AN ARCHITECTURE FOR NETWORK PATH SELECTION

A Thesis
Presented to
The Academic Faculty

by

Murtaza Motiwala

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2012

AN ARCHITECTURE FOR NETWORK PATH SELECTION

Approved by:

Dr. Nick Feamster, Advisor
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Mostafa Ammar
College of Computing
Georgia Institute of Technology

Dr. Dave Andersen
Computer Science Department
Carnegie Mellon University

Dr. Ellen Zegura
College of Computing
Georgia Institute of Technology

Date Approved: 19 December 2011

To my parents Zarina and Mohsin,

and to my brother Mustafa,

for their belief and encouragement.

ACKNOWLEDGEMENTS

Research is rarely done alone, and this thesis is a product of interaction with a number of people I have had the fortune to be part of my life. They have all had direct or indirect contributions to this final product and I wish to use this opportunity to acknowledge them.

My advisor, Prof. Nick Feamster has had the most influence on my research work and there would be no exaggeration in saying that I would not have had much success in grad school if it was not for Nick's constant guidance and presence in my career. Nick's motivation to push the boundaries and challenge conventional thinking have been instrumental in driving my research. His constant availability was something that really helped me in my initial years and I am extremely grateful to Nick for keeping his faith even during unproductive periods.

I would also like to thank the thesis committee. The insightful comments and feedback helped me to improve the thesis. I was extremely lucky to get to work with Prof. Santosh Vempala during my PhD. His amazing enthusiasm and encouragement were extremely useful during the initial years of my PhD. Anukool Lakhina hosted me at his startup in India, and helped me develop the cost model framework as part of my thesis, for which I am grateful. I have also had the opportunity to learn and seek advice from Prof. Jennifer Rexford, which has been very beneficial. I would also like to thank Vijay Gill and Ed Crabbe who hosted me at Google for my internship and gave me the opportunity to see how networking is deployed in the real-world.

Over the years, I have had the fortune to meet and become friends with a number of extremely bright and hard-working graduate students. My colleagues in the Networking Lab have been my family away from home. Among them Amogh, Partha, Ahmed, Srikanth, Sam, Hyojoon, Robert, Illias, Saiedeh, Samantha have tolerated me over the years and I

am thankful for their company. Valas, apart from being my labmate, my housemate, has also been a very close friend. I have relied on his support to help me through the ups and downs in graduate school. Also friends outside the lab, Binh, Rohan, Qingyang, Vandana who have made me feel at home and helped me escape when networking research got a bit too much. Also, special mention to Mukarram, who has been a close friend and mentor throughout my PhD years. He has always been available when I needed advice.

I would like to acknowledge Mohammed, my friend since school, and I am very grateful that he followed me to US. He has been a pillar of support and encouragement. Last but not the least, I would like to acknowledge and dedicate this thesis to my family. My parents, Zarina and Mohsin, who have cherished every step of my academic life. My brother, Mustafa, for always being proud of my achievements. My entire extended family who have always believed in my abilities. I have been truly fortunate to have such a loving and supporting family.

BIBLIOGRAPHIC NOTES

Chapters 2 contains material from our paper [57]. Chapter 3 contains material from our papers [62, 60, 61]. Chapter 4 contains material from the implementation and evaluation sections of our paper [57]. Chapter 5 contains material from our paper [59] and additional material on a cost-based path selection architecture. The implementations described in the dissertation are publicly available at [3].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Traditional routing protocols select paths based on static link weights and converge to new paths only when there is an outright reachability failure (such as a link or router failure). This design allows routing scale to hundreds of thousands of nodes, but it comes at the cost of functionality: routing provides only simple, single path connectivity. Networked applications in the wide-area, enterprise, and data center can all benefit from network protocols that allow traffic to be sent over multiple routes en route to a destination. This ability, also called *multipath routing*, has other significant benefits over single-path routing, such as more efficiently using network resources and recovering more quickly from network disruptions.

This dissertation explores the design of an architecture for path selection in the network and proposes a "narrow waist" interface for networks to expose choice in routing traffic to end systems. Because most networks are also business entities, and are sensitive to the cost of routing traffic in their network, this dissertation also develops a framework for exposing paths based on their cost. For this purpose, this dissertation develops a cost model for routing traffic in a network. In particular, this dissertation presents the following contributions:

- **Design of path bits, a "narrow waist" for multipath routing.** Our work ties a large number of multipath routing proposals by creating an interface (*path bits*) for decoupling the multipath routing protocols implemented by the network and end systems (or other network elements) making a choice for path selection. Path bits permit simple, scalable, and efficient implementations of multipath routing protocols in the network that still provide enough expressiveness for end systems to select alternate paths. We demonstrate that our interface is flexible and leads to efficient network

implementations by building prototype implementations on different hardware and software platforms.

- **Design of path splicing, a multipath routing scheme.** We develop, path splicing, a multipath routing technique, which uses random perturbations from the shortest path to create exponentially large number of paths with only a linear increase in state in a network. We also develop a simple interface to enable end systems to make path selection decisions. We present various deployment paths for implementing path splicing in both intradomain and interdomain routing on the Internet.

- **Design of low cost path-selection framework for a network.** Network operators and end systems can have conflicting goals, where the network operators are concerned with saving cost and reducing traffic uncertainty; and end systems favor better performing paths. Exposing choice of routing in the network can thus, create a tension between the network operators and the end systems. We propose a path-selection framework where end systems make path selection decisions based on path performance and networks expose paths to end systems based on their cost to the network. This thesis presents a cost model for routing traffic in a network to enable network operators to reason about "what-if" scenarios and routing traffic on their network.

CHAPTER I

INTRODUCTION

The Internet is composed of many Autonomous Systems (ASes) and the interconnections among the ASes makes the interdomain topology. Border Gateway Protocol (BGP) [75] is the interdomain routing protocol used to construct interdomain paths. Each AS has forwarding devices or routers that route traffic. ASes typically use an intradomain routing protocol like OSPF [63] or IS-IS [65] to construct intradomain routes.

Routing protocols compute paths to provide reachability. Traditional routing protocols (both intradomain and interdomain) have been carefully designed to compute paths without creating forwarding loops; routing protocols are designed to create a shortest path forwarding tree for every destination. This simplicity has made routing scale to hundreds of thousands of routers, but it also comes at a cost: there is no mechanism for selecting alternate paths in the network, and the routing protocols only trigger the selection of a new path when an outright failure of a link or network device occurs.

Applications that run at the edges of the network are agnostic about the routing protocols that compute end-to-end paths. This decoupling has enabled both the rapid evolution of diverse applications at the edge [89] and of different network technologies for building complex networks. Unfortunately, this decoupling also has a cost: routing protocols lack knowledge of the end-to-end performance of each path. Also, applications cannot *request* an alternate path in case the current path does not fit its requirements.

Applications have different notions of availability that depend on a number of metrics such as available bandwidth, packet loss, jitter, and latency. A network path that works for a particular application could be completely unusable for some other application depending on their notions of availability. This distinction has become even more stark with the

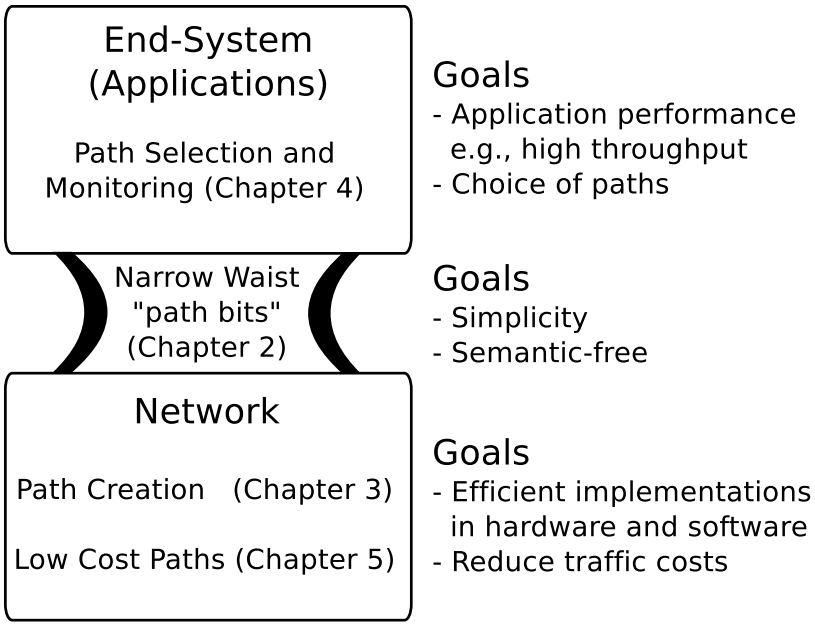


Figure 1.1: Outline of dissertation: Interaction among the different components of the path selection architecture.

proliferation of real-time applications like voice over IP, online multiplayer games (about 6% of the total Internet traffic in US in 2010 is real-time [47]). Because current routing protocols have no knowledge of application performance the paths that they compute may not be the *best* path for all applications using it.

Several studies have demonstrated that the default path on the Internet is often sub-optimal [66]. Unfortunately, a link with a high loss rate (*e.g.,* 5%) may not trigger path reconvergence and traffic from a large number of source-destination pairs could potentially be sharing that link. Many real-time applications or even bulk-transfer applications would find the path completely or partially unusable. This path could still be usable by certain other applications (*e.g.,* e-mail or web traffic). Thus, giving applications more control over the path their traffic takes in the network is beneficial [9, 85, 41].

Unfortunately, providing applications (or end systems) the ability to select paths in the network may cause problems, since end systems and network operators sometimes have conflicting goals. Applications are concerned with using the *best* available network path for carrying their traffic, whereas network operators are concerned with reducing the cost of

carrying traffic in the network [37]. Providing applications with control over path selection can conflict with the network operator's goals of routing traffic on paths which are low cost. Thus, there is a tension between providing applications ability to select paths and the need for the network operators to control their traffic costs that needs to be resolved [24].

This dissertation develops a network path selection architecture, which uses a semantic-free interface, as shown in Figure 1.1, with the following division of labor.

- **Networks provide a choice of paths**

  Routing protocols must create and maintain a large number of diverse paths in the network. There are many proposals for multipath routing protocols [99, 96, 52], and some like ECMP (Equal Cost Multi-Path) [44] are widely deployed. Unfortunately, a number of such proposals suffer from not being able to provide a large choice of paths for a small increase in resulting state in the routers. We develop *path splicing* [62, 60], a multipath routing primitive that uses random perturbations from the default shortest path to create an exponential number of paths in the network. The alternate paths have several desirable properties *e.g.*, their path lengths are comparable to the length of the shortest path (*small stretch*) and they have high path diversity to be able to recover from a large combination of failures. We show how to construct these paths in a network by perturbing existing link weights that are assigned to each link in the network, and running existing shortest path routing protocols to compute the alternate paths in the network. We also extend path splicing to the interdomain setting with minimal changes to BGP, and without exchanging any additional BGP messages [61]. We modify the BGP route selection procedure to select multiple policy-compliant routes from the routes already received from neighboring networks.

- **Applications make path selection decisions**

  Applications can best make a decision regarding the performance they observe on a

particular network path; enabling applications to select paths can greatly benefit application performance. We propose *path bits* as a "narrow waist" for implementing multipath routing in the Internet [57]. The *path-bits* interface is simple and scalable, and imposes no semantics on the underlying network to interpret them while selecting a network path. We demonstrate by implementing a variety of multipath routing protocols, on both hardware and software platforms, to show the benefits of such a narrow waist architecture. *Path bits* achieve decoupling by letting routing protocols do what they can do best: construct a large number of diverse paths in the network and leave the task of making path selection decisions to applications (or end-systems) that best understand their own availability requirements. The lack of semantics implies that applications (or end-systems) need a way to discover paths. We implement extensions to the end-system to support path bits and then build a number of monitoring applications that can benefit different types of applications.

To resolve the tension between application requirements for selecting paths and the network operators need to control the cost of routing traffic in their network, this dissertation develops *cost-based path selection* by the network to restrict the choice of available paths to the end-systems based on the cost of routing traffic on the particular path.

- **Cost-based path selection** Routing traffic has associated costs. Reasoning about the cost of routing traffic requires a cost model. Towards this goal, we develop a cost model that an operator could use to ensure that the network only exposes low-cost alternate paths to end systems. The cost model is also useful in justifying a number of "what-if" decisions. Our cost model is generic to be applicable to a variety of networks like access networks, transit providers. The proposed path selection system would act as a black box that takes the cost model as input along with other information like routing, network topology, traffic matrix and outputs a set of paths that the network can expose to end-systems.

1.1 Thesis Contributions

This dissertation makes the following contributions:

- We identify the need for a standard interface for path selection in the Internet and describes the design of one such architecture (*path bits*), which is semantic-free.
- We present *path splicing* as a multipath routing scheme that provides exponential path diversity with only a linear increase in state. We also show how path splicing's path selection mechanism can be mapped easily to the path-bits interface.
- We implement a number of multipath routing schemes, in both hardware and software. We make our implementations available for researchers to experiment with implementing their own custom path creation mechanisms using the path-bits interface.
- We develop a holistic traffic cost model for a network, to allow operators to attribute costs to traffic in their network. This model enables a cost-based path selection mechanism that operators can use to expose paths to end-systems by also taking into consideration the cost of sending traffic on the different paths.

1.2 Roadmap

The rest of the thesis is organized as follows. The relations between the different chapters in the thesis are as shown in Figure 1.1.

- Chapter 2 presents the "narrow-waist" architecture for enabling path selection in the Internet. We focus there on the motivation and high-level design requirements from such an interface.
- Chapter 3 presents *path splicing*, a multipath routing primitive that enables routing protocols to create large number of alternate paths in the network.
- Chapter 4, presents the implementation of different path selection mechanisms that use the "narrow-waist" in both hardware and software. We also describe extensions

required to end-systems to efficiently utilize the paths exposed by the network to benefit large number of applications.

- Chapter 5 presents a holistic traffic cost model for network to attribute cost to traffic flows and then show how to augment the path creation mechanism in the network to respect traffic costs.
- Chapter 6 presents concluding remarks from this dissertation and lists few directions for future research.

How to read this thesis? The best way to read the thesis would be to read the chapters in the serial order to grasp all the ideas that we develop as part of the thesis. Another way to read the thesis would be to read Chapter 2 and then jump to Chapter 4 to read about the design and implementation of the path selection architecture described in Chapter 2. The reader can then choose to either read Chapter 3 or Chapter 5 in any order, followed by Chapter 6. Figure 1.2 illustrates the alternate reading suggestion for this dissertation.

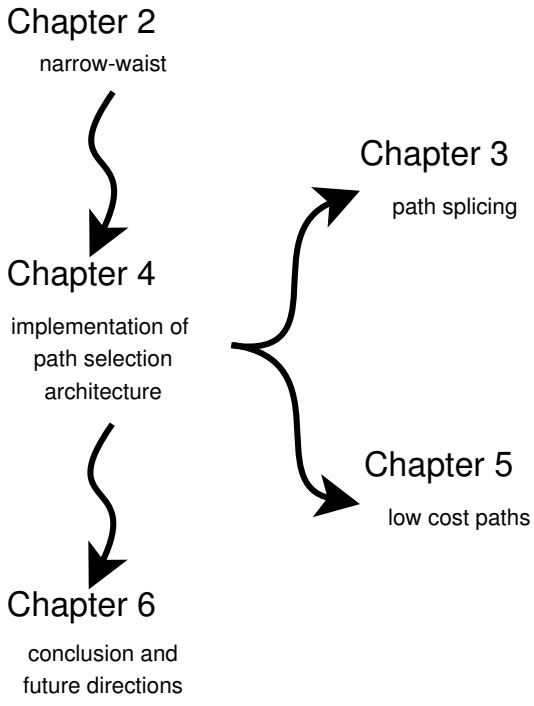


Figure 1.2: Alternate suggestion for reading this thesis

CHAPTER II

DESIGN OF AN INTERFACE FOR NETWORK PATH SELECTION

2.1 Introduction

Networked applications in the wide-area, enterprise, and data center can all benefit from network protocols that allow traffic to be sent over multiple paths en route to a destination. This mechanism, called *multipath routing*, can reduce latency, improve throughput, or improve robustness to network failures. Applications that can benefit from multipath routing range from real-time applications such as network voice over IP and video to bulk-transfer applications; notably, each of these obtains a different benefit from multipath routing *e.g.*, voice over IP can pick a path which has lower latency and jitter to improve the voice quality and a bulk-transfer application can simultaneously utilize multiple paths to achieve higher throughput.

Providing these desired benefits are numerous *mechanisms* for multipath routing, each of which may be more or less beneficial to different classes of applications [60, 96, 99, 36]. Each multipath mechanism has typically come with its own, unique way of allowing applications to specify a path to use. The unfortunate consequence of this is that there is neither a standard multipath interface, nor a set of applications ready to make use of any multipath mechanisms that could become available. The lack of a flexible, widely-applicable interface inhibits adoption of multipath mechanisms (there exist no applications that can use them) and imposes high barriers for researchers attempting to compare different approaches.

The premise of this part of the thesis is that many multipath implementations can be adapted to use a common application interface: a *narrow waist for multipath routing*. Below the narrow waist, multipath routing schemes can evolve, and network service providers

can replace one multipath routing scheme for another. Above the waist, any application can gain access to multipath routing capabilities, as long as its socket layer conforms to the interface specified by this narrow waist. This interface embodies a natural separation between the network, which provides access to multiple paths; and the end-systems and applications, which can use these multiple paths.

To be successful, this narrow waist must meet four requirements: It must be *general* enough to support a wide set of applications, *powerful* enough to take advantage of the capabilities provided by multipath mechanisms, and *easy to adopt* in applications without major rewriting. Finally, it must admit *efficient implementation* in networking hardware and software.

Though, admittedly, there can be a variety of path-selection interface that can be developed, the design this thesis proposes for this narrow waist is an opaque string of *path bits* that provides simple but powerful semantics: two packets with the same destination address, but with different path bits, will (with high probability) take different paths to the destination. At the end-system, the networking stack sets the path bits, and modifies them when it wants to use a different path (*e.g.*, in the event of failure or performance degradation); in the network, routers interpret these bits as a "selector" for different outgoing interfaces to a destination. Figure 2.1 summarizes how the right narrow waist can decouple the multipath protocols below the waist and the applications and users of the multipath above the waist.

In our design of *path bits*, we make an explicit choice concerning the amount of control that an end system has over the paths that its traffic takes en route to a destination. Path bits do not explicitly select hops along an end-to-end path, but instead correspond to *some* path. Making path bits opaque provides an interface to applications that divorces the interface from any specific implementation of multipath routing. This abstraction is based on the insight that end systems typically do not care about the specific sequence of hops that traffic takes through the network, as long as they can have easy access to a set of good paths or

alternately can avoid bad (*e.g.*, lossy or failed) paths [9, 41]. Although we recognize that associating some semantics to path bits may afford some benefits, we explicitly choose not to do so.

Although making path bits opaque could require applications to perform trial-and-error to discover a desirable path (*e.g.*, testing alternate paths by setting different combinations of path bits), keeping the path bits relatively free of semantics keeps the interface generic with respect to the types of path properties that an application might want, thereby making the interface both simple and future-proof. This simplicity allows the path bits interface to remain fixed as application requirements and multipath mechanisms evolve. An application can still discover the path bits to use for the path it wants, albeit in a slightly more ad hoc manner than would be necessary with an explicit path selection interface.

To simplify presentation, we focus on the design decisions and properties of the narrow waist. We also show how many existing multipath routing protocols can be mapped to a *path-bits* interface. We show how our design achieves two goals:

1. *Decouple the end systems and multipath routing mechanism so that multipath mechanisms can evolve independently from the applications that use them.*
2. *Provide a simple interface to applications that allows them to achieve application-appropriate benefits from multipath.*

The rest of this chapter is organized as follows. Section 2.2 presents the background and motivates the need for a narrow-waist. Section 2.3 presents the design of the narrow waist and Section 2.4 summarizes the chapter. We follow up on our goal of evaluating how the interface achieves our goal of allowing for efficient implementations of multipath routing protocols in both hardware and software in Chapter 4.

2.2 Background and Motivation

As with the narrow waist in the network stack itself, a narrow interface for multipath routing could accommodate diversity and evolution on both sides of the interface. We describe

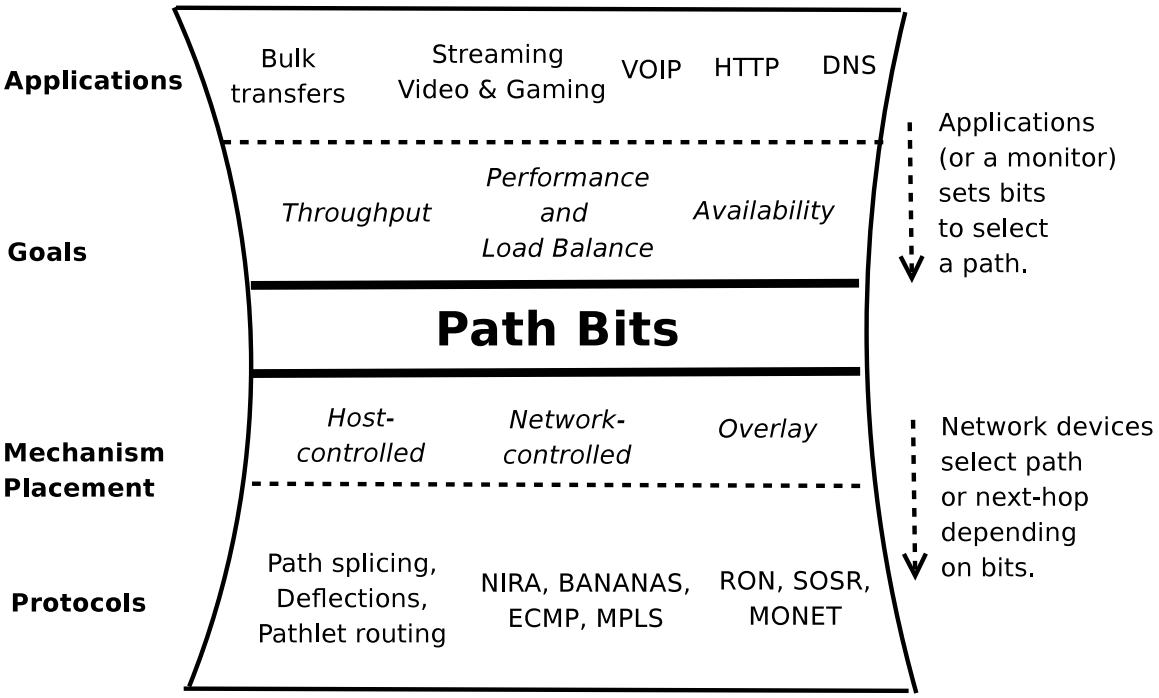


Figure 2.1: *Path bits* serve as a narrow interface between applications that want to use multipath routing and the different implementations of multipath routing.

several multipath routing proposals to show the need for a consistent, narrow interface between applications and multipath routing implementations. Figure 2.1 depicts several "below the waist" multipath routing protocols and "above the waist" applications that can benefit from a choice of paths.

