. It indicates that the price of a resilient virtual link can be set below the price of a non-resilient virtual link without any loss of revenue for the provider. However, this is under the condition that $f = 1.0$, which means that the price of a pair of physically disjoint links is the same as the price of two regular virtual links.
Resilience Provisioning in Provider-Based Virtual Private Networks

between the same overlay points.

To understand what a reasonable $f$-value would be, we measured $f$ by comparing the number of accepted demands when satisfying them with either a pair of physically disjoint virtual links or a pair of regular virtual links. It was observed that $f$ grows slowly starting from 1.0. When the network starts to saturate, it jumps to about 1.2, after which it slows down and flattens out at 1.26 in both networks. Using $f = 1.26$, it can be seen in Figure 4.7 that the price of a resilient virtual link in Network A can be set at about 23% above the price of a regular virtual link whereas in Network B, as shown in Figure 4.8, the minimum price of a resilient virtual link is essentially the same as the price of a regular virtual link, if enough RDs are present.

$K$ drops earlier and more steeply when more NRDs are present. The plot representing 1700 NRDs drops from the very beginning as a result of the higher degree of network efficiency achieved in the PI scenario.

\textbf{Figure 4.7} $K$ in Network A with 15 overlay points and $f \in \{1.0, 1.3, 1.6\}$.
Resilience Provisioning in Provider-Based Virtual Private Networks

4.4 Conclusion

In this chapter we looked at how a network provider’s revenue depends on where resilience is implemented in resilient VPNs. Our experiments show that, given enough demand for resilient traffic, the provider can double its revenue through offering resilient virtual links at the same price as a pair of disjoint virtual links. The customers will benefit as well when using a VPN built with resilient virtual links since it is less complex and less costly to set up and maintain than a VPN built with redundant resources and resilience mechanisms.

The difference in price between disjoint virtual links and regular links is essential for the interpretation of the results. Our measurements show that the price of a disjoint virtual link must be up to 26% higher than the price of a regular virtual link to sustain the same revenue. The results also show that the price of one resilient virtual link can be set below half the price of a pair of disjoint virtual links with sustained revenue.
Chapter 5

Resilience Differentiation in Programmable Virtual Private Networks

Service and application requirements on network resilience have increased over the past few years. New online services such as e-commerce and connection-oriented interactive real-time services require higher network resilience than more traditional offline services. Programmable VPNs are a special kind of VPN that delivers both virtual links and virtual sites. Virtual sites can be used by customers to implement functionality “in the cloud” such as customized routing and content adaptation. Programmable VPNs promise fast and easy provisioning of new services but very little effort has been made to address new services’ diverse resilience requirements in a programmable VPN context.

Chapter 4 showed that there are several advantages when resilience mechanisms are implemented by the network provider in the physical network. This chapter further discusses issues related to resilience provisioning in programmable VPNs with a focus on resilience differentiation. This chapter presents a set of general guidelines that apply to resilience-differentiated programmable VPN architectures. The contents covered in this chapter have been published in (Rosenbaum et al., 2004). A case study is used to illustrate how the proposed guidelines can be met by extending an existing programmable VPN architecture with resilience-differentiated properties. The presented architecture effectively integrates resilience with the full-mesh and hi-
Network service survivability, or resilience, has attracted increasing interest over the past few years. The main reason is that new online services such as e-commerce and connection-oriented interactive real-time services require ever-increasing network availability to meet their users’ expectations while other off-line services such as email have low resilience requirements. Such diverse resilience demands add new requirements on the network infrastructure. To increase efficient utilization of network resources, recovery scheme design should take the different resilience requirements into account. Traffic engineering methods (Awduche et al., 2002) are a requirement to efficiently provision resilience in a network. Recovery mechanisms can be implemented at multiple network layers. They can even be in operation simultaneously. The authors of (Demeester et al., 1999) present some guidelines on where to implement recovery mechanisms and how to coordinate them if they are applied in more than one layer. In general, lower layers such as wavelength-division multiplexing (WDM) and synchronous digital hierarchy (SDH) provide very fast recovery while higher layers like IP and MPLS provide higher resource efficiency, flow and QoS granularity.

As mentioned in Chapter 2 programmable VPNs are promoted as a solution for fast and easy provisioning of new innovative services over the Internet. The basic idea
Resilience Differentiation in Programmable Virtual Private Networks

is to provide multiple programmable VPNs over one physical infrastructure. Each virtual network customer (VNC) can install and run customized code in virtual nodes to control routing and support application-specific tasks, such as content adaptation and monitoring. A number of different programmable VPN architectures have been proposed over the past few years, for example Tempest (Rooney et al., 1998), Virtual Active Network (Brunner and Stadler, 2000) and the Programmable Virtual Network (PVN) architecture (Nguyen et al., 2002). However, the resilience issues related to a programmable VPN infrastructure have not been addressed.

End-to-end resilience differentiation in overlay infrastructures introduces a new dimension, as the end-to-end resilience provisioning can be split between the network provider and its VNCs. Some of the challenges to providing resilience differentiation in programmable VPNs arise because a VNC controls its own routing. Furthermore, it is important to uphold PVN integrity and privacy. Traffic must never leak out of nor into a PVN. Scalability issues must be considered as well. A programmable VPN infrastructure can potentially provide hundreds of PVNs, each supporting thousands of end-to-end flows.

This chapter addresses general resilience differentiation issues related to programmable VPNs. A set of general guidelines that apply to programmable VPN architectures is presented. To illustrate how the guidelines can be met, a resilience differentiation enhancement of the PVN architecture (Nguyen et al., 2002) is proposed as a case study.

5.2 Resilience Issues in Programmable Virtual Private Networks

A programmable VPN realizes a number of overlay networks over one physical infrastructure. Each overlay network is owned and managed by a VNC, while the physical infrastructure is owned and managed by a network provider (NP). This separation introduces new aspects on end-to-end resilience provisioning. The goal is to satisfy end-user needs for end-to-end resilience and QoS in an efficient way. What is considered efficient varies, but in general it can be measured in terms of scalability,
Resilience Differentiation in Programmable Virtual Private Networks

Table 5.1 Resilience provisioning models in programmable VPNs.

<table>
<thead>
<tr>
<th>Recovery implementation</th>
<th>End-user signaling</th>
<th>Scalability</th>
<th>Flexibility</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>NP</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>NP</td>
<td>VNC</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>VNC</td>
<td>VNC</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

flexibility and overhead combined with a target environment. A programmable VPN is a two layered network managed by different administrative entities. Therefore, the end-to-end resilience can be provisioned in a number of different ways. Resilience provisioning consists of two parts: end-user signaling to establish resilient end-to-end paths and provisioning of recovery mechanisms.

Whether the VNC or the NP should provision one, both or neither of these parts is an open issue that depends on the target environment. Table 5.1 shows the possible provisioning models and how they relate to scalability, flexibility and overhead. As shown, NP provisioning of recovery mechanisms is efficient in terms of overhead because the same recovery mechanisms will be shared by all VNCs. In the NP/NP model, a full mesh of virtual links must be established between a VNC’s virtual nodes. Thus the number of virtual links is bounded by $O(n^2)$, where $n$ is the number of virtual nodes, resulting in low scalability. The other two models scale better; the number of virtual links is bounded by $O(n)$ as an effect of the customized routing. The NP/VNC model offers the highest flexibility. In this model, the VNCs can choose to implement their own recovery mechanisms or buy them from the NP. If a VNC chooses to provision its own recovery mechanisms, it is essential that the virtual links used during normal and recovery operation are physically disjoint. Otherwise it might not be possible to recover from a single link or node failure.

When either the NP or a VNC provisions both recovery mechanisms and end-user signaling, there is no practical difference compared to a non-overlay infrastructure. Methods described in the literature (Autenrieth and Kirstadter, 2002; Sharma et al., 2003; Andersen et al., 2001) can be applied directly. However, when end-to-end resilience provisioning is split between the NP and its VNCs, the situation becomes
more complicated. The interface between the NP and the VNCs must be very clear and fulfill a number of requirements, as discussed below.

### 5.2.1 Split End-to-End Resilience Provisioning

Here, the NP implements a set of recovery mechanisms but no end-user signaling; that responsibility is effectively pushed to the VNCs. However, the NP must provide resilience hooks to the VNCs. The hooks are used by the VNC to specify the desired resilience level when mapping end-user traffic onto its virtual links. Each VNC must be able to select an appropriate set of resilience hooks on a link-by-link basis because they may have different resilience requirements for different links.

Generally, finer resilience granularity implies higher realization costs in the network. There is a clear tradeoff between resilience granularity and realization cost. The need for fine granularity indicates that the actual implementation of recovery schemes should be located in the network layer (Demeester et al., 1999; Sharma et al., 2003), that is in IP or MPLS. Because provisioning of resilience differentiation requires some grade of traffic engineering, the underlying network should be based on MPLS or a protocol with similar traffic engineering properties.

The authors of (Autenrieth and Kirstadter, 2002) propose a definition of resilience levels in terms of resilience classes (RCs). A RC specifies a maximum service disruption time for a set of expected failures such as all single link and node failures. An RC may also specify the allowable reduction of QoS during recovery operations. A network provider offers a limited set of well defined resilience classes to its VNCs, for example the set of resilience classes shown in Table 5.2. In the table, RC1 offers the highest resilience guarantees with a maximum service disruption time of 100 milliseconds and no QoS reduction. RC2 guarantees to recover within 0.5 seconds with possible temporary QoS reduction. Resilience class RC3 offers recovery within five seconds and essentially permanent QoS reduction (until normal operation is resumed). Finally, class RC4 has no recovery guarantees nor does it promise to provide any QoS constraints. These RCs can be extended and further subdivided if so desired. The authors also present resilience signaling extensions to the two dominant QoS models proposed by the Internet Engineering Task Force (IETF): Differentiated
Table 5.2 Example set of resilience classes.

<table>
<thead>
<tr>
<th>Resilience Class</th>
<th>RC1</th>
<th>RC2</th>
<th>RC3</th>
<th>RC4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resilience req.</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>None</td>
</tr>
<tr>
<td>Recovery time</td>
<td>&lt; 100 ms</td>
<td>&lt; 0.5 s</td>
<td>&lt; 5 s</td>
<td>N/A</td>
</tr>
<tr>
<td>QoS</td>
<td>Equivalent</td>
<td>Temp. reduced</td>
<td>Reduced</td>
<td>None</td>
</tr>
</tbody>
</table>

Resilience Differentiation in Programmable Virtual Private Networks

How many resilience classes and what resilience levels to choose depend on the network provider’s situation and expected VNC demands. A programmable VPN architecture should not restrict the number of supported resilience classes.

When a network failure such as a link or node failure occurs it may result in redirection of a number of flows. How the redirection is carried out and how long it takes depends on the recovery mechanism that protects the affected flows. In ordinary networks, this would not impose any problem. In the case of programmable VPNs, however, the redirected flows are owned and controlled by VNCs, not the network provider. It is of vital importance that the redirection is handled transparently to the VNCs. This is a direct implication of the resilience guarantees offered to the VNCs. The guarantees imply that the NP takes full responsibility to maintain operation in case of a link or node failure. Hence, the redirection must be handled in the physical network, transparent to the VNCs.

Another very important aspect of resilience differentiation in programmable PVNs is security. The integrity and privacy of PVNs must be maintained during network recovery as well as during normal operation. Packets must never leak out of nor into a PVN. The consequences of misrouted traffic between PVNs are potentially much worse than in non-overlay infrastructures because layer-three addresses can be reused in different PVNs.

VNCs will have access to a PVN in which the virtual links are provided with resilience hooks. The VNCs can apply any resilience signaling protocol such as the extended DiffServ and IntServ protocols proposed in (Autenrieth and Kirstadter, 2002) to provision end-to-end resilience. A VNC may choose to implement its own recov-
ery schemes even though it could buy resilience-classified hooks from the network provider. In such cases the VNC should be notified when a virtual link goes down or perceives degraded QoS.

A summary of the architectural guidelines for resilience-differentiated programmable VPNs are presented below. The architecture:

- must be based on a network infrastructure that offers traffic engineering capabilities;
- must support a limited but sufficiently large set of resilience classes;
- must provide VNCs with hooks to a suitable subset of the supported resilience classes;
- must provide VNC-transparent traffic redirection during recovery operation;
- must uphold PVN integrity and privacy;
- should notify the VNCs when a virtual link failure or QoS degradation occurs.

5.3 Resilience Differentiation in the PVN Architecture

5.3.1 The PVN Architecture

The PVN architecture (Nguyen et al., 2002) is an MPLS-based approach that introduces programmability in a carrier-grade environment. It basically provides VNCs with a full set of virtual MPLS network resources including labels, links and nodes. The main goals of the PVN architecture are:

- **Scalability** achieved through a fast data transit path and efficient active packet identification and extraction.

- **Flexibility** to support a wide range of applications, including those with very high processing requirements, without compromising scalability.
– Migration path through gradual provisioning of processing capability and expansion of active network applications over legacy infrastructure.

Figure 5.1 shows the PVN node architecture with two Active Processors. The Active Processors virtualize all resources exposed to the VNCs and enforce policies in the MPLS abstraction layer. Execution Environments (EEs) represent virtual sites (or nodes). An EE has access rights to a set of virtual links and a contiguous block of virtual labels. A label switched router (LSR) is an MPLS switch that conforms to the MPLS standard. Active Processors are connected to a LSR via a set of LSPs and a management port. The LSPs carry packets that either require some active processing in the node or originate from EEs in the node’s Active Processors. There are two distinct types of flows; one type carries active packets destined to an EE inside the node, the other type carries transit traffic.

The PVN architecture uses a two-level label stack. The outer (top) label denotes an MPLS tunnel between two PVN nodes while the inner (bottom) label represents a VNC controlled LSP. The inner label is used by the network provider’s control plane to identify both a VNC and the VNC’s label. Hence, it is possible to multiplex LSPs controlled by different VNCs onto the same MPLS tunnel. Furthermore, the inner label is used by the forwarding plane to either divert active packets to the corresponding EE or to an outgoing MPLS tunnel. MPLS tunnels start and terminate in LSRs while VNC controlled LSPs start and terminate in EEs.

5.3.2 MPLS-Based Recovery

MPLS has attracted considerable attention because of its traffic engineering properties (Awduche, 1999; Xiao et al., 2000) and its ability to achieve fast and efficient recovery. A framework for MPLS recovery (Sharma et al., 2003) has been published by the IETF MPLS working group. It discusses issues related to fast and reliable fault detection and recovery mechanisms in MPLS networks. It concludes that MPLS-based recovery can detect a failure and recover traffic on a time scale comparable to synchronous Optical Network (SONET), which is about 50 milliseconds. On top of fast recovery, MPLS provides both high flow recovery granularity and QoS granularity.
A number of different recovery schemes have already been developed (Huang et al., 2002; Kodialam and Lakshman, 2000). Each recovery mechanism has its own properties in terms of recovery time, QoS preservation, resource utilization and overhead. Which recovery scheme to choose for a given resilience class is an open issue, although some guidelines are given in (Autenrieth and Kirstadter, 2002). If more than one resilience class is provided by a network provider, it is likely that more than one recovery scheme will be implemented in the network.

5.3.3 Resilience Extensions

To extend the PVN architecture with resilience differentiation, care must be taken so the architecture’s original design goals, as outlined in Section 5.3.1, remain fulfilled. The resilience extension targets the two-layered approach where the network provider supplies resilience hooks to the VNCs who in turn provide the end-user signaling. How the VNCs provision the end-user signaling is beyond the scope of this thesis.

The Resilience Module in Figure 5.1 represents the major part of the architecture extension. It is inserted as a middleware layer between the management ports of the Active Processors and the LSR. It contains three major functional blocks. First, it provides a set of recovery engines. A recovery engine is responsible for computing resilient QoS connections, monitoring service paths and reacting to service path failures. These three functions will collectively be referred to as a recovery mechanism. Second, it provides a recovery redirection block that offers redirection support to the recovery engines while hiding eventual redirections from the management block. The label information base (LIB) in legacy LSRs is often designed to support fast packet forwarding and hence is not very flexible. Therefore, the resilience redirection layer implements an extended variant of the LIB called ELIB. The ELIB provides the additional functionality required in the control plane. Note that the forwarding plane still uses the LIB. Finally, the Resilience Module provides a management block. This block process all configuration requests initiated by the EEs.

A VNC buys virtual links and interfaces from the network provider. A virtual link connects two EEs, or virtual nodes, with each other and a virtual interface is used
to transmit and receive data over a virtual link. Instead of purchasing one virtual interface per virtual link, as in the original architecture, a set of virtual interfaces is purchased. Each virtual interface corresponds to a specific resilience class.

The abstraction layer in the Active Processor maps the virtual interfaces to a tunnel protected by a suitable recovery engine, which typically provides the same resilience class as the virtual interface. However, it can be mapped to an engine that provides higher resilience.

Figure 5.2 shows a simple subset of a PVN infrastructure with two programmable nodes. The set of grey-colored MPLS tunnels, marked with capital letters, are configured in the nodes by the network provider. Another set of LSPs, marked with small letters, are configured by the VNC residing in the displayed EEs. LSPs $C$, $D$ and $G$ can be considered as tunnel extensions to reach the appropriate Active Processors but are not conceptually part of the tunnels.

Packets following LSP $p$ are regarded as active packets as they are diverted to an EE. Likewise, packets on LSP $r$ are considered active because they originate from an EE.
Note that packets may be active in one programmable node while inactive in others.

The Resilience Module (RM) in node 1 supervises tunnel $E$. The Resilience Module is configured to switch all traffic in tunnel $E$ to tunnel $E'$ if a physical link or node failure occurs along tunnel $E$. Observe that tunnels $E$ and $E'$ can be routed over several intermediate LSRs including other programmable nodes. The switch-over time depends on the recovery engine used.

Figure 5.3 shows a snapshot of the ELIB and LIB configurations on PVN node 1 and PVN node 2 in Figure 5.2. When a tunnel failure occurs and the corresponding recovery engine initiates a redirection request, two things take place in the redirection layer. First, the corresponding entry in the ELIB is modified to point to the new desired tunnel $E'$, as shown in Figure 5.3. Second, all entries in the LSR’s LIB that point to tunnel $E$ are re-mapped to tunnel $E'$. Whether tunnel $E'$ is preestablished or dynamically signaled and allocated when a failure occurs depends on the recovery mechanism implemented by the supervising recovery engine. When a recovery engine detects that a tunnel is down or perceives degraded QoS it notifies the management block. The management block forwards the notification to all EEs that have a virtual interface mapped to the failing tunnel.

When the Management block in the Resilience Module receives a configuration request from an EE it translates the virtual interface to a tunnel (label) before the request is forwarded to the redirection layer. The redirection layer then checks if there is any active redirection for that tunnel, in which case the substitution is made before the actual LIB is configured in the LSR. To illustrate this procedure, suppose that tunnel $E$ has been redirected to $E'$ when the setup request of LSP $n$ is made in PVN node 1, see Figure 5.2 and Figure 5.3. First, the management block translates the virtual interface to tunnel $E$. Next, the redirection layer configures its ELIB to swap label $n$ with label $q$ and forward the packets over tunnel $E$. However, tunnel $E$ is currently redirected to tunnel $E'$, so $E'$ is substituted for $E$ before the request is forwarded to the LIB. In PVN node 2, no special precautions are required, thus both the ELIB and the LIB are configured to swap label $q$ with label $t$ and forward over tunnel $H$. 
Consider the incoming packet in PVN node 1 shown in Figure 5.2. The top label $A$ is popped and the inner label $n$ is swapped with label $q$ before the packet is forwarded on tunnel $E'$ according to PVN node 1’s LIB configuration, as shown in Figure 5.3. When the packet arrives in PVN node 2, its top label $E'$ is popped and its inner label $q$ is swapped with label $t$ before it is forwarded on tunnel $H$. The ELIB is never used to perform any forwarding actions in the LSRs.