2.2.1 Path Selection Mechanisms

Multipath routing exposes multiple paths for each destination to each end system; we use the term to refer to any scheme that does so either for intra-domain or inter-domain routing. Multipath routing can improve failure recovery by allowing end systems to react to failures more quickly than the underlying routing system would. If a multipath routing system allows an end host to use multiple paths simultaneously, it can also improve throughput. We survey various types of multipath routing mechanisms.

Many mechanisms and environments In *source-controlled routing*, the source controls the path that its traffic takes through the network. Sources annotate packets to signal information about a choice of network path. The control ranges from implicit—from an opaque interface much like path bits—to explicit source routing. Implicit approaches include routing deflections [99] and path splicing [60, 62]. Explicit methods, such as pathlet routing [36], allow the source to specify the path that traffic takes en route to the destination, by, for example, specifying a sequence of virtual nodes ("vnodes").

Some schemes, such as IP fast reroute [79] and MPLS fast reroute [21], operate entirely *within the network*. They maintain multiple paths to act as backups in response to failures. With equal cost multipath routing (ECMP) [44], the network dynamically balances traffic across equal cost intra-domain paths. In other network-controlled multipath schemes, nodes along the path annotate packets with failure information; other routers along this path use this information to route around paths that include the failed node or link [54].

Some network-controlled multipath routing mechanisms operate across domains. In MIRO [96], networks request access to alternate paths from neighbors. Similarly, R-BGP [52] and Anomaly-Cognizant Forwarding (ACF) [28] add functions to the network that enable it to send traffic over a backup path when a failure occurs; neither of these approaches provides control directly to applications or end systems, although path bits could augment these approaches to allow an end-host to explicitly select a backup path in the absence of path failures. BANANAS [49] allows networks to stitch together an inter-domain path using a single Path ID, in the same way that a single MPLS label can be used in interdomain MPLS. This Path ID or MPLS label is similar in spirit to path bits. NIRA [98] allows edge networks some control over end-to-end paths; because path selection in NIRA is explicit, assumes that hosts use provider-based addressing, and requires significant changes to packet headers, it would be difficult to accommodate NIRA with a path-bits interface.

Data centers are also increasingly using multipath routing to improve utilization and enable fast recovery from failures. Many recent data center architecture proposals design

the data center network to have many parallel paths between each pair of servers [38, 42, 1]. TRILL proposes using ECMP-based multipath routing with special bits set by the switches in the TRILL header. VL2 uses ECMP internally, and its network design relies on implementing Valiant Load Balancing (VLB) where a server randomly picks an intermediate switch to forward the packet to the destination. This scheme essentially performs a two "hop" path selection, and can be mapped to using path bits.

Many implementations There are many options for implementing multipath routing protocols in hardware and software. The multipath interface should not be specific to a particular implementation technology (hardware, software, DRAM, TCAM, etc); we evaluate our design using several options that exist today in order to gain confidence that the mechanism will work on future options as well. Supercharged PlanetLab offers programmable network processors [87]. OpenFlow [64] allows a controller to install flow table entries that can match fields in each packet header and direct traffic out certain switch ports. This abstraction could implement a multipath routing scheme based on path bits: extra bits in a packet-header field (*e.g.*, MAC address, IP ID, VLAN ID) could be used to index into different flow-table entries. Similarly, path bits demultiplex packets into different forwarding tables on a NetFPGA card [2]. Of course, multipath routing protocols can also be implemented in software. Any interface for multipath routing should make it easy to realize a variety of multipath routing implementations in both hardware and software. In Chapter 4, we demonstrate how path bits can afford a wide variety of implementations in both hardware and software with a variety of reference implementations in Click, OpenFlow, and NetFPGA.

2.2.2 Diverse Application Needs

Applications differ in both their requirements and their ability to take advantage of multiple paths. We discuss a set of applications that could credibly benefit from using different criteria for the ways they use paths. Because this list of applications is likely to evolve

Table 2.1: Many multipath routing protocols map easily to the path bits interface. We implement the ones in bold and describe their implementation details in Chapter 4.

Multipath Scheme	Representation with Path Bits
Path splicing (Chapter 3)	IP ID field stores the "splicing bits"; TTL field indexes in to those bits at each hop.
Routing Deflections [99]	IP ID field and TTL fields are used to calculate the index into the deflection table at the router
ECMP [44]	Hash the (src ip, dst ip, path bits) tuple instead of just (src ip, dst ip) to select one of the equal cost paths
TRILL [1]	End-systems or switches can set bits in the TRILL header to use ECMP-based multipath routing in the network.
VL2 [38]	VLB mechanism can be implemented using path bits. Path bits can also be used by ECMP deployed within the network to select path to the intermediate switch and then the final destination.
BCube [42]	BCube also relies on multiple paths between different end-systems in a datacenter environment. The BCube multipath mechanism can be implemented using path bits in the packet header.
Pathlet Routing [36]	Encode each pathlets (*i.e.*, a sequence of virtual nodes) onto a set of opaque bits
MIRO [96]	Path bits can be included in the IP ID field, or as an interdomain MPLS tag to indicate the tunnel to use for forwarding packet to an intermediate AS different from the default.

as new applications emerge, a good interface to multipath routing should be agnostic to the criteria applications use to select paths and the ways that applications use the available paths. We consider several representative examples.

Improving throughput. Many file transfer systems balance data transfers across multiple TCP streams to improve throughput [45, 100, 86], and recent work is exploring a multipath TCP congestion control standard [94].

Improving quality or responsiveness Voice applications often use a weighted combination of loss, latency, and jitter to evaluate path quality, using the Mean Opinion Score, or MOS, to make this judgement [85]. Depending on the path quality, these applications might use multipath to avoid degradation due to packet loss or failures, to reduce end-to-end latency, or to bond two low-bandwidth channels to be able to use a higher-quality encoding. Low-bandwidth interactive applications (*e.g.*, telnet) might simply send two copies of each packet, wasting bandwidth to improve responsiveness.

Improving availability Another reason that applications may use multipath routing is to improve availability in the face of failure. The Domain Name System can benefit from sending queries over different paths *to different servers*, allowing it to avoid failed DNS servers as well as failed paths [10]. A video client, on the other hand, may want fast failover to a new path in the event that the current path fails. Although such clients might also use multiple paths concurrently, they may not wish to pay the implementation complexity of doing so. Selecting a path based upon the success of a TCP SYN packet provides a simple, effective way to load balance requests upon multiple paths and avoid many path failures [10, 4, 41]. Although this technique is best-suited for small, stateless requests such as HTTP traffic, its simplicity makes it attractive for some applications.

2.3 Designing the Narrow Waist

We describe the design of the narrow waist (Section 2.3.1) and its properties (Section 2.3.2).

2.3.1 Path Bits Design

Design overview Each packet carries a string of *path bits* that identifies some path in the network. Routers select an appropriate outgoing interface for each packet based on both the destination IP address and the path bits. End systems use path bits to influence the forwarding decisions that routers along the path make. Path bits may reside in an additional header or in unused fields in the IP header.

We are not the first to propose an interface like path bits—indeed, it is inspired by the interface used by several multipath routing designs (*e.g.*, routing deflections [99] and path splicing [60]). Our intention in this section, and for the remainder of this chapter, is to generalize the design of this interface and explore its practicality for real applications in Internet-like environments. In developing a general design for path bits, we explore two main questions: (1) What semantics should the bits carry and how much control should they give to end hosts? and (2) How many bits should the interface have?

Decision #1: Minimal semantics At one extreme, path bits might denote an entire source route; at the other, the bits might simply encode the desire for a source to have a new path. We design path bits to have minimal semantics.

To provide a simple, scalable interface to hosts and to balance control between end hosts and the network, *path bits should be opaque*; they should not explicitly encode any specific path. Rather, they should provide the property that *changing the bits will, with high probability, yield a different path to the destination*. This abstraction is based on the insight that end systems typically do not care about the specific sequence of hops that traffic takes through the network, as long as they can have easy access to a set of good paths or alternately can avoid bad (*e.g.*, lossy or failed) paths [9, 41].[1]

Opaque bits still offer some flexibility: The bits might explicitly encode information about how each node along the path should forward traffic (*e.g.*, having a fixed number of bits per hop), or they might simply encode a request for the network to change the path (a request that could be encoded in a small number of bits, or even a single bit). Encoding only minimal semantics in the path bits also allows independently operated networks to retain some control over how network devices along the path interpret bits, allowing them

[1] One often-hypothesized requirement is that a path might wish to avoid going through a particular location or country; such an application is compatible with a path bits specification. In contrast, expressing a requirement *for* a particular route is more difficult, but such requirements mostly arise in a functional context, such as wishing to route traffic through a firewall [90], a capability beyond the intent of our architecture to provide.

to retain some autonomy concerning routing policy and traffic engineering.

We aim to provide syntactic and semantic isolation between applications and multipath mechanisms by imposing two minimal constraints on the semantics:

- **Per-host consistent path selection** The network must interpret the path selection information such that the same path preference expression will result in the same path choice until a routing reconfiguration occurs. Consistent path selection allows hosts to learn path properties and to ensure that flows that should receive similar treatment will follow the same path. The interface does *not* specify that this information be consistent across end systems. This choice is pragmatic—ECMP and similar mechanisms already provide such a guarantee. It does, however, preclude shared path information approaches, such as SPAND [78].
- **Ability to explore alternate paths** The interface should allow end systems to *explore* available paths in the network. Just as any multipath routing protocol might not expose every route to a destination, the path bits interface need not expose every path. Rather, it should facilitate exploring enough paths to allow a variety of applications to find working paths.

Decision #2: Small number of bits The number of path bits could range from a single bit to $l \log_2 k$ bits, where l represents the maximum number of hops along any network path and each hop has as many as k bifurcations. Using only a single bit (*i.e.*, to indicate that the path should change) offers only coarse control, but is obviously compact. The opposite extreme—encoding each hop as a sequence of bits—offers maximal flexibility because the bits encode the forwarding choice at every hop. In our prototype implementations of the path-bits interface, we opted to use only sixteen bits. We explain this rationale below.

The path bits are included in packet headers and hence must not introduce substantial overhead; on the other hand, the bits should also be expressive enough to give end systems sufficient options for exploring alternative paths. Picking a *specific* number is obviously an

engineering tradeoff, but practical limits suggest, that the number should be constant—not based upon the path length—and that it should be small, but not so small that it forces implementations to maintain complex mappings between the bits and the resulting path. Using a small number of bits also allows them to be embedded in the IP ID or TOS fields of the IPv4 header; in the case of IPv6, packets already include a flow label that could be used to carry path bits. We describe our implementation using IP ID field in the IPv4 header in Chapter 4. A small number of bits can still provide sufficient control and flexibility at each hop. For example, a router could select among n possible outgoing interfaces to a destination based on the output of a hash function, $H(s,d,p)$, where (s,d) represents the source-destination pair and p represents the path bits.

Can a small number of bits still support many multipath implementations? Can an interface that provides only minimal semantics still be useful? Table 2.1 explains how path bits map to existing multipath routing protocols. Our implementations of three different multipath protocols in software and hardware (described in Chapter 4) demonstrate that the interface is powerful and flexible enough to support efficient implementations of many protocols in both hardware and software, as well as varied approaches to path monitoring and selection. Our evaluation in emulated settings and on real-world paths (Chapter 4) shows that, despite not providing explicit choice over the network path, path bits still provide applications enough control to quickly find alternate paths.

2.3.2 Design Properties

The path bits design has two desirable properties: (1) *It decouples the end systems from the specific multipath routing mechanisms* so that multipath mechanisms can evolve independently from the applications that use them, and online service providers can use different multipath routing protocols without having to rewrite applications. As a corollary, it also decouples the interface from the mechanisms that use the interface, so that the multipath protocols can evolve independently of the interface. (2) *It provides a simple interface to*

applications and affords a simple implementation.

Property #1: Decoupling End Hosts from Protocols Decoupling end-hosts path selection from the underlying protocols and mechanisms that implement multipath routing achieves the following goals:

- **Interoperation among different multipath protocols** The interface allows different networks to provide different multipath routing protocols, while permitting an end host to take advantage of multipath routing across two or more such domains. A key requirement for interoperability is ensuring that the interface makes minimal assumptions about how the path selection information is interpreted within the network. Backwards compatibility with conventional routing approaches is also one of our goals and follows naturally from this requirement.
- **Shared control between end hosts and the network** The interface balances control between end systems and routing protocols. End systems should be able to improve throughput or balance load without introducing unpredictable or oscillatory traffic patterns.

In contrast, a control mechanism that tells routers which path to select (*e.g.*, classical source routing), or indicates to the routers a set of properties that the chosen path must satisfy (*e.g.*, the original ToS bits in the IP header) couples the application and the semantics of path selection. As a result, applications built to such an interface could assume a degree of control that might not be available with other protocols, and the network's routing protocols could become reliant upon receiving this information from applications. Future applications would be forced to provide sufficient information for this interface, and future path selection mechanisms would be required to provide (a superset of) the existing semantics.

Property #2: Simple Interface and Implementation The path-bits interface is simple, easy-to-use, easy-to-interpret at end systems, and easy for network devices to implement,

regardless of the choice of the underlying multipath routing protocol implementation.

- **Easy and flexible for end-system applications** Path bits shield applications and hosts from the complexity of the underlying multipath routing, but still provide considerable flexibility. For example, an application programmer may wish to bind a flow to a path, only changing the path when a failure occurs; another application programmer may wish to split a flow across multiple paths to improve throughput. The path-bits interface enables both of these modes.
- **Easy to implement on network devices** Because decoding the path bits signal in each packet is straightforward and does not require much state in the routers and switches beyond that required by the underlying multipath routing protocol, we could easily and efficiently implement three multipath routing protocols that use the path-bits interface, in both programmable hardware and software routers. The interface also scales well with multipath mechanisms that expose many paths: The memory requirements and processing required to map from the path bits to routes remains constant as the number of available paths increases.

2.4 Summary

We have motivated the problem which is the premise of the thesis, that of a unified narrow waist interface for path selection in the Internet. Despite a large body of work in providing multiple paths in the network, we believe that lack of a standard multipath interface is hampering development of compelling applications and pervasive deployment of multipath routing protocols in the Internet. As we illustrate, many networked applications can benefit from access to multiple paths for improved performance and rapid failure recovery. A unifying interface for path selection that divorces the applications from the actual multipath details could lead to independent evolution of applications utilizing path selections and of underlying multipath routing protocols.

We describe the design of *path bits*, a narrow waist for multipath routing—a standard

interface that makes minimal demands of applications and of the network—that we believe will enable evolution of protocols and implementations below the waist, and applications above it. Path bits is simply a string of opaque bits included in the packet header, inserted by the application or end-system. The interpretation of the *path bits* to select a path is left to the particular multipath implementation. The *only* requirement is that changing the path bits should with high probability select a different path to the destination.

The *path-bits* interface is simple and easy for applications to use. It is general and admits efficient implementation in both hardware and software. We will discuss more about these implementations and their evaluation in Chapter 4. We will also show how the simplicity of the interface permits different path monitoring implementations at the end-system that can suit different applications. We make our implementations available as the first framework that allows both different multipath algorithms and different monitoring and recovery frameworks in a common context [3].

In the next chapter we present a multipath routing scheme, *path splicing* that also allows end-systems to be able to influence the path their traffic can take in the network by using a separate *shim* header in the packets. This path selection interface of path splicing can be mapped easily to the path-bits interface. We will present prototype implementations of path splicing using path-bits in Chapter 4.

CHAPTER III

CREATING NETWORK PATHS: PATH SPLICING

3.1 Introduction

The narrow waist assumes that the network (or the routing protocol) creates multiple paths in the underlying network. In this chapter, we describe the design and evaluation of a multipath routing primitive, *path splicing*, that can also benefit from a *path-bits* interface. Path splicing uses a set of bits, *splicing bits* in the packet header to allow end-systems control over path selection. This can be easily mapped to *path-bits* interface.

Multipath routing, which provides nodes access to multiple paths for each destination, can increase availability by providing fast (or simultaneous) access to backup paths; it can also improve capacity by increasing the number of paths that endpoints can use to communicate with one another. As Internet applications demand higher availability and faster recovery from failures, multipath routing and pre-computed backup paths have emerged as promising mechanisms for recovering from failures.

Despite the need for, and the promise of, multipath routing, many such schemes require considerable precomputation to achieve even a small number of paths through the network. Two obstacles have hindered many multipath routing solutions; the first is *scalability*. Existing schemes typically compute a small number of backup paths that can protect against certain failure scenarios, but they do not provide recovery from many others. Instead, the routing system should provide much stronger guarantees: Unless the underlying network is partitioned, the routing system should provide at least one path that allows endpoints to communicate. The second obstacle is *control*: an endpoint (or intermediate point) should have some ability to change the path or paths that it uses to send traffic to each destination. Unfortunately, granting too much control to end systems can interfere with traffic

engineering and may potentially result in traffic oscillations [71].

This chapter presents the design, implementation, and evaluation of a new routing primitive called **path splicing**, a scalable mechanism for providing network nodes or endpoints access to a very large number of alternate paths. Path splicing has three key features: (1) it constructs multiple routing trees over a single fixed physical topology; (2) it allows traffic to take a path that switches between these trees at intermediate hops en route to the destination; (3) it allows end systems to change the forwarding path by changing a small number of additional bits in the packet header. Intermediate nodes can also change the path on which traffic is forwarded. These building blocks, of course, could apply to any routing protocol. In this chapter, we study them in the context of intradomain and interdomain routing.

We explore how path splicing can improve availability according to two metrics: *reliability* and *recovery*. Reliability measures whether the routing information that is disseminated between routers reflects the connectivity in the underlying topology. In other words, it measures whether the paths that each router knows create a connected graph in the underlying topology, even when links or nodes in the underlying topology fail. Recovery measures how quickly endpoints can re-establish working paths with one another by finding a working path in among the available choices in the routing tables. Our evaluation demonstrates that, with just a few slices, path splicing can achieve reliability that approaches that of the underlying graph (*i.e.*, the best possible), and that, in the face of failures, end systems can discover a new working path within two trials (which are independent and can be run simultaneously), even without any knowledge about the location of the failure. The actual time to recover from a failure, of course, also comprises the time to detect the existence of a failure, which we do not consider in this work. Our results suggest that, when combined with a fast failure detection mechanism, path splicing can provide end systems with enough resilience to quickly recover from failures without waiting for dynamic routing protocols to converge to a new working path.

To illustrate why path splicing can be so effective, consider Figure 3.1. A conventional

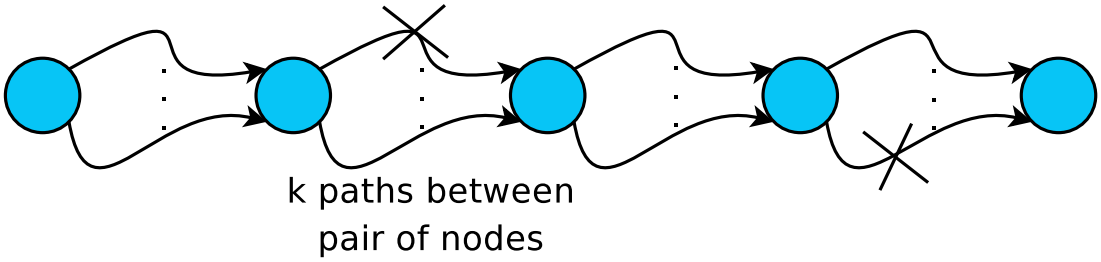


Figure 3.1: With k paths between the pairs of nodes, any k failures, one on each path disconnects the network. With splicing, a graph cut must be created to disconnect the network.

routing algorithm would compute one path between the nodes at each end. Multipath routing typically aims to compute k edge-disjoint paths between these nodes. Unfortunately, if at least one link fails on each path, the nodes may become disconnected, *even if the underlying topology remains connected*. Path splicing computes multiple paths and also allows traffic to change paths at intermediate nodes, thus "splicing" paths together. By providing access to these spliced paths, path splicing can sustain connectivity in the face of many more link and node failure scenarios. In Figure 3.1, the pair of nodes on each side of the graph will become disconnected if a link fails on each of the k edge-disjoint backup paths. With path splicing, k links must fail in the same cut to create a disconnection, a much less likely event (since this is only one specific way in which all k paths could be broken). If we assume that links fail at random, then $O(k \log k)$ failures will disconnect all k paths with high probability[1], and the probability of a cut is exponentially small.

Despite its conceptual simplicity, path splicing faces several practical challenges. First, splicing forwards traffic along paths that do not constitute a single tree to a destination, which creates the possibility for paths to contain loops. We show, both analytically and empirically, that in practice these loops are neither persistent nor long. Second, splicing gives end hosts some control over where traffic is forwarded, which can interfere with operators' traffic engineering goals and potentially cause oscillations if all end systems forward traffic

[1]This result follows from the coupon collector problem.

over the same set of links. Path splicing's interface for path selection carries no explicit semantics about the actual path, however, which means that end systems have no mechanism or incentive to select the same alternate path when a path fails. Our experiments show that spliced paths do not adversely affect the traffic distribution or load across the network links. Finally, there is an inherent tradeoff between the extent to which alternate slices provide paths with a diverse set of edges and the additional latency ("stretch") incurred along the spliced paths. For intradomain routing, path splicing can achieve near-optimal reliability with a stretch of about 30%; for interdomain routing, splicing can achieve near-optimal reliability with negligible stretch in terms of the number of AS hops.

Although this chapter focuses on how splicing applies to Internet routing (specifically, we focus on applications of splicing to both intradomain and interdomain routing), the mechanism is general and could certainly be applied in other contexts (*e.g.*, routing in datacenter networks or overlays). We discuss possible extensions to datacenter networking in Chapter 6. This chapter explores how path splicing can improve availability by facilitating rapid recovery from failures; however, splicing is useful in any scenario that requires access to multiple paths. We also defer the details of implementing path splicing, on both hardware and software platforms, to Chapter 4.

The rest of the chapter is organized as follows. Section 3.2 summarizes our design goals. Section 3.3 presents related work. Section 3.4 provides an overview of path splicing and describes the high-level properties of the technique. Section 3.5 describes how splicing can be applied to intradomain routing, and Section 3.6 describes an extension of splicing to interdomain routing. Section 3.7 presents experiments that quantify how splicing improves both reliability and recovery, and explores splicing's effects on and interactions with traffic. Section 3.9 describes a possible implementation path for splicing, as well as security concerns, and Section 3.10 concludes.

3.2 Design Goals

To achieve high availability, routing must exploit the underlying diversity of the network graph. Routing should maintain paths between nodes in the network unless the underlying network graph itself is disconnected. Current routing protocols, which are typically single-path, cannot achieve this. The challenge in providing multiple paths in the network to provide high path diversity is to disseminate the information about the multiple paths in a simple, scalable fashion. Specifically, a routing system should have the following design goals:

- **High reliability.** A routing protocol should allow nodes to maintain information about connectivity between pairs of network nodes, even as nodes or links in the network fail. (Section 3.2.1)

- **Fast recovery.** In addition to providing many alternate paths, the routing protocol should allow end systems to discover and use these alternate paths. (Section 3.2.2)

- **Small stretch.** The alternate paths should not be significantly longer, in terms of latency or number of hops, than the default path. (Section 3.2.3)

- **Control to end systems.** End systems should have some control over the paths that traffic uses. (Section 3.2.4)

The rest of this section describes these goals in more detail and formally defines metrics that we use to evaluate them.

3.2.1 High Reliability

Many attempts to improve reliability through diverse, multiple paths have operated without a clear definition of either reliability or path diversity, although they have typically implicitly assumed an "operational" definition of masking path failures along paths between endpoints. To capture the effect of increasing path diversity on the actual availability of

the network, we introduce a formal metric for *reliability*, which describes how the graph behaves under failure. Reliability essentially measures the extent to which nodes in an underlying graph remain connected when nodes or edges in the underlying graph fail. We first formally define reliability.

Definition 3.1 (Reliability)**.** *For a given graph G, and any* $0 \leq p \leq 1$*, let* $R(p)$ *denote the fraction of node pairs that are disconnected when each edge fails independently with probability p. Reliability is then represented as a function* $y = R(x)$*, where x ranges from* 0 *to* 1.

The intuition behind reliability is that it reflects the probability that the graph experiences disconnection given that links in the graph fail with certain probabilities. While we could certainly represent reliability as a binary metric (*i.e.*, does the graph remain connected or not?), it is convenient to talk about reliability in terms of the fraction of node pairs become disconnected when a certain fraction of edges fail.

This metric has an edge version and a vertex version. We have stated the edge version, but the vertex variant is quite similar. Note that this metric can apply to any graph, including the underlying network graph; we can assess the reliability of a routing protocol by comparing the reliability achieved by the routing protocol to that of the underlying graph. To achieve high reliability (*i.e.*, to attain a reliability curve that mirrors as closely as possible that of the underlying graph), a routing protocol should exploit the path diversity that exists in the underlying graph.

Conventionally, previous routing protocols have achieved high path diversity by providing systems access to node-disjoint paths. However, paths do not need to be completely node disjoint to provide high reliability (particularly if edges are failing, as opposed to nodes). To capture this property, we quantify the diversity that is achieved by two paths using a notion we call *novelty*. Essentially, the novelty of two paths is the fraction of edges between the two paths that are distinct.

Definition 3.2 (Novelty)**.** *Given a* $(source, destination)$ *pair, let* P_s *be the path with fewer edges and* P_l *be the path with more edges. Formally, novelty is*

$$1 - \frac{|P_l \cap P_s|}{|P_s|}$$

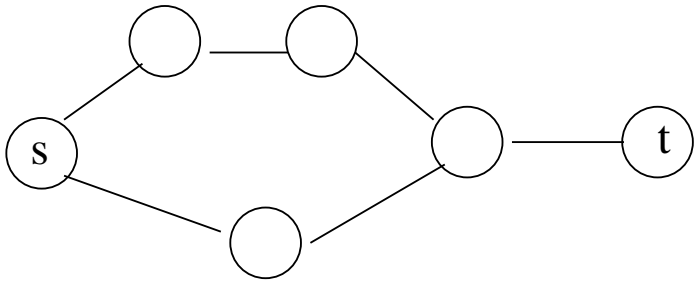


Figure 3.2: Two partially disjoint paths; the novelty value is $2/3$.

Novelty provides a diversity metric for any two paths between a source-destination pair. Note that novelty captures disjointness in some fashion: For example, two paths that are completely edge disjoint will have novelty 1. Figure 3.2 shows two partially disjoint paths between two nodes s and t. This pair of paths has a novelty value of $2/3$: The two paths share one link in common, and the length of the shorter path is 3. As with reliability, novelty has a vertex version, but we focus on the edge version in this thesis. In our experiments, we use novelty to quantify the diversity of the paths in each alternate slice relative to the original shortest path.

3.2.2 Fast Recovery

Simply achieving high reliability is not of much use if the routing system cannot quickly discover working paths when nodes or edges fail. Beyond simply achieving *high reliability*, a routing system should quickly, scalably, and simply provide working paths to nodes and end systems when links or nodes fail. We define the time it takes for a pair of nodes to establish a working path after a failure has occurred the *recovery time*.

Definition 3.3 (Recovery Time)**.** *Recovery time is the time that the routing system takes to re-establish connectivity between a* $(source, destination)$ *pair after the existing path has failed.*

In the absence of pre-computed backup paths or other "fast recovery" techniques, the recovery time is simply the convergence time of the routing protocol (*i.e.*, the time it takes to re-establish a working path after a failure has occurred). In the case where backup paths are available, however (*e.g.*, in the cases of fast reroute and path splicing), recovery can be faster than convergence time, because a failure can trigger an immediate failover to a backup path.

When we consider recovery time of path splicing, we are interested in quantifying how long it takes for end systems to discover alternate working paths after a failure occurs. Recovery time should ideally be measured in units of time and include both the detection time (*i.e.*, the time taken to detect a failure) and the time to discover a new working path. Without a complete implementation, however, it is difficult to express recovery time in units of time. For our evaluation in Section 3.7, we express recovery in terms of number of trials—the number of recovery attempts before a working path is found. One could estimate recovery time as detection time plus the recovery time, where recovery time is the number of trials required for recovery divided by the number of trials that can be executed in parallel.

3.2.3 Low Stretch

Routing protocols should provide access to alternate paths that are not significantly longer than the "default" path between those nodes, both in terms of the actual latency of the alternate paths and in terms of the number of hops that they traverse. We define a notion of *stretch* to quantify the additional latency that is incurred by alternate paths over the default path.

Definition 3.4 (Stretch)**.** *Stretch is defined as the ratio of the latency on a path (between a pair of nodes) in the perturbed topology to the ratio of the shortest path (between the same pair of nodes) in the original topology.*

We use total path cost as a proxy for latency. Path diversity and stretch are somewhat

conflicting goals. Thus, we must generate slices to have low stretch, but high novelty. An easy approach to creating paths with high novelty with acceptable stretch is to create slices at random (*i.e.*, by using random link weights for creating each slice). Selecting link weights in this way would lead to paths with high stretch.

3.2.4 Control to End Systems

The notions of *availability* and *failure* are specific to the application sending traffic along these paths. In the case of real-time applications such as VoIP, it matters if the packets cannot reach the destination in a certain bounded time. For other applications (*e.g.*, bulk file transfer), these constraints may matter less, but end systems may wish to find paths with high throughput. Because end systems have differing requirements for what constitutes a "good" path, building a "one size fits all" routing system that provides good paths to all applications without taking input from the end systems themselves about the quality of paths is difficult.

If an end system deems some path in the network to be non-functional or detrimental to application performance, it should be able to signal to the network the desire to send its traffic along a different path. Of course, because network operators have traffic engineering goals and constraints of their own, the routing system should provide this control without introducing too much instability to the offered traffic load in the network.

3.3 Related Work

We survey related work in three areas—multihoming and multi-path routing, fast recovery schemes, and overlay networks—and explore the tradeoffs of each of these recovery schemes in terms of processing overhead, storage overhead, recovery time, and required modifications to existing routers.

Multihoming and multipath routing. Multihoming and multi-path routing provide nodes multiple paths for exchanging traffic. Various mechanisms manipulate routing to take better advantage of multiple underlying network paths [23, 46]. These schemes can operate without changing hosts or routers but are more coarse-grained, since they still only forward traffic along one path to each destination at any time. Perlman designed a routing protocol that floods routes in a way that is robust to Byzantine failure [67]. MIRO [96] and R-BGP [52] allow networks to discover additional interdomain routes to recover from failure. MIRO provides more explicit control over the AS path that traffic travels to a destination (*e.g.*, it allows a network to explicitly select the ASes that its traffic traverses) and it requires no modifications to the data plane (*i.e.*, packet headers or forwarding functions), but it requires establishing additional state at routers for each alternate path and additional out-of-band control-plane signaling, which may make it too heavyweight as a general recovery mechanism. R-BGP provides similar interdomain failure recovery as splicing, without requiring any modifications to the packet headers. Like splicing, it requires additional state in forwarding tables like splicing. Unlike splicing, however, R-BGP provides only local recovery at routers.

Path splicing relates to multi-topology routing, which precomputes backup topologies for specific failures by removing edges from the underlying topology or by setting high costs on some edges [13, 35, 53]; in contrast, path splicing computes alternate paths for *arbitrary* failure combinations. Path splicing allows traffic to traverse multiple topologies along a single path, whereas multi-topology routing only allows traffic to switch topologies once en route to the destination. It also allows end systems to divert traffic along different paths. Aspects of multi-topology routing have been standardized [70], and Cisco has recently incorporated a related mechanism called multi-topology routing into their IOS routing platform [22]; a small variant could ultimately enable path splicing.

Fast recovery and reroute. Path splicing uses bits in the IP header to affect how routers along a path forward traffic to a destination. This mechanism is similar to the "deflection" mechanism recently proposed by Yang *et al.* [99]. Although path splicing's mechanisms for deflecting traffic along a new end-to-end path are similar, we show in Section 3.7 that path splicing achieves more path diversity than this deflection mechanism with considerably shorter paths. Establishing parallel backup paths resembles various techniques proposed by the IETF routing working group [77] and router vendors, including load balancing mechanisms such as equal-cost multipath [44], link protection mechanisms such as MPLS Fast Reroute [21], IP Fast Reroute [79] (as well as various optimizations [80, 14]), but fast reroute requires manual configuration and requires additional routing state for each link or node to be protected. Furthermore, rerouting is triggered only by local failure detection, not by end systems. Failure-carrying packets (FCPs) carry information about failed links; this information allows routers to re-route data packets around failed links [54]. Like fast reroute mechanisms, FCPs allow routers to circumvent node and link failures without waiting for the routing protocol to reconverge, but the mechanism only provides local recovery and requires inserting large amounts of information into packets as well as potentially expensive dynamic computation.

Improving reliability with overlays. Overlay networks can improve diversity by routing traffic on alternate paths above the network layer [9, 11, 41]. Others have investigated how to improve connectivity by strategically placing overlay nodes within a single ISP [19]. Splicing provides a similar recovery capability without requiring continual probing of alternate paths.

3.4 Path Splicing: Main Idea

Path splicing is a general mechanism for giving end systems access to multiple paths composed from multiple routing trees. Any instantiation of path splicing relies on the following three aspects:

1. *Generate many alternate paths by running multiple routing protocol instances.*[2] Instead of running a single instance of a routing protocol over a topology, routers run k routing protocol instances on the same topology, each with a slightly different configuration. The goal is to design the configuration of the routing protocol instances such that the trees to each destination do not share many edges in common. Every node then stores k forwarding table entries for each destination (one corresponding to each tree).
2. *Allow traffic to switch between paths at intermediate hops.* Rather than routing traffic over a single topology, path splicing allows traffic to switch topologies at any intermediate hop along the path. Thus, rather than having k options, a source gains access to considerably more paths to a destination (in theory, as many as k^l, where l is the number of hops on a path between the source and destination).
3. *Give end systems the control to switch paths.* To select a path, an end system includes *splicing bits*[3] in the packet, along with the packet's destination. These splicing bits control which of the k forwarding tables is used at each hop en route to the destination. In later sections, we describe several possible designs for the splicing bits.

Path splicing has many possible realizations in various contexts. For example, it does not mandate the use of any particular routing protocol, nor does it specify how alternate topologies are generated. In the rest of this thesis, we study path splicing in the context of Internet routing. Section 3.5 discusses the application of path splicing to intradomain routing; Section 3.6 discusses path splicing in the context of interdomain routing. In each case, the methods for generating alternate paths are slightly different, but both share the above three properties.

[2] We describe splicing as running k routing protocol instances for conceptual simplicity. Later, we describe how the same function can be achieved by only running a single routing protocol instance.

[3] As noted earlier, this is similar to path bits concept.

3.5 Intradomain Path Splicing

In this section, we describe the design of *path splicing* in the context of intradomain routing. However many of the design features described in this section are also generally applicable for splicing on other types of networks. We also define some of the terminology we use when talking about splicing in the later sections.

3.5.1 Control Plane

The first step in splicing is to create a set of slices for the network. A *slice* is essentially a set of shortest path trees for a particular *view* of the network graph.

Constructing slices. The path splicing *control plane* computes multiple routing trees based on perturbations of the underlying network topology. The control plane comprises of two main components: (1) random perturbations of link weights to help deflect traffic off the shortest paths for some gains in diversity; and (2) pushing these routes in the data plane so that they can be used by the routers in making forwarding decisions.

Conventional shortest paths routing is designed to route traffic along low-cost paths, but it may create bottlenecks between various source-destination pairs. To allow endpoints to discover paths other than shortest paths between any two nodes in the network, path splicing creates routing trees that are based on *random link-weight perturbations*.

Path splicing perturbs link weights based on the original weight of the link to ensure that the length of the new shortest path is not very long compared with the original shortest path (*stretch*). The following expression defines the link weight perturbations:

$$L'(i,j) = L(i,j) + \text{Weight}(a,b,i,j) \cdot \text{Random}(0, L(i,j)) \tag{3.1}$$

where $L(i,j)$ is the original link weight of the link from nodes i to j, $\text{Weight}(a,b,i,j)$ is a function of some properties of nodes i and j (*e.g.*, the degrees of the nodes), a and b are constants and $\text{Random}(0,L(i,j))$ is a random number chosen in the range of 0 to $L(i,j)$.

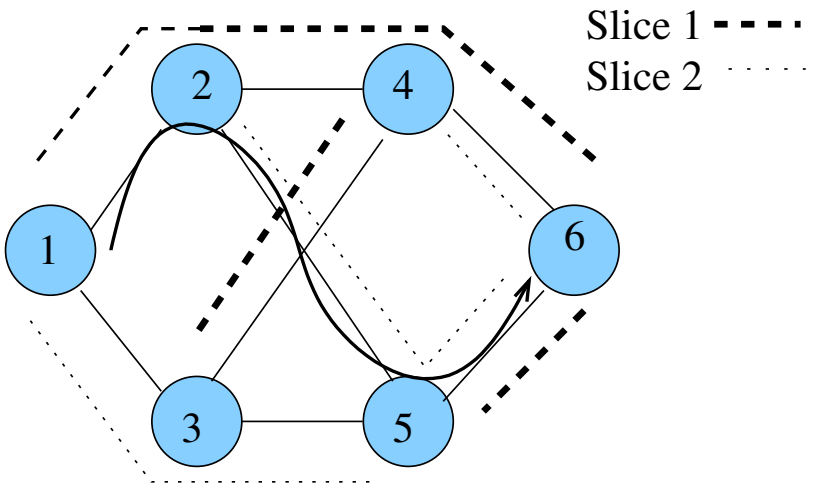


Figure 3.3: Example of path splicing: The two different slices shown with dotted lines on top of the original topology reflect two different trees, both rooted at node 6. Traffic can reach node 6 by traversing one or more trees.

The nature of the perturbation can be changed by using different Weight() and Random() functions. The particular Weight() function used will have an effect on the types of shortest paths selected by the shortest-path algorithm.

Degree-based perturbations of link weights. The function Weight(a,b,i,j) is selected to be a linear function of the sum of the degrees of i and j, *i.e.*

$$\forall_{i,j}\text{Weight}(a,b,i,j) = f_{ab}(degree(i)+degree(j))$$

where f_{ab} is a linear function in $degree(i)+degree(j)$ ranging from a to b. This function will cause the perturbations to depend on the end vertices i and j of a link. Links connected to nodes with a high degree may be perturbed more than links connected to nodes with smaller degree, which reduces the likelihood of many shortest paths using the same link. To describe a degree-based perturbation, we use the notation *Degree-Based* $[a,b]$, where a and b correspond to the minimum and maximum values that can be taken by the Weight(i,j) function. The intuition behind degree-based perturbations is to discourage the use of links between high-degree nodes, introducing more diverse path choices.

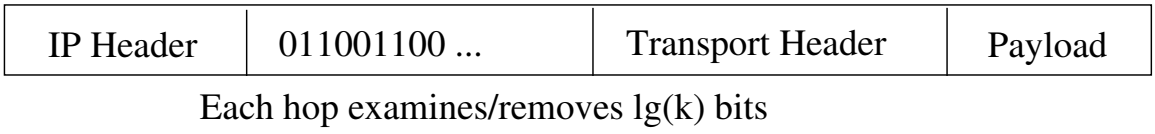


Figure 3.4: The path splicing header sits between the IP and the transport headers, facilitating incremental deployment: routers without path splicing simply forward traffic based on the IP header.

3.5.2 Data Plane

Once we have precomputed multiple slices in the network, a spliced path can be constructed by "splicing" together path segments from one or more slices. For example, as shown in Figure 3.3, a spliced path from node 1 to 6 is constructed by starting on slice 1 and then switching to slice 2 at the next hop (node 2). Thus, a spliced path is composed of multiple path segments from different slices. It is also easy to construct, since at each hop an independent forwarding decision could be made to either let the packet be forwarded on the same slice or switch to another slice. As we describe further, the packet could carry splicing bits (shown in Figure 3.4), which dictate the slice on which the packet is to be forwarded at each hop along the path. Because each hop stores the forwarding table entries (FTEs) for each slice in a separate forwarding table, the bits can index the forwarding table to use (since a forwarding table corresponds to a slice).

Header format End systems insert a "shim" *splicing header* in between the network and transport headers. End systems can set *splicing bits* in this header to control the path taken by the packets in the network by indicating, for each hop, which forwarding table should be used to forward the packet en route to the destination.

We propose a simple encoding where the shim header contains, for n hops along the network path, $\lg(k)$ bits that indicate an index into the forwarding table that should be used to forward the traffic at that hop, where k is the number of slices used to splice the network paths. Thus, if the size of the splicing header is $n \cdot \lg(k)$ bits, then the header

Algorithm 1 Algorithm for forwarding packets

$dst \Leftarrow$ destination IP address
$src \Leftarrow$ source IP address
$fwdbits \Leftarrow$ splicing bits from the shim header
if $fwdbits > 0$ **then**
  slice $\Leftarrow fwdbits$ & $(2^k - 1)$
else
  slice $\Leftarrow Hash(src, dst)$
end if
$fwdbits >> \lg(k)$
nexthop $\Leftarrow Lookup(dst, slice)$
forward packet to next-hop

allows the packet to switch between k slices for as many as n hops along the network. Our experiments in Section 3.7 indicate that reliability of path splicing approaches the *best possible* reliability (as limited by the underlying network topology) with only about 4 or 5 slices. Given that most network-level paths are typically less than 30 hops [15], even this inefficient encoding would require only $30\lg(4) = 60$ bits. Other encodings could reduce the overall size of the splicing header.

Forwarding algorithm. As shown in Algorithm 1, to forward packets, each node along the path: (1) reads the rightmost $\lg(k)$ bits from the splicing header to determine the forwarding table to use for forwarding the packet; and (2) shifts the bitstream right by $\lg(k)$ bits to allow subsequent hops to perform the same operation.