### 5.4 Guideline Conformance

The guidelines presented in Section 5.2 are all met by the proposed resilience-differentiation extension:

- The PVN architecture is based on MPLS and therefore offers a sufficient level of traffic engineering.

- The network provider can use multiple recovery engines simultaneously to meet any desired number of supported resilience classes. The extended architecture can even support resilience classes with recovery times as low as 50 milliseconds.

- The virtual interfaces provide the necessary resilience hooks for a VNC to map end-user flows to different resilience classes.

- The Recovery redirection block provides the required VNC transparency during recovery operation.

- The VNC identification is based on the inner label, hence it is independent of the outer label (tunnel). This guarantees that packets do not leak into nor out of a PVN.

- When a recovery engine detects tunnel failure or QoS degradation, it notifies the Management block in the Resilience Module, which in turn makes sure that the affected EEs are notified.
Figure 5.2 MPLS tunnels and VNC controlled LSPs.

Figure 5.3 ELIB and LIB configurations.
The proposed extensions also meet the PVN architecture’s main goals as outlined in Section 5.3.1. The extension leaves both the fast transit path and active packet identification and extraction principles unaltered. It does, however, introduce a larger set of MPLS tunnels: one extra tunnel for each peering PVN node and supported resilience class. A consequence is that the size of the LIB increases and the maximum number of possible independent MPLS tunnels decreases. This fact should be considered when determining the resilience granularity. The ELIB is only used in the control plane, hence it does not affect the forwarding performance in the LSRs.

The proposed resilience extension has no negative impact on flexibility. In fact, the extension enhances flexibility. An even wider range of services and applications can be supported, because the PVN architecture now provides resilience differentiation to the VNCs. Finally, the migration path is not affected by the resilience extension. However, recovery mechanisms that require forwarding functionality beyond the MPLS standard might not be supported. A legacy LSR is not required to provide any additional functionality than specified in the MPLS standard.

### 5.5 Implementing VPN Services Using PVNs

The extended PVN architecture can be used to provision resilient L2 and L3 VPNs. The architecture basically provides mechanisms for provisioning resilient QoS connections. Virtual sites or EEs can be used by the network provider to implement any VPN service, such as VR, BGP/MPLS VPN, VPWS and VPLS. For instance, if all EEs in a specific PVN run a VR and are given access to ports that physically connect to customer sites, then that particular PVN is a direct implementation a VR VPN. In this scenario, the EEs are administered by the network provider rather than the customer. Similarly, if all EEs in a PVN simply bridge a set of resilient interfaces to ports that physically connect to a customer site, then that PVN is an implementation of VPWS. Furthermore, different PVNs can implement different types of VPNs and thus multiple VPN services can coexist in a single resilient PVN architecture.
5.6 Conclusion

New services and applications create an increasingly large diversion of network resilience requirements. Programmable VPNs aim to improve fast and easy provisioning of new services and applications but have so far left resilience-related issues unexplored. This chapter discussed those issues and provided a set of guidelines that a resilience-differentiated programmable VPN architecture should conform to. An extension to the PVN architecture was proposed as a case study to show how the guidelines could be met while upholding the architecture’s original design goals. In this chapter we introduced a new concept called recovery engines. A recovery engine is responsible for computing resilient QoS connections, monitor service paths and react upon service path failures. A recovery engine is typically used to implement a specific level of resilience and QoS. The presented architecture also provides a generic platform for implementing one or more VPN services such as VPWS, VPLS, VR and BGP/MPLS VPNs.
Chapter 6

Dynamic Routing of Resilient QoS Connections

In Chapter 5 we introduced a new MPLS-based resilience-differentiated PVN architecture that provides resilient QoS connections and virtual sites. One central feature of this architecture is the recovery engine. A recovery engine is responsible, among other things, for computing resilient QoS connections. In this chapter, we consider how resilient QoS connections can be effectively computed when the QoS constraints are bandwidth and end-to-end delay with a focus on network performance.

The current frameworks for computing resilient QoS connections described in the literature use a preprocessing step either to convert the end-to-end delay into an effective bandwidth or to prune all links that do not meet the bandwidth constraint. Both frameworks therefore reduce the problem to one with a single QoS constraint, bandwidth or end-to-end delay respectively. We argue that the current frameworks result in poor network performance and suggest a new framework that exploits the dependency between the end-to-end delay, chosen path and provisioned bandwidth.

Then, given our new framework, we present a new generic algorithm that decomposes the problem into subproblems where existing algorithms can be used. Next we propose two new linear programming formulations, and show that they achieve higher network performance than decomposed approaches in both 1+1 and 1:1 recovery contexts. These new formulations are not intended to be used in a production environment as they have long run times, up to 2.5 minutes on average per request,
but they can be used to benchmark heuristics that in turn can be used in production. In Chapter 7 we develop new algorithms that improve the run time and hence become attractive for a production environment. The work covered in this chapter has been published in (Rosenbaum et al., 2005).

An introduction to the problem of computing resilient QoS connections is given in Section 6.1. Then we provide a formal problem formulation in Section 6.2 followed by a number of decomposed computation approaches in Section 6.3 and combined computation approaches in Section 6.4. Section 6.5 presents our simulations, and discusses the results. Conclusions are given in Section 6.6.

### 6.1 Introduction

Dynamic routing of resilient QoS connections in MPLS networks is motivated by network providers’ needs to satisfy customers who want both QoS and resilience guarantees simultaneously for their aggregated flows. A resilient QoS connection is a logical connection between two network nodes, an ingress and an egress node, that is resilient to network failures and meets the specified QoS constraints.

In this chapter we consider a common recovery framework in which a resilient QoS connection is protected from end to end in a single-link failure model (Suurballe and Tarjan, 1984; Kodialam and Lakshman, 2003; Norden et al., 2001). Furthermore, we assume that the network provider uses MPLS and that requests for resilient QoS connections are routed dynamically one by one as they arrive, without any knowledge of future requests. In the given framework, each resilient QoS connection is realized by two link-disjoint LSPs from the ingress to the egress node. Thus when a request for a resilient QoS connection arrives, the network provider must first compute the two LSPs and then set them up in the network. The LSP setup can be handled using RSVP-TE (Awduche et al., 2001a) or CR-LDP (Jamoussi et al., 2002). The focus of this chapter is on the path computation.

A typical scenario where a customer needs both QoS guarantees and resilience for its aggregated flows is when the customer uses real time applications like e-commerce
and VoIP in a provider-based virtual private network (VPN). Here, the aggregate flows are forwarded along a resilient QoS connection between two customer sites attached to ingress and egress nodes in the provider network. When a customer wants to connect two sites, the customer requests a new resilient QoS connection between them from its network provider. The problem that a network provider faces when such a request is received, is how to compute two link-disjoint LSPs from the ingress to egress node such that both paths meet the QoS constraints.

Motivated by network providers’ needs to provide bandwidth guaranteed services that are both delay-sensitive and resilient, we focus this chapter on the routing problem of resilient QoS connection subject to bandwidth guarantee and end-to-end delay constraints. As described in Chapter 2, the current approaches use a preprocessing step to reduce the problem to one with only a single constraint. One approach converts the end-to-end delay to an effective bandwidth (Kodialam and Lakshman, 2003; Kodialam and Lakshman, 2001; Li et al., 2002; Xu et al., 2004; Alicherry and Bhatia, 2004; Norden et al., 2004). The other approach prunes all links that do not have enough residual capacity to accommodate the requested bandwidth (Orda and Sprin-son, 2004; Bejerano et al., 2005). Hence, the problem is reduced to a problem with only a bandwidth or an end-to-end delay constraint.

Observing that the end-to-end delay depends on both the chosen path and the provisioned bandwidth, a new approach to the described routing problem can be taken. In the new approach, this dependency is exploited so that the two link-disjoint paths are computed dynamically with their respective bandwidths. Thus, a solution to the routing problem of resilient QoS connections will not only contain two LSPs but their respective bandwidths as well. This approach has been successfully applied to compute non-resilient QOS connections.

There are basically two ways to approach the routing problem for resilient QoS connections with dynamic bandwidth allocation. One way is to decompose it into problems where existing solutions already exist. For instance, one can decompose the problem into QoS routing of non-resilient connections where known solutions like the ones presented in (Ma and Steenkiste, 1997; Q. Ma and P. Steenkiste, 1998; Pornavalai et al., 1998; Korkmanz and Krunz, 2003) can be applied. Another example
is to decompose it into a problem of routing resilient bandwidth guaranteed connections (Suurballe and Tarjan, 1984; Kodialam and Lakshman, 2003; Norden et al., 2001) and end-to-end delay verification (Stiliadis and Varma, 1998). Decomposing the problem inevitably trades accuracy for simplicity, hence a given decomposition might not find a solution even though one exists. It is therefore important to look at the combined way to approach the problem whereby the two LSPs and their respective bandwidths are computed simultaneously.

The decomposed approach has not yet been fully understood in the literature and the combined approach is to our knowledge not addressed at all. In this chapter we discuss the two approaches and propose three new algorithms, one generic algorithm used to implement decomposed approaches and two new linear programming (LP) formulations that implement the combined approach. The LP formulations presented are not suitable for a production environment because of their high complexity. However, they are suitable for benchmarking approximation algorithms that in turn can be used in production.

6.2 Problem Formulation

In this chapter we are concerned with the problem of routing connections that carry an aggregate of flows with both QoS and resilience requirements in a label switched network. In particular, we consider an end-to-end recovery scenario under a single-link failure model where two link-disjoint LSPs must be set up from the ingress to the egress node for each admitted connection request. We will call the two LSPs service path $P_s$ and backup path $P_b$. In our framework, connection requests are routed one by one as they arrive with no knowledge of future requests. When a connection request arrives it is only admitted to the network if the provider can find a service path and a backup path that both meet the QoS requirements, otherwise the request is rejected.

Motivated by network providers’ needs to provide bandwidth guaranteed connections that are both delay sensitive and resilient, we will use bandwidth guarantee and bounded end-to-end delay as our QoS constraints. We will refer to such connections as resilient delay bounded connections.
Formally, we can specify a resilient delay-bounded connection request as a 4-tuple $\text{Req} = (s, d, T\text{Spec}, R\text{Spec})$, where the first field $s$ is the ingress node, $d$ is the egress node, $T\text{Spec}$ is the aggregate traffic specification defined by a vector $(M, r, t, b)$ where $M$ is the maximum packet size, $r$ is the sustainable rate, $t$ is the peak rate and $b$ is the burst tolerance. $R\text{Spec}$ specifies the upper end-to-end delay bound $d_m$. Thus given a resilient delay bounded connection request and a network modeled by $G$, the problem is to find a service path and a backup path from $s$ to $d$ such that both the $T\text{Spec}$ and $R\text{Spec}$ is satisfied on both paths.

To approach this problem we need a method that describes how the end-to-end delay relates to a given path and the provisioned bandwidth. One such method is deterministic network calculus (Le Boudec, 1998), which provides a deterministic worst case end-to-end delay guarantee. One drawback of the deterministic approach is that it provides an over-estimate of the bandwidth requirement because it does not take statistical multiplexing into account. There are some recent efforts in creating a stochastic network calculus counterpart (Li et al., 2003a; Ciucu et al., 2005), but the results are still under development. We have therefore chosen to use the deterministic network calculus where mature understanding of the theories exist. Note that whether the deterministic or stochastic framework is used does not affect the arguments presented herein. In addition, the optimization formulations in Section 6.4 can be readily adapted if we are to use the stochastic framework though the formulations may not remain linear.

In the deterministic network calculus framework it has been proved that the general form of QoS routing of non-resilient connections is NP-complete (Jaffe, 1984). Stiliadis et al. showed that when the QoS constraints are bandwidth guarantee and end-to-end delay, the problem can be reduced to polynomial time through introducing latency rate (LR) servers (Stiliadis and Varma, 1998). A LR server is a network node that runs a packet scheduler like Weighted Fair Queuing (WFQ) (A. Demers and Shenker, 1989; Parekh and Gallager, 1993), Fair Weighted Fair Queuing (WF$^2$Q) (Bennet and Zhang, 1996), Self-Clocked Fair Queuing (SCFQ) (Golestani, 1994), Virtual Clock (VC) (Zang, 1990) or Frame-based Fair Queuing (Stiliadis and Varma, 1996). They showed that, in a network consisting of LR servers, a tight upper bound
of the end-to-end delay $D_m$ on a path $P$ from $s$ to $d$ is given by:

$$D_m = \frac{(t - R)}{(t - r)} \cdot \frac{b}{R} + \sum_{(i,j) \in P} \left( \frac{M_{ij}}{R} + \frac{M_{ij}^m}{C_{ij}} + \text{prop}_{ij} \right)$$

(6.1)

where $C_{ij}$ and $\text{prop}_{ij}$ are respectively the capacity and the propagation delay of link $(i, j)$, $M_{ij}^m$ is the maximum packet size of all LSPs that use link $(i, j)$, and $R$ is the minimum of the allocated bandwidths associated with the LSP in the traversed nodes. In other words $R = \min(R_1, R_2, ..., R_{|P|})$ where $r \leq R_i \leq t$ is the allocated bandwidth for the LSP in node $i$ and $|P|$ is the length of $P$. However, in this chapter we are considering LSPs where all nodes reserve the same amount of bandwidth and hence $R_j = R_i, \forall i, j \in P$.

In Equation (6.1) the first term represents the shaper delay at the ingress node. The second term, the sum, corresponds to the queuing-related delay induced by transmission buffers and propagation delays. No transmission buffers are considered in $d$ because that is the egress node.

As evident in Equation (6.1), the end-to-end delay bound depends not only on the chosen path but also on the provisioned bandwidth $R$. Evidently, increasing the provisioned bandwidth $R$ results in shorter end-to-end delay and vice versa. In this chapter we seek to exploit the dependency between the end-to-end delay, chosen path and bandwidth when satisfying a resilient delay-bounded connection request.

Now, the problem we consider in this chapter can be formulated as follows. Given a network graph $G = (V, E)$, where $V$ is a set of LR servers and $E$ is a set of edges connecting the servers, and a request $\text{Req} = (s, d, TSpec, RSpec)$, find two link-disjoint LSPs $P_s$ and $P_b$, from $s$ to $d$, and their respective bandwidths $R_s$ and $R_b$, such that both $TSpec$ and $RSpec$ are satisfied.

When very high resilience is required, a 1+1 recovery scheme should be used. In this scheme, traffic on a connection is forwarded along both the service and backup path simultaneously and the egress node then listens on the path with the strongest signal. This scheme does have the advantage of achieving very fast recovery from
link failures but at the cost of doubling the traffic volume.

If the resilience requirement is relaxed, it is possible to use the more bandwidth-conservative recovery scheme 1:1, in which traffic is switched to the backup path only after a failure has been detected on the service path. Because the backup path is idle as long as the service path is in operation the backup path can be shared between different service paths as long as the service paths are link disjoint.

We will address the problem of routing resilient delay bounded connections for both 1+1 and 1:1 recovery, that is without and with backup path resource sharing. Independently of the routing scheme, an algorithm that solves this problem must compute not only two LSPs but also their respective bandwidths. The problem can be approached in two ways. One way is to decompose the problem into subproblems to which known solutions already exist. The other way is to search for the two LSPs and their respective bandwidths simultaneously.

### 6.3 Decomposed Computation

In this section we approach the problem of routing resilient delay bounded connections through decomposition. The problem can be decomposed in a number of different ways.

One approach is to first convert the end-to-end delay bound into an effective bandwidth and then search for two disjoint bandwidth-guaranteed LSPs simultaneously, using a known algorithm such as the ones presented in (Kodialam and Lakshman, 2003) and (Suurballe and Tarjan, 1984). However, when converting the end-to-end delay bound into an effective bandwidth, we need to know the path that will be used. Because we do not have the path at hand we need to estimate or guess the path. If we want to be sure that the computed path will not breach the end-to-end delay bound we must make a pessimistic estimate and use the longest (measured in end-to-end delay) possible loop-free path from \( s \) to \( d \). Hence, the resulting effective bandwidth will be higher than required in most cases because the computed paths are likely to be shorter than the longest path. This approach can be refined in a number of differ-
ent ways; nevertheless, it will always be subject to an estimated path. Clearly, this
decomposition does not exploit the end-to-end delay dependency on the chosen path
and provisioned bandwidth.

Another way is to decompose the problem into a subproblem of finding a single QoS
constrained LSP $P_s$ and its associated bandwidth $R_s$, from $s$ to $d$ in $G = (V, E)$
using a known algorithm such as the one presented by Steenkiste et al. (Q. Ma and
P. Steenkiste, 1998) or by Pornavalai et al. (Pornavalai et al., 1998). Once $P_s$ is
computed, the second LSP $P_b$ and its associated bandwidth $R_b$, is computed using
the same algorithm but on network graph $G' = (V, E \setminus E(P_s))$, where $E(P_s)$ is the
set of edges on $P_s$. Although this approach is simple, it has been shown that joint
computation of two bandwidth guaranteed LSPs leads to higher network performance
than computing them in sequence (Suurballe and Tarjan, 1984). Here, the end-to-end
delay dependency on the chosen path and provisioned bandwidth is exploited when
using the algorithm provided by Steenkiste et al. but not when using Pornavalai’s
algorithm.

Yet another way to decompose the routing problem for resilient delay-bounded con-
nections is to first search for two disjoint paths, again using a known algorithm (Kodi-
alam and Lakshman, 2003; Suurballe and Tarjan, 1984), with a bandwidth at least as
high as the requested sustainable rate $r$. If two paths are found, then if the end-to-end
delay bound can be verified on both paths the request is admitted into the network,
otherwise it is rejected. This decomposition does not exploit the dependency of end-
to-end delay on chosen path or provisioned bandwidth.

One way to possibly improve the performance of this decomposition, is to exploit the
dependency between the end-to-end delay and allocated bandwidth. The baseline is
that increasing the bandwidth allocation shortens the end-to-end delay. Thus, iterat-
ing a path-finding algorithm using an increasing amount of bandwidth can improve
the chances of finding two paths that meet the end-to-end delay requirement. This
approach is expressed in Algorithm 6.1.

The algorithm takes a network $G = (V, E)$ and a request $Req = (s, d, TSpec, RSpec)$
for a delay bounded resilient connection as input and returns a service path $P_s$ and
Dynamic Routing of Resilient QoS Connections

Algorithm 6.1: Generic backup path $P_b$, with their respective bandwidths $R_s$ and $R_b$. The algorithm first initializes the bandwidth $bw$ to the sustainable bandwidth $r$ as specified in the request’s TSpec. Then while $bw$ is less than or equal to the peak rate $t$ it first searches for two disjoint paths through a call to $FindPaths$. $FindPaths$ is typically implemented using existing algorithms. If it is run in a 1+1 recovery routing context an algorithm like the one proposed by Suurballe can be used (Suurballe and Tarjan, 1984), which finds two bandwidth-guaranteed, link-disjoint paths from $s$ to $d$ that have the lowest combined cost. For a 1:1 recovery context, a backup bandwidth-sharing algorithm like the one proposed by Kodialam et al. (Kodialam and Lakshman, 2003) is preferably used.

If $FindPaths$ fails to find two paths then the algorithm returns false, because if $FindPaths$ failed to find two paths using $bw$ it is certain to fail using some bandwidth greater than $bw$.

Given that $FindPaths$ finds two paths, the end-to-end delay bound is checked in $VerifyDelay$. $VerifyDelay$ uses Equation (6.1) to verify the delay and returns true if and only if $D_m \leq d_m$ on both $P_s$ and $P_b$ when $R = bw$. Now, if both paths pass the end-to-end delay bound test, their respective bandwidths $R_s$ and $R_b$ are calculated using $Trim$. The reason for trimming the bandwidths is that the bandwidth $bw$ used to calculate the paths is an upper bound on the required bandwidth. $Trim$ basically
calculates a more precise bandwidth using Equation (6.1) with $D_m = d_m$, $P = P_s$ and $P = P_b$ respectively. After the bandwidths have been trimmed, the algorithm returns true indicating that the request can be admitted to the network.