In the default case, an end system sets the splicing bits in the splicing header to direct traffic along a path in a single routing tree (*i.e.*, as would be the case with a conventional routing protocol). A network can achieve some load balance if sources select their initial slices at random: in the absence of failure, a different subset of all sources can route traffic in each perturbed slice, achieving better "spread" of traffic across the network than could be obtained by routing all traffic along a single tree. We evaluate the effects of splicing on traffic in the network in Section 3.7.7.

Splicing bits carry *no explicit semantics*; this characteristic has two important implications. First, it allows path splicing to scale well, since end hosts never need to learn the details of actual paths through the network; rather, they simply use the splicing bits as an opaque identifier for some path, and they can change the path through the network simply by changing the splicing bits. We believe that this function is sufficient: end systems tend to care less about the specific hop-by-hop details about the paths their traffic is traversing than they do about whether or not they can route around a poorly performing (or faulty) path with high likelihood.

Because splicing bits control which path segments from the different slices are used to construct a spliced path, the selection of the bits determines whether an end-to-end path could be found between two nodes for which the path on the default path is disconnected. Our evaluation shows that even an extremely simple choice for the splicing bits ensures that end systems will be able to find an available path within two trials.

Because the splicing bits are opaque and have no explicit semantics (*e.g.*, they do not specify node addresses for a path), path splicing is incrementally deployable: routers that have implemented path splicing can inspect the splicing header and route packets out a different outgoing interface based on the rightmost $\lg k$ bits in the header. Nodes along the path that do not support path splicing simply forward data packets as they normally would, based on the destination IP address in the IP header.

Failure recovery. When a failure occurs, traffic must be redirected to a different slice; an end host can perform this redirection simply by changing the bits in the splicing header, which will cause an end-to-end path to the destination to be spliced from a different set of slices. This redirection could be performed by either a node along the path that detects the failure or the end system, end systems can detect poorly performing paths from a variety of causes (*e.g.*, queueing, packet loss, etc.), and they are better equipped to detect when traffic should be deflected off of a poorly performing *end-to-end* path.

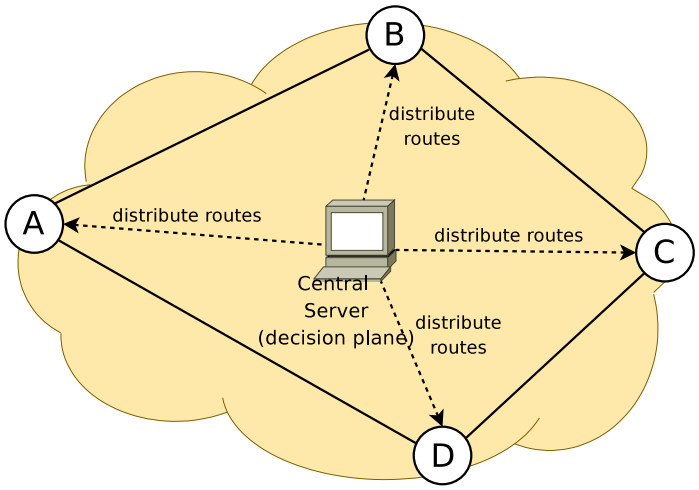


Figure 3.5: 4D-Style deployment of path splicing. The *decision* plane consists of a centralized server, which computes slices and distributes the computed routes to the routers via the *dissemination* plane.

There are many possible ways to attempt recovery. Perhaps the simplest approach is for an end host to select a new random bit-string for the splicing header upon detection of a failure, which will cause traffic to be sent, with high probability, along a completely different path, thus avoiding the cause of the faulty path. If an end system were able to determine the location of a failure, however, it could change only the bits in the splicing header that were needed to divert traffic around the failure. As a third option, an end system could divert traffic to a different slice at an early point along the path (*i.e.*, close to the source) so as to divert traffic to a network slice that avoided the failure with high likelihood.

Nodes in the network can also take advantage of splicing to divert traffic from default paths during network failures or high congestion. If a router detects that the next-hop for a particular destination is unreachable, it can send the packet on some other connected slice while waiting for the routing protocol to converge.

3.5.3 Deployment: Path splicing in 4D

Path splicing could be deployed within the context of the 4D architecture [97], as shown in Figure 3.5. The 4D architecture factors the network architecture into *decision, dissemination, discovery* and *data* planes. The *decision* plane computes the routes for each corresponding slice, possibly at a centralized server. Instead of running a routing protocol, the *decision* plane computes routes from the topology information it receives from the *discovery* plane and uses perturbations of that discovered topology to create routes. The server then uses the *dissemination* plane to disseminate these routes to the nodes in the network. Given enough slices, a network that computes and disseminates paths with path splicing could mask a significant set of link and node failures without requiring the routing protocol to recompute new routes. In other words, path splicing implemented in the 4D context could eliminate the need for any intradomain routing protocol, beyond simple topology discovery.

3.5.4 Optimizations

We describe few optimizations to make the splicing implementation more efficient in the network. We implement few of these in our hardware implementation of splicing described in Chapter 4.

Single routing protocol instance. It is easy to think of path splicing as running multiple instances of the routing protocol, where each instance runs with a slightly perturbed version of the topology. Unfortunately, running multiple instances of a routing protocol introduces additional unnecessary overhead including additional routing messages, as well as resource consumption on the nodes running multiple instances of the routing software.

Instead, we can implement path splicing within the context of a single routing protocol instance, with a few minor modifications. As in any intradomain routing protocol, each node would discover the complete network topology via link-state advertisements. Each

node could then generate multiple variants of this topology by perturbing the weights on each edge in the graph *in the same way as on other nodes in the topology* and could compute forwarding tables for each slice locally, without having to run multiple routing protocol instances to advertise perturbed link costs.

Single forwarding table. The basic splicing scheme requires inserting FTEs corresponding to each slice in a separate forwarding table at each node, essentially having a forwarding table for every slice. Given that every node has a fixed number of neighbors, there could be many common entries for a particular destination among the different forwarding tables. For example, if a node has only two neighbors and we compute 3 slices, then at least two of them will have the same next hop for any destination. Thus, maintaining separate forwarding tables for every slice can lead to inefficient use of memory. One possible optimization involves having only a single forwarding table for all slices and maintain a separate column which records the different slices for which a particular entry is valid.

Embed splicing bits into the IP header. As we have described path splicing, the splicing bits explicitly control which slice each node on the path should use to forward traffic. In this case, the size of the shim header is proportional to the length of the path. To reduce this overhead, the splicing bits could instead be encoded in a smaller number of bits and embedded into the type-of-service and IP ID fields in the IP header; each router could then select the slice on which to forward traffic based on, say, a hash of these bits (and possibly also the source and destination IP address).

3.6 Interdomain Path Splicing

This section describes the application of path splicing to interdomain routing. Interdomain splicing can be deployed without modifying BGP's message format and with no additional routing messages. In fact, it can be deployed using only a single BGP instance.

The key idea involves exploiting the fact that each router learns one BGP route to each

destination *per session*, and most BGP-speaking routers already have multiple BGP sessions to neighboring routers. Rather than selecting a single best route per destination, a router inserts the best k routes for each destination; a packet's splicing bits can then directly indicate which of these k routes a router should use to forward traffic to each destination. This section describes the control-plane and data-plane modifications to routers, and practical considerations (*e.g.*, ensuring that spliced BGP routes do not violate business policy).

3.6.1 Control Plane

Routers typically learn multiple routes to any given destination prefix both from neighboring ASes and from other routers within the same AS (via internal BGP), as shown in Figure 3.6. Some of these routing table entries may correspond to alternate highly disjoint paths in the network. Routers may thus *already* learn multiple diverse routes for each destination. Today, BGP selects only a single best route for each destination prefix. Instead, a router could select the best k routes and push them into the forwarding table. The splicing bits in a packet then index to the appropriate FTE at each hop. Using splicing bits to access alternate FTEs contrasts with existing multipath interdomain routing schemes (*e.g.*, MIRO [96], R-BGP [52]), which rely on the control plane to discover and exercise these alternate routes.

A naïve approach for selecting the top k best routes would be to repeat the route selection k times, each time removing the best route and pushing it into the IP routing table of the router. A more efficient approach would be to modify the BGP decision process to select the k best routes instead of a single best route.

Algorithm 2 shows a more efficient algorithm for selecting the top k BGP routes but requires modifications to the BGP route selection process. The process of updating the routes in the IP routing table of the router will also need to be modified to update the appropriate forwarding table whenever the route corresponding to that table changes.

Algorithm 2 Modified BGP route selection process for interdomain splicing

```
Type ⇐ highest-local-pref
All ⇐ all routes for a given prefix
k ⇐ number of slices
call Sel_routes(Type, All, k)
// Definition of function Sel_routes
procedure Sel_routes(Type, All, k)
Best ⇐ NULL
if Type == highest-local-pref then
   RSet ⇐ Routes with highest local pref (All)
   if |RSet| < k then
      k ⇐ (k − |RSet|)
      // Order these routes based on the next priority rule
      Best ⇐ Best ∪ Sel_routes(shortest-as-path, RSet, |RSet|)
      All ⇐ All − RSet
      // Select remaining routes from next highest local pref
      Best ⇐ Best ∪ Sel_routes(highest-local-pref, All, k)
   else
      Best ⇐ Best ∪ Sel_routes(shortest-as-path, RSet, k)
      return Best
   end if
else if Type == shortest-as-path then
   RSet ⇐ Routes with shortest AS path length (All)
   // Similar to the code in the above if statement
   .
else if Type == lowest-orig-type then
   Rset ⇐ Routes with lowest origin type (All)
   .
else if Type == lowest-med-value then
   Rset ⇐ Routes with lowest MED value (All)
   .
end if
```

3.6.2 Data Plane

Unlike intradomain splicing, interdomain splicing uses alternate routes already in the BGP routing tables to achieve path diversity. However, the data plane of the router needs to be modified to support path splicing.

Splicing bits As described before, an end system inserts splicing bits into the packet header; the ingress and egress routers in each AS read these bits to determine how to forward the packet, as shown in Figure 3.6. The ingress router learns multiple paths to

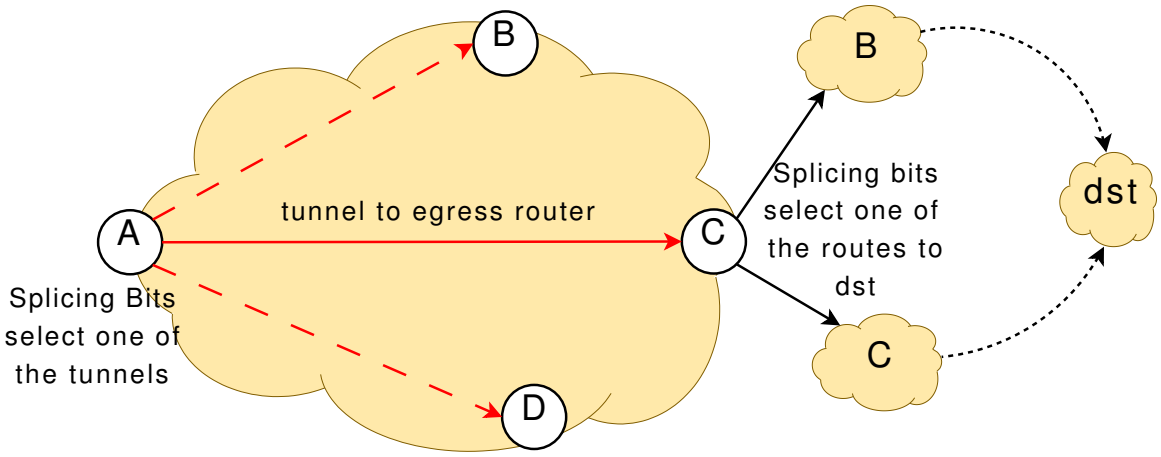


Figure 3.6: Interdomain splicing. The bits at the ingress router select the egress router to use. The packet is tunneled to the egress router and from there one of the external routes is used to forward the packet to a neighboring AS.

a destination prefix from the various border routers using iBGP (either via a "full mesh" iBGP or via its connections to multiple route reflectors) and thus may learn multiple exit points ("egress routers") from the network for each destination prefix. For each packet, an ingress router reads the rightmost $\lg(k)$ routing bits to determine which egress router should receive the packet and tunnels the packet to one of the egress routers. Similarly, an egress router learns multiple routes to a destination from the various border routers of the neighboring ASes via eBGP. It uses the rightmost $\lg(k)$ routing bits to determine which of the k eBGP-learned routes (*i.e.*, which FTE) to use.

As with intradomain splicing, the ingress or egress router removes the rightmost bits from the splicing header to allow the next router that supports interdomain splicing to read the next rightmost bits. Using this approach, an n-hop AS path requires $2n \cdot \lg(k)$ routing bits. To further reduce overhead, interdomain splicing can also use an encoding that is similar to those described in Section 3.5.2.

When a packet arrives at an AS's ingress router, that ingress router uses the bits to select one of the routes in its forwarding table for the corresponding destination. This operation also requires no additional modifications to BGP: it effectively corresponds to selecting an egress router from that AS (*i.e.*, the "next-hop" route attribute for different routes in

the table). The router then sends the packet along the IGP path to the destination, where intermediate routers may route to the same egress router; the path to that egress router may also, in fact, be spliced using intradomain splicing. When the packet reaches an AS's egress router, that router then inspects the bits to select one of the routes learned from BGP sessions to neighboring ASes.

When a packet is being tunneled to an egress router from the ingress router, the intermediate routers in the network along the path may use intradomain splicing to reach the egress router. Note that the same splicing bits could be used for intradomain as well as interdomain splicing. Splicing bits act as an index to the corresponding forwarding table entry; this function requires additional logic to read the splicing bits and to use them to select an entry from the corresponding forwarding table and use that for forwarding the packet.

Forwarding tables The data plane for interdomain splicing can be implemented in the following way:

- **Simple case: Multiple forwarding tables.** A router's routing table contains one route for each destination per BGP session. In interdomain splicing, routers select the k best paths from the RIB and insert these entries into forwarding tables on the router line cards, Interdomain splicing requires that the line cards provide support for multiple forwarding tables.
- **Optimization: Single forwarding table.** Creating k copies of the forwarding tables could introduce significant memory overhead on line cards, given the large (and growing) size of the default-free BGP routing tables. However, note that in many cases, the next-hop for a destination may be the same for different slices. In these cases, FTEs could be coalesced to save space, similar to how routers can coalesce FTEs for contiguous IP prefixes that use the same outgoing interface. In future work, we will study the extent to which this coalescing can reduce this overhead.

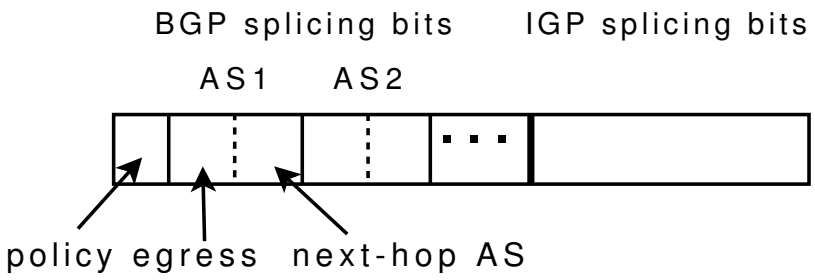


Figure 3.7: Structure of splicing bits for intradomain and interdomain splicing.

Interdomain and intradomain splicing Path splicing's *splicing bits* must direct traffic along an end-to-end path that ultimately traverses multiple domains. To achieve this function, these bits must carry semantics for both interdomain and intradomain paths as shown in Figure 3.7. Additionally, the interdomain paths that splicing takes must also comply with ISPs' business policies. To achieve this function, we divide the splicing bits into several segments. The first segment is for interdomain routing (*i.e.*, selecting at both ingress and egress routers which alternate paths to use); the second segment is for intradomain routing. We envision that the interdomain splicing bits will be used at each hop along the path to the destination; in contrast, the same intradomain bits can be re-used in different ISPs along the end-to-end path.

Finally, we use a single bit in the packet header to indicate whether the packet has traversed a "peer" or "customer" edge (in the parlance of Gao-Rexford [34]); if this bit is set, the interdomain bits can *only* be used to select a BGP route through a customer AS. Routers can easily implement this mechanism by dividing the forwarding table into two separate tables: routes to provider and peer ASes, and routes to customer ASes. A router sets this bit before it sends a packet along a customer or peer edge. With this additional bit set, the interdomain splicing bits will be used to select only routes from the latter forwarding table. This mechanism ensures that all interdomain paths are valley-free.

3.6.3 Loop Prevention and Detection

Because interdomain splicing constructs a single end-to-end interdomain path from multiple routing trees, interdomain paths can also have loops. Here we describe examples where interdomain splicing can introduce loops, as well as mechanisms for preventing them.

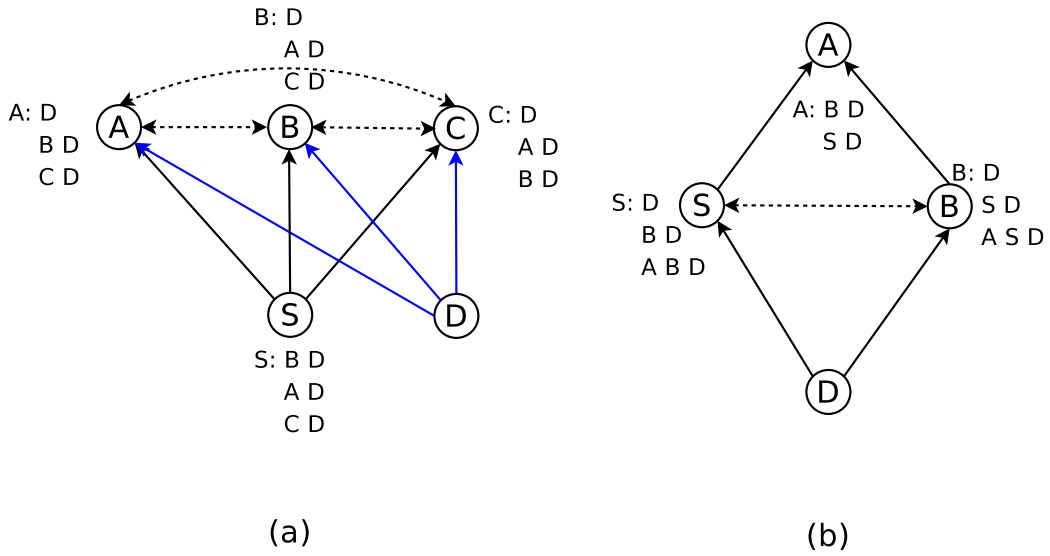


Figure 3.8: Splicing can cause loops: In the figure, S is the source AS and D is the target AS. The solid lines show the customer-provider kind of relationship among the ASes while the dotted lines show peering relationship among the ASes. On the side of each AS, the learned routes to AS D are listed. An AS picks one of the routes based on the routing bits in the packet header.

Loops can occur among peers or in customer-peer-provider relationships, as illustrated in Figure 3.8. In Figure 3.8(a), consider the scenario in which AS *S* chooses the route through one of its providers to forward the packet to *D*. When the packet reaches a provider (A, B, or C), even though each AS has a direct path to D, the routing bits may cause the packets to be forwarded among a sequence of peers, resulting in a forwarding loop. Note that such a loops can occur *even if standard preference and filtering rules are applied* [34], since the actual forwarding graph is an overlay of three separate policy-compliant routing trees. Figure 3.8(b) shows the possibility of a 3-hop loop between ASes *S*, *A* and *B* which can arise if *S* selects a path from its provider *A*, while *B* chooses a path through its peer *S*.

In each of these cases, a packet will not loop forever, because there are a limited number of routing bits in the header. Once the routing bits get exhausted, the packet is forwarded on the default loop-free BGP path. Nevertheless, in the interdomain case, we treat *any* loop as undesirable, since even a loop involving two ASes may traverse a significant distance. Accordingly, we propose the following two mechanisms to limit the extent and occurrence of forwarding loops.

Solution 1: Include AS Path in the packet One approach to detecting loops involves inserting 16-bit hashes of each of the first four hops of the AS path that the packet traverses in its header. We choose four since most paths in the Internet pass through four ASes or fewer [55]. An AS's border router can examine these bits and avoid selecting a next-hop AS that has already been visited by the packet (unless there is no other route available). This mechanism does not prevent loops altogether, but it does limit the extent to which packets can be caught in a loop. Of course, the mechanism only prevents short loops (*i.e.*, those less than four AS hops), but the average length of Internet paths and standard policy constraints (*i.e.*, preferring customer routes over peer routes, etc.) make long loops unlikely.

Solution 2: Deflection Counter To deal with larger loops, we introduce a 2-bit "counter" in the header. We observe that forwarding loops can only occur when a packet is deflected from a best customer (or peer) route to a peer/provider (or provider) route (*i.e.*, AS-level loops are not likely on spliced valley-free paths, except for the case of an all-peer loop). Accordingly, we introduce a *deflection counter* to limit the number of times a packet is deflected from its most preferred class of routes: If a router in some AS has a best path to a destination through its customer but it instead chooses a peer or provider path for forwarding a packet (or a provider route instead of a peer route), that router decrements deflection counter of the packet. This mechanism bounds the number of times the packet can be deflected and prevents a packet from being repeatedly forwarded "uphill" (which would be required for a persistent loop). An end system that has no tolerance for loops may

set this counter to zero; increasing this counter increases a router's flexibility in choosing paths that are not policy-compliant, at the cost of increased potential for routing loops.

3.6.4 AS-level forwarding consistency

In interdomain splicing, traffic might be forwarded along any of the top k best routes for a prefix, but the AS announces only a single best route to its neighbors. Some might view using a route that was not announced to its neighbors as a violation of protocol semantics, but we note that an AS will use a non-default path only if the splicing bits in the packet explicitly request this behavior or if the default path has failed. We also note that, even today, the AS-level forwarding path is by no means guaranteed to match the advertise AS path, and many such violations occur in practice [92].

3.7 Evaluation

This section evaluates path splicing in terms of the reliability it achieves, the ability to allow paths to quickly recover from failures of nodes and links, the latency stretch of the resulting paths, the reliability when only a fraction of ASes deploy it, the frequency of loops in spliced paths, and the effects on traffic. Table 3.1 summarizes the results of our evaluation. We find that path splicing provides high reliability and rapid recovery from failures and provides end systems access to a large number of low-latency, relatively loop-free paths. We also find that path splicing balances traffic across links in the network in much the same fashion as the "base" set of link weights and, to some extent, even balances this traffic slightly more evenly.

3.7.1 High Reliability

This section presents the results for reliability experiments performed with splicing for intradomain and interdomain networks. We find that, in both cases, path splicing achieves reliability that approaches that of the underlying network.

Table 3.1: Path splicing: Summary of results.

Result Summary	Section
Reliability with splicing approaches optimal. For intradomain splicing, 5 slices and for interdomain splicing, only 2 slices achieve near-optimal reliability.	3.7.1
Splicing has fast recovery. An end system can recover from failure in about 2 trials when trying splicing bits at random.	3.7.2
Perturbations achieve high novelty with low stretch. Intradomain splicing has an average stretch of 20% while gaining 80% paths which are different from the original. For interdomain, the average hop stretch is only 3.8% when 5% of AS links have failed.	3.7.3
Splicing provides better recovery than routing deflections. Path splicing with only 5 slices can provide better recovery than routing deflections [99] with bounded stretch. Path splicing generally provides much shorter recovered paths, and the recovered paths have much lower variance in terms of stretch.	3.7.4
Splicing is incrementally deployable. Splicing offers significant benefits even if only a fraction of ASes deploy it.	3.7.5
Loops are rare. Forwarding loops are transient and infrequent. In intradomain splicing, we observe only 1 loop longer than 2 hops and *no persistent loops*, even with 10% of links failed.	3.7.6
Splicing causes minimal disruption to traffic. Splicing does not have much adverse effect on traffic in the network. Our evaluation using real traffic data on Abilene shows that total load on links increases only by 4% on average.	3.7.7

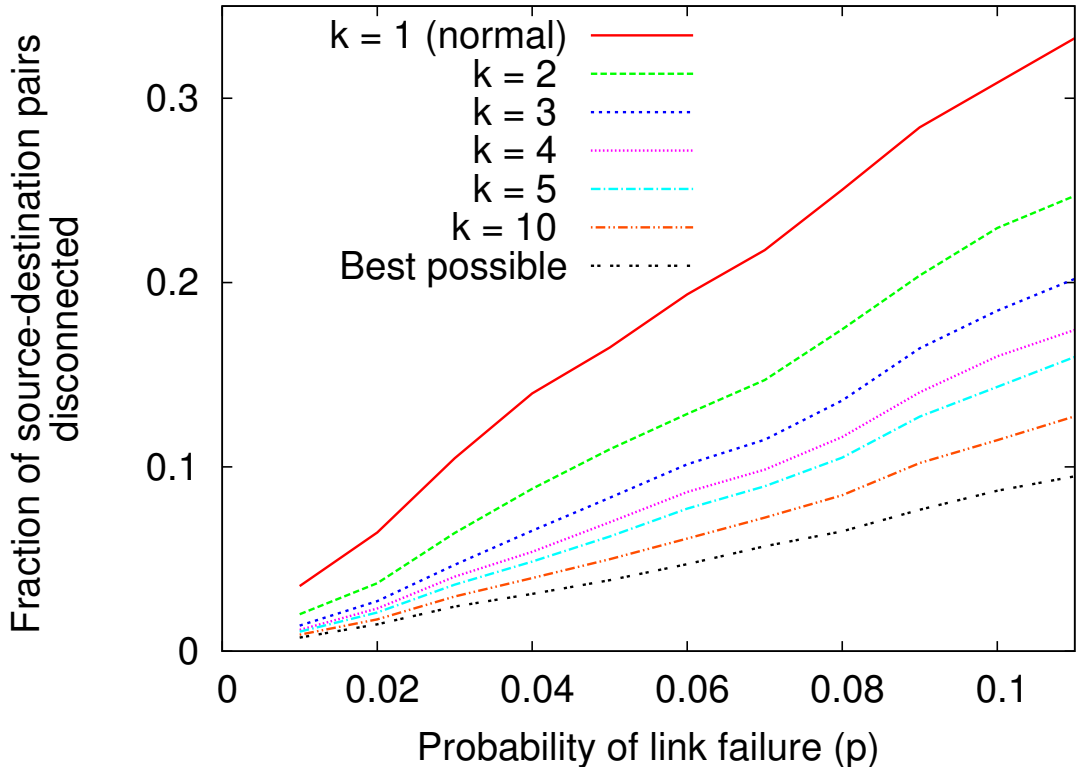


Figure 3.9: Reliability of path splicing for the Sprint topology.

Intradomain splicing To evaluate the reliability of path splicing under a variety of link-failure scenarios, we implemented a simulator that takes as input a "base" network topology (with link weights) and outputs the different shortest paths trees for that network using degree-based perturbations. To simulate link failures, we removed each edge from the underlying graph with a fixed failure probability. We used the Sprint backbone network topology inferred from Rocketfuel, which has 52 nodes and 84 links [83]. We computed the *reliability curves* for graphs generated using path splicing and compared this characteristic both to "conventional" shortest paths routing and to that of the original underlying graph, whose reliability reflects the *best possible* reliability that could be achieved by any routing protocol.

A spliced graph with k slices is constructed by taking the union of the k slices, each of which is a random perturbation, generated as described in the previous section. Next, we

Table 3.2: Sprint topology: Reliability for single node failures

Slices	Fraction of pairs disconnected
k = 1	0.0256
k = 2	0.0204
k = 3	0.0187
k = 5	0.0176
Best possible	0.0171

remove each edge from the graph independently with probability p. We start with $k = 1$, evaluate the reliability for the resulting graph, increase k to 2 (*i.e.*, add edges to the graph by taking the union of the two graphs) and evaluate the reliability of the resulting graph by failing the *same set of links* (simulating the effects of a link failure in the underlying network). We perform this process $1,000$ times; in other words, for each k and p, we construct a k-slice graph with appropriate edges "failed", and compute the average reliability for those $1,000$ trials.

Figure 3.9 shows the reliability curves for Sprint using degree-based perturbations with Degree-based$(0,3)$. Adding just one slice (*i.e.*, increasing k to 2) significantly improves reliability; adding more slices improves reliability further. Figure 3.9 demonstrates that even with just a few slices (*i.e.*, 5) and a simple scheme for generating alternate graphs (*i.e.*, link-weight perturbations), *the reliability of path splicing approaches the reliability of the original underlying network.* We also performed a reliability experiment for single node failures and found similar results. Table 3.2 summarizes the results for the experiment.

Interdomain splicing To evaluate the reliability of interdomain splicing, we used C-BGP [72], an open-source BGP routing solver. C-BGP takes as input a policy-annotated graph of ASes and calculates the interdomain routes for each AS. For our experiments, we use a 2,500 node policy-annotated AS graph generated by Dimitropoulos *et al.* [25]. Once C-BGP computes the interdomain routes, we removed AS edges at random with probability p. Next, on this modified AS graph, we checked for connectivity between random pairs of

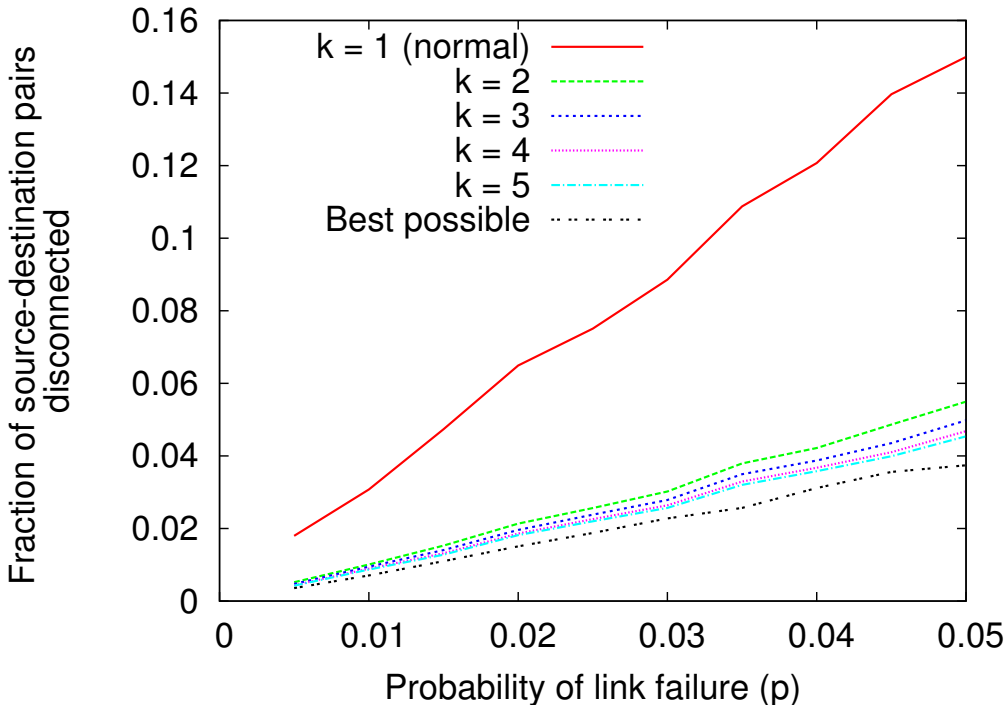


Figure 3.10: Reliability using a 2,500 node policy-annotated Internet AS graph.

ASes in the graph (testing reliability for all pairs is not tractable).

In cases where the default path was disconnected, we checked to see if a "spliced" path existed for the disconnected AS pair using up to k choices for the next-hop. We repeated this process 50 times for each value of p and k. Figure 3.10 shows the average fraction of pairs disconnected for a range of values for p and k. We observe that adding just one more slice significantly improves the reliability of the AS graph. For the "best possible" case, we evaluated reliability for the base graph (without policy restrictions). The reliability curve for interdomain splicing that respects policy is so close to the *best possible* reliability curve, which demonstrates that BGP, even with policy restrictions, has near-optimal path diversity if multiple routes are used. Path splicing can thus exploit this diversity without violating AS-level policies or any modifications to BGP message format.

3.7.2 Fast Recovery

In this section, we demonstrate how an end system or a network node can quickly recover from failures by selecting spliced paths in the network at random. We evaluate two approaches to recovery: *end-system recovery* is network-agnostic and relies on the end system (*e.g.*, user, proxy, edge router) to initiate recovery; *network-based recovery* assumes that the node in the network can detect a failure on an incident link and initiate recovery by diverting traffic to a different slice. To generate a spliced graph with failures on the Sprint topology, we use a simulation setup similar to the one for the intradomain reliability experiment in Section 3.7.1. We only show results from *end-system recovery*.

For all disconnected source-destination pairs, we evaluate whether splicing allows pairs of nodes to discover working alternate paths. If splicing can recover the path in five or fewer trials (we assume that the end system or node could run these trials either in sequence, in parallel, or even in advance), we consider the path recoverable. As discussed in Section 3.2, our simulations do not allow us to explicitly compute recovery time in terms of seconds, but we can estimate what this time might be from the number of trials: Because it would take about one round-trip time to estimate whether a new set of splicing bits resulted in a functional path, we can estimate the recovery time as the number of trials times the round-trip time, divided by the number of trials that the system makes in parallel.

If a user typically makes the attempts one after another till he/she finds a working path, then the recovery time would be the number of attempts times the time for each trial. If the number of attempts are small, then the user could also make the trials in parallel and the recovery time would simply be the time needed to make a trial. The time to make a trial translates to the time spent by the end-user after sending a packet to conclude that the path is "non-workable". TCP typically declares a data packet as lost, if the ACK is not received within the time-out interval which is a function of the round-trip time (RTT) of the path. Thus, recovery time would be at least the RTT of the path.

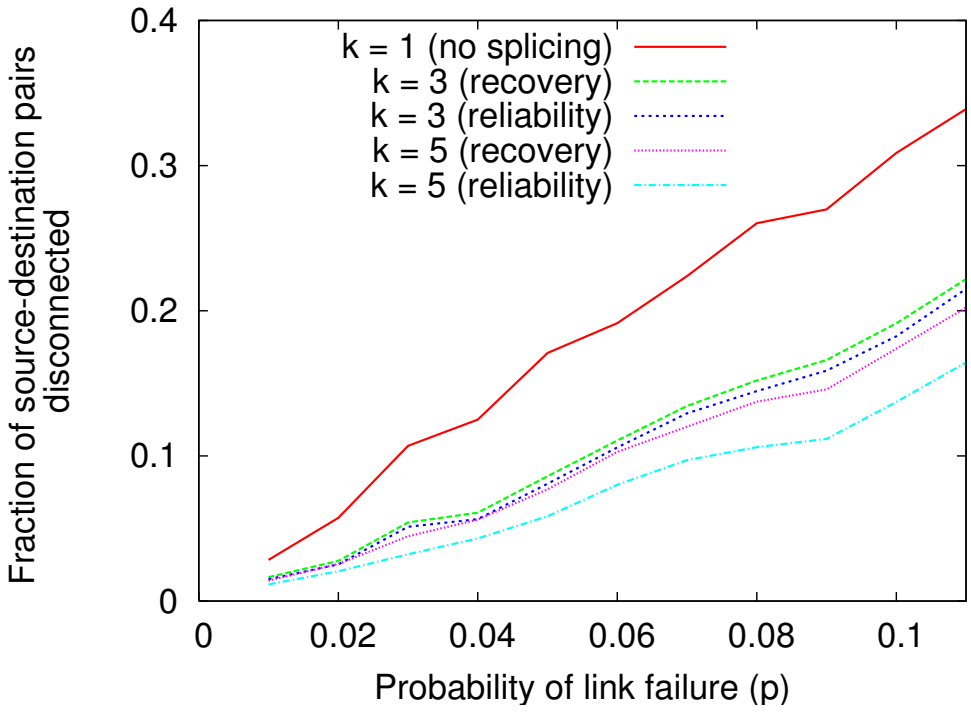


Figure 3.11: Recovery using *end-system recovery* and Sprint topology.

End-system recovery. Figure 3.11 shows the recovery where the end system controls the spliced path to the destination. In our experiments, we used a header that allows 20 hops to be spliced. For a failed path, the new shim header (*i.e.*, the splicing bits) is constructed as follows: A coin is tossed for every hop in the shim header; if the result is a head, a different slice is selected at random for that hop (*i.e.*, at every hop we switch slices with 0.5 probability). We check to see if a failed path can be recovered in fewer than 5 trials. The average number of trials in any case where splicing could recover from the failure was slightly more than 2. Paths were on average 1.3 times longer (in terms of path cost) compared to the shortest path in the "base" topology; the resulting paths typically used about 50% more hops compared to the original shortest path. In any particular slice, 99% of all paths in each tree had stretch less than 2.6. Figure 3.12 shows recovery for interdomain splicing. The recovery is slightly worse because we consider only policy-compliant paths as recoverable. These results show that splicing provides effective recovery, even with the simplest possible recovery scheme and no knowledge about the location of failures.

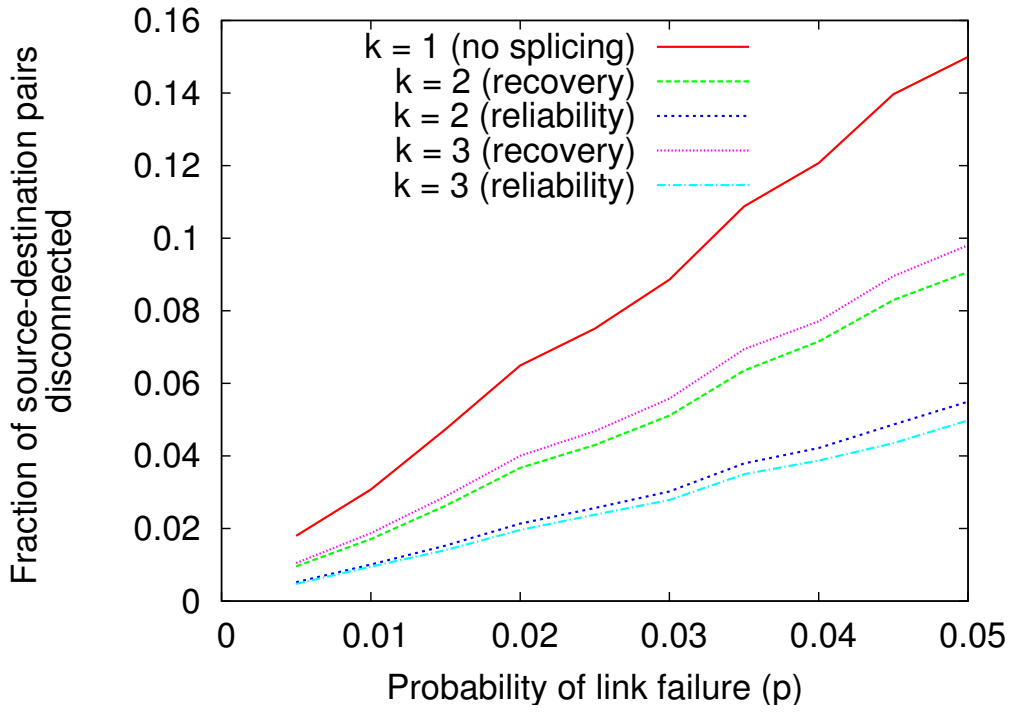


Figure 3.12: Recovery using *end-system recovery* and a 2,500 node policy-annotated Internet AS graph.

To understand how these recovery numbers compare to a simpler scheme that simply tries to recover by using one of k paths at the source (closer to what a simple multipath scheme might do), we compared path splicing to a recovery scheme that selects one slice at the first hop and does not switch at intermediate hops. We found that splicing's end-system recovery still exhibits slightly better recovery: With 2 slices and a 10% failure probability, splicing was able to recover about 7% more paths. This margin may, in some cases, not justify the additional cost of path splicing, but path splicing may also be able to perform better with a more sophisticated recovery scheme that uses specific information about the location of network failures.

Network-based recovery. Figure 3.13 shows results for network-based recovery: When a router x receives packets destined to d with next-hop y and discovers that link (x, y) has failed, it finds in its forwarding table an alternate slice with a connected next-hop for d (if

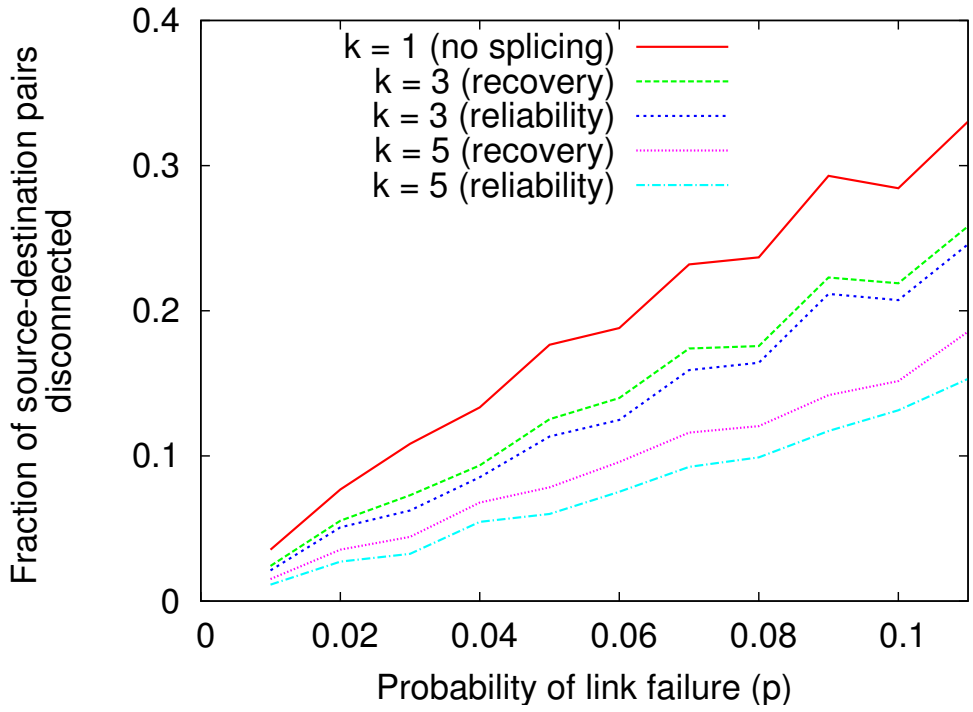


Figure 3.13: Recovery using *network-based recovery* and Sprint topology.

one exists). If a path between two endpoints is discovered using this process we consider the path recoverable. All paths between connected endpoints need not be recoverable since the packet could end up in a dead-end from where there is no connected next-hop to reach the destination, due to the specific slices selected by the routers. The average stretch for network-based recovery was 1.33; there were 55% more hops in the recovered paths.

3.7.3 High Novelty, Low Stretch

Recall from our design goals in Section 3.2 that the paths generated in each slice should have low stretch and high novelty. Our evaluation shows that, for intradomain splicing, random perturbations achieve reasonable novelty while keeping the stretch of each slice—and the stretch of the overall spliced paths—low.