If either of the two paths did not meet the end-to-end delay bound, then the bandwidth $bw$ is adjusted to a higher value. The granularity of the bandwidth adjustment is controlled through the constant $q$. Higher values of $q$ mean smaller adjustments to $bw$ and consequently more iterations. Thus $q$ can be used as a tradeoff mechanism between solution precision and run time. If no paths have been found that satisfy the end-to-end delay bound even when the peak rate is used, the algorithm returns false indicating that the request must be rejected.

This solution approach cannot guarantee that it will find a solution if it exists. The reason is that it tries to find two paths using the same bandwidth. Assume that there are exactly two paths $P_1$ and $P_2$ in $G = (V, E)$ from $s$ to $d$ and that $bw = k$ in the current iteration of Algorithm 6.1. Assume furthermore that the two paths’ respective residual capacities are $H_{P_1} \leq k$ and $H_{P_2} > k + \Delta$ for some $\Delta > 0$, also let their respective delay bounds be $D_{P_1}^k \leq d_m$, and $D_{P_2}^k > d_m$. Now, in the algorithm, $FindPaths$ will return $P_1$ and $P_2$ because both have enough residual capacity to host the request, but $VerifyDelay$ will return false because $P_2$ does not comply with the end-to-end delay bound. The algorithm therefore adjusts the bandwidth to $bw = k + \epsilon < t$ for some $\epsilon \in (0, \Delta]$ and calls $FindPaths$ again. This time $FindPaths$ will return false because $P_1$ does not have enough residual capacity and the request is subsequently rejected even if $D_{P_2}^{k+\epsilon} \leq d_m$ is true and a solution exists, namely $P_s = P_1$ and $P_b = P_2$ with their respective bandwidths $R_s = k$ and $R_b = k + \epsilon$.

### 6.4 Combined Computation

In this section we approach the online routing problem of resilient delay-bounded connections through combined computation of paths and bandwidths. Hence, when a request arrives, the algorithm searches for a service path $P_s$ and a backup path $P_b$ along with their respective bandwidths $R_s$ and $R_b$ simultaneously. We base these
algorithms on linear programming and provide two LP formulations, one for the 1+1 and one for the 1:1 recovery scheme, the difference being that in the 1:1 recovery scheme backup path resources can be shared between different service paths as long as the service paths are link disjoint.

In the LP formulations presented below we allow the provisioned bandwidth to adjust up to the requested peak rate $t$, because we base our end-to-end delay calculations on Equation (6.1), which has $t$ as a theoretical upper bound. However, the requested peak rate $t$ can be much greater than the requested sustainable rate $r$. Therefore a tighter upper rate bound can be chosen to possibly improve the overall network performance. To make a fair comparison when using the presented LP formulations as a benchmark for approximation algorithms, the same upper rate bound should be chosen for both the approximation algorithm and for the LP formulation.

### 6.4.1 1+1 Recovery

When using 1+1 recovery routing, the bandwidth reserved on the backup path cannot be shared amongst different service paths. However, algorithms that do not implement any backup resource sharing can indeed be used in a 1:1 recovery context as well. Here, backup paths can be used to forward preemptive traffic as long as the service path is in operation. The problem we address in this section is: Given a network $G = (V, E)$ and a request $(s, d, TSpec, RSpec)$, find a service path $P_s$ and an exclusive link-disjoint backup path $P_b$ from $s$ to $d$ and their respective bandwidths $R_s$ and $R_b$ such that both the TSpec and RSpec are met on both $P_s$ and $P_b$. Furthermore, we want the total cost of the two paths to be minimized.

Let the cost of using link $(i, j)$ on the service path be defined as $R_s F_{ij}^R$, where $F_{ij}^R$ is the link weight. $F_{ij}^R$ can typically be used to balance the network load where $F_{ij}^R$ is chosen to reflect the amount of residual capacity on link $(i, j)$. The more residual capacity the lower the value of $F_{ij}^R$. Similarly, let the cost of using link $(i, j)$ on the backup path $P_b$ be defined as $R_b F_{ij}^R$. Let $x_{ij}$ be a decision variable that indicates if link $(i, j)$ is used on $P_s$ or not. Define $x_{ij}$ to be 1 if $x_{ij} \in P_s$ and 0 otherwise. Likewise, we define $y_{ij}$ to indicate if link $(i, j)$ is used on $P_b$ or not, with a value of 1 if $y_{ij} \in P_b$ and 0 otherwise. Given these decision variables, the cost of using link
Dynamic Routing of Resilient QoS Connections

(i, j) on the service path can be formulated as \( z_{ij} = R_s F^R_{ij} x_{ij} \) and on the backup path as \( w_{ij} = R_b F^R_{ij} y_{ij} \). Note that \( z_{ij} \) is a nonlinear function of decision variables \( R_s \) and \( x_{ij} \) and likewise \( w_{ij} \) is a nonlinear function of \( R_b \) and \( y_{ij} \). We will show later how \( z_{ij} \) and \( w_{ij} \) can be reformulated as linear expressions.

To constrain \( P_s \) and \( P_b \) and their respective bandwidths \( R_s \) and \( R_b \) so that the given delay bound \( d_m \) is met, we use Equation (6.1). The baseline is that \( d_m \geq D_m \) and because we are considering linear programming we need to transform Equation (6.1) into a linear expression. First we multiply both sides with \( R \) which gives us:

\[
R d_m \geq b \cdot \frac{(t - R)}{(t - r)} + \sum_{(i,j) \in P} \left( M + R \left( \frac{M^m_{ij}}{C_{ij}} + prop_{ij} \right) \right) \tag{6.2}
\]

Furthermore, we need to convert the expression into a more general form where the sum is over all links in the network, not only the links on the path. Using the service path \( P_s \) as an example we obtain:

\[
R_s d_m \geq b \cdot \frac{(t - R_s)}{(t - r)} + \sum_{(i,j) \in E} \left( M x_{ij} + \left( \frac{M^m_{ij}}{C_{ij}} + prop_{ij} \right) R_s x_{ij} \right) \tag{6.3}
\]

We can transform the nonlinear expression \( R_s x_{ij} \) in Equation (6.3) into a linear expression through the observation that \( R_s x_{ij} = z_{ij} \frac{1}{F^R_{ij}} \). Furthermore, substituting the constant \( \frac{1}{F^R_{ij}} \left( \frac{M^m_{ij}}{C_{ij}} + prop_{ij} \right) \) with \( K_{ij} \) gives us the following linear expression for the end-to-end delay bound on the service path:

\[
R_s d_m \geq b \cdot \frac{(t - R_s)}{(t - r)} + \sum_{(i,j) \in E} \left( M x_{ij} + K_{ij} z_{ij} \right) \tag{6.4}
\]

where \( z_{ij}, x_{ij} \) and \( R_s \) are decision variables. Equation (6.4) can be used to compute the end-to-end delay bound on the backup path as well after substituting \( z_{ij}, x_{ij} \) and \( R_s \) with \( w_{ij}, y_{ij} \) and \( R_b \) respectively.

Now we are ready to present the LP formulation. The objective is to minimize the
combined cost of the service path and backup path as expressed in Equation (6.5).

\[
\min \left( \sum_{(i,j) \in E} z_{ij} + \sum_{(i,j) \in E} w_{ij} \right)
\]  

(6.5)

under the following constraints:

\[
z_{ij} = \begin{cases} 
R_s F_i^R R_{ij} & \text{if } x_{ij} \geq 1, \\
0 & \text{otherwise}.
\end{cases}
\]  

(6.6)

\[
w_{ij} = \begin{cases} 
R_b F_i^R R_{ij} & \text{if } y_{ij} \geq 1, \\
0 & \text{otherwise}.
\end{cases}
\]  

(6.7)

\[
r \leq R_s \leq t
\]  

(6.8)

\[
r \leq R_b \leq t
\]  

(6.9)

\[
R_s \leq H_{ij}, \quad \text{if } x_{ij} \geq 1, \forall (i, j) \in E
\]  

(6.10)

\[
R_b \leq H_{ij}, \quad \text{if } y_{ij} \geq 1, \forall (i, j) \in E
\]  

(6.11)

\[
R_s d_m \geq b \left( \frac{t - R_s}{t - r} \right) + \sum_{(i,j) \in E} (M x_{ij} + K_{ij} z_{ij})
\]  

(6.12)

\[
R_b d_m \geq b \left( \frac{t - R_b}{t - r} \right) + \sum_{(i,j) \in E} (M y_{ij} + K_{ij} w_{ij})
\]  

(6.13)

\[
\sum_j x_{ij} - \sum_j x_{ji} = 0, \quad \forall i \neq s, d
\]  

(6.14)

\[
\sum_j x_{sj} - \sum_j x_{js} = 1
\]  

(6.15)

\[
\sum_j x_{dj} - \sum_j x_{jd} = -1
\]  

(6.16)

\[
\sum_j y_{ij} - \sum_j y_{ji} = 0, \quad \forall i \neq s, d
\]  

(6.17)

\[
\sum_j y_{sj} - \sum_j y_{js} = 1
\]  

(6.18)

\[
\sum_j y_{dj} - \sum_j y_{jd} = -1
\]  

(6.19)

\[
x_{ij}, y_{ij} \in \{0, 1\}, \quad \forall (i, j) \in E
\]  

(6.20)
\[ x_{ij} + y_{ij} \leq 1, \quad \forall (i, j) \in E \]  

Equations (6.6) and (6.7) provide the cost for using link \((i, j)\) on the service and backup path respectively. For readability we formulate them as logical expressions, but it is straightforward to convert them into linear expressions, see Appendix A. In general we provide the logical formulation in the remainder of this chapter for \(if - then\) and \(if - then - else\) expressions to improve readability.

Equations (6.8) and (6.9) provide the limits for the provisioned service and backup bandwidths. Followed by the link capacity constraints given in Equation (6.10) and (6.11), which say that the provisioned bandwidth on link \((i, j)\) must be smaller than the link’s residual capacity, \(H_{ij}\). These constraints cannot be formulated as \(R_s x_{ij} \leq H_{ij}\), because that is a nonlinear expression. The end-to-end delay constraints for the service and backup path are given in Equation (6.12) and (6.13), which correspond to Equation (6.4).

Equations (6.14) - (6.19) are the flow preservation constraints for the service path and backup path. Equation (6.20) assures that \(x\) and \(y\) only take on values 0 and 1. The last constraint, Equation (6.21) states that the service and backup paths must be disjoint.

### 6.4.2 1:1 Recovery

When using 1:1 recovery, the resources reserved on the backup path can be shared amongst different service paths as long as the service paths are link disjoint (Kodialam and Lakshman, 2003; Norden et al., 2001; Lau and Jha, 2004b). Thus the problem in this scenario is: given a network \(G = (V, E)\) and a request \((s, d, T\text{Spec}, R\text{Spec})\), find a service path \(P_s\) and a shareable link-disjoint backup path \(P_b\) from \(s\) to \(d\) and their respective bandwidths \(R_s\) and \(R_b\) such that both the TSpec and RSpec are met. Furthermore, we want the total cost of the two paths to be minimized.

The cost \(z_{ij}\) of using link \((i, j)\) on the service path is defined in the same way as in the LP formulation for the 1+1 recovery scheme. Before we can define the cost of
Dynamic Routing of Resilient QoS Connections

using link \((i, j)\) on the backup path, we need to look at the implications of backup path bandwidth sharing.

We are concerned with resilience under single link failures, hence the baseline for sharing is that as long as two or more service paths do not have any links in common they can share the same backup path resources.

**Example 1:** Suppose that we have two link-disjoint service paths \(P_{s1}\) and \(P_{s2}\) \((P_{s1} \cap P_{s2} = \emptyset)\), that use \(R_{b1}\) and \(R_{b2}\) units of bandwidth respectively on their backup paths. Now if both \(P_{s1}\) and \(P_{s2}\) use link \((u, v)\) on their backup paths, then it is sufficient to allocate \(\max(R_{b1}, R_{b2})\) units of backup bandwidth on \((u, v)\) because \(P_{s1}\) and \(P_{s2}\) will not both fail at the same time.

In the 1+1 recovery scenario it was sufficient to keep track of the residual link capacities. Here, we need to keep track of all currently admitted requests in the network as well, including their associated service and backup paths and respective provisioned bandwidths. Let \(A_{ij}\) represent all admitted requests that use link \((i, j)\) on their service path and let \(B_{uv}\) represent all admitted requests that use link \((u, v)\) on their backup path.

Now we define \(\phi_{ij}^{uv}\) as the set of requests that use link \((i, j)\) on their service path and link \((u, v)\) on their backup path, as expressed in Equation (6.22).

\[
\phi_{ij}^{uv} = A_{ij} \cap B_{uv} \tag{6.22}
\]

Furthermore, define \(\delta_{ij}^{uv}\) as the amount of backup bandwidth on link \((u, v)\) that depends on requests that use link \((i, j)\) on their service path. \(\delta_{ij}^{uv}\) is given by the sum of the backup bandwidths used by requests in \(\phi_{ij}^{uv}\).

\[
\delta_{ij}^{uv} = \sum_{k \in \phi_{ij}^{uv}} R_{b}^{k} \tag{6.23}
\]

where \(R_{b}^{k}\) is the provisioned backup bandwidth by request \(k\). Note that \(\delta_{ij}^{uv}\) does not express how much backup bandwidth is reserved on link \((u, v)\), it only tells how
much backup bandwidth on link \((u, v)\) depends on requests that use service paths linking \((i, j)\). However, the reserved backup bandwidth \(N_{uv}\) on link \((u, v)\) is always the highest value of \(\delta_{ij}\), that is \(N_{uv} = \max_{(i,j) \in E} \delta_{ij}\).

**Example 2:** Looking back at Example 1, we see that \(\delta_{ij}\) for link \((u, v)\) is either 0, \(R_{b1}\), or \(R_{b2}\), assuming that no other admitted requests use \((u, v)\) on their backup path. \(N_{uv}\) is simply the largest of these three values. What if the two service paths in the example are only partly link disjoint? Say that \(P_{s1} \cap P_{s2} = \{(e, f)\}\). Then for any link \((u, v)\) that is shared by their backup paths, we have that \(\delta_{ef} = R_{b1} + R_{b2}\) so \(N_{uv} = \max(0, R_{b1}, R_{b2}, R_{b1} + R_{b2}) = R_{b1} + R_{b2}\). Hence, no sharing will take place.

In general, the amount of bandwidth to reserve on backup link \((u, v)\) when admitting a request using \(R_b\) units of bandwidth on the backup path depends on how much bandwidth can be shared on link \((u, v)\). Shareable bandwidth is the difference between the already allocated backup bandwidth \(N_{uv}\) and the current maximum dependency of this bandwidth to links used by the current request’s service path \(P_s\). The shareable bandwidth can thus be expressed as \(N_{uv} - \max_{ij \in P_s} \delta_{ij}\). More specifically, the shareable bandwidth on link \((u, v)\) when used to backup link \((i, j)\) is \(N_{uv} - \delta_{ij}\). If \(R_b\) exceeds the shareable bandwidth, we refer to the difference \(R_b - (N_{uv} - \delta_{ij})\) as the excess bandwidth.

Now we define \(\Theta_{ij}\) as the cost of using link \((u, v)\) to backup link \((i, j)\) as:

\[
\Theta_{ij}^{uv} = \begin{cases} 
R_b F_{ij}^{F} & \text{if } R_b \leq N_{uv} - \delta_{ij}, \\
(N_{uv} - \delta_{ij}) F_{ij}^{F} + (R_b - (N_{uv} - \delta_{ij})) F_{ij}^{R} & \text{otherwise.}
\end{cases}
\]  

(6.24)

Equation (6.24) says that the cost to use link \((u, v)\) to backup link \((i, j)\), when there is no excess bandwidth, is \(R_b F_{ij}^{F}\). Here \(F_{ij}^{F}\) is a constant used to weight the use of *free* bandwidth in a similar way as \(F_{ij}^{R}\) is used to weight explicit bandwidth reservations. Even though a link potentially can be used on a backup path without explicitly reserving more bandwidth, it does increase the bandwidth dependencies. Thus we treat it as a network resource that should be balanced. If there is excess bandwidth, the cost \(\Theta_{ij}^{uv}\) is the sum of the cost for the *free* portion and the cost for the excess...
portion.

Now we can calculate the cost $q_{uv}$ of using link $(u, v)$ on the backup path when the service path is $P_s$ as $q_{uv} = \max_{(ij) \in P_s} \Theta_{ij}^{uv} = \max_{(ij) \in E} \sigma_{ij}^{uv}$, where $\sigma_{ij}^{uv}$ equals $\Theta_{ij}^{uv}$ if and only if link $(i, j)$ is used on the service path and link $(u, v)$ is used on the backup path, otherwise it equals 0.

Definitions of $x_{ij}$ and $y_{ij}$ in the following LP formulation are the same as in the 1+1 recovery scenario. Now the LP formulation can be stated as follows. The objective is to minimize the combined cost of the service path and backup path as expressed in Eq (6.25):

$$\min \left( \sum_{(i,j) \in E} z_{ij} + \sum_{(u,v) \in E} q_{uv} \right)$$  (6.25)

subject to the following constraints

$$z_{ij} = R_s F_{ij} R_{ij} x_{ij} = \begin{cases} R_s F_{ij} R_{ij} & \text{if } x_{ij} \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$  (6.26)

$$r \leq R_s \leq t$$  (6.27)

$$R_s \leq H_{ij}, \text{ if } x_{ij} \geq 1, \forall (i,j) \in E$$  (6.28)

$$R_s d_m \geq b \frac{(t - R_s)}{(t - r)} + \sum_{(i,j) \in E} (M x_{ij} + K_{ij} z_{ij})$$  (6.29)

$$q_{uv} \geq \sigma_{ij}^{uv}, \forall (i,j) \in E$$  (6.30)

$$\sigma_{ij}^{uv} = \begin{cases} \Theta_{ij}^{uv} & \text{if } x_{ij} + y_{uv} \geq 2, \\ 0 & \text{otherwise.} \end{cases}$$  (6.31)
Dynamic Routing of Resilient QoS Connections

\[ \Theta_{ij}^{uv} = \begin{cases} R_b F_{ij}^F & \text{if } R_b \leq N_{uv} - \delta_{ij}^{uv}, \\ (N_{uv} - \delta_{ij}^{uv}) F_{ij}^F + (R_b - (N_{uv} - \delta_{ij}^{uv})) F_{ij}^R & \text{otherwise}. \end{cases} \] (6.32)

\[ r \leq R_b \leq t \] (6.33)

\[ R_b \leq H_{uv} + N_{uv} - \delta_{ij}^{uv}, \text{ if } x_{ij} + y_{uv} \geq 2 \] (6.34)

\[ R_b d_m \geq b \left( \frac{t - R_b}{t - r} \right) + \sum_{(i,j) \in E} (M y_{ij} + K_{ij} w_{ij}) \] (6.35)

\[ w_{ij} = R_b F_{ij}^R y_{ij} = \begin{cases} R_b F_{ij}^R & \text{if } y_{ij} \geq 1, \\ 0 & \text{otherwise.} \end{cases} \] (6.36)

\[ \sum_j x_{ij} - \sum_j x_{ji} = 0, \quad \forall i \neq s, d \] (6.37)

\[ \sum_j x_{sj} - \sum_j x_{js} = 1 \] (6.38)

\[ \sum_j x_{dj} - \sum_j x_{jd} = -1 \] (6.39)

\[ \sum_j y_{ij} - \sum_j y_{ji} = 0, \quad \forall i \neq s, d \] (6.40)

\[ \sum_j y_{sj} - \sum_j y_{js} = 1 \] (6.41)

\[ \sum_j y_{dj} - \sum_j y_{jd} = -1 \] (6.42)

\[ \sum_j y_{ij} \leq 1, \quad \forall i \neq d \text{ and } \forall j \neq s \] (6.43)

\[ \sum_j y_{is} = 0 \] (6.44)

\[ \sum_j y_{jd} = 0 \] (6.45)

\[ x_{ij}, y_{ij} \in \{0, 1\}, \quad \forall (i,j) \in E \] (6.46)
\[ x_{ij} + y_{ij} \leq 1, \quad \forall (i, j) \in E \quad (6.47) \]

The service path constraints are given in Equations (6.26)–(6.29), which are identical to the ones in the 1+1 recovery scenario. However, the backup path constraints are different because we consider backup path resource sharing in this scenario.