Intradomain splicing. We show the results of our stretch and novelty experiments using the Sprint topology. We vary the Weight(a,b,i,j) function from Equation 3.1 (Section 3.2)

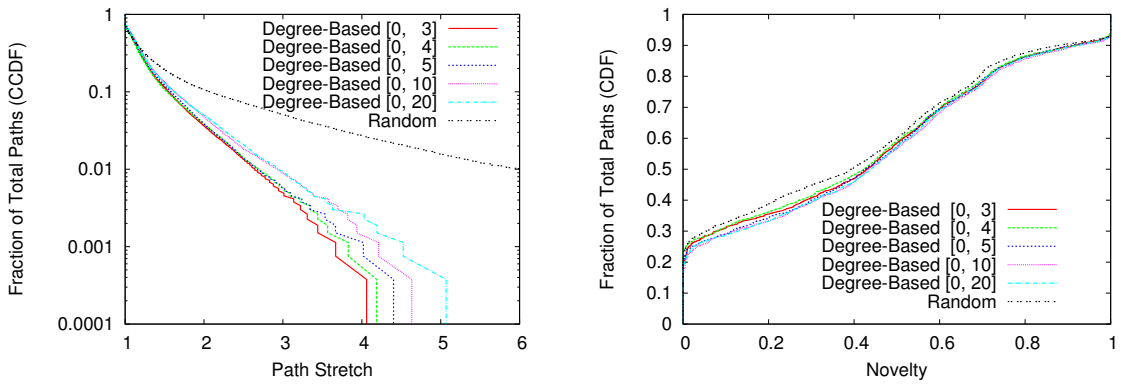


Figure 3.14: Stretch and novelty for *degree-based* perturbations of the paths in the Sprint topology.

and observe its effects on novelty and stretch. We also compared the results of degree-based perturbations with the random case in which link weights are set randomly in the range of [0, 5000]. For these experiments, we ran the simulator to generate 100 different slices for different values of b with $a = 0$, in Weight(a,b,i,j), which controls the magnitude of the perturbations.

Figure 3.14 shows the stretch and novelty for the Sprint topology with degree-based perturbations; each line reflects a different Weight(a,b,i,j) function. Degree-based perturbations achieve almost as much novelty as random link weight settings, but with far less stretch (particularly in the worst case). For example, in the case of *Degree-Based*$[0,3]$, the average stretch is only 1.2; the worst-case stretch is also far better than the random link-weight settings. In fact, only about 3.5% of paths have stretch of more than 2. The corresponding average novelty value for the slices for degree-based perturbations is 0.41 and 80% of paths have one or more links different than those in the original shortest paths. Increasing the value of the Weight() function results in small improvements in novelty but higher stretch.

Uniform perturbations also have low stretch, but they provide less novelty than degree-based perturbations. For example, the average stretch for the case of Weight$() = 1$ is only 1.03. The corresponding average novelty for this case is 0.22. On average, 57% of paths

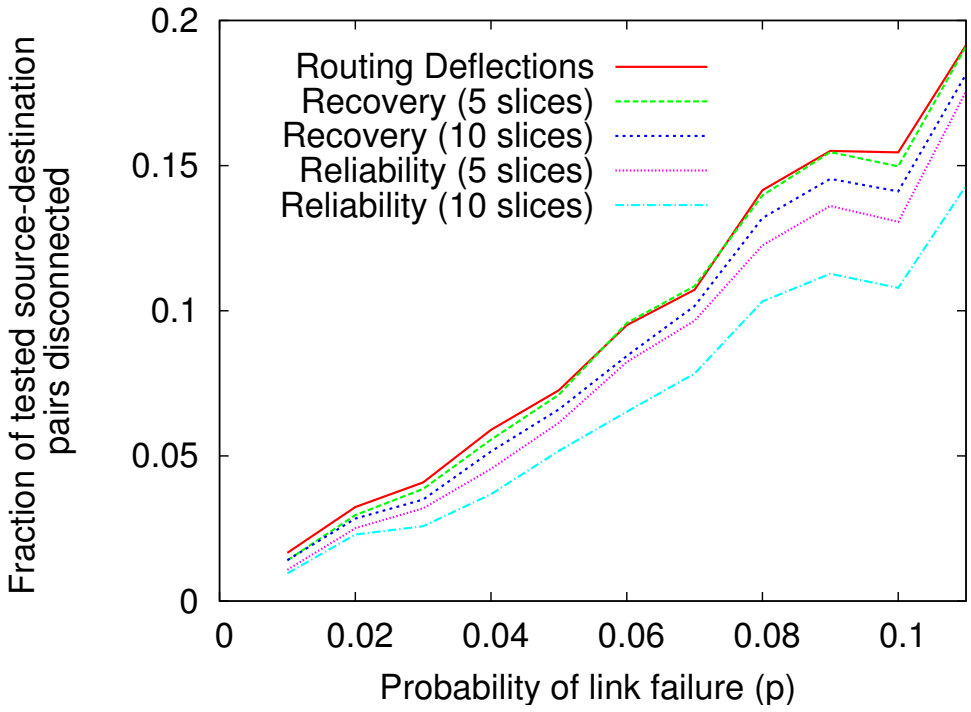


Figure 3.15: Comparison of recovery for splicing vs. routing deflections with stretch < 2.

differ by one link or more from the original shortest paths.

Not only is the stretch of the paths in each slice low, but the stretch of the actual spliced paths after recovery is also low. In the case of end-system recovery, paths were on average 1.3 times longer in delay compared to the shortest path in the "base" topology; the resulting paths typically use about 50% more hops compared to the original shortest path. In any particular slice, 99% of all paths in each tree have stretch of less than 2.6. The average stretch network-based recovery was 1.33, while there were 55% more hops in the recovered paths; these numbers are slightly higher compared to the end-system recovery scheme.

Interdomain Splicing. We computed the average hop-count stretch for the interdomain reliability experiment in Section 3.7.1. The hop-count stretch with 5% of the AS links failed was only 1.038, or 3.8% more hops than in the default AS paths.

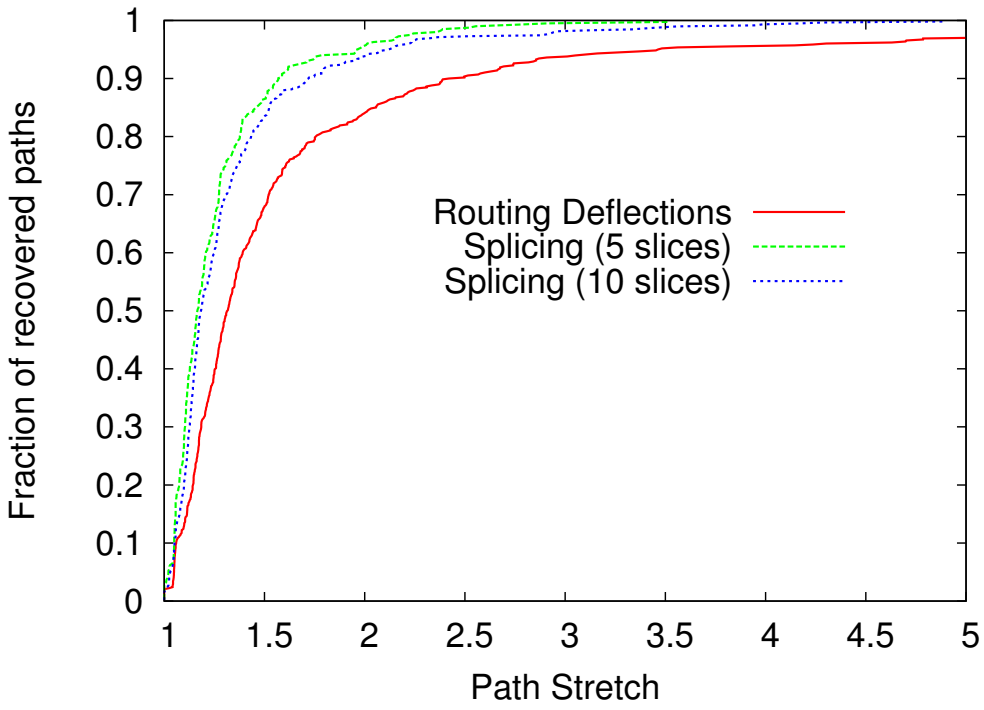


Figure 3.16: Comparison of stretch for recovered paths for splicing vs. routing deflections.

3.7.4 Comparison to Routing Deflections

We compared the end-system recovery achieved by intradomain path splicing to that achieved by the routing deflection mechanism proposed by Yang *et al.* [99]. We re-implemented the deflection routing system and compared the reliability achieved by this scheme to that achieved by path splicing. Previous work on routing deflections does not consider the stretch of the resulting paths and considers *all* possible recovered paths. With routing deflections, the number of neighbors that a node can potentially send a packet to is not bounded, whereas in path splicing it is bounded by the number of slices; hence, routing deflections may require significantly more storage. To provide a fair comparison between the two schemes, we consider a path "recovered" only if it has a stretch of less than 2. Figure 3.15 shows the recovery achieved by path splicing for different numbers of slices compared to routing deflections. Path splicing recovers more paths than routing deflections using just five slices.

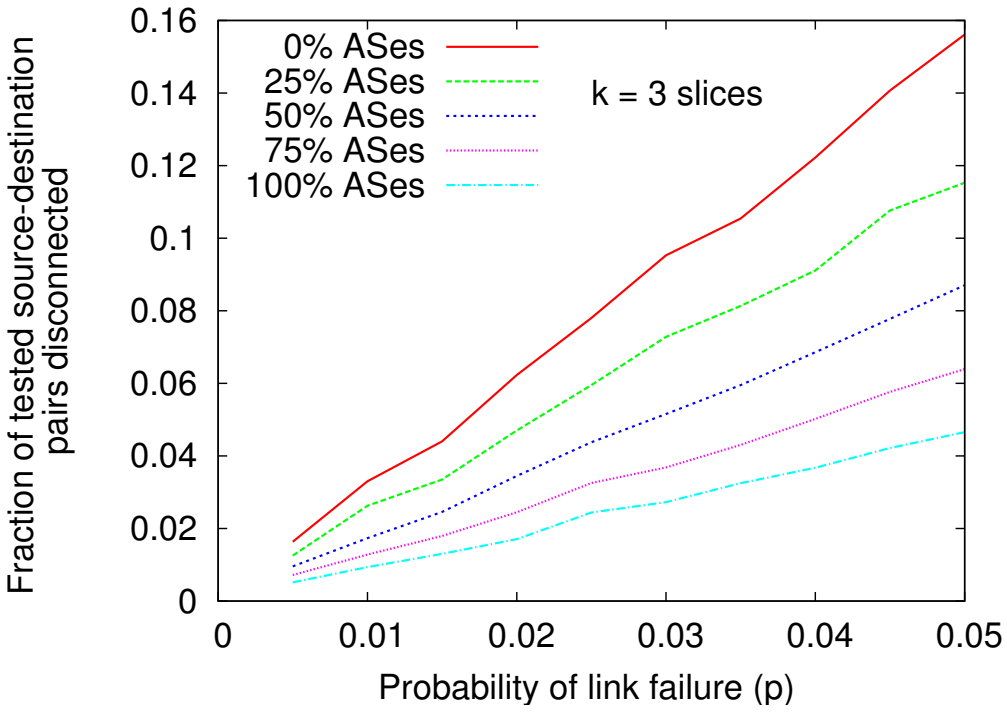


Figure 3.17: Interdomain path splicing: Incremental deployment.

In addition to directly comparing recovery, we compared the stretch of the recovered paths using each of the schemes for this experiment. Figure 3.16 shows the resulting statistics. The results show that path splicing can recover paths that have lower stretch than the stretch of the paths recovered using routing deflection. Path splicing generated paths with an average stretch of 1.26, whereas the path stretch using routing deflections was 1.78. Path splicing also generates shorter paths more consistently: the variance of stretch values for paths generating using path splicing was 0.09; in contrast, the variance of stretch for recovered paths using routing deflections was 4.83.

3.7.5 Incremental Deployability

Interdomain splicing requires ASes to independently decide to deploy additional functionality. It is reasonable to ask, then, how well interdomain splicing would perform if only a fraction of ASes deployed it. Our experiments show that path splicing provides significant

benefits even if only a small fraction of a fraction of ASes deploy it. To evaluate the benefits of partial deployment, we use the same AS topology as in the interdomain reliability experiments. We fixed the number of slices and performed the reliability experiment as before; for each experiment, we let only a fraction of ASes select an alternate AS-level path if the next-hop on the default route has failed. We evaluate reliability for five levels of deployment: 0% to 100% with 3 slices, as shown in Figure 3.17. Reliability improves significantly even if only 25% of the ASes deploy interdomain splicing. We expect that the benefits might be even higher if all "Tier-1" ISPs deployed splicing.

3.7.6 Infrequent (and avoidable) Loops

Because traffic is not forwarded along a single routing tree, splicing does create the potential for transient forwarding loops if some precautions are not taken. Forwarding loops are a concern because they increase the total length of the end-to-end path, and they also unnecessarily use extra network capacity and node resources (note that these detriments are the same as paths with longer stretch; we have already shown that spliced paths have reasonable stretch).

Fortunately, certain recovery strategies can avoid *persistent* forwarding loops entirely. First, a persistent loop would require the splicing bits to be repeated in exactly the right sequence. Second, in the design we presented in Section 3.5.2, the splicing header will eventually run out of splicing bits as each node shifts $\lg(k)$ bits from the header; at this point, the packet stays in the same tree to the destination. Second, paths that never switch back to a previously used slice would never contain persistent forwarding loops of any length; recovery strategies could pick only these paths. Although it would not necessarily prevent transient loops entirely, restricting the number of switches between slices that any packet takes would also limit the likelihood of loops significantly. Our evaluation shows that loops were quite infrequent. Using network-based recovery, there was less than 1 loop on average with length greater than 2 when recovering from the case where the network

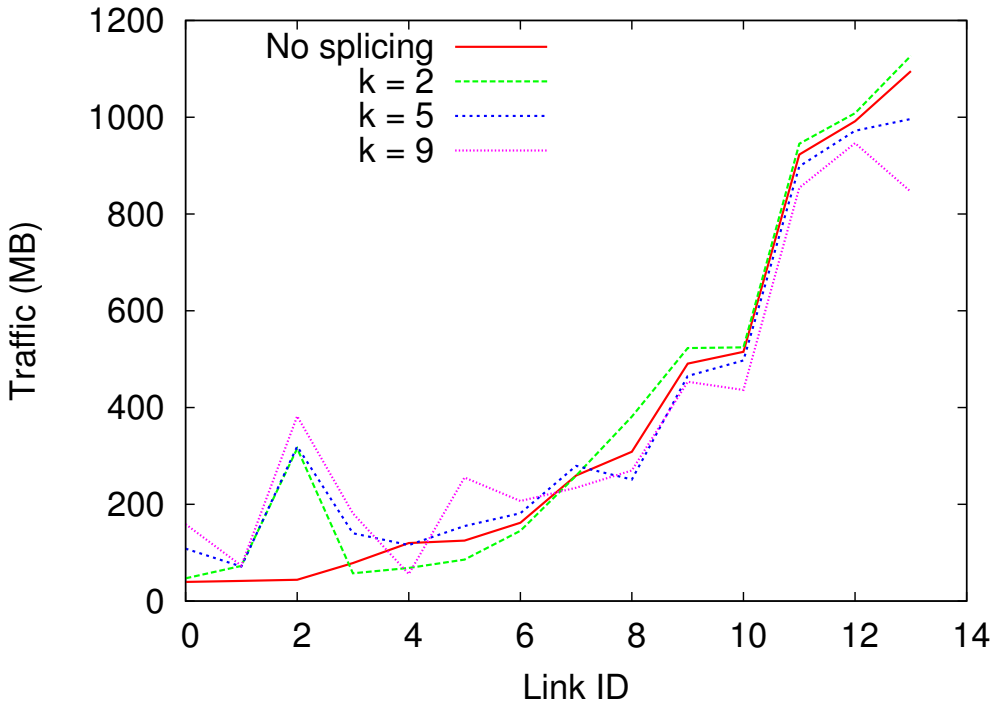


Figure 3.18: Abilene Network: Effect of splicing on traffic in the network using real traffic traces.

had 10% of links failed. Two-hop loops occurred more frequently (about one per 100 trials for $k = 2$, and about one in ten trials for higher values of k). Using any of the schemes discussed above could eliminate loops entirely, at the cost of restricting the paths available for recovery.

3.7.7 Minimal Disruption to Traffic

We studied the effects of splicing on traffic loads within a single ISP. We extended C-BGP to support intradomain path splicing and provided C-BGP with BGP routing tables, IGP configurations, and NetFlow traffic traces for the Abilene network; we then used it to determine the traffic load on each link in the network in the default case and for various instantiations of splicing. Abilene has only 11 nodes and 14 links, but we ran our experiments using this network because it makes routing and traffic data publicly available.

For the experiment, we create k slices for the Abilene topology in C-BGP; we used degree-based perturbations to generate the slices. C-BGP computes shortest paths for each

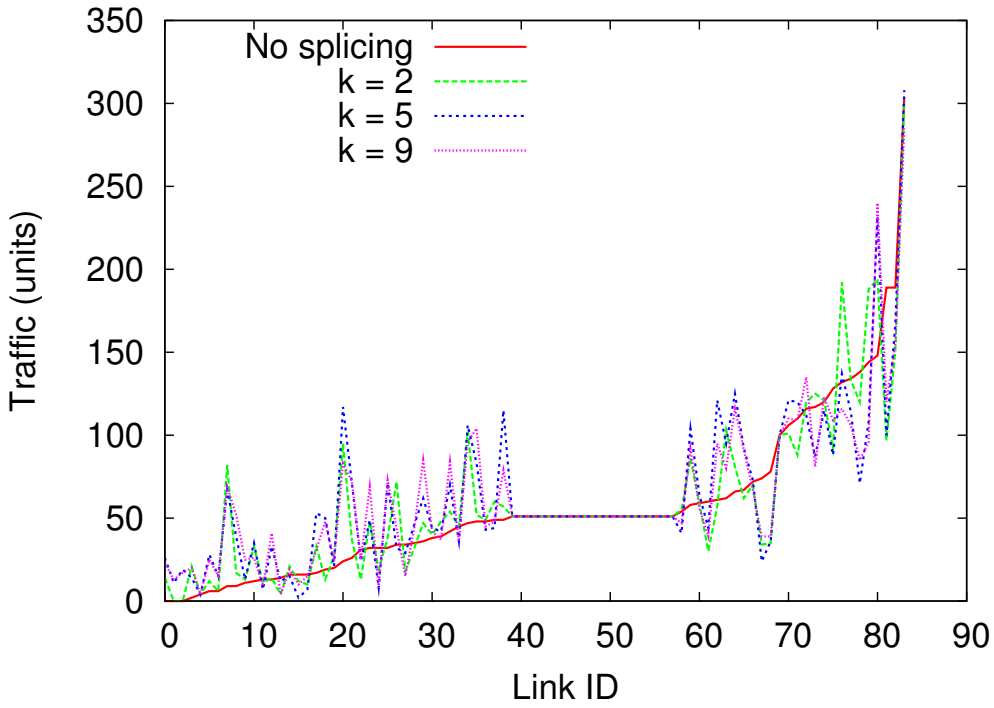


Figure 3.19: Sprint Network: Effect of splicing on traffic in the network using synthetic traffic.

slice and loads the routes into the respective forwarding tables on each of the nodes. Next, we load the BGP routing table dumps obtained from Abilene on each of the nodes. We then "play" 5-minute NetFlow traces through the network; we load a NetFlow trace onto each node that corresponds to the traffic collected from the node in the actual Abilene network.

For every packet reflected in the trace statistics, C-BGP selects a slice based on the hash value of the source and destination IP addresses in the packet. So traffic is split randomly among the k slices. Figure 3.18 shows the resulting link loads.[4] We also performed a similar experiment using the Sprint topology and a synthetic traffic matrix, which consisted of unit traffic for all node pairs. Figure 3.19 shows the results of this experiment.

The plots sort links on the basis of their load in the case without splicing and show the corresponding load on the same links using splicing. The plots demonstrate that splicing does not cause significant adverse effects on traffic. Splicing can increase stretch if traffic is

[4]We repeated the experiment with different 5-minute NetFlow packet traces and found similar results.

routed on paths other than the shortest path in the network. As a result, the sum of the load on the links in the network will be higher when using splicing. Fortunately, the utilization is not that much greater: the sum of the load on the links is on average only about 4% higher (and never more than 10% higher) than without splicing. In the Sprint network, traffic under splicing is 9% higher on average (and never more than 12%).

3.8 Proofs

In this section, we present sketches of proofs to show how splicing is ability to achieve high reliability with a low, bounded stretch and also that the probability of forwarding loops occurring on spliced paths is rare. Our analysis shows that the number of slices required to achieve near-optimal connectivity with bounded stretch scales well with the size of the graph. Specifically, Theorem 3.8.1 shows that the number of slices required to achieve connectivity that is close to that of the underlying graph scales as $\log n$, where n is the number of nodes in the graph.

3.8.1 Reliability Analysis

Fix a maximum allowable stretch D. Then, for each pair of vertices s,t, we consider the subgraph $G(D,s,t)$ induced by paths of length at most D from s to t. Let $\chi_G(D,s,t)$ be the connectivity of this graph and $\chi_G(D) = \min_{s,t} \chi_G(D,s,t)$. We show that the connectivity of the paths used by path splicing approaches $\chi(D)$ (*i.e.*, that of the underlying graph).

Theorem 3.8.1. *Let H denote the union of k shortest path trees to a destination t, each obtained from a graph G by independent random perturbations of the link weights uniformly in the range $(L, 2DkL)$. Then for any $k > c_0 \log n$, with high probability, the connectivity of H is at least $c_1 \chi_G(D)$ where c_0, c_1 are universal constants and n is the number of nodes in the graph.*

We only give the idea of the proof here. We argue that every cut of H has $\Omega(k)$ edges. This uses two ideas: (1) there are at most $n(n-1)/2$ mincuts in an undirected graph with

n vertices and at most $2^l n(n-1)/2$ cuts of with at most l times the minimum number of edges (2) An cut C with $|C|$ edges has an edge subset set of support $\Omega(|C|)$ with the property that each edge is chosen roughly uniformly in a shortest path tree with perturbed weights. Combining (1) and (2) along with a Chernoff bound gives the claimed result.

3.8.2 Stretch Analysis

In this section, we show that stretch is bounded and that, as a consequence, long forwarding loops are unlikely.

Theorem 3.8.2. *Assume the perturbations of a link i with original weight L_i are uniform in the range $[-cL_i, cL_i]$. Consider a packet traveling from source s to destination t that has made m hops of perturbed lengths $L' = (L'_1, \ldots, L'_m)$ on a single slice and reached a node u. Let P be a shortest path from s to t. Then, for any $r > 1$,*

$$\Pr\left((1-c)d(u,t) \leq ||P||_1 - ||L'||_1 + \frac{rc}{\sqrt{3}}||P||_2\right) \geq 1 - \frac{1}{r^2}.$$

Proof. We begin with a simple probabilistic bound on the perturbed length of any fixed path. Let X_i be the perturbed length of a traversed link with original length L_i. Then $\mathsf{E}(X_i) = L_i$. Further,

$$\begin{aligned}\mathsf{Var}(X_i) &= \mathsf{Var}(X_i - L_i))^2 \\ &= \mathsf{E}((X_i - L_i)^2) - \mathsf{E}(X_i - L_i)^2 \\ &= \mathsf{E}(Y_i^2)\end{aligned}$$

where Y_i is uniform in the interval $[-cL_i, cL_i]$. Thus,

$$\begin{aligned}\mathsf{Var}(X_i) &= \frac{\int_{-cL_i}^{cL_i} y^2\, dy}{2cL_i} \\ &= \frac{c^2}{3}L_i^2.\end{aligned}$$

Let $X = \sum_{i=1}^m X_i$. Using Chebychev's inequality, we have

$$\Pr\left(|X - L| \geq r\frac{c}{\sqrt{3}}|L|_2\right) < \frac{1}{r^2}.$$

Let $d'(.,.)$ denote the shortest path distances with perturbed weights in the current slice. Let P be a shortest path between s and t with the original weights.

$$\begin{aligned} d(u,t) &\leq \frac{1}{1-c} d'(u,t) \\ &= \frac{1}{1-c}(d'(s,t) - ||L||_1) \\ &\leq \frac{1}{1-c}(d(s,t) + \frac{rc}{\sqrt{3}}||P_{st}||_2 - ||L||_1) \end{aligned}$$

where the equality follows from the property of a shortest path and the inequality holds with high probability using the above analysis. □

3.9 Discussion and Open Issues

This section explores the changes both to hosts and to routers that would be required to deploy and evaluate various aspects of splicing in practice (*e.g.*, recovery time).

Changes to routers. Path splicing requires changes to the forwarding plane in routers in order to support multiple routes for a destination and the ability to select one of those routes based on the splicing bits. Recently, multi-topology routing has been standardized [70], and router vendors are also supporting this function [22, 56]. The basic forwarding mechanism required for splicing is very similar to multi-topology routing. We expect that the data-plane implementation of splicing will entail only a small extension to MTR. Additionally, we have developed a Click element that uses bits in the IP ID and type of service fields and to index into separate forwarding tables generated by the path splicing control plane; we plan to use this in conjunction with the changes to end systems described below to evaluate the recovery time of splicing in practice.

Changes to end systems. Path splicing relies on a failure detection mechanism before it can find a new working path. As we discussed in Section 3.7, detection could take place either at the routers themselves (as it is done today with other recovery mechanisms, such as fast reroute) or at end hosts (which might allow for recovery from different classes of

"failures", such as paths that exhibit high packet loss or jitter, as well as those that might exhibit complete outages). Instrumenting applications to take advantage of path splicing will require designing and developing mechanisms for receiving information about path quality as well as an extension to the sockets API for setting splicing bits in the packet headers.

Automatic tuning. Previous work has examined ways to tune network routing protocols to achieve desirable properties (*e.g.*, low congestion) [32, 33], but these mechanisms are offline and potentially quite sensitive to small changes in the network topology (*e.g.*, link or node failures). Similarly, planned maintenance events on links or nodes in the network require careful manipulation of the routing protocol parameters to minimize disruption and reduce the possibility of congesting certain links. By generating multiple non-overlapping routing trees on the same topology and splitting traffic across those trees, a single link failure (or planned maintenance event) will, on average, disrupt fewer end-to-end paths (we examine this property empirically in Section 3.7).

Alternate slicing and recovery mechanisms. Rather than generating slices at random, each slice could be configured with some consideration of the edges that were already covered by other slices. Other approaches to generating backup trees (*e.g.*, multi-router configuration, multi-topology routing) might be used to achieve reliability with fewer slices. Similarly, other approaches for setting splicing bits could result in even faster recovery; particularly if end systems have additional information about the location of a failure along a path from auxiliary monitoring systems [31, 58].

Adversarial concerns. An adversary could set splicing bits that send packets into a forwarding loop, thus wasting resources. This attack seems unlikely, because it requires an adversary to actually discover splicing bits that will induce a loop. An adversary cannot use the splicing bits to create arbitrary loops. Path splicing gives end systems some control

over the paths that traffic takes, which introduces the possibility that all end systems will react the same way upon seeing a faulty network path. If all end systems choose the same backup path when a link or node fails, the resulting traffic shifts could introduce congestion on certain links. Because each end system selects a new sequence of forwarding bits at random, we expect that traffic will disperse evenly across the network topology upon failure recovery; still, examining the effects of failures on traffic dynamics deserves further study.

3.10 Summary

In this chapter, we presented the design and evaluation of *path splicing*, a primitive for increasing reliability by composing routes from multiple routing protocol instances. Path splicing has three salient features that can be applied to any routing protocol: (1) Nodes run multiple routing protocol instances (or, alternatively, a single instance with many variants of the underlying configuration) to obtain alternate paths to each destination; (2) Intermediate nodes can forward traffic to the destination on any of these slices, effectively allowing traffic to "switch midstream"; (3) End systems can switch the path along which their traffic is sent using opaque bits in the packet header.

We have applied path splicing to both intradomain and interdomain routing and evaluated its ability to allow end systems to find alternate paths when links fail. Our experiments show that running just a few slices in parallel allows path splicing to achieve reliability that is close to that of the underlying graph (*i.e.*, as long as endpoints remain connected in the underlying graph, there will be some spliced path that connects them). We have also demonstrated that even simple recovery schemes, such as switching slices without prior knowledge of failure at intermediate hops, allows end systems to realize this reliability using alternate paths with small stretch.

Path splicing can be deployed on existing routers with small modifications to existing multi-topology routing functions. We also foresee many possible applications to other routing protocols (*e.g.*, wireless, overlay routing) and to many other applications that could

take advantage of having access to multiple paths in parallel. In Chapter 4, we present several implementations of path splicing, in both software and hardware platforms. We also demonstrate the simplicity and flexibility of the path-bits interface by implementing the splicing path selection using path bits. We discuss extensions of path splicing in datacenter environment in Chapter 6.

CHAPTER IV

NETWORK AND END SYSTEMS SUPPORT FOR PATH BITS

4.1 Introduction

We have so far presented the motivation and design of *path bits* in Chapter 2 and the design of a multipath routing primitive, *path splicing*, that can be easily mapped to a path-bits interface, in Chapter 3. In this chapter, we present the implementation of the narrow waist. We explore two questions: (1) Can an opaque path-selection mechanism be generalized to support many multipath routing protocols and implementations (in both hardware and software)? and (2) Can end systems use an opaque path-selection mechanism to effectively and quickly discover good alternate network paths in practice, on real Internet paths for real applications? Our evaluations show that "blind" path selection works well: For example, on the Level 3 Rocketfuel topology with a 5% link-failure rate, multipath routing using the path bits interface required only 1.3 more attempts on average to recover from a failed path than a technique than an optimal approach that knew the exact failure location.

In particular, our "narrow waist" design achieves two goals:

1. *Decouple the end systems and multipath routing mechanism so that multipath mechanisms can evolve independently from the applications that use them.* We evaluate this goal by implementing several multipath routing mechanisms that expose the simple path bits interface. Our evaluation shows that many existing multipath routing implementations can be controlled using path bits, and that path bits affords a simple design and implementation of these mechanisms. We also describe how to implement path bits by using existing fields in the IP header.

2. *Provide a simple interface to applications that allows them to achieve application-appropriate benefits from multipath.* To evaluate this goal, we show how several

existing applications can be easily modified to take advantage of the path bits interface. We also show that, if path bits are managed by the kernel, the path bits interface allows existing applications to use multipath routing with no modifications whatsoever.

We make three contributions. First, we propose a generic interface between a network that supports multiple paths and for end-systems to be able to efficiently select and use those multiple paths. Second, we demonstrate the simplicity and efficiency of the interface by implementing a number of recent multipath routing architectures on a variety of software and hardware platforms. Third, we implement extensions to the end host to support path bits manipulation in the kernel; using that interface we implement simple monitoring techniques that can benefit from the path bits interface to make intelligent path selection decisions and demonstrate the utility of such a framework for applications like failure recovery, improved throughput and better reliability for real-time applications. We show how the path bits interface at the end-system could be integrated with either a passive or active monitoring framework.[1]

The rest of this chapter is organized as follows. Section 4.2 presents implementations of different multipath routing schemes using the path-bits interface. Section 4.3 presents our prototype implementation of supporting path bits at end systems and describes our path monitoring implementations. Section 4.4 presents our evaluation of the path monitoring schemes and Section 4.5 concludes the chapter.

4.2 Supporting Path Bits in the Network

We describe the mechanisms that network devices need for supporting path bits, as well as the implementations we have done on both software and hardware platforms.

[1] Some of the multipath schemes we consider defer forwarding decisions to end hosts rather than networking elements [54, 96, 44]. Path bits applies to these approaches as well, although in our exposition, we focus on end-host path selection.

4.2.1 Network Support

Forwarding devices along the path interpret the path bits in a packet to direct traffic to the appropriate outgoing interface. We provide network support for different multipath routing protocols by having the bits serve as an index into different forwarding table entries and changing *how* the path bits are used to index into the different forwarding table entries. Routing deflections [99] could be implemented at a router by using the path bits to switch forwarding to a pre-computed next-hop that is closer downstream to the destination. Path splicing [60] could be implemented by using the bits as an index into one of k pre-computed slices. Pathlet routing [36] could be implemented by pre-computing labels for pathlets, and setting up those paths in the forwarding tables, and using the bits to index into different pathlets. A variant of ECMP could be implemented by installing multiple paths of almost equal length into the forwarding table and using the path bits to select one of multiple possible next hops (see Table 2.1).

A key design decision for implementing the narrow waist at network devices is *which* bits the devices should use to index into different forwarding table entries. Options include the IP ID and TOS fields in the IP header to the VLAN ID in the layer-two header. The main criteria are: (1) every network device along the path that interprets path bits should interpret the same set of bits; (2) the path bits should not affect other functions in the network. For our Click implementation, we used the TTL field instead of the TOS field, since we ran our experiments on Emulab, which filters non-standard TOS values in the IP headers of packets. Similarly, our OpenFlow implementation uses the VLAN ID instead of the IP ID and TOS fields because the current implementation of OpenFlow does not allow rules for these fields.

4.2.2 Network Implementations

To understand the generality, power, and efficiency of the path bits interface, we implemented three multipath routing schemes on a mix of four different platforms: software-based implementations in Click, and hardware implementations on NetFPGA, OpenFlow [64], and the Intel IXP network processor. One of the goals of this exercise is to demonstrate that the simple interface provided by path bits—and its function as an index into multiple forwarding tables—can support a variety of multipath routing schemes in few lines of code, using only modest resources. In all cases, we embed path bits in the IP ID field in the IP header. The TTL field value is used to index into the appropriate set of bits in the IP ID field by the routers along the path. All of our implementations, along with details on configuration, are publicly available on our website [3].

Click-based software router We implement three multipath schemes as Click elements: Path splicing, Routing Deflections, and a modified version of ECMP that we call ECMP++. The Click modules for these are available for use by researchers on our Web site [3]. Figure 4.1 describes the basic structure of the Click modules of our implementation. The implementation has a *path bits extractor* component that extracts the path bits from the packet header (the IP ID and TTL fields in our case). We have a custom implementation for each multipath scheme, which interprets the path bits based on the particular multipath scheme. Finally, the custom code derives an index number which corresponds to a choice of the forwarding table that must be used to lookup the destination. Our design generalizes to implementations on other platforms as well, as we show later in this Section.

- **Path Splicing** Figure 4.2 shows the experimental topology with an example of how path bits can indicate a path to the routers in the network. The IP ID field encodes the forwarding tree that each router along the path should use to forward the packet en route to the destination. Routers read the appropriate bit position in the IP ID field according to the index in the TTL field; the IP ID serves as an index into the

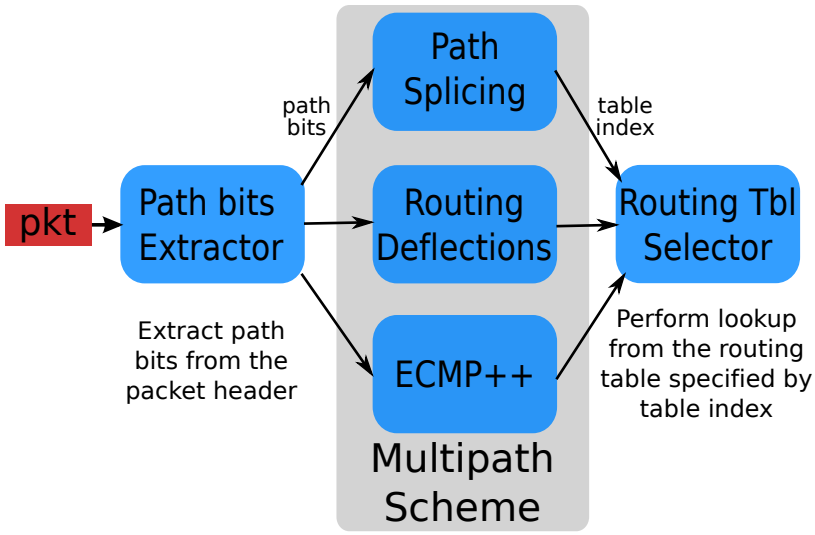


Figure 4.1: Click implementations of three different multiple routing schemes using path bits.

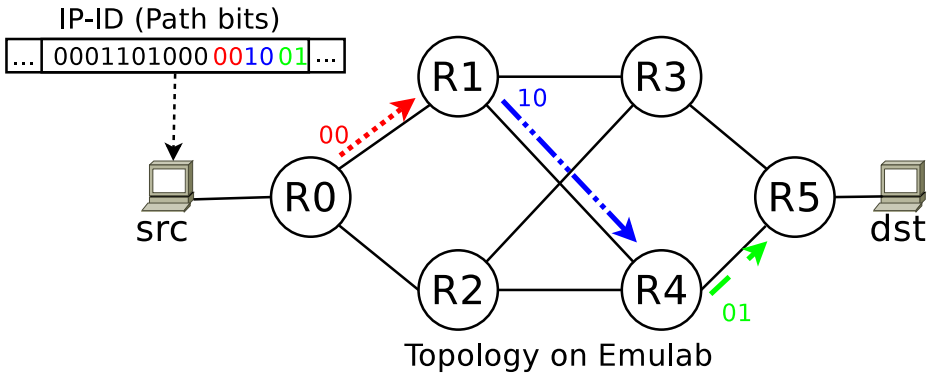


Figure 4.2: Topology used for Emulab experiments. The arrows and corresponding path bits show the next hop for reaching the destination along an example path.

forwarding table ("slice") that the router should use to forward the packet. As mentioned above, we use the IP ID and TTL fields to carry the path bits. The code for the Path Splicing implementation consists of a new Click element that reads the IP ID field in the packets and selects the appropriate routing table to use for forwarding the packet. *This element (PathSplicing) required only seven semicolon-containing lines of C++*, as shown in Figure 3(a). Lines 11–12 extract the table index number from the IP ID and TTL fields of the packet. The Click element then outputs the packet on the appropriate output port of the element (Line 15).

```
1  void PathSplicing::push(int port, Packet *p_in)
2  {
3    const click_ip *ip_in = p_in->ip_header();
4    assert(ip_in);
5
6    //Path Bits Extractor
7    uint16_t path_bits = ntohs(ip_in->ip_id);
8    uint8_t ttl = ip_in->ip_ttl;
9
10   //Splicing Specific to determine the slice number
11   uint8_t bits_index = ttl % 8;
12   uint16_t table_index = ((path_bits & (0x0003 << (2 *
         bits_index))) >> (2 * bits_index)) & 0x0003;
13
14   //Push the packet out to the output
15   output(table_index).push(p_in);
16 }
```

(a) Click element code.

```
1  elementclass RoutingTables
2  { |$src,$dst|
3    input -> rtable :: RadixIPLookup($src 0, $dst 1, 0/0 2,);
4    rtable[0] -> [0] output;
5    rtable[1] -> [1] output;
6    rtable[2] -> [2] output; }
7
8  //Create routing tables
9  rtable0 :: RoutingTables(src_ip, dst_ip);
10 rtable1 :: RoutingTables(src_ip, dst_ip);
11 ...
12 //PathSplicing connected to
13 //the appropriate routing table
14 splicing::PathSplicing();
15 splicing[0] -> DecIPTTL-> rtable0;
16 splicing[1] -> DecIPTTL-> rtable1;
17 ...
18 //Connections for table 0
19 splicing[0] -> to_router0
20 splicing[1] -> to_router4
21 splicing[2] -> Unstrip(14) -> ToHost();
22 .... //remaining routing tables
```

(b) Click configuration file.

Figure 4.3: Path Splicing.

Each router has a Click configuration file that specifies the connections for the multiple routing tables at the router and connects the output of the Click element *PathSplicing* with the appropriate forwarding table. Figure 3(b) shows the Click configuration for router R2 in Figure 4.2. Lines 15–17 direct the output of *PathSplicing* to the appropriate routing table, whose connections are specified for example in Lines 19–21 in Figure 3(b).

- **Routing Deflections** Routing Deflections [99] uses the bits in a similar manner as Path Splicing. Each router has a *deflection set* consisting of next hops that may be used for a packet depending on the packet's previous hop and destination address. Thus, the deflection set is a function of (*ingress interface*, *destination ip address*). The IP ID and TTL fields index into the deflection set at each router, as described in [99]. We implemented Routing Deflections by precomputing the deflection set for each router and including it in the Click configuration files. We also implemented a Click element that reads the IP ID and TTL fields to output the index number in the deflection set. The Click element is similar to the code snippet shown in Figure 3(a), and as shown in Figure 4(a), is about nine semicolon-containing lines of C++. Lines 11–15 compute the *tag*, which is extracted only if the TTL is greater than 160 and less than 200. The tag is then used to compute the index into the deflection set for the incoming interface and destination IP address (Line 19).

  Figure 4(b) shows the Click configuration for the Routing Deflections elements at Router *R*2. The configuration is similar to the Path Splicing configuration, except the configuration now specifies a deflection set for each neighbor, as opposed to individual routing tables per slice. Lines 3–7 configure the *RoutingDeflection* element and the connections to the routing tables (*deflection set*) for packets coming from neighbor *R*0.

- **ECMP++** ECMP++ is a version of ECMP where the outgoing interface for a packet

```
void RoutingDeflection::push(int port, Packet *p_in)
{
  const click_ip *ip_in = p_in->ip_header();
  assert(ip_in);

  //Path Bits Extractor
  uint16_t path_bits = ntohs(ip_in->ip_id);
  uint8_t ttl = ip_in->ip_ttl;

  //Routing Deflections specific
  path_bits = path_bits & 0x03ff; //use only the last 10 bits
  uint16_t tag = 0;
  if(ttl > 160 && ttl < 200) {
    tag = path_bits;
  }
  uint16_t table_index = (tag % prime) % size;

  //Push the packet out to the output
  output(table_index).push(p_in);
}
```

(a) Click element code.

```
....
//Deflection set for neighbor R0
deflect_0::RoutingDeflection(2);
rtable_R0_0::RTable(src_ip, dst_ip);
rtable_R0_1::RTable(src_ip, dst_ip);
deflect_0[0] -> DecIPTTL -> rtable_R0_0;
deflect_0[1] -> DecIPTTL -> rtable_R0_1;

//Deflection set for neighbor R3
deflect_3::RoutingDeflection(2);
rtable_R3_0::RTable(src_ip, dst_ip);
rtable_R3_1::RTable(src_ip, dst_ip);
...

//Connections for pkts coming from neighbor R0
rtable_R0_0[0] -> to_router0;
rtable_R0_0[1] -> to_router4;
rtable_R0_0[2] -> Unstrip(14) -> ToHost();

rtable_R0_1[0] -> to_router0;
rtable_R0_1[1] -> to_router3;
rtable_R0_1[2] -> Unstrip(14) -> ToHost();
.... //Similarly for other neigbors
```

(b) Click configuration file.