Equations (6.30) and (6.31) together express the cost of using link \((u, v)\) on the backup path, while Equation (6.32) specifies the cost of using link \((u, v)\) to backup link \((i, j)\), as defined in Equation (6.24).

The backup bandwidth bounds are given in Equation (6.33) where the upper bound \(t\) is determined by the the delay calculation formula provided in Equation (6.1).

The capacity constraints are given in Equation (6.34), which basically says that the provisioned bandwidth on the backup path must be smaller than or equal to the sum of the residual capacity and the shareable bandwidth.

Equations (6.35) and (6.36) bound the provisioned bandwidth on the backup path from below so that the delay constraint is met. These constraints are identical to the constraints in Equations (6.13) and (6.7) for the 1+1 recovery scenario.

The flow preservation constraints, Equations (6.37)–(6.46), are the same as in the 1+1 recovery scenario as well. If \(F^F_{ij}\) is 0 then the possibility of using a link “for free” on the backup path potentially creates loops. To prevent this, we need a few additional constraints. Equation (6.43) ensures that at most one incoming link per node on the backup path is used, Equation (6.44) prevents use of incoming links to the source node and Equation (6.45) prevents use of outgoing links from the destination node on the backup path. Finally, Equation (6.47) provides the constraints for disjointness, which again are the same as in the 1+1 recovery scenario.
6.5 Simulation Results

In this section we present simulations that aim to outline the performance difference between the dynamic and static bandwidth allocation approach to the routing problem of resilient delay bounded connections. Furthermore, the simulations aim to reveal the computation times required for the proposed LP formulations.

The two approaches are compared in both the 1+1 and 1:1 recovery contexts. In the dynamic bandwidth allocation approach, we use the presented LP formulations in their respective routing contexts. In the static bandwidth allocation approach, we use Algorithm 6.1 in both routing contexts. To make it behave as a static bandwidth allocation algorithm, we prevent it from iterating by choosing \( q \) less than one. Furthermore, in the 1+1 recovery routing context, we implement \( \text{FindPaths} \) using an algorithm proposed by Suurballe (Suurballe and Tarjan, 1984), which computes two link-disjoint paths that have the least combined cost. In the 1:1 recovery context we implement \( \text{FindPaths} \) using an LP formulation called Complete Information Scenario (CIS) proposed by Kodialam et al. (Kodialam and Lakshman, 2003). CIS uses backup bandwidth sharing and, like Suurballe’s algorithm, it computes two link-disjoint paths that have the lowest combined cost. Neither of these two algorithms provides a load balancing mechanism, so for the sake of fairness, we “turn off” the load balancing in our proposed LP formulations by selecting \( F_{ij}^R = 1 \) and \( F_{ij}^F = 0 \). All code is written in C++ and CPlex, an optimization package from ILOG.

The network we modeled in the simulations is shown in Figure 6.1. It represents a typical ISP network with 18 nodes and 60 unidirectional links. This network was used for comparable simulations in (Kar et al., 2002; Apostopoulos et al., 1998). The capacity is set to 4 Mbps and the propagation delay to 1 millisecond on all links. Similar results to those presented here were obtained running the simulation on two other networks. In the simulation, requests for resilient delay-bounded connections arrive one by one according to a Poisson distribution with mean arrival rate \( \lambda \). Admitted requests remain in the system for a time span drawn from an exponential distribution with mean \( \mu \). We chose to keep \( \mu \) constant at one time unit and \( \lambda \) to 55 requests per time unit in the 1+1 recovery context and to 85 requests per time unit in the 1:1
recovery context, which gives a rejection rate of about 5% when the delay bound is relaxed.

The simulation emulates resilient delay-bounded connections used for voice applications. Assuming that the voice applications use a G.726 codec with a bit rate of 40 kbps, we find a maximum packet size $M$ of 84 bytes. The other parameters in the TSpec, $r$, $t$ and $b$ are chosen to be respectively 400 kbps, 1000 kbps and $18M = 1512$ bytes, which gives a maximum buffer delay at the ingress node of about 30 milliseconds. The end-to-end delay bound $d_m$, as specified in the RSpec, is used as a variable ranging from 50 – 200 milliseconds.

In our simulations we want to see how the network performance of the static bandwidth allocation approach is affected by different values of the provisioned bandwidth. We run the simulations three times, choosing the provisioned bandwidth to be either 400, 700 or 1000 kbps by initializing $bw$ to the chosen bandwidth in Algorithm 6.1.

Once the network has reached a steady state, the measurements commence and run for 15 time units. We measure the rejection rate, which is plotted as a function of the end-to-end delay bound $d_m$. A solid line represents the dynamic bandwidth allocation approach and a dotted line represents the static bandwidth allocation approach.

The average computation time per accepted request in the 1:1 recovery scenario is up to 146 seconds and in the 1+1 scenario up to 0.5 seconds on a 3.2 GHz Intel CPU with 2 GB RAM memory. Because of the inherent complexity of the resilient delay-bounded routing problem, the computation time in some cases exceeds 40 hours for
one request in the combined computation approach. Therefore, we limit the allowed computation time to 30 minutes per request. If this time limit is reached before a solution is found, the request is rejected. The only rejections because of the maximum time limit in our simulations occurred at 50 milliseconds in the 1:1 recovery context for the dynamic bandwidth allocation approach, where three requests were rejected.

Looking at the results from the simulation in the 1+1 recovery context shown in Figure 6.2, we see that as expected, the dynamic bandwidth allocation performs better than the static scheme. This is particularly true when the end-to-end delay bound is tight.

Note that when static rates of 700 and 1000 kbps are used, better network performance is achieved than when using the sustainable rate 400 kbps for delay bounds tighter than 80 and 60 milliseconds respectively. However, when the delay bound is relaxed, static bandwidth allocation with 400 kbps outperforms the other two rates with a rejection rate of about 6\% versus 11\% and 17\%. The explanation is that, when

---

**Figure 6.2** Dynamic vs. static bandwidth allocation in a 1+1 recovery context, $\lambda = 55$. 

using for instance a rate of 700 kbps, any link with less residual capacity than 700 kbps will not even be considered during the path computation no matter how relaxed the delay bound is. Furthermore, when the delay bound is tighter than 80 milliseconds, it apparently pays off to look for paths that can use up to 700 kbps because the higher rate makes it more likely that the paths comply with the delay bound. Remember that the bandwidth is trimmed, so that the bandwidth actually reserved in the network is in most cases less than 700 kbps per admitted request. Clearly there is a tradeoff between the chosen static bandwidth and the network performance in the presence of an end-to-end delay bound.

Given a static rate of 400 kbps we see that the impact that the end-to-end delay bound has on the network performance of the two approaches decreases rapidly until the end-to-end delay bound increases to about 100 milliseconds, after which they perform similarly. Hence, using dynamic bandwidth allocation only makes sense when the delay bound has an impact on the path computation. What happens is that the problem is effectively transformed into a problem of pure bandwidth-guaranteed resilient connections when the delay bound is relaxed. In other words, if the delay

Figure 6.3 Dynamic vs. static bandwidth allocation in a 1:1 recovery context, $\lambda = 85$. 
bound is relaxed enough, it will not be breached no matter how long the chosen path is.

Another interesting point in the graph is that for the static bandwidth allocation approach, the rejection rate increases slightly from about 100 milliseconds onwards. One explanation for this is that around 100 milliseconds, requests that require long paths to be setup are rejected because of the delay bound and thus leaving more resources available for “shorter” requests. This effect subsides when the delay bound is relaxed, resulting in the slight increase of the rejection rate.

We see similar patterns in the results for the 1:1 recovery context shown in Figure 6.3. It should be noted that even though the arrival rate is about 50% higher than in the 1+1 recovery context, the dynamic bandwidth allocation algorithm performs better than the dynamic bandwidth allocation algorithm in the 1+1 context as an effect of backup resource sharing.

One noticeable difference between the two routing contexts is that the improvement from using dynamic bandwidth allocation is higher in the 1:1 recovery context. The reason is that the backup path resource sharing encourages “long” backup paths, because backup resources can potentially be used for free or at a subsidized cost. Longer paths are less likely to comply with the delay bound and hence are more likely to be rejected. The same reason is behind the high rejection rate for the static bandwidth allocation approach with rate 400 kbps when the delay bound is tight.

The response times of the LP formulations are long, up to 3 minutes in the 1+1 recovery context and more than 40 hours in the 1:1 context. The average is about 0.5 seconds in the 1+1 and 2.5 minutes in the 1:1 recovery context. The poor response times make the LP formulations unsuitable for production environments, which typically require a guaranteed response time in fractions of a second. However, they do result in very good network performance and thus they are suitable as benchmark tools for approximation algorithms.
6.6 Conclusion

In this chapter we addressed the problem of routing resilient delay-bounded connections. We proposed a new approach to solving this problem in which the two paths and their respective bandwidths are computed dynamically. The dynamic bandwidth allocation approach exploits the dependency between the end-to-end delay, chosen path and bandwidth allocation. To realize our approach, we presented three algorithms, one of them a generic algorithm that decomposes the problem into problems where known solutions already exist. The other two algorithms are LP formulations that implement a combined approach where the paths and their respective bandwidths are computed simultaneously.

Our initial results show that the network performance improves significantly when using a dynamic bandwidth allocation approach as opposed to the current static bandwidth allocation approach. Tighter delay bounds result in greater improvements. The improvement is more significant in the 1:1 recovery context than in the 1+1 context. The presented LP formulations are not suitable for a production environment as their response times are too long, up to 2.5 minutes on average. They are however, suitable for benchmarking approximation algorithms that in turn can be used in production. In Chapter 7 we will develop algorithms that address the run time issue in the 1+1 and 1:1 recovery context without backup resource sharing strategies. To develop algorithms that address the run time issue in the 1:1 recovery context with backup resource sharing is left as future work.
Chapter 7

Efficient Algorithms for Computing Resilient QoS Connections

Chapter 6 presented a new framework for computing resilient QoS connections where the QoS constraints are bandwidth and end-to-end delay. In that framework, requests for resilient QoS connections are satisfied by computing two link-disjoint paths and their respective bandwidths such that the given constraints are met. Chapter 6 also presented a new generic algorithm that decomposes the given problem into subproblems that can be solved using existing algorithms, and two new LP formulations that address the combined problem. Simulations showed that the combined approach outperforms the decomposed approach in terms of network performance. However, the presented LP formulations’ long run times make them unsuitable for a production environment.

In this chapter, we develop new algorithms in the framework presented in Chapter 6 that improve run times while sustaining high network performance. The goal is to develop algorithms that can be used in production, hence we need to put very stringent requirements on run times. Furthermore, in this chapter we focus on algorithms that can be used in both the 1+1 and 1:1 recovery contexts, thus we will not use any backup resource-sharing strategies. We will further subject the returned paths to an optimality criterion such that either the returned paths’ combined cost or the number of hops is minimized.

We first provide a sketch of an algorithm for computing resilient QoS connections
that systematically uses a shortest-path algorithm to compute non-resilient connections. Here, “shortest” refers to a path with either minimum cost or minimum number of hops. As no algorithms that compute non-resilient QoS connections with sufficiently short run times have been proposed in the current literature, we develop a new algorithm and heuristic that compute non-resilient QoS connections. The run time of the heuristic is comparable with Dijkstra’s shortest-path algorithm, which is \(O(V \log V + E)\) where \(E\) is the number of links and \(V\) is the number of nodes in the network. Our simulations show that our heuristics achieve at least as high network performance as the current state-of-the-art algorithm, which has a \(O(E^2 V)\) run time. Next, we provide a detailed description of our new algorithm for computing resilient QoS connections that has a \(O(V^2 \log V + E V)\) run time. Even though the proposed algorithms have very short run times, they achieve at least as good network performance as other algorithms including the corresponding LP formulation presented in Chapter 6. The proposed algorithms are therefore attractive for an operative environment.

The reminder of this chapter is organized as follows. An introduction follows. Then we provide a sketch of the algorithm that computes resilient QoS connection in Section 7.2. Section 7.3 presents new algorithms and their heuristics that compute non-resilient QoS connections. Then Section 7.4 gives the details of the algorithms that compute resilient QoS connections. Conclusions are given in Section 7.5.

## 7.1 Introduction

In this chapter we develop new algorithms for computing resilient QoS connections in the framework proposed in Chapter 6. In this framework, resilient QoS connections are protected end to end under a single-link failure model by computing two link-disjoint paths from the ingress to egress nodes (Suurballe and Tarjan, 1984; Kar et al., 2002; Norden et al., 2001; Li et al., 2003b). Furthermore, the framework assumes that the network provider uses MPLS and that requests for resilient QoS connections, subject to bandwidth guarantee and end-to-end delay constraints, are routed dynamically one by one as they arrive, without any knowledge of future re-
quests.

As observed in Chapter 6 the end-to-end delay depends on both the chosen path and the provisioned bandwidth. Exploiting this dependency, two link-disjoint paths can be computed dynamically with their respective bandwidths such that both paths meet the given bandwidth and end-to-end delay constraints. Thus, a solution to the routing problem of resilient QoS connections in this context not only contains two LSPs but their respective bandwidths as well. The approach whereby the dependency between end-to-end delay, chosen path and provisioned bandwidth is exploited has successfully been used in the context of computing non-resilient QoS connections (Q. Ma and P. Steenkiste, 1998; Stiliadis and Varma, 1998). To the best of our knowledge, this approach has not yet been used to compute resilient QoS connections in the current literature.

Our aim in this chapter is to provide algorithms that can be used in production, hence our design goals are both network performance and low run time, as opposed to the aim in Chapter 6 where the focus was on network performance alone. Furthermore, we aim to develop algorithms that can be used in both the 1+1 and 1:1 recovery contexts, thus we will not consider any backup resource-sharing strategies in this chapter. The first contribution in this chapter is a new polynomial-time algorithm that computes two link-disjoint paths and their respective bandwidths such that they both meet the given QoS constraints. Furthermore, the returned paths are subject to an optimality criterion of either minimized combined cost or number of hops. The proposed algorithm systematically uses a shortest-path algorithm to compute non-resilient QoS connections; this algorithm computes a single path and its bandwidth such that the returned path meets both the given bandwidth and end-to-end delay constraints. The previously proposed state-of-the-art algorithm that computes non-resilient QoS connections runs in $O(E^2 V)$ time (Q. Ma and P. Steenkiste, 1998). This is unfortunately too expensive for our purposes, as we have very stringent run time requirements. Hence, a second contribution in this chapter is a new algorithm that computes non-resilient QoS connections and its heuristics that run in $O(V \log V + E)$ time.
7.2 Sketch of an Algorithm for Computing Resilient QoS Connections

In Chapter 6, we presented an LP formulation that computes resilient QoS connections without any backup resource-sharing strategies and thus can be used in both the 1+1 and 1:1 recovery contexts. However, the long run time of the proposed formulation makes it unsuitable for an operational environment. Our aim in this chapter is to develop an algorithm that improves on the run time while maintaining high network performance, so that it can be used in a production environment.

Our approach is to deal with the additive end-to-end delay constraint in an additive manner. In other words, we want to build up two paths from \( s \) to \( d \) one hop at a time and adjust the bandwidth as we go, until we either reach \( d \) or until either path must be dismissed because it does not meet the given constraints.

The basic idea in our approach is to use a two-layered algorithm. The bottom layer computes a non-resilient QoS path and its required minimum bandwidth such that the QoS constraints are met. The algorithm in the top layer systematically uses colors and the bottom layer algorithm to extend two paths from \( s \) towards \( d \). The colors serve two purposes, to make sure the two paths will be link disjoint when the algorithm returns and to communicate restrictions to the bottom layer algorithm. Three colors are used, grey, blue and yellow, where grey is the “default” color, meaning that if neither blue nor yellow is present then the link or node is grey. The problem is now to find one blue and one yellow link-disjoint path from \( s \) to \( d \) and their respective bandwidths such that both paths meet the given QoS constraints. A sub-objective is to find two paths that have the shortest combined distance from \( s \) to \( d \), where the distance is measured either in minimum cost or number of hops.

In the remainder of this chapter links are considered unidirectional, we will refer to link \((u, v)\)’s upstream node \( u \) as the tail node and downstream node \( v \) as the head node.

Intuitively, the top layer algorithm works as follows: first find a shortest path \( P_b \) from \( s \) to \( d \). Once found, the first link on \( P_b \), that is the link with tail node \( s \), is
Efficient Algorithms for Computing Resilient QoS Connections

removed and a second shortest path $P_y$ is computed for the modified network. Once $P_y$ has been found, the removed link is reinserted into the network. The two paths will meet at some point towards $d$ in the network. If not before, they will meet at $d$, in which case both paths have been found. We refer to the node where the two paths meet as the rendezvous point (RV). More specifically, if $RV \neq d$ then RV is the tail node of the first common link on $P_b$ and $P_y$. This is illustrated in Figure 7.1 a). In the figure, the $<$ arrowheads indicate the current link directions. Next, all links and nodes on $P_b(s \leadsto RV)$ and $P_y(s \leadsto RV)$ are reversed and colored blue and yellow respectively, as illustrated in Figure 7.1 b). The requirement on $P_b(s \leadsto RV)$ and $P_y(s \leadsto RV)$ is that they must be link disjoint. Hence, a node can be traversed by both paths and thus be both blue and yellow at the same time, whereas a link cannot. A node $u$ that is being colored blue will remember the path segment $P_b(s \leadsto u)$, which is the first segment on $P_b(s \leadsto u \leadsto RV \leadsto d)$. Likewise, if it is colored yellow it will remember $P_y(s \leadsto u)$. A node that is colored both blue and yellow will remember both path segments. A stored path segment at node $u$ contains more information than just the path but for now we omit the full state information for clarity.

We know that the two paths join at RV and given that $RV \neq d$ we know that the two paths share the first downstream link $(RV, v)$, that is $P_b(s \leadsto RV \rightarrow v \leadsto d)$ and $P_y(s \leadsto RV \rightarrow v \leadsto d)$. Link $(RV, v)$ is now removed and two new shortest paths, $P'_b$ and $P'_y$ are computed from RV to $d$ as shown in Figure 7.1 c).

Once $P'_b$ and $P'_y$ have been computed, the top layer algorithm will have a total of four paths from $s$ to $d$. It now selects the pair of paths, $(P'_b, P'_y)$ or $(P_b, P_y')$, that has the combined shortest distance from $s$. Suppose that the former pair has a combined shorter distance than the latter. The algorithm then modifies $G$ so that the blue trail becomes $RV' \leadsto u \leadsto s$, rather than the current blue trail $RV \leadsto u \leadsto s$. Therefore, the algorithm “uncolors” nodes and links on segment $P_b(u \leadsto RV)$ and restores the original link direction. Then the algorithm colors and reverses links and nodes on segment $P'_b(u \leadsto RV')$, after which we will have the desired blue trail $RV' \leadsto u \leadsto s$. It then reassigns the blue path $P_b$ to $P'_b$ and reinserts link $(RV, v)$. Next, all links and nodes on path segment $P_y(RV \leadsto RV')$ are reversed and colored yellow. The
result is depicted in Figure 7.1 d).

This process is repeated until $RV = d$, after which we will have one blue and one yellow trail from $d$ to $s$. The final step is to reverse the colored links and return the two colored paths, and their required minimum bandwidths. The returned paths will be disjoint and will meet the given constraints.

The bottom layer shortest-path algorithm takes a network $G = (V, E)$, a request $Req = (s, d, TSpec, RSpec)$ and a color $col \in \{blue, yellow\}$ as input and uses the required minimum bandwidth $R \in [r, t)$ as a variable when computing a shortest path from $s$ to $d$ that meets the given constraints, that is, not only the path is returned but the required minimum bandwidth $R$ as well. If $col$ is blue, the returned path is guaranteed not to include any yellow links and vice versa.

To illustrate how the shortest-path algorithm operates, assume that we search for the blue path $P'_b$ from $RV$ to $d$ in Figure 7.1 b). Now, when the shortest-path algorithm follows a blue link in searching for the blue path, instead of updating the blue head node it uses the head node’s stored blue path segment $P_b(s \leadsto u)$. In effect, when a blue link is followed, the algorithm will be moving along the blue trail from $RV$ towards $s$. Eventually the algorithm will divert from the blue trail, say at node $u$ on its way to $d$. Therefore, one might think that the resulting path would be $P'_b(RV \leadsto u \leadsto d)$. However, the resulting path will be $P'_b(s \leadsto u \leadsto d)$, because when the search for the blue path diverted from the blue trail at node $u$ it expanded $u$’s memorized state, that is $P_b(s \leadsto u)$ rather than $P'_b(RV \leadsto u)$.

Next, we present the bottom layer algorithm and its heuristic that compute non-resilient QoS connections in Section 7.3. Then in Section 7.4 we present the details of the top layer algorithm we sketched in this section.

### 7.3 Computing Non-Resilient QoS Connections

In this section we present a new algorithm that we call Dynamic Bandwidth Shortest Path (DBSP). The problem that DBSP addresses is: given a network $G = (V, E)$, a request $Req = (s, d, TSpec, RSpec)$ and a color $col \in \{blue, yellow\}$, find a
Figure 7.1 Illustration of the top layer algorithm.
single shortest path and its required minimum bandwidth from \(s\) to \(d\) that meets the constraints given by \(T\text{Spec}\) and \(R\text{Spec}\) such that no link on the path has a color other than \(col\) and grey. Furthermore, the returned path \(P\) is a shortest path in the sense that it has the shortest distance from \(s\) measured either in minimum cost or number of hops. Each node \(u\) in \(G\) on the returned path will hold complete state information about the initial path segment \(P(s \rightsquigarrow u)\). We must ensure that the returned path not only conforms to the rate-related parameters given in the \(T\text{Spec}\) but also to the end-to-end delay as specified in the \(R\text{Spec}\). The only explicit constraints put on the provisioned bandwidth \(R\) are therefore that \(R\) is at least as high as the sustainable rate and at most as high as the peak rate, that is \(R \in [r, t]\). Thus the returned path might require more bandwidth than the sustainable rate.

The basic approach we take in this chapter is to exploit the relationship between the provisioned bandwidth and the end-to-end delay, as detailed in Equation (6.1). In particular we exploit the fact that the relationship is additive, which implies that a link can be added to a path and tested whether or not the now extended path conforms to the delay constraint in constant time. Conversely, we can find the required minimum bandwidth to provision on the extended path so that it complies to the delay constraint in constant time.

Using this method, we start at the ingress node \(s\) and add one link at a time, adjusting the bandwidth as we go until we either reach the egress node \(d\) or have to dismiss the path. There are two reasons to dismiss a path: the bandwidth may need to be adjusted beyond the path’s residual capacity to meet the constraints; or the bandwidth may need to be adjusted beyond the peak rate \(t\) given in the \(T\text{Spec}\), which is an upper theoretical limit in Equation (6.1). In the equation, \(R\) must be smaller than \(T\), thus we have \(R \in [r, t]\).

The required minimum bandwidth to provision, \(R_{P_u}\), on a given path \(P_u\) from \(s\) to node \(u\) can be derived from Equation (6.1). Rearranging Equation (6.1) we obtain:

\[
D_m = \frac{t - R_{P_u}}{R_{P_u}} \cdot \frac{b}{l - r} + |P_u| \cdot M \cdot \sum_{(i,j) \in P_u} \left( \frac{M_{ij}^m}{C_{ij} + \text{prop}_{ij}} \right) \tag{7.1}
\]

where \(|P_u|\) is the length of path \(P_u\). After substituting the constant \(\frac{b}{l - r}\) with \(Z\) and
defining $K_{P_u}$ as:

$$K_{P_u} = \sum_{(i,j) \in P_u} \left( \frac{M_{ij}^m}{C_{ij}} + \text{prop}_{ij} \right)$$  \hspace{1cm} (7.2)

the expression given in Equation (7.1) can be reduced to

$$D_m = t - R_{P_u} \cdot Z + |P_u| \cdot \frac{M}{R_{P_u}} \cdot K_{P_u}$$ \hspace{1cm} (7.3)

Now, extracting $R_{P_u}$ in Equation (7.3) we obtain an expression showing how the provisioned minimum bandwidth depends on the chosen path:

$$R_{P_u} = \frac{t \cdot Z + M \cdot |P_u|}{D_m - K_{P_u} + Z}$$ \hspace{1cm} (7.4)

where $t$, $Z$, $M$ and $D_m$ are constants independent of the actual path. Note that both $|P_u|$ and $K_{P_u}$ can be incrementally calculated in constant time as the path is extended. Say that node $v$ is added to $P_u$ as the next hop from node $u$, then $|P_v| = |P_u| + 1$ and:

$$K_{P_v} = \sum_{(i,j) \in P_v} \left( \frac{M_{ij}^m}{C_{ij}} + \text{prop}_{ij} \right)$$

$$= \sum_{(i,j) \in P_u} \left( \frac{M_{ij}^m}{C_{ij}} + \text{prop}_{ij} \right) + \frac{M_{uv}^m}{C_{uv}} + \text{prop}_{uv}$$ \hspace{1cm} (7.6)

$$= K_{P_u} + \frac{M_{uv}^m}{C_{uv}} + \text{prop}_{uv}$$ \hspace{1cm} (7.7)

Furthermore, we define the cost of using link $(i, j)$ as $c_{ij} = R \cdot F_{ij}$, where $F_{ij}$ is a constant that reflects the link weight. Typically, $F$ can be used to load-balance the network, discouraging use of expensive links and encouraging use of cheap links.

Now, the cost of using path $P_u$ can be calculated as $c_{P_u} = \sum_{(i,j) \in P_u} (R_{P_u} \cdot F_{ij}) = R_{P_u} \cdot \sum_{(i,j) \in P_u} F_{ij} = R_{P_u} \cdot F_{P_u}$. Here we see that the cost of using path $P_u$ is cumulative. Suppose again that we extend path $P_u$ with link $(u, v)$ to yield $c_{P_u} = R_{P_v} \cdot \sum_{(i,j) \in P_v} (F_{ij}) = R_{P_v} \cdot (F_{P_u} + F_{uv})$. Hence, the cost can also be computed in constant time as the path is extended.

We define a Path Candidate (PC) as a container that holds the following information about a path $P_u$ from $s$ to $u$:

- the actual path $P_u$ from $s$ to $u$;
Table 7.1 Functions defined on a PC $x$ at node $u$.

<table>
<thead>
<tr>
<th>$P(x)$</th>
<th>path from $s$ to $u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p(x)$</td>
<td>length of $P(x)$</td>
</tr>
<tr>
<td>$H(x)$</td>
<td>residual capacity on $P(x)$</td>
</tr>
<tr>
<td>$K(x)$</td>
<td>as defined in Equation (7.2)</td>
</tr>
<tr>
<td>$R(x)$</td>
<td>required minimum bandwidth to be provisioned on $P(x)$ as given by Equation (7.4)</td>
</tr>
<tr>
<td>$F(x)$</td>
<td>accumulated weight along $P(x)$</td>
</tr>
<tr>
<td>$h(x)$</td>
<td>residual capacity after provisioning: $H(x) - R(x)$</td>
</tr>
<tr>
<td>$c(x)$</td>
<td>the cost of using path $P(x)$, given by $R(x) \cdot F(x)$</td>
</tr>
</tbody>
</table>

- the path length $|P_u|$;
- the minimum residual capacity $H_{P_u}$;
- $K_{P_u}$ as defined in Equation (7.2);
- the required minimum bandwidth, $R_{P_u}$ to be provisioned;
- the cumulative link weights $F_{P_u}$.

A PC at node $u$ thus represents an initial segment on a potential path from $s$ to $d$ that traverses node $u$. We define a number of functions on a PC $x$ at node $u$ as given in Table 7.1. Note that all these functions can be implemented in constant time.

One can expect that there are several distinct paths from $s$ to $u$. Because each distinct path can be represented by a unique PC we will, in that case, have several PCs at node $u$. We are only interested in PCs that have the potential to be extended to the egress node $d$. Furthermore, we want to dismiss redundant PCs. Consider two PCs $x$ and $y$ at node $u$: $y$ is defined as redundant if

- $y$ can be extended to $d \Rightarrow x$ can be extended to $d$
- $x$ can be extended to $d \Rightarrow y$ can be extended to $d$

That is, if $y$ can be extended to the egress node $d$ then $x$ can too but if $x$ can be extended to $d$ then $y$ may or may not.

We need a way to determine whether a PC is redundant. There are two reasons for extensions of a PC to fail. First if there is not enough residual capacity on the links ahead and secondly if the bandwidth must be adjusted so that the minimum residual
capacity on the path is breached. In both cases, the residual capacity on the path plays an important role and in the latter case the current required minimum bandwidth becomes important as well. A higher required minimum bandwidth indicates that the required minimum bandwidth grows faster because all PCs originate from the same source node and thus if two PCs at node $u$ have the same residual capacity after provisioning, then after being extended over the same link to node $v$, the PC with the higher required minimum bandwidth at node $u$ will have less residual capacity at node $v$.

In other words, more residual capacity on a path results in a higher chance of reaching the egress node. Moreover, lower required minimum bandwidth also improves the chance of reaching the egress. Thus the functions $h(x)$ and $R(x)$ are measures of how likely a PC is to extend all the way to the egress.

Therefore, given two PCs $x$ and $y$, we say that $y$ is redundant if $h(y) < h(x)$ and $R(y) > R(x)$. If $h(x) = h(y)$ and $R(x) = R(y)$, then they have equal chance of reaching the egress node and thus $x$ and $y$ are mutually redundant. It is therefore sufficient to keep either one of them. In all other cases, we need to keep both to ensure that a path from $s$ to $d$ is eventually found if one exists.

We will say that $y$ can overtake $x$ if either or both of the following conditions are true: $h(y) > h(x)$ and $R(y) > R(x)$. The intuition behind this is illustrated in Figure 7.2, which shows two PCs, $x$ and $y$ at node $u$ and their respective extensions $x'$ and $y'$ over the link $(u,v)$. Even though $x$ has more residual capacity than $y$, we need to keep both of them. The reason is that after they are extended over link $(u,v)$ with residual capacity 10, $y'$ will make $x'$ redundant and hence $x'$ will be dismissed at node $v$.

It is possible that several feasible paths can be found from $s$ to $d$. This raises the question of which path to choose. One common way is to choose the path with the lowest cost; if several such paths exist the path with the most residual capacity after provisioning is chosen. Another way is to choose the path with the minimum hop count and if several such paths exist, the path with the most residual capacity after
Figure 7.2 Intuition behind *overtaking*.

provisioning is chosen. We refer to the former path selection criterion as MinCost and the latter as WidestShortest, in consistent with the terminology used by Ma and Steenkiste (Q. Ma and P. Steenkiste, 1998).

Now we are ready to define an order between PCs. Given the MinCost path selection criterion and two PCs $x$ and $y$, the order is defined as:

$$
x < y = \begin{cases} 
  \text{true} & \text{if } c(x) < c(y), \\
  \text{true} & \text{if } c(x) = c(y) \text{ and } h(x) > h(y), \\
  \text{false} & \text{otherwise}, \\
  \text{true} & \text{if } c(x) = c(y) \text{ and } h(x) = h(x) \text{ and } R(x) < R(y), 
\end{cases}$$

(7.8)

where the first line states that $x$ is smaller than $y$ if it has lower cost. The second condition states that if $x$ and $y$ have the same cost but $x$ has more residual capacity after provisioning then again is $x$ smaller than $y$. The third condition states that if $x$ and $y$ both have the same cost and residual capacity after provisioning, then $x$ is smaller than $y$ if $x$ has lower required minimum bandwidth. In other words, two PCs are considered equal if and only if they have the same cost, the same residual capacity after provisioning and the same required minimum bandwidth. If two PCs are equal, they have the same chance of reaching the egress node and furthermore, if they reach the egress node, they will do so at the same cost. Similarly, given the
WidestShortest path selection criterion and two PCs $x$ and $y$ the order is defined as:

$$
x < y = \begin{cases} 
true & \text{if } p(x) < p(y), \\
true & \text{if } p(x) = p(y) \text{ and } h(x) > h(y), \\
true & \text{if } p(x) = p(y) \text{ and } h(x) = h(x) \text{ and } R(x) < R(y), \\
false & \text{otherwise},
\end{cases}
$$

(7.9)

where the only difference to the MinCost comparison function is that the cost function $c$ in Equation (7.8) has been replaced by the path length function $p$.

In the rest of this chapter, “shortest path” refers to a path with the shortest distance from $s$ measured by either the cost function or the path length function.

### 7.3.1 Path Candidate Set

It is clear that nodes will be associated with several PCs, even though we can probably discard some of them. Therefore, we associate each node with a Path Candidate Set (PCS), which is a strictly increasing ordered set of PCs for the initial segment on a potential path from $s$ to $d$. We denote the PCS at node $u$ as $PCS_u$. Furthermore, we require that all PCs in a PCS can potentially overtake all comparably smaller PCs in the same PCS. Each PC in $PCS_u$ is a candidate for the initial segment of the shortest path from $s$ to $d$ that goes through $u$, that is $s \sim u \sim d$. Moreover, if there is a shortest path from $s$ to $d$ via $u$, then $PCS_u$ will contain a PC that represents the initial segment on that shortest path.

The PCs in a PCS are subject to a number of rules. Let $PCS_u = \{x_0, x_1, ..., x_n\}$. Now, we know from the definition of PCS that the elements in $PCS_u$ are strictly increasing, so for any $i \in [0, n - 1]$ and $j \in [i + 1, n]$, $x_i < x_j$. Furthermore, each element has the possibility of overtaking all other elements in the same PCS that are smaller. As shown above, there are two ways a larger PC can overtake a smaller PC further along the path, either through having more residual capacity after provisioning or to grow more slowly. At least one of the two following statements must be true:
Assuming that we use the comparison operator defined in Equation (7.8), then there are $c_0, c_1, ..., c_m$, $m \in [0, n]$ distinct costs associated with the $n + 1$ elements in $PCS_u$. Without loss of generality we can make $c_0 < c_1 < ... < c_m$. Furthermore, we define the Equal Distance (ED) set as the set of PCs in PCS with equal cost, that is $ED^u_j = \{ x \in PCS_u | c(x) = c_j \}$, $j \in [0, m]$. Thus, $PCS_u$ can be expressed as $ED^u_0 \cup ED^u_1 \cup ... \cup ED^u_m$.

Likewise, when using the comparative operator defined in Equation (7.9), there are $p_0, p_1, ..., p_m$, $m \in [0, |V|]$ distinct path lengths associated with the $n + 1$ elements in $PCS_u$. Again, without loss of generality we can make $p_0 < p_1 < ... < p_m$. Here, we define the ED set as the set of PCs in PCS with equal path length, that is $ED^p_j = \{ x \in PCS_u | p(x) = p_j \}$, $j \in [0, m]$. Thus, $PCS_u$ can be expressed as $ED^p_0 \cup ED^p_1 \cup ... \cup ED^p_m$.

What is the internal structure of the set $ED$? Its elements are sorted in a strictly increasing order, and hence no two elements compare equal. We also know that they all have the same distance from $s$, which is given in either cost or path length. Furthermore, we know that no element is redundant when compared to smaller elements, that is a larger element in $ED$ must have a chance to overtake a smaller element somewhere down the track. Hence the following rules apply to all elements $\{ x_i \in ED | x_{i+1} \in ED \}$:

\[
\begin{align*}
  h(x_i) &< h(x_{i+1}) \\
  R(x_i) &> R(x_{i+1})
\end{align*}
\]  

The first rule results from the dominance of the residual capacity after provisioning over the required minimum bandwidth. It is strictly decreasing because if $h(x_i) = h(x_{i+1})$ then the element with the higher required bandwidth, that is $x_{i+1}$ because $x_i < x_{i+1}$, does not belong in the original PCS at all because $x_{i+1}$ then would be redundant. As a consequence of the second rule, the only way a consecutive element
can overtake its predecessor further down the track is to grow more slowly, that is \( R(x_i) > R(x_{i+1}) \).

If we are given a strictly increasing ordered list \( L = \{x_0, x_1, ..., x_n\} \) of PCs, how do we select PCs from that list to form a PCS? The only requirement on elements in the list is that they are strictly increasing; this requirement applies to a PCS as well. However, a PCS has the additional requirement that an element must have the possibility of overtaking all comparably smaller elements. Clearly, then, we do not need to change the order of elements from the list when we are constructing the PCS, but we probably need to discard a number of them.

Given \( ED_i \) and an element \( \{x \in L \mid x > y, \forall y \in ED_i\} \), \( x \) can overtake all elements in \( ED_i \) if some \( y_j \in ED_i \) exists such that either \( h(y_j) > h(x) > h(y_{j+1}) \) and \( R(y_j) > R(x) \), or \( h(y_j) > h(x) \) and \( R(y_j) > R(x) \) for all \( y_j \in ED_i \). To overtake all elements in PCS, \( x \) needs to fulfill these requirements on all \( ED_i \in PCS \).

The idea that we will pursue to construct a PCS from an ordered list \( L \) is to add one ED at a time starting with \( ED_{i=0} \), where the distance is given by the first element in the list \( L \), that is either \( c_0 = c(x_0) \) or \( p_0 = p(x_0) \), depending on which comparison operator definition we use. Then \( i \) is stepped up for each PC in the list with a longer distance than the previous PC. Once we have formed all EDs we simply append them together to form the PCS.

Clearly the first PC \( x_0 \) will be added to \( ED_0 \). The second PC \( x_1 \) either can overtake \( x_0 \) or not. If not, it is discarded. If it can overtake the previous element and has the same distance then it is appended to \( ED_0 \). If \( x_1 \) has longer distance than \( x_0 \) then \( x_1 \) is added to \( ED_1 \) with \( c_1 = c(x_1) \) or \( p_1 = p(x_1) \).

Generalizing the procedure, assume we currently work on \( ED_i \) and evaluate element \( x_j \). We first check whether \( x_j \) can overtake all elements in all EDs according to the procedure described above. If not it is discarded. Otherwise if \( c(x_j) = c_i \) or \( p(x_j) = p_i \) then it is appended to the end of \( ED_i \), otherwise \( ED_i \) is closed and \( ED_{i+1} \) created with \( c_{i+1} = c(x_j) \) or \( p_{i+1} = p(x_j) \) to which \( x_j \) is appended.
What is the run time of the above procedure? Assume that $L$ has $n$ elements of which $k$ have distinct distances. Then we will have $k$ EDs with at most $n - k$ elements. Hence, finding whether element $x_j \in L$ can overtake all elements in an ED will take at most $O(\log(n - k))$ time and we have $k$ such operations. Thus it will take at most $O(k \log(n - k))$ time. Because $k$ is bounded by $n$ we have $O(n \log(n - n)) = O(n)$. There are $n$ elements in the list, so the total run time becomes $O(n^2)$. If we are measuring distance through the path length function, $k$ is bounded by $|V|$ and thus the run time is improved to $O(n |V| \log(n - |V|))$.

### 7.3.2 The Dynamic Bandwidth Shortest-Path Algorithm

In this section we introduce DBSP, which, given a network graph $G = (V, E)$, a request $Req = (s, d, TSpec, RSpec)$ and a $col \in \{\text{blue, yellow}\}$, finds a single shortest path and its required minimum bandwidth from $s$ to $d$ that meets the requirements given by TSpec and RSpec such that all links on the returned path are either $col$ or grey. That is, if $col$ is blue then no links on the returned path will be yellow and vice versa. Furthermore, a sub-objective of DBSP is to minimize the returned path’s distance from $s$, that is $\min(c(P_s) + c(P_b))$ or $\min(p(P_s) + p(P_b))$, depending on which distance measure is used.

We base DBSP on Dijkstra’s shortest-path algorithm (Cormen et al., 1990) for two reasons. Firstly, it has an appealing run time $O(|E| \log |V|)$ when implemented with a binary heap and $O(|V| \log |V| + |E|)$ when implemented with a Fibonacci heap (Cormen et al., 1990). Secondly it builds up a shortest-path tree from $s$ by extending paths one hop at the time.

The major difference between DBSP and Dijkstra’s shortest-path algorithm is that each node in DBSP is associated with a set of shortest-path candidates, the PCS (see Section 7.3.1, as opposed to Dijkstra, where each node only holds at most one shortest-path candidate. This difference theoretically increases DBSP’s run time, but as we shall see, in practical terms the run time of DBSP can be improved to an amortized constant time difference to Dijkstra’s shortest-path algorithm with negligible degradation of network performance.
In DBSP, each node is associated with a PCS initialized to the empty set. The algorithm repeatedly pulls the node with the smallest PC into the cloud and relaxes its neighbors until all nodes are in the cloud.

The relaxation step updates the neighbors’ PCSs. Suppose that we relax node $u$’s neighbor $v$. Each PC in $PCS_u$ is first extended with link $(u, v)$ and if they still comply with the given constraints they are then considered for addition to $PCS_v$ to improve $PCS_v$. In DBSP it may happen that $v$ is already in the cloud. This is not a problem in the original Dijkstra’s shortest-path algorithm where the estimated shortest path to a node $v$ is deterministically computed when the node is pulled into the cloud. In other words, once a node is pulled into the cloud its estimate will never improve and all possible shortest paths to the egress node $d$ via $v$ will use that same initial segment from $s$ to $v$. However, when using DBSP, a shortest path from $s$ to $d$ via $v$ may use different initial segments from $s$ to $v$ depending on the rest of the path from $v$ to $d$. Thus a node that is in the cloud can still be improved and this improvement must be propagated towards $d$.

To cope with this problem, DBSP initializes each node as dirty and later when it is pulled into the cloud, it is set to clean. If a node in the cloud subsequently has its PCS improved in the relaxation procedure it is marked as dirty again. After all nodes currently reachable from $s$ have been pulled into the cloud, nodes marked as dirty are pulled out again and the relaxation procedure is rerun. This process is repeated until no more dirty nodes appear in the cloud.

Iteration over dirty nodes introduces a side effect. Consider the two PCs $x$ and $y$ at node $u$, shown in Figure 7.3. Assume that $x$ is clean and $y$ dirty. That is, $x$ reached $u$ in a previous iteration while $y$ is being considered for addition for the first time. Furthermore, assume that $x$ and $y$ have the same distance (path length) and growth rate but $y$ has more residual capacity after provisioning than $x$. Then $y$ will push $x$ out of $PCS_u$, because $y$ makes $x$ redundant and consequently $u$ will “forget” $x$. Now, if we extend $y$ over link $(u, v)$, its residual capacity may be such that it becomes the residual capacity bottleneck link on both PC extensions, that is on $x'$ and $y'$. Because $x$ was extended in a previous iteration, $PCS_v$ already contains $x'$. When $y'$ is considered for addition to $PCS_v$, it will compare equal to $x'$, because $x'$
and $y'$ now have the same residual capacity after provisioning and therefore they are mutually redundant. Here $y'$ will be dismissed rather than $x'$, because $x'$ came first.

Thus, we have a situation where the extension of $x$ can end up as the winner but when following that path through the network to extract the corresponding PC at each node we will see that node $u$ cannot retrieve $x$ because node $u$ forgot $x$. To prevent this, instead of forgetting clean PCs that are pushed out of a PCS such PCs are added to a history list. When a node is asked to retrieve a specific PC it first looks in its PCS and if not found there, it checks its history list. One can argue that it is better to replace $x'$ at node $v$ with $y'$ because they compare equal and thus have the same chance of reaching the destination node. However, doing so will result in more iterations over dirty nodes and thus degrade the overall run time.

Another problem that we encounter in DBSP, which is not present in the original Dijkstra shortest-path algorithm, is that $d$ can suddenly turn out to be unreachable with respect to the current bandwidth. Even though DBSP starts out with a connected graph $G$ where each link has at least $r$ units of residual capacity, as the algorithm progresses towards $d$ the required minimum bandwidth may be adjusted so that $d$ becomes unreachable. This is not a problem in Dijkstra’s algorithm because the bandwidth remains constant throughout. To cope with this problem DBSP uses an additional check: if $d$ is not in the cloud once the relaxation completes, DBSP concludes that there is no path from $s$ to $d$ that meets the given constraints and hence returns false.

The DBSP algorithm is given in Algorithm 7.1. It takes a directed network graph $G$, 

![Figure 7.3 Illustration of the need for history lists.](image-url)
an ingress node \( s \) and egress node \( d \) along with a TSpec, a RSpec and a color \( \text{col} \) as input. DBSP returns \( true \) if it has found a path with the shortest distance from \( s \) to \( d \) in \( G \) that meets both the TSpec and RSpec, otherwise it returns \( false \). In the given description of DBSP we are using the comparative operator with path length as its measure of distance. Simply substitute the path length function \( p(x) \) with the cost function \( c(x) \) to use the cost measure of distance instead.

DBSP starts with a call to \( \text{Initialize} \), shown in Algorithm 7.2. \( \text{Initialize} \) first marks all nodes in \( G \) as dirty. Then if node \( u \) has the same color as the input color then the colored PC is retrieved and added to \( PCS_u \). Otherwise, if \( u \) is the source node, an elementary PC (zero distance) is added to \( PCS_u \), or else a PC with infinite distance is added to \( PCS_u \).

Next, DBSP initializes the cloud \( Cloud \) to all nodes in \( G \); in effect all nodes in \( Cloud \) are now dirty. It then enters a loop that begins by pulling all dirty nodes out of \( Cloud \) and storing them in \( Q \). Then it enters a new loop which is the actual relaxation step. The node \( u \) with the shortest distance is first extracted from \( Q \). If it shows that the shortest distance from the cloud to the closest node is infinite then DBSP concludes that the cloud already contains all nodes reachable from \( s \) and breaks the while loop. If the distance is less than infinite, node \( u \) is marked as \( clean \) and pushed into the cloud. Now, each neighbor \( v \) to \( u \) is relaxed before the while-loop wraps around.

When either \( G \) has been partitioned or all nodes in \( Q \) have been pushed into the cloud, DBSP checks whether there are any \( dirty \) nodes in the cloud. If so, the \( dirty \) nodes are pulled out again and the relaxation procedure is repeated. Once the cloud only contains \( clean \) nodes, the repeat loop terminates. If \( d \) is not in the cloud, it cannot be reached from \( s \) and DBSP returns false. Otherwise, the shortest path is represented by the first PC in \( PCS_d \), that is \( PCS_d[0] \). The required path can be extracted from that PC using the \( P() \) function, similarly the required minimum bandwidth can be extracted using the \( R() \) function.

We now turn our attention to \( \text{Relax} \), given in Algorithm 7.3, which updates \( u \)'s neighbors. Suppose that node \( v \) is reached via node \( u \) over link \( (u, v) \). Now, if link \( (u, v) \) is colored either blue or yellow, node \( v \) must not be updated and thus \( \text{Relax} \)
Input: \( G, TSpec, RSpec, s, d, col \)

Output: \( G, P, R \)

\[ G \leftarrow \text{Initialize}(G, s, col); \]
\[ Cloud \leftarrow V[G]; \]
\[ \text{repeat} \]
\[ Q \leftarrow \text{all dirty nodes in } Cloud; \]
\[ \text{while } u \leftarrow \text{ExtractMin}(Q) \text{ do} \]
\[ \quad \text{if } p(PCS_u[0]) = \infty \text{ then break; } \]
\[ \quad u \leftarrow \text{clean; } \]
\[ \quad Cloud \leftarrow Cloud \cup u; \]
\[ \quad \text{foreach } v \in Adj[u] \text{ do} \]
\[ \quad \quad \text{Relax}(u, v, TSpec, RSpec); \]
\[ \quad \quad \text{decrease key in } Q \text{ for } v; \]
\[ \text{until no dirty nodes in } Cloud; \]
\[ \text{if } d \notin Cloud \text{ then return false;} \]
\[ P \leftarrow P(PCS_d[0]); \]
\[ R \leftarrow \max(TSpec.r, R(PCS_d[0])); \]
\[ \text{return true; } \]

Algorithm 7.1: DBSP

---

Input: \( G, s, col \)

Output: \( G \)

\[ \text{foreach } u \in G \text{ do} \]
\[ \quad u \leftarrow \text{dirty; } \]
\[ \quad PCS_u \leftarrow \emptyset; \]
\[ \quad \text{if } u \text{ has color } col \text{ then} \]
\[ \quad \quad \text{retrieve } col\text{-colored PC and add it to } PCS_u; \]
\[ \quad \text{else if } u = s \text{ then} \]
\[ \quad \quad \text{add elementary PC to } PCS_u; \]
\[ \quad \text{else} \]
\[ \quad \quad \text{add PC with } \infty \text{ distance to } PCS_u; \]
\[ \text{return } G; \]

Algorithm 7.2: Initialize
returns. If the link is grey, each PC in $PCS_u$ must be extended over link $(u, v)$, and if an extended PC complies with both the given peak rate $t$ and the available residual capacity, it becomes a potential candidate to $PCS_v$ and hence should be considered for addition. $Relax$ first builds a strictly increasing ordered list $L$. $L$ is initialized with PCs that already are in $PCS_v$ before all potential new candidates extended from $PCS_u$ are inserted. If a new candidate is found equal to an element already in the list the new candidate is discarded. This way, PCs that made their way into $PCS_v$ first take precedence over newer equally good candidates. Once all PCs have been considered, the resulting list $L$ is used to construct $PCS_v$ according to the method described in Section 7.3.1.

Taking a closer look at $Relax$, it starts by initializing two PCs, $x$ and $y$, after which it initializes a strictly increasing ordered list $L$ to $PCS_v$. Then in the $foreach$ loop it extends each PC in $PCS_u$ to $v$ over link $(u, v)$ and marks the extended PC as dirty. On line 13 the required bandwidth on the extended path is computed and compared to the available residual capacity. If it passes the comparison, its required bandwidth is then compared to the peak rate $t$, as specified in the $TSpec$. If it passes this test too, it is inserted into $L$, otherwise $x$ is simply discarded, because it does not meet the end-to-end delay bound even when the peak rate is used.

When all successfully extended PCs have been added to $L$, $Relax$ enforces the PCS requirement on it according to the procedure described in Section 7.3.1. $Relax$ starts by initializing $PCS_v$ and $ED$ to the empty set. Then each element $x$ in $L$ is considered for addition to the current $ED$ in increasing order. Next, $Relax$ checks whether $x$ can overtake all previously added elements, that is all elements in $PCS_v$ and $ED$. If so, $x$’s path length is compared to $ED$’s length. If it is longer, then $ED$ is appended to the end of $PCS_v$ and reassigned to the empty set. Clearly we are using the comparison operator defined in Equation (7.9); if we were using the comparative operator defined in Equation (7.8), then this line would test if $c(x) > c(EC)$ instead.

Next, if $x$ is dirty, then node $v$ must also be set to dirty to ensure that it will be pulled out of the cloud again if $v$ happens to be in the cloud already. $lastDirty$ is also updated to track the most recently added dirty node. $x$ is then added to the current $ED$. 
If $x$ is clean but could not overtake all PCs already in $PCS_v$, then it has been pushed out by a dirty PC, so if $x$ has the same required bandwidth as the one pushing $x$ out, $x$ is added last in $v$’s history list. Once all elements in $L$ have been considered for addition, the current $ED$ is appended to the end of $PCS_v$.

If DBSP finds a path, that path is guaranteed to contain only links colored either grey or $col$. Assume that DBSP takes a graph $G$ with grey, blue and yellow links and nodes and that the given color is blue, then the returned path must not contain any yellow links. Now, Initialize initializes yellow nodes with an infinite distance PC rather than retrieving the node’s yellow PC. If the node is blue as well, then the blue PC is retrieved. Thus, after initialize returns, all PCSs contain either an infinite PC or a PC representing a blue path segment. Furthermore, when Relax is called it first checks whether the given link is grey or not. If it is not, then Relax simply returns without relaxing the head node and thus leaves the head node’s PCS unaltered. In effect no PCS will be updated with a PC that contains a yellow link and thus the returned path will not contain any yellow links.

### 7.3.3 DBSP Complexity

In this section we will analyze the run time of DBSP. We will start with Initialize, followed by Relax, and then finally DBSP.

#### 7.3.3.1 Initialize Analysis

Initialize is built with one foreach loop that iterates through all $|V|$ nodes in $G$. Each node is initialized to dirty and its $PCS$ is cleared; both these operations are constant time. Next is a constant-time test and in either case a PC is added to the empty $PCS$ which again is a constant-time operation. Hence, the total run time for Initialize is $O(|V|)$.

#### 7.3.3.2 Relax Analysis

The first two lines in Relax run in constant time. The initialization of $L$ on line 4 takes $O(|L| \log |L|)$ time if $L$ is implemented as a binary heap and $PCS$ as a linked list, implicitly ordered by the way it is built up. In the following foreach loop, a
Input: $u, v, TSpec, RSpec$

Output: none

if $(u, v)$ not GREY then return;

$x \leftarrow \text{nil}$;

$y \leftarrow \text{elementary PC}$;

$L \leftarrow PCS_v$;

**foreach** $x \in PCS_u$ **do**

$P(y) \leftarrow P(x) \cup v$;

$H(y) \leftarrow \min(C_{uv}, H(x))$;

$K(y) \leftarrow K(x) + \frac{M_{uv}}{C_{uv}} + \text{prop}_{uv}$;

$p(y) \leftarrow p(x) + 1$;

$R(y) \leftarrow \text{Equation (7.4)}$;

$F(y) \leftarrow F(x) + F_{uv}$;

$y \leftarrow \text{dirty}$;

if $\max(R(y), TSpec.r) \leq H(y)$ then

if $R(y) < TSpec.t$ then

insert $y$ into $L$;

$PCS_v \leftarrow \emptyset$;

$ED \leftarrow \emptyset$;

$lastDirty \leftarrow \text{nil}$;

**foreach** $x \in L$ **do**

if $x$ overtakes all elements $\in \{PCS_v \cup ED\}$ then

if $p(x) > p(ED)$ then

append $ED$ to $PCS_v$;

$ED \leftarrow \emptyset$;

if $x$ is dirty then

$v \leftarrow \text{dirty}$;

$x \leftarrow \text{clean}$;

$lastDirty \leftarrow x$;

append $x$ to $ED$;

else if $x$ is clean then

if $R(x) = R(lastDirty)$ then

add $x$ to $v$’s history;

append $ED$ to the end of $PCS_v$;

Algorithm 7.3: Relax
number of constant time operations are applied to each element on lines 6 through 12. Then follow two constant-time tests and an insertion of $y$ into $L$ on line 15. The insertion takes $O(\log |L|)$ time. The foreach loop is run $|L|$ times and thus we spend $O(|L| \log |L|)$ time inserting elements into $L$.

The algorithm then has three constant-time initializations and a foreach loop on line 19. The foreach loop iterates over $L$ once for each of its $|L|$ elements, starting with the smallest. The first line in the foreach loop, line 20, tests whether the current element can overtake all other elements already added. As shown in Section 7.3.1 that operation takes $O(|L|)$ time. If the result is true, another test is made on line 21 that runs in constant time, after which there is a constant-time append operation adding the current $ED$ to $PCS_v$ and a constant-time assignment on line 23. Now, if the current element is dirty, three more constant-time operations are made on lines 25 through 27. Then on line 28 $x$ is added to the end of the current $ED$, again a constant-time operation. Next, if the element does not overtake all elements already added but it is clean, then after a constant-time check $x$ is added to the end of $v$’s history list, and all these operations are constant time. Therefore, the run time of the foreach loop is $O(|L|)$ and the loop is executed $|L|$ times giving a total run time of $O(|L|^2)$. On the last line in Relax $ED$ is added to the end of $PCS_v$, which is a constant-time operation. The total run time for Relax is dominated by the second foreach loop and thus Relax runs in $O(|L|^2)$ time. $|L|$ is bounded by $2 \cdot |PCS|$ and therefore Relax’s run time can be expressed as $O(|PCS|^2)$.

7.3.3.3 DBSP Analysis

In this section we analyze the run time of DBSP. It begins with a call to Initialize, which has run time $O(|V|)$, as shown in Section 7.3.3.1. Next the non-ordered set Cloud is initialized to all $|V|$ nodes in $G$, which is a $O(|V|)$ operation.

Then inside the repeat loop, $Q$ is initialized to all the dirty nodes in Cloud; this operation takes $O(|V| \log |V|)$ time if $Q$ is implemented as a binary heap. Then on line 5, the node with the smallest PC is extracted from $Q$, which takes $O(\log |V|)$ time. The extraction is performed $|Q|$ times and thus will take a total of $O(|V| \log |V|)$ time. Then the extracted element $u$ is tested for its distance, which is a constant time
operation. If the result is true, the while loop is interrupted. Otherwise \( u \) is set to \textit{clean}, which again is a constant operation, before it is pushed into the cloud on line 8. Pushing \( u \) into the cloud takes constant time because \( u \) simply is added to the end of the unstructured \textit{Cloud}.

Next follows a \textit{foreach} block on line 9. Here, each outgoing link is followed to \( u \)’s neighbors. Then \textit{Relax} is called for each neighbor to \( u \). A call to \textit{Relax} takes \( O(|PCS|^2) \) time. \textit{Relax} will be called exactly once for each node’s outgoing links and we do this for each node in \( G \), that is exactly once for each of \(|E| \) links. Thus the total computation of \textit{Relax} will take \( O(|E||PCS|^2) \) time. After a node has been relaxed, its key, which is the smallest PC’s distance, must be updated in \( Q \). Decreasing the key in a binary heap takes \( O(\log |V|) \) time and is executed at most \(|E| \) times, thus this operation will consume \( O(|E|\log |V|) \) time. The \textit{while} loop as a whole thus has a run time of \( O(|E|\log |V| + |E||PCS|^2) \) given that the graph is connected.

The while loop is executed once in each iteration of the \textit{repeat} loop. How many times is the \textit{repeat} loop executed? It is executed every time dirty nodes have been found in \textit{Cloud}. A node in \textit{Cloud} is marked as \textit{dirty} if and only if its \( PCS \) has been improved because it was first pushed into the cloud. An improvement travels at least one hop in each iteration. Furthermore, an improvement never revisits nodes it has already traversed. In the worst case, then, an improvement will trigger the \textit{repeat} loop \(|V| \) times. Altogether, DBSP therefore has a run time of \( O(|E||V|\log |V| + |E||V||PCS|^2) \).

The priority queue \( Q \) can be implemented using a Fibonacci Heap instead of a binary heap. Each \textit{ExtractMin} operation then takes \( O(\log |V|) \) time and the ‘decrease key’ operation on line 11 takes \( O(1) \) time. The run time of the \textit{while} loop is then bounded by \( O(|V|\log |V| + |E| \cdot |PCS|^2) \). The \textit{while} loop is executed at most \(|V| \) times resulting in a total run time for DBSP of \( O(|V|^2 \log |V| + |E| \cdot |V| \cdot |PCS|^2) \).
7.3.4 DBSP Complexity Reduction

In this section we look at how to improve the run time of DBSP and how those improvements affect the network performance. In the ideal case, we would like to achieve a run time of the same order as Dijkstra’s shortest-path algorithm which is \( O(|V| \log |V| + |E|) \) when implemented with a Fibonacci heap. There are two major factors that contribute to the complexity difference between DBSP and Dijkstra’s algorithm. Firstly, Relax handles a list of shortest-path candidates as opposed to a single value, and thus introduces a quadratic dependency on the PCS size, whereas the original Dijkstra algorithm performs constant-time relaxations. Secondly, a node that already has been pushed into the cloud can still have its shortest path estimates updated and thus trigger a rerun of the relaxation procedure, as opposed to Dijkstra’s algorithm in which a node in the cloud never has its shortest path estimate updated. The iteration over dirty nodes potentially adds \(|V|\) iterations over the while loop on line 5.

In this light, it seems reasonable to try to somehow limit the size of the PCS and to limit the number of iterations over dirty nodes. Hence, we need to understand how these entities behave in different network scenarios. We therefore ran a set of simulations and observed the PCS size, the number of iterations over dirty nodes and the number of dirty nodes in each iteration. We simulated DBSP using both the MinCost and WidestShortest path selection criteria and call them MC-DBSP and WS-DBSP respectively.

We are primarily interested in scenarios that cover network sizes used by Internet Service Providers (ISPs). Therefore, we ran our simulations in three different network sizes: small, medium and large. The small network is shown in Figure 6.1, Chapter 6. It represents a typical ISP network with 18 nodes and 30 bidirectional links, that is 60 unidirectional links. This network has been used for comparable simulations in (Kar et al., 2002; Apostopoulos et al., 1998). The other two networks we used were generated according to the Waxman model (Waxman, 1998) using the BRITE topology generator (Medina et al., 2001). The medium-sized network has 50 nodes and 300 unidirectional links and the large network has 100 nodes and 800
unidirectional links. The link capacity in all networks are 4 Mbps. The propagation delays are uniformly distributed between 5 and 150 microseconds and the link weights $F_{uv}$ are uniformly distributed in $[1, 10]$.

In the simulations, delay-bounded requests arrive one by one according to a Poisson distribution with mean arrival rate $\lambda$. Admitted requests remain in the system for a time span drawn from an exponential distribution with mean $\mu$. We choose to keep $\mu$ constant at one time unit and adjust $\lambda$ so that the rejection rate becomes about 10%, which is an arrival rate of 155, 860 and 2200 requests per time unit for the small, medium and large networks respectively.

The simulation emulates delay bounded connections used for voice applications. Assuming that the voice applications use a G.726 codec with a bit rate of 40 kbps, we have a maximum packet size $M$ of 84 bytes. The other parameters in the TSpec, $r$, $t$ and $b$, are chosen to be respectively 400 kbps, 1000 kbps and $b = 18M = 1512$ bytes, which gives a maximum buffer delay at the ingress node of about 30 milliseconds. The end-to-end delay bound $d_m$, as specified in the RSpec, is 200 milliseconds.

We will first look at how the PCS size, number of dirty iterations and number of dirty nodes in those iterations varies when using MC-DBSP. Figure 7.4 shows how the PCS size is distributed in the three different networks. The results have been normalized with respect to the total number of samples. Two lines are drawn for each network size, one dashed line representing relative PCS sizes and a solid line representing the cumulative function. The PCS size is sampled each time a Relax operation completes. The most interesting point shown in Figure 7.4 is that over 99% of all PCSs have a size smaller than 8 in all three networks. This strongly suggests that one can limit the PCS size to 8 and still maintain high network performance.

How the number of iterations over dirty nodes varies with the network size is shown in Figure 7.5. Again two line styles are used to represent the relative number of iterations, the dashed line, and the cumulative function of the number of iterations, which is solid. As shown, DBSP iterates less than 10 times over dirty nodes for very close to 100% of all requests for the small and medium sized networks. For the large
network 99.9% of all requests required less than 15 iterations. However, this alone is not sufficient data to deduce whether or not it is wise to limit the number of iterations to a constant number and what number to choose. We need to understand how many nodes are dirty in each iteration as well. Figure 7.6 shows how the percentage of dirty nodes varies with the number of iterations in the three networks. Again we have two line styles. The solid line represents the average percentage of dirty nodes as a function of the iteration number. The dashed line expresses 95% confidence. For instance, 4% of the nodes in the large network (= 4 nodes) were dirty on average in the 6th iteration and at most 19% (= 19 nodes) are dirty in 95% of the 6th iterations. For the small network, the 95% confidence curve forms steps, because of the low number of nodes, and the steps represents 4, 3, 2 and 1 dirty nodes. The medium network shows a similar stepwise pattern whereas the large network has a smoother curve.

Now, we want to limit the number of iterations while maintaining high network performance. We see in Figure 7.6 that at most 4% of the nodes are dirty on average after 6 iterations in all three networks. At the same time we can see in Figure 7.5 that DBSP iterates at most six times in 99.9% and 98% of cases respectively in the small and medium network. In the large network, eight iterations only cover about 90% of all requests and about 99% after 15 iterations. Choosing a maximum number of iterations larger than 8 will have very little impact on the run time in small and medium sized networks. In the large network we see that about 99% of all requests require less than 15 iterations over dirty nodes. Thus choosing a maximum of 15 iterations rather than 8 has little impact on the run time because less than 4% of the nodes are dirty on average in each iteration. Hence limiting the maximum number of iterations to 15 seems to be a good tradeoff between accuracy and run time. We will therefore approximate MC-DBSP by limiting the PCS size to eight and choose a maximum of 15 iterations over dirty nodes.

We now turn our attention to WS-DBSP. Figure 7.7 shows both the distribution (dotted line) and the cumulative function of the PCS size in the three networks. It is interesting to note that over 99% of all PCS sizes are smaller than four, which is half
the size of the PCSs when using MC-DBSP. Looking at the number of dirty iterations in Figure 7.8, we see that over 99% of all runs require less than 9 iterations over dirty nodes, even in the large network. Furthermore, very close to 100% of all requests required less than 10 iterations in all three networks. This explains the sudden jump of the average number of nodes in the large network at the 16th iteration in Figure 7.9. Here, only one sample required 16 or more iterations in the simulation and apparently that one sample contained 5% of the nodes in the network (= 5 nodes). As a consequence, the 95% confidence interval coincides with the average at 16 iterations and beyond. Given this data, it seems to be a good idea to limit the PCS size to five and to choose a maximum of 10 iterations over dirty nodes for the WS-DBSP approximation.

In conclusion, we will approximate MC-DBSP by limiting the PCS size to eight and the number of iterations over dirty nodes to 15; we will call this approximation MC-DBSP-A. Similarly we will call the WS-DBSP approximation WS-DBSP-A and limit WS-DBSP-A’s PCS size to five and the number of iterations over dirty nodes to 10. Limiting the PCS size and number of iterations to constant values significantly improves the theoretical run time of DBSP. First of all, the run time of Relax becomes $O(1)$, so the while loop in DBSP now runs in $O(|E| \log |V| + |E|) = O(|E| \log |V|)$ time. Secondly, the repeat loop is run at most $O(1)$ times. As a result, the proposed approximations of DBSP run in $O(|E| \log |V|)$ time. If DBSP is implemented with a Fibonacci heap rather than a binary heap, then the run time of the approximations becomes $O(|V| \log |V| + |E|)$.

### 7.3.5 DBSP Performance

In this section we examine how DBSP and its approximations perform in a networking context. We implement DBSP using both the MinCost (MC-DBSP) and Widest-Shortest (WS-DBSP) path selection criteria and their respective approximations MC-DBSP-A and WS-DBSP-A, as described in Section 7.3.4. To contrast the performance, we compare them to two other algorithms; Generic Dijkstra and the current state-of-the-art algorithm for computing non-resilient QoS connections proposed by Ma and Steenkiste (Q. Ma and P. Steenkiste, 1998) called Iterative Bellman–Ford
Efficient Algorithms for Computing Resilient QoS Connections

\textbf{Figure 7.4} PCS size in a small, medium and large network using the MinCost path selection criterion.

\textbf{Figure 7.5} Dirty iterations in a small, medium and large network using the MinCost path selection criterion.
Figure 7.6 Dirty nodes in each iteration in a small, medium and large network using the MinCost path selection criterion.

Figure 7.7 PCS size in a small, medium and large network using the WidestShortest path selection criterion.
Figure 7.8 Dirty iterations in a small, medium and large network using the WidestShortest path selection criterion.

Figure 7.9 Dirty nodes in each iteration in a small, medium and large network using the WidestShortest path selection criterion.
(IBF).

- **Generic Dijkstra** is an adaption of the generic algorithm given in Algorithm 6.1. We implement $\text{FindPaths()}$ with Dijkstra’s shortest-path algorithm such that it returns only one path. $\text{VerifyPaths}$ simply verifies that the computed path complies to the end-to-end delay constraint using Equation (6.1). $\text{Trim}$ trims the bandwidth using Equation (7.4) so that no more bandwidth than necessary is provisioned. To help the algorithm find a path that is most likely to meet the end-to-end delay constraint we compute the link cost to reflect the link delay. The cost for link $(u, v)$ is computed as:

$$c(u, v) = F_{uv} \cdot \left( \frac{M_{bw}}{bw} + \frac{M_{m}}{C_{uv}} + \text{prop}_{uv} \right)$$

(7.14)

where $M_{m}$ and $\text{prop}_{uv}$ are assigned according to Section 7.3.4 and $bw$ is the bandwidth used in the current iteration. The link weight $F_{uv}$ is for the sake of fairness chosen to be one, because WidestShortest path selection disregards link weights. As the bandwidth $bw$ changes in each iteration the link costs are recomputed each time. We also choose to limit the number of bandwidth adjustments to 20 by selecting $q = 20$.

- **IBF** basically iterates over all residual link capacities in the network and computes a shortest path for each such capacity. Once all paths have been computed, a path is selected according to a path selection criterion. Ma and Steenkiste proposed four different criteria:

  - **MinCost** – this is the path selection criterion used in this chapter.
  
  - **Widest Shortest** – select a feasible path with the minimum hop count. If more than one such path exist then choose the one leaving the most residual capacity. The difference from the WidestShortest path selection criterion used in this chapter is that we choose the path with the most residual capacity after path provisioning rather than before provisioning.
  
  - **Shortest Widest** – select a feasible path with the maximum reservable bandwidth and if several such paths exist, then choose the one with the minimum hop count.
Table 7.2 Arrival Rates (λ).

<table>
<thead>
<tr>
<th>Network</th>
<th>Delays (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td>small</td>
<td>90</td>
</tr>
<tr>
<td>medium</td>
<td>535</td>
</tr>
<tr>
<td>large</td>
<td>1300</td>
</tr>
</tbody>
</table>

- **Shortest Delay** – select a feasible path with the minimum end-to-end delay if the maximum reservable bandwidth is reserved (same as the path residual capacity). If several such paths exist, then choose the one with the minimum hop count.

The authors showed that IBF performs best when using the WidestShortest path selection criterion, thus we implement IBF using this path selection criterion. The run time of IBF is $O(|E|^2|V|)$.

We evaluate the algorithms in the network contexts described in Section 7.3.4. We measure the performance by observing the rejection rate. We measure the rejection rate in the three network scenarios for end-to-end delays of 100, 200 or 300 milliseconds. The arrival rate $\lambda$ is tuned so that WS-DBSP’s rejection rate is about 10% for each end-to-end delay. The tuned arrival rates are given in Table 7.2. Unfortunately we did not have enough computing resources to evaluate the performance of IBF in the large network because of IBF’s long run times. To evaluate the performance under different network loads, we fix the end-to-end delay bound at 200 milliseconds and measure the rejection rate in the medium sized network while choosing different arrival rates, $\lambda \in \{795, 825, 860\}$. These arrival rates result in rejection rates of about 2, 5 and 10% respectively for WS-DBSP.

The results in the small network are shown in Figure 7.10. We see that they are all similar except for Generic Dijkstra, which rejects about 1% more of the requests. It is interesting to note how similarly WS-DBSP and its approximation WS-DBSP-A perform; this is expected because the limited number of iterations over dirty nodes and the size of the PCS cover very close to 100% of the unrestricted WS-DBSP.
Similarly, MC-DBSP and its approximation MC-DBSP-A perform very similarly.

Figure 7.11 shows the rejection rates in the medium sized network. We see that both WS-DBSP and its approximation have a performance edge over the other algorithms including IBF. Here it becomes evident that the WidestShortest path selection criterion, as represented by WS-DBSP(-A) and IBF, achieves higher performance than the MinCost selection criterion represented by MC-DBSP and its approximation. One can also see that the MC-DBSP(-A) and Generic Dijkstra have poor performance at 300 milliseconds end-to-end delay. The reason for this is possibly that they are more greedy and thus do not spread the network load as well as the algorithms representing a WidestShortest path selection criterion.

In the large network, shown in Figure 7.12, we see again that the WidestShortest path selection criterion outperforms both the MinCost selection criterion and Generic Dijkstra. Again DBSP and its approximations perform very similarly.

For the performance under different work loads shown in Figure 7.13, we see a similar pattern. DBSP and its approximations perform very similarly. Again IBF performs as well as or slightly worse than WS-DBSP(-A).

Our results strongly suggests that: (1) the performance of the proposed approximation with WidestShortest path selection criterion performs at least as well as IBF; (2) the proposed caps on the PCS sizes and the number of iterations over dirty nodes result in a negligible network performance degradation. The major strength of WS-DBSP-A and MC-DBSP-A is their attractive run times, $O(|V| \log |V| + |E|)$, which is a major improvement compared to IBF’s run time $O(|E|^2 |V|)$. As a consequence of the run time improvement DBSP-A can be used in large networks.

### 7.4 Dynamic Disjoint Shortest Paths

In this section we will present a Dynamic Disjoint Shortest Paths (DDSP) algorithm that, given a network graph $G = (V, E)$ and a request $Req = (s, d, TSpec, RSpec)$ for a resilient QoS connection, finds two link-disjoint paths, $P_s$ and $P_b$ from $s$ to
Efficient Algorithms for Computing Resilient QoS Connections

Figure 7.10 Rejection rates as a function of end-to-end delay in the small network.

Figure 7.11 Rejection rates as a function of end-to-end delay in the medium network.
Efficient Algorithms for Computing Resilient QoS Connections

Figure 7.12 Rejection rates as a function of end-to-end delay in the large network.

Figure 7.13 Rejection rates as a function of arrival rate in the medium network.
$d$ and their respective required minimum bandwidths, $R_s$ and $R_b$, such that both paths meet the given TSpec and RSpec. Furthermore, a sub-objective of DDSP is to minimize the returned paths’ combined distance, that is $\min (c(P_s) + c(P_b))$ or $\min (p(P_s) + p(P_b))$.

As explained in Section 7.2 we approach this problem using a two-layered algorithm. The top layer will systematically extend two disjoint paths towards $d$. Once $d$ is reached, DDSP has found two link-disjoint paths from $s$ to $d$ and their required minimum bandwidths such that they both meet the given constraints. Each time the two paths are extended, DDSP will call a bottom layer algorithm that computes a single shortest path and its required minimum bandwidth twice. Because the bottom layer algorithm will be heavily used it is essential that it computes the path efficiently both from a network resource perspective and perhaps even more importantly from a run time perspective. The DBSP approximations presented in Section 7.3.2 have both these traits and therefore we will use DBSP as the bottom layer algorithm.

DDSP is shown in Algorithm 7.4. The intuition behind DDSP is given in Section 7.2. DDSP starts by coloring all nodes and links in the given network $G$ grey. Then it initializes the rendezvous point $RV$ to source node $s$ before it copies $G$ to $G_b$. $G_b$ will be used when computing blue paths and to retrieve specific path information when coloring nodes in $G$ blue. Next DDSP computes $P_b$ using DBSP. When $P_b$ has been found, DDSP copies all nodes and links but the first link on $P_b$ to $G_y$, which subsequently is used when computing the yellow path $P_y$. If either of the two paths were not found DDSP returns $false$ because there are not two disjoint paths from $s$ to $d$.

On line 8 a while loop is entered and repeated until the rendezvous point $RV$ is $d$ after which the two paths have been found and their respective required minimum bandwidths are extracted on line 27 and 28. Inside the while loop, DDSP first calculates where the blue and yellow paths meet, $RV'$. Then, on line 10, DDSP colors and reverses all nodes and links on the blue path in $G$. Note that when a node $v$ is colored blue in $G$, the shortest path information $P_b(s \leadsto v)$ is extracted from $G_b$. This is repeated for the yellow path $P_y$. Now, if the new rendezvous point is $d$ there is no need to continue so DDSP breaks the while loop. Otherwise two new shortest
paths must be computed from $RV'$ to $d$. $G_b$ is reassigned to $G$ after the downstream link from $RV'$ on $P_b$ is removed. Then, a new blue path $P'_b$ is computed on line 14. The following two lines compute a new yellow path from $RV'$ to $d$ in the same way.

If DDSP failed to find both the blue and yellow paths, there are not two paths from $s$ to $d$, so DDSP gives up and returns false. If at least one set of two paths was found it computes which of the two pairs of paths $(P_b, P'_y)$ and $(P'_b, P_y)$ has the combined shortest distance from $s$ on line 18. In this case we use the path length function $p$ to measure the distance. If the former pair has lowest cost then DDSP first computes at which node $u_y$ path $P'_y$ diverts from $P_y$. Then it reverses and “uncolors” all yellow nodes and links on $P_y(u_y \leadsto RV)$ in $G$. Next it updates $P_y$ to $P'_y$. If the latter pair has a combined shorter distance to $s$ then DDSP updates the blue entities in the same manner. The last step in the while loop is to reassign the old rendezvous point $RV$ to the new rendezvous point $RV'$.

When the while loop terminates, the two paths with the combined minimum distance from $s$ to $d$ are $P_b$ and $P_y$ and their respective bandwidths can be extracted after retrieving the blue and yellow PCs from node $d$ in $G$.

### 7.4.1 Analysis

In this section, we analyze the run time of DDSP. Because it uses DBSP we will refer to DBSP’s run time as $O(DBSP)$.

The first line in DDSP, as shown in Algorithm 7.4, colors all links and nodes in $G$ grey; this operation takes $O(|V| + |E|)$ time. The following two lines run in constant time. Line 4 and 6 each takes $O(DBSP)$ time. Line 5 takes $O(|E| + |V|)$ time. The if statement runs in constant time. Next, DDSP enters a while loop on line 8. Each iteration of the while loop brings $RV$ at least one hop closer to $d$, thus the while loop will be executed at most $|V|$ times. Inside the while loop on line 9 the new rendezvous point is computed; this can be done in $O(|V|)$ time. One way to do it is that given a copy of the input graph, $G_{grey}$, color all links on $P_b$ blue in the graph and then follow $P_y$ in $G_{grey}$, until we hit a blue link, in which case $RV'$ is the tail node of that link. If $d$ is reached without traversing any blue links then the two paths do not
Input: $G, TSpec, RSpec, s, d$
Output: $P_b, R_b, P_y, R_y$

1. color all links and nodes in $G$ grey;
2. $RV \leftarrow s$;
3. $G_b \leftarrow G$;
4. $P_b \leftarrow \text{DBSP}(G_b, s, d, TSpec, Rspec, \text{blue})$;
5. $G_y \leftarrow G \setminus \{\text{first link on } P_b\}$;
6. $P_y \leftarrow \text{DBSP}(G_y, s, d, TSpec, Rspec, \text{yellow})$;
7. if $P_b = \text{nil}$ or $P_y = \text{nil}$ then return false;
8. while $RV \neq d$ do
   9. $RV' \leftarrow P_b$ joins $P_y$;
   10. reverse and color grey links and nodes blue on $P_b(s \leadsto RV')$ in $G$;
   11. reverse and color grey links and nodes yellow on $P_y(s \leadsto RV')$ in $G$;
   12. if $RV' = d$ then break;
   13. $G_b \leftarrow G \setminus \{(RV', u) \in P_b\}$;
   14. $P'_b \leftarrow \text{DBSP}(G_b, RV', d, TSpec, Rspec, \text{blue})$;
   15. $G_y \leftarrow G \setminus \{(RV', u) \in P_y\}$;
   16. $P'_y \leftarrow \text{DBSP}(G_y, RV', d, TSpec, Rspec, \text{yellow})$;
   17. if $P'_b = \text{nil}$ and $P'_y = \text{nil}$ then return false;
   18. if $p(P_b) + p(P'_y) \leq p(P'_b) + p(P_y)$ then
      19. $u \leftarrow \text{node where } P'_y \text{ diverts from } P_y$;
      20. reverse and erase yellow from links and nodes on $P_y(u \leadsto RV')$ in $G$;
      21. $P_y \leftarrow P'_y$;
   22. else
      23. $u \leftarrow \text{node where } P'_b \text{ diverts from } P_b$;
      24. reverse and erase blue from links and nodes on $P_b(u \leadsto RV')$ in $G$;
      25. $P_b \leftarrow P'_b$;
      26. $RV \leftarrow RV'$;
   27. $R_b \leftarrow R(\text{get blue PC from } d \in G)$;
   28. $R_y \leftarrow R(\text{get yellow PC from } d \in G)$;
   29. return true;

Algorithm 7.4: DDSP
share any links and RV’ is $d$. Once done, color all blue links grey again so that $G_{grey}$ can be reused in future iterations.

Then on line 10, reversing all grey links on $P_b$ takes $O(|V| \log |V|)$ time and coloring all links and nodes on $P_b$ blue takes $O(|V|)$ time. The same run time applies to coloring and reversing nodes and links on line 11. Then follows a constant-time comparison on line 12, followed by an assignment of $G_b$ that takes $O(|E| + |V|)$ time and line 14 that takes $O(DBSP)$ time. These run times are repeated on line 15 and 16. Line 17 takes constant time. The cost calculations and comparisons on the following line also take constant time. Then, finding the diversion node on line 19 takes $O(|V|)$ time. Reversing and coloring nodes and links on the next line takes $O(|V| \log |V|)$ time. Line 21 takes constant time. If lines 19−−21 are not computed, lines 23−−25 will be and vice versa, either way gives the same run time. The last statement in the while loop takes constant time. Thus the while loop takes a total of $O(|E| + |V| \log |V| + DBSP)$ time. The while loop is executed $O(|V|)$ times and thus the DDSP’s run time is $O(|E| \cdot |V| + |V|^2 \log |V| + |V| \cdot DBSP)$.

Using the original DBSP algorithm, DDSP’s run time becomes $O(|E| |V| + |V|^2 \log |V| + |V| (|V|^2 \log |V| + |E||V||PCS|^2)) = O(|V|^3 \log |V| + |E||V|^2 |PCS|^2)$. However, using the suggested approximation algorithms, we have a total run time for DDSP of $O(|E| |V| + |V|^2 \log |V| + |V| (|V| \log |V| + |E|)) = O(|E| |V| + |V|^2 \log |V|)$.

### 7.4.2 DDSP Performance

In this section we evaluate the performance of DDSP. We implement DDSP using both MC-DBSP-A and WS-DBSP-A and refer to the two implementations as WS-DDSP and MC-DDSP respectively. To benchmark DDSP we use three other algorithms, our LP formulation presented in Section 6.4.1, Generic Suurballe and Generic 2Dijkstra, which are adaptations of the generic algorithm given in Algorithm 6.1.

- **LP formulation** – is a linear programming formulation that computes a pair of disjoint paths and their required minimum bandwidths such that they have the combined lowest cost. This objective is similar to the one implemented by MC-DDSP. However, MC-DDSP will pick the path that has the most residual
capacity after provisioning if several pair of paths exist with the same cost.

- **Generic Suurballe** – is an adaptation of the generic algorithm given in Algorithm 6.1. It implements *FindPaths* using Suurballe’s algorithm, which computes a pair of disjoint paths that have the lowest combined cost (Suurballe and Tarjan, 1984). *VerifyPaths* simply verifies that the two computed paths comply with the end-to-end delay constraint using Equation (6.1). *Trim* trims the bandwidths using Equation (7.4) so that no more bandwidth than necessary is provisioned. To help the algorithm find paths that are the most likely to meet the end-to-end delay constraint we compute the link cost in the same way as for Generic Dijkstra in Section 7.3.5. We choose to limit the number of bandwidth adjustments to 20 by selecting $q = 20$.

- **Generic 2Dijkstra** – is also an adaptation of the generic algorithm given in Algorithm 6.1. *FindPaths* is implemented using Dijkstra’s shortest-path algorithm to first compute one shortest path, then remove all links on the found path from the network before computing a second shortest path. *VerifyDelay* and *Trim* are implemented in the same way as in Generic Suurballe. The link costs are computed in the same way as in Generic Suurballe and the maximum number of bandwidth adjustments is set at 20 ($q = 20$).

We evaluate the algorithms in the network contexts presented in Section 7.3.4. We use the rejection rate as the network performance measure. We measure the rejection rate in the three network scenarios while choosing the end-to-end delay to be either 100, 200 or 300 milliseconds. The arrival rate $\lambda$ is tuned so that WS-DDSP’s rejection rate is about 10% for each end-to-end delay. The tuned arrival rates are given in Table 7.3. Unfortunately we did not have enough computing resources to evaluate the performance of the LP formulation in either the medium sized nor the large network because of its extensive run time requirement. To evaluate the performance under different network loads, we measure the rejection rate in the medium sized network for different arrival rates, $\lambda \in \{325, 350, 420\}$. These arrival rates result in rejection rates of about 3, 5 and 10% respectively for WS-DDSP.
Looking at the rejection ratios in the small network shown in Figure 7.14, we see that WS-DDSP performs best, closely followed by MC-DDSP. We can also see that Generic Suurballe performs slightly better than Generic 2Dijkstra. It is interesting to note that MC-DDSP performs at least as well as the LP formulation. The reason MC-DDSP has a performance edge on the LP formulation is possibly because it picks paths that have more residual capacity if several paths with the same cost exist.

Turning to the medium sized network in Figure 7.15, we see a similar pattern, where WS-DDSP performs better or comparably to MC-DDSP, whereas both generic algorithms reject several per cent more requests. This difference worsens in the large network shown in Figure 7.16. Again, WS-DDSP performs best followed by MC-DDSP in the large network.

In Figure 7.17 the rejection rate is plotted as a function of the arrival rate $\lambda$. Here too, we note that WS-DDSP performs best, closely followed by MC-DDSP. The two generic algorithms perform similarly to each other, rejecting about 2% more requests than DDSP for all sampled arrival rates, which indicates that the arrival rate does not affect the comparative performance between the algorithms.

## 7.5 Conclusion

In this chapter we looked at the problem of finding resilient QoS connections in a network consisting of latency rate servers. An algorithm addressing this problem should not only achieve high network utilization but have a low run time as well so that it can be used in a production environment.
Efficient Algorithms for Computing Resilient QoS Connections

Figure 7.14 Rejection rates as a function of end-to-end delay in the small network.

Figure 7.15 Rejection rates as a function of end-to-end delay in the medium network.
Figure 7.16 Rejection rates as a function of end-to-end delay in the large network.

Figure 7.17 Rejection rates as a function of arrival rate in the medium network.
To address this problem we designed a two-layered algorithm. The bottom-layer algorithm called DBSP is used to find a non-resilient QoS connection by computing a single shortest path subject to either minimum cost or number of hops, and its required minimum bandwidth such that the path meets the given QoS constraints. The top-layer algorithm called DDSP addresses the original problem of finding a resilient QoS connection through systematic use of three colors and DBSP to compute two link-disjoint paths and their respective bandwidths such that both paths meet the given QoS constraints. Moreover, the returned paths are subject to either minimum cost or number of hops.

One drawback of DBSP is its high theoretical run time. After observing DBSP’s run time complexities, we proposed an approximation DBSP-A for each optimality criterion that has a $O(|V| \log |V| + |E|)$ run time bound, which is comparable to Dijkstra’s shortest-path algorithm. Our extensive simulations show that DBSP-A performs very similarly to DBSP and at least as well as the current state-of-the-art algorithm that has a $O(|E|^2 |V|)$ run time. Our simulations also show that a minimum hop count optimality criterion, that is WidestShortest, performs better than minimum cost, which is in accordance to the findings made by Ma and Steenkiste (Q. Ma and P. Steenkiste, 1998). DBSP and its approximations can be used in a more general context to compute non-resilient QoS connections without modification.

The run time of DDSP is $O(|V|^2 \log |V| + |E| |V|)$ and our simulations show that DDSP performs at least as well as other approaches, including its LP formulation counterpart presented in Section 6.4.1. Not surprisingly, DDSP achieves higher network performance when using the minimum hop count optimality criterion than when using the minimum cost optimality criterion.

In this chapter we used the provided peak rate as the upper bound on the bandwidth adjustments but a tighter upper bound can be chosen to possibly improve further on the network performance. The low run times of DBSP-A and DDSP combined with their high network performances make them suitable for a production environment even for networks as large as 100 nodes and 800 links.
Chapter 8

Conclusion

It is increasingly important to meet new service and application demands for resilient QoS connections. In this context we first pointed out a number of issues that have not yet been satisfactorily addressed in the current literature. Then we addressed those issues starting with a quantitative study that confirms the intuition that both VPN customers and network providers benefit when resilience is implemented by the network provider in the physical network than when implemented by its customers in their VPNs. We then provided a MPLS based programmable VPN architecture that delivers resilient QoS connections with customizable levels of resilience and QoS. Moreover, the proposed architecture delivers virtual sites that can be used by VPN customers to implement functionality “in the cloud”. To meet network providers’ traffic engineering goals, it is important to provide resilient QoS connections effectively. Here, effective refers to both high network performance and low computational complexity.

We further proposed a new framework for computing resilient QoS connections in a 1+1 and 1:1 end-to-end recovery context. In our framework bandwidth and end-to-end delay constraints are considered simultaneously as opposed to reducing the problem to one with only a single constraint in a pre-processing step, which is the approach taken in the current literature. Given our framework, we first constructed a new generic algorithm that can be used to decompose the problem into sub-problems where known algorithms can be applied. Then we proposed two new LP formulations. These formulations only meet the first aspect of “effective” as they achieve
high network performance but suffer from high computational complexity, which makes them unsuitable for an operational environment. To improve on the run time we proposed two very efficient algorithms that achieve at least as high network performance as their LP formulation counterpart but with appealing run times that make them suitable for an operational environment such as in our proposed VPN architecture.

8.1 Resilience Implementation Scenarios

Though it is intuitive that resilience can be more efficiently implemented by the network provider in the physical network than by its customers in their VPNs no quantitative study has been presented confirming this intuition. Thus, we conducted such a study. Our approach was to measure the network provider’s revenue in two different resilience implementation scenarios. In one scenario, the customers implement resilience in their VPNs through purchasing a pair of disjoint QoS connections from the network provider for each VPN connection. In the other scenario resilience is implemented by the network provider and the customers purchase one resilient QoS connection for each VPN connection. Our results show that the network provider can potentially double its revenue when implementing resilience given enough requests for resilient VPN connections. The difference in price between disjoint QoS connections and regular QoS connections is essential for the interpretation of our results. Our measurements show that the price of a disjoint QoS connection must be up to 26% higher than the price of a regular QoS connection to sustain the same revenue. The results also show that the price of one resilient QoS connection can be set below half the price of a pair of disjoint QoS connections with sustained revenue. The customers benefit as well when using an overlay network built with resilient QoS connections since it is less complex and less costly to set up and maintain than a VPN built with redundant resources, resilience and failure discovery mechanisms. This study confirms the intuition that it can be more efficient to implement resilience in the physical network than in the overlay networks and that both the network provider and its customers benefit from this implementation scenario.
8.2 Resilience-Differentiated Programmable VPN Architecture

We discussed issues that relate to resilience-differentiation in programmable VPN architectures. As a case study we propose a new MPLS based programmable VPN architecture that delivers both resilient QoS connections and virtual sites. Virtual sites allow customers to implement functionality “in the cloud”. The most apparent applications for virtual sites customized routing and re-coding/multicasting multimedia streams. The proposed architecture introduces a new concept called recovery engines, which are responsible for computing resilient QoS connections, initiate LSP signaling and monitoring as well as react to service path failures. The architecture can serve as a platform to provide a wide range of new and old VPN services such as VPWS, VPLS and various types of L3 VPNs. It is scalable in terms of number of supported VPNs and allows a smooth integration into an existing MPLS network through a well defined migration path and use of legacy LSRs.

8.3 Resilient QoS Connection Computation

Well defined mechanisms have been proposed to establish and monitor LSPs. One hot topic in the literature is how to compute resilient QoS connections. An algorithm that computes resilient QoS connections should achieve high network performance as well as having low computational complexity so that it can be used in a production environment.

Today two different approaches are used to compute resilient QoS connections in the presence of both bandwidth and end-to-end delay constraints. Both approaches reduce the problem to one with either a bandwidth or an end-to-end constraint in a pre-processing step. We argue that a more efficient approach is to consider both constraints simultaneously. This approach has been proven successful in the context of computing non-resilient QoS connections. Using this approach, a solution does not only contain two disjoint paths but their required minimum bandwidths as well such that the given constraints are met. Using this framework we first propose a generic
algorithm that decompose the original problem into sub-problems that can be solved efficiently using existing algorithms. Next we develop two new LP formulations that can be applied in a 1+1 and 1:1 end-to-end recovery context. The proposed LP formulations compute two link-disjoint paths from the ingress to the egress and their respective bandwidths such that they both satisfy the given bandwidth and end-to-end delay constraints. Furthermore, the returned paths are optimal in the sense that they combined induce minimum cost.

Our simulations show that the LP formulations achieve higher network performance than decomposed approaches. The tighter end-to-end delay bound, the higher network performance gain. In a network with 18 nodes and 60 unidirectional links the run time per request is up to 3 minutes in the 1+1 recovery context and up to 40 hours in the 1:1 routing context where a backup resource sharing strategy is applied. The average run time of our proposed LP formulations are 0.5 seconds in the 1+1 recovery context and 2.5 minutes in the 1:1 recovery context. These run times are well beyond acceptable for an operational environment. However, the presented LP formulations can be used to benchmark approximation algorithms.

To improve on the run time, we proposed two new algorithms, MC-DDSP and WS-DDSP, that operates in our new framework. Here, we used two different optimization criterion. One that minimizes the combined cost, which is a similar optimization criteria to the one used by the LP formulations. However, if two pair of paths exists with the same cost the one with the most residual capacity is chosen rather than selecting a one at random. The other optimality criteria we use is to select the pair of paths that have the combined minimum number of hops and if two such pairs exist, the one with the most residual capacity is chosen. This optimality criteria has been shown to outperforms the minimum cost criteria in the non-resilient QoS connection routing context (Ma and Steenkiste, 1997; Q. Ma and P. Steenkiste, 1998). We observed similar performance differences between the two optimality criterion in the resilient QoS connection context. DDSP does not implement any backup resource sharing and thus DDSP can be used for both the 1+1 and 1:1 recovery types. DDSP systematically use an algorithm that computes non-resilient QoS connections.

The current state-of-the-art algorithm that computes non-resilient QoS connections
is called IBF and has a $O(|E|^2 |V|)$ run time, which unfortunately is too expensive for our purposes. Thus, we propose two new algorithms, MC-DBSP and WS-DBSP, and their respective approximations MC-DBSP-A and WS-DBSP-A that compute a single path and its required minimum bandwidth such that the path meets the given bandwidth and end-to-end delay constraints. The computed path is subject to the same optimality criteria as DDSP. The run time of DBSP-A is comparable to Dijkstra’s shortest-path algorithm: $O(|V| \log |V| + |E|)$, which is a substantial improvement on IBF’s run time. Our simulations show that DBSP-A achieves at least as high network performance as IBF.

The run time of DDSP is $O(|V|^2 \log |V| + |E| |V|)$ when implemented with DBSP-A. We show through extensive simulations that MC-DDSP and WS-DDSP perform better than both decompositions and their proposed LP formulation counterpart. The run times of both DDSP and DBSP coupled with their high network performance make them attractive for an operational environment and thus suitable candidates for computing resilient QoS connections in recovery engines residing in the proposed programmable VPN architecture.

### 8.4 Future Work

As new and existing services’ demand for high levels of resilience and QoS continues to grow, network providers are continuously challenged to efficiently implement flexible support for differentiated resilience and QoS. The challenge is extended to the research community to develop new network architectures, methods and algorithms that are flexible enough to allow new and existing networked services to evolve. Work presented in this thesis provide a clear path for several future research directions.

#### 8.4.1 Resilience Integration in VPN Provisioning Models

In Chapter 3 we outlined a number of ongoing research issues that relate to resilience integration in the sink-tree, hierarchical and sink-tree provisioning models. In this thesis we addressed the first two provisioning models in an end-to-end recovery scope.
leaving segmented and local recovery scopes as future work. Resilience integration in the sink-tree provisioning model is also an interesting research area that needs to be addressed in future work. What sets the sink-tree provisioning model apart from the other two models is that paths originating from different sites are merged in the core network, which raise issues related to choosing buffer sizes and provisioned bandwidth along the merged paths.

Future work also need to address issues related to starvation in the recently proposed customized VPN tree provisioning model as described in Section 3.6.1. Dynamic resizing, where link-loads are monitored and resources like bandwidth allocation dynamically changes throughout the lifetime of a virtual connection also needs further investigation, especially in the context of resilient VPN services.

8.4.2 Programmable Virtual Private Networks

Programmable VPNs allow customers to implement services that require customized routing and content adaptation “in the cloud”. This idea is an extension of the programmable network concept. In this thesis we presented a programmable VPN architecture that enables network providers to efficiently provide customers with virtual private label switched networks. In essence, a customer has the full power of a label switched network at its dispense. What types of services that will be developed on this platform in the future is impossible to predict. Some apparent applications are creating more resource efficient VPN topologies, multicasting streaming media and multimedia conferences whereby multimedia streams are merged in more optimal geographic locations to increase the perceived service quality.

However, services that require virtual sites will not be developed if there is no platform to run them on and conversely, network providers will not offer virtual sites before any such services have been developed. Thus, future work aim to implement the presented architecture in the Smart Internet Technology CRC’s (SmartInternet, ) testbed to enable development of services that require resilience, QoS, customized routing and or content-adaptation, such as the Smart Internet Technology CRC’s Virtual Café service.
Other future directions in this area are to further refine the presented architecture and to construct new architectures that deliver similar functionality.

### 8.4.3 Computing Resilient QoS Connections

In this thesis we proposed a new framework for computing resilient QoS connections. We developed a number of algorithms that operate in the proposed framework that show very promising results both in terms of network performance and computational complexity. However, when backup resource sharing is considered algorithms with low computational complexity still need to be developed. We focused our efforts on algorithms that compute two links disjoint paths in an end-to-end recovery scope. Thus, future work is to extend the methods described in our framework to segmented and local recovery scopes as well.

### 8.4.4 Probabilistic Recovery Analysis

In this thesis we used a simple single link failure model that can be generalized to single node failures as well. The assumption that there is only one failure in the network at any one time is a serious simplification of real life networks. Future work should investigate what probability a single-link failure model has to recover traffic in realistic network environments. Such investigations should be extended to find the probability of successful failure recovery when two, three, four etc backup paths are protecting a service path. Further extensions of these investigations are to develop new algorithms that compute multiple backup paths for each service path.
Bibliography


Appendix A

If-Then Conversion

A logical expression like $if \ f(x_1, x_2, ..., x_n) > 0 \ then \ g(y_1, y_2, ..., y_m) \geq 0$ can be transformed into a linear expression through introducing a variable $t \in \{0, 1\}$ and a constant $M$ chosen so that $M > f$ for all possible values of $f$ and $M > g$ for all possible values of $g$. Now the logical $if - then$ expression can be transformed into:

$$-g \leq Mt \quad \text{(A-I)}$$
$$f \leq M(1 - t) \quad \text{(A-II)}$$

Furthermore, an expression $if - then - else$ can be divided into two $if - then$ expressions. Consider the following, $if \ f > 0 \ then \ g \geq 0 \ else \ h \geq 0$ can be divided into $if \ f > 0 \ then \ g \geq 0$ and $if \ f \leq 0 \ then \ h \geq 0$. Now if $f$ is an integer function then $f \leq 0$ can be written as $1 - f > 0$ after which the described transformation can be directly applied.