Figure 4.4: Routing Deflections.

```
1   void ECMPplus::push(int port, Packet *p_in)
2   {
3     const click_ip *ip_in = p_in->ip_header();
4     assert(ip_in);
5
6     //Path Bits Extractor
7     uint16_t path_bits = ntohs(ip_in->ip_id);
8     uint8_t ttl = ip_in->ip_ttl;
9
10    //ECMP++ specific
11    uint8_t index = ttl % 8;
12    uint16_t bits = ((path_bits & (0x0003 << (2 * index))) >> (2 *
          index)) & 0x0003;
13    int src_addr = ip_in->ip_src.s_addr;
14    int dst_addr = ip_in->ip_dst.s_addr;
15    int table_index = Hash(src_addr, dst_addr, bits);
16    table_index = table_index%2;
17
18    //Push the packet to the output
19    output(table_index).push(p_in);
20  }
```

Figure 4.5: ECMP++.

is determined based on a hash of (*src ip, dst ip, path bits*) in the packet header. The path bits are included as part of the IP ID field in the packet header; the TTL field can be as in the path splicing implementation to help the routers index in the IP ID field to read the path bits corresponding to the router. Figure 4.5 shows the implementation of this Click element; in the interest of space, we have not shown the Click configuration for this setup. It is similar to that for other multipath routing implementations. Lines 7–11 are same as in implementation of path splicing, while Lines 12–16 determine the output port (either 0 or 1) by hashing the ip addresses with the path bits.

For evaluating our end-system support for path bits we used the above software implementations of multipath routing schemes. We now move to describing how path bits can be supported on hardware platforms as well.

Openflow-based implementation We describe our design and implementation of Path Splicing using OpenFlow. OpenFlow is suited for deployment in an enterprise network or in a datacenter network environment, where all the network elements (*i.e.*, the routers and switches) are owned by a single entity. OpenFlow setup consists of switches which support the OpenFlow specification and an OpenFlow controller. The controller has a full view of the topology and can communicate with the OpenFlow switches to install forwarding rules. The controller also receives any frames that do not match the forwarding rules in the switches.

Using the switch topology information, the OpenFlow controller computes multiple spanning trees for the network topology and installs the appropriate forwarding rules in the OpenFlow switches. When a new end host sends a frame to the switch that it is attached to (*e.g.*, for an ARP request), the switch will not find a matching rule in the flow table and forwards the frame to the controller. The controller uses this frame to learn the location (*i.e.*, switch and port) of the host. The controller then installs multiple forwarding rules on *all* the switches in the network, based on the spanning trees that it has computed. Subsequently, as other hosts send frame to this host, the switches forward them using the rules installed by the controller.

Because current OpenFlow switches allow specifying forwarding rules based on limited number of fields in the Ethernet frames or IP packets, we cannot use the IP ID or TTL fields for the path bits. Instead, we use the source VLAN ID tag field to carry the path bits; this ID actually lends itself to a natural mapping between a single network that provides multiple paths and a network that is overlaid with multiple networks. Using the VLAN ID field also preserves semantics at layer two and higher, since no other fields in the Ethernet frame or IP header are modified. Unfortunately, unlike the IP header fields, the VLAN ID is not modifiable by applications. To allow hosts to modify the VLAN ID, hosts must implement a module that copies *path bits* to the VLAN ID tag field in the packets when they are sent on the network. This design is feasible, because an enterprise (or datacenter operator) has

much tighter control over the host operating systems.

We have implemented the above design with a custom NOX controller [40] and reference software switches for a three switch and two host network topology, similar to NetFPGA testbed topology shown in Figure 4.2. We are in the process of implementing the host modifications to allow setting the VLAN ID.

Implementing path splicing using OpenFlow requires additional space in the switch flow tables, as well as additional communication overhead with the controller. If there are k trees and N active hosts in the network, then we need $k \cdot N$ rules in the flow table of every switch. By comparison, a classical learning switch that is part of a single spanning tree maintains N entries in its bridge table. Second, the network incurs overhead in terms of communication with the controller. If there are M switches in the network, the controller sends $k \cdot M$ messages: one message per switch per spanning tree. The host's switch only forwards the first frame from a host to the controller, so this overhead is fixed. Our current implementation does not refresh the rules or expunge stale entries; these functions are important if hosts are silent for extended periods, leave the network, or relocate in the network. Implementing these features requires $k \cdot M \cdot N$ messages per refresh cycle across all switches.

NetFPGA-based implementation We implemented Path Splicing and Routing Deflections using NetFPGA [2]. Our implementations are loosely based on the implementation for building a fast, virtualized data plane with NetFPGA [12]. We implemented these schemes on the Xilinx Virtex-II Pro 50 FPGA.

The **path splicing** implementation instantiates four forwarding tables and four ARP tables in the base router. Because a destination IP address can exist in multiple forwarding tables, the implementation requires separate ARP tables to have different ARP entries for same IP address. The implementation uses four 32-entry TCAMs and four 32-entry ARP tables. The lookup modules are implemented using SRL16e; ARP CAMs are implemented

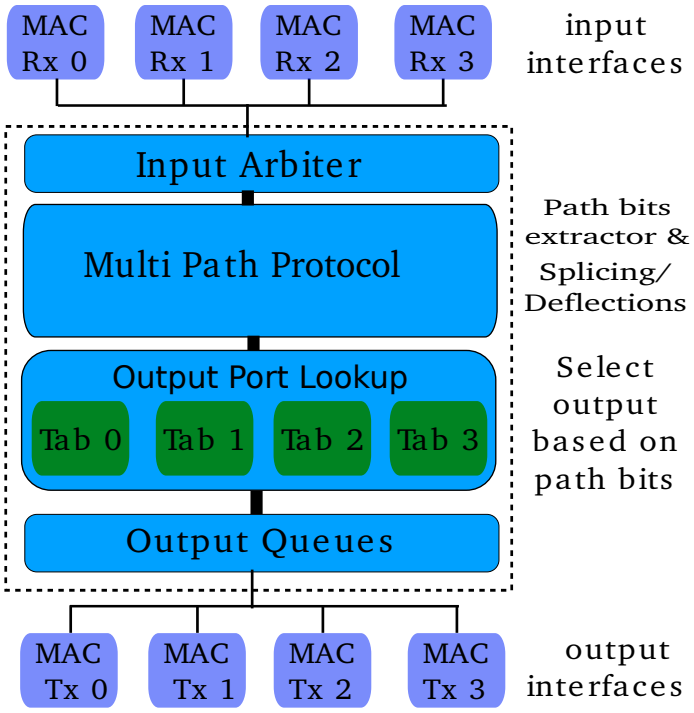


Figure 4.6: Router pipeline for the NetFPGA implementation of path splicing.

using dual port Block RAM (BRAM). These 32-entry tables correspond to available resources on the NetFPGA base router implementation: the card has one 32-entry TCAM longest-prefix match module with its lookup table and one 32-entry ARP table.

The implementation creates the path bits using three bits from the TTL field, plus the IP ID field. The high three bits are used to divide the lower 16 bits into eight entries. Each entry determines which of the four routing tables to use at each hop. We use on-chip memory to store the forwarding tables. For a base router, we use the reference router implementation from the NetFPGA group [2]. Figure 4.6 shows the base router implementation and the modules that are added or modified in the reference design. The path splicer performs collects the path bits and informs the output port lookup module which forwarding table to use to determine the next hop.

By separating forwarding table selection from forwarding, the *multipath module* in Figure 4.6 allows designers to use any kind of path bits selection mechanism that can act on the packet header. We use this feature to ease the implementation of both path splicing

and routing deflections. As observed the common denominator among different multipath schemes is to have separate forwarding tables for different paths; different multipath algorithms merely change how the path bits are used to select among these forwarding tables. Our *multipath* module can be used easily to implement a new multipath routing scheme by changing the table selection register shown in Figure 4.6.

4.3 End-system Support

This section describes the design and implementation of the end-system support for the path-bits interface. In Section 4.3.1, we explain key design decisions for allowing end systems to set (and modify) path bits in packets. Our implementation of path bits runs on a Linux end host, which we describe in Section 4.3.2, and comprises (1) the *path bits manager*, a kernel interface for manipulating path bits; and (2) a *socket capture library* for providing path bits support to unmodified legacy applications. We also implement a few simple path monitoring agents that use path bits to monitor and select paths. Section 4.3.3 describes the interface between the path bits manager and monitoring agents that can induce the path bits manager to change a flow's path bits.

4.3.1 Software Interface Design Decisions

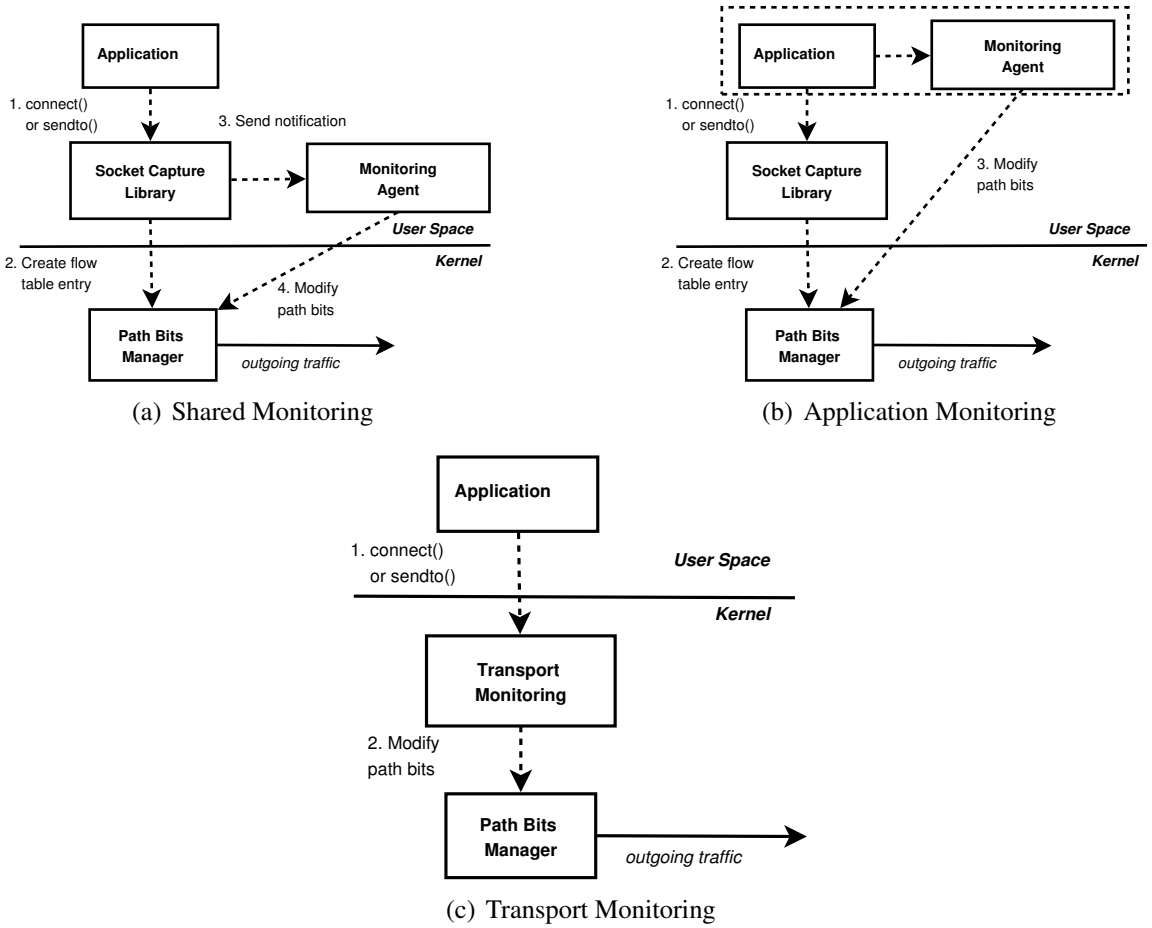
Our primary goal for the software interface is to balance *ease-of-use*—making it as close to transparent as possible for applications to benefit from path bits—with *flexibility*—the ability for an application to meet precisely its unique needs from the underlying multipath capabilities. To meet this goal, our interface is based upon three high-level design decisions:

1. **The kernel controls the assignment of path bits to packets.** Application programmers often think in terms of sockets or flows. Furthermore, particularly when using TCP, they may not have the control or timing needed to decide on a packet-by-packet basis what path bits to assign. As a result, we place the kernel in charge of setting

the per-packet path bits, and define an interface for either the application or a higher-level protocol (*e.g.*, TCP) to modify them. We have implemented one such mapping using a Click [50] kernel module called the *path bits manager*, which replaces the networking stack and incorporates a table that maps flows to path bits.

2. **It should be easy for either senders *or* receivers to trigger a path bits change, as needed.** The sender sets path bits, but the receiver can directly measure the quality of the paths that reach it. Some protocols such as TCP already provide the end-to-end feedback that the sender needs to determine path quality, but sometimes the receiver may wish to trigger a path change on the forward path (*e.g.*real-time applications like VoIP or online games might want such control). To make this option easier for applications to use, we add a feedback mechanism to the path bits manager at the sender and receiver to allow the receiver to trigger a change of the path bits for packets sent to it (assuming the sender allows this). An implementation of this trigger would be to change the bits on the forward flow whenever the bits on the reverse flow change.

3. **Share the implementation work of monitoring path quality.** Many applications may have similar requirements for path quality (and therefore, similar criteria for selecting and changing paths). A shared implementation of path quality monitoring could relieve application developers of the need to implement their own monitoring and decision logic to achieve objectives such as high throughput or low latency. Multipath-aware applications, or those with specific requirements such as concurrent multipath use, could instead use lower-level flow-binding mechanisms. As a proof of concept, we implement two simple end-system path monitoring agents that set path bits for applications. The first actively sends packet trains to monitor path quality, and the second passively monitors application traffic to estimate the path quality. We also provide a simple in-kernel failure recovery mechanism for those applications



(a) Shared Monitoring

(b) Application Monitoring

(c) Transport Monitoring

Figure 4.7: Interaction of the end-system components for different types of monitoring.

that care only about availability. We describe these implementations in Section 4.3.3.

4.3.2 Implementation

End-system support for path bits consists of an interface for applications to set and modify the path bits corresponding to the traffic flows that belong to the application. This requires the end-system kernel to maintain path bits corresponding to the active traffic flows (*path bits manager* as shown in Figure 4.7). We implement this feature using Click running as a kernel module. Our Click implementation provides an RPC interface for interacting with the path bits manager from user-space. Applications can modify these bits corresponding to their traffic flows. We realize that all applications may not care about fine-grained control

over the network paths as long as the paths are of *reasonable* quality. In order to facilitate this, we implement, as a prototype, a user-space shared monitoring application that monitors paths on behalf of the applications and a socket capture library that can trigger monitoring of paths on behalf of the application.

Figure 4.7 shows an overview of the extensions we make to end hosts to support the path bits interface. This framework has two components: (1) a kernel interface to access the path bits maintained by the path bits manager; (2) a socket capture library for running legacy applications. We describe these components below:

- **Path Bits Manager**

  Our implementation of path bits manager runs in Click [50] as a kernel module to replace the Linux network stack. The path bits manager consists of a table that stores information about which path bits are currently used for a particular flow. The Click module captures packets from the network interface, matches the packet flow identifier with the entries in the path bits manager table, and applies the corresponding path bits if there is a match. The Click module also provides an RPC-based API to read, modify, and delete entries from the path bits manager. The module supports wild-card entries for any tuples in the traffic flow that are over-ridden if a specific entry for a flow exists in the table. For example, to add bits for a particular TCP flow, *add_tcp_bits(flow_identifier, path_bits)* is used, which sets the appropriate path bits for the TCP flow. If the *flow_identifier* has the source port number zeroed then the path bits manager will match all IP flows to the particular destination port (unless a more specific match is present). The wild-card is only allowed for applications running with root permissions. The monitoring agent implementations use this API to interact with the path bits manager.

- **Socket Capture Library**

  Legacy applications can use the path-bits interface using a *socket capture library*. To

keep track of a set of desirable paths for each application, we implement a socket capture library that intercepts `connect()` calls for TCP and `sendto()` calls for UDP flows. The library determines when new flows are initiated from the end host, as shown Figure 4.7. If applications need explicit control of their paths, they can make use of the API provided by the operating system to modify their path bits. All modern operating systems support the dynamic linker option, which we exploit to dynamically link our socket capture library to an unmodified application binary. For example, in GNU/Linux, we can use the environment variable `LD_PRELOAD` to specify our custom library to load for the application binary. Similar provisions are available in Microsoft Windows and Mac OS X operating systems. The **path bits manager** kernel module can check if the process making the RPC call owns the flow to prevent unauthorized applications from modifying the path bits for a flow.

Now, we show how to build a path monitoring and selection framework using the above components.

4.3.3 Path Monitoring and Selection

Our design makes it easy to implement several distinct mechanisms for the end host to monitor and select paths. We implement three proof-of-concept monitoring mechanisms to demonstrate the variety of implementation choices available using path bits:

- **Shared monitoring**, which estimates the path quality for applications running on the end host, and sets the path bits on their behalf;
- **Application monitoring**, implemented directly by the applications themselves;
- **Transport monitoring**, which is provided by the end host networking stack to all applications.

Other possibilities include using a combination of the above techniques. Our goal in investigating these techniques is not to advocate any particular technique, but rather to

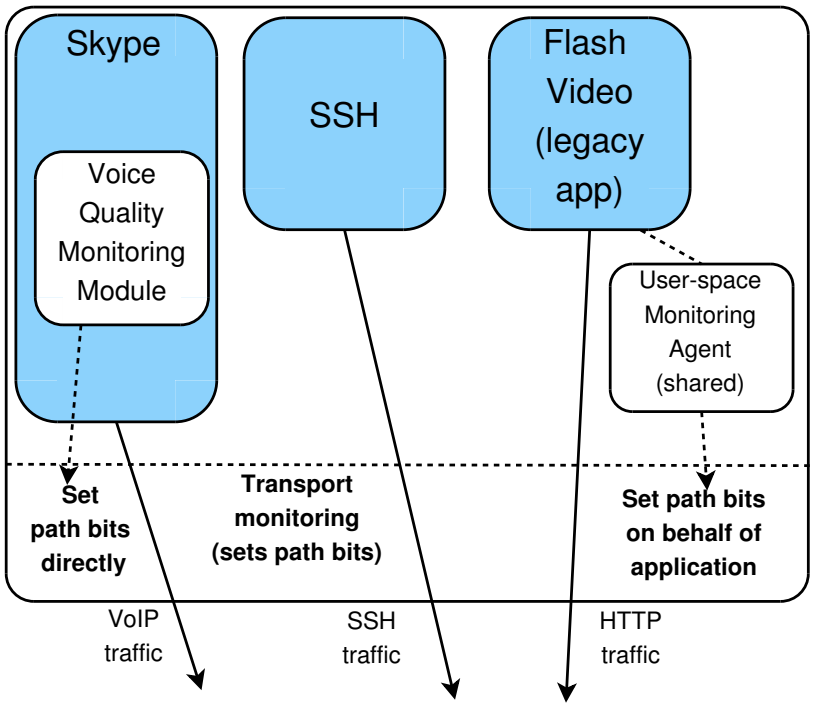


Figure 4.8: How path bits enable building customized path monitoring mechanisms at the end host.

emphasize the power and flexibility offered by the path bits interface. Figure 4.8 shows the three path-selection mechanisms. A real-time application like Skype can build its own monitoring module. An application like ssh, which may not care about anything other than simple connectivity could use the default recovery provided by the TCP stack at the end host. It also enables running unmodified legacy applications, by running a shared monitoring agent as a separate user-space process.

Shared monitoring application A common monitoring application can be implemented for applications running at the end system that choose to delegate the path monitoring and selection decision to some other application. To demonstrate the utility and simplicity of path bits for this purpose, we implement two simple prototype monitoring applications. We note that these monitoring applications are only suggestive of the possibilities of doing path monitoring with the help of path bits. The first application uses active probes to determine path performance and also to probe alternate paths so that the application traffic can be

moved to an alternate path quickly when there is a perceived degradation in path quality from the desired. The second monitoring application passively monitors the application traffic to measure the performance. These applications demonstrate that the *interface* works and is general.

For both of these monitoring agents, an important parameter is the time the *path evaluation interval*, which is that the agent waits before deciding to initiate a path switch.

1. **Active Monitoring**

   When an application initiates a flow, the socket capture library sends a notification to the active monitoring agent. The current implementation receives the identification of each flow (the four tuple <src ip, dst ip, src port, dst port>), as well as acceptable thresholds for the latency and loss rate for the flow for each application type. The application can specify how many alternate paths to monitor for any given flow. This could be implemented using `setsockopt` by the application or done by the socket capture library on behalf of the application.

   The agent monitors a set of paths for performance and picks the best one for the application. The agent then periodically sends packet trains on each of the paths in the set, and records and maintains their performance in a monitoring table for the particular flow. If the performance of the current path falls below the application's performance thresholds, the agent tells the *path bits manager* to switch paths. The monitoring agent periodically replaces paths in its monitoring table that do not meet the thresholds with new random paths. The monitoring agent interacts with the path bits manager to set path bits for the monitored flows. We evaluate the impact of the number of packets send in a single probe on the ability of the monitoring agent to effect path selection in Section 4.4.2.

2. **Passive Monitoring**

   Active probing may observe different performance than the application traffic. There

is a rich literature on passively observing application traffic to determine the quality of the network path the traffic is using [7, 8, 48]. The passive monitoring agent only monitors TCP flows. As with the active monitoring agent, the passive agent receives a notification from the socket capture library about a flow; it then passively monitors the flow by capturing the flow packets using the standard `libpcap` interface. As with the active monitoring agent, the passive monitoring module can trigger a path change for the traffic flow via the path bits manager. To track performance of a flow, the agent applies an EWMA to calculate the round-trip time of the flow. it also uses the technique described in Allman *et al.* [7] to estimate the loss rate of the TCP flow: it counts the number of retransmitted TCP segments, discounting the retransmits that the sender may have sent that do not correspond to packet loss. We implement this passive monitoring technique in fewer than 300 lines of Ruby.

A critical choice in passive monitoring is how aggressively an implementation will switch to a new path upon perceiving poor performance, vs. waiting to have more confidence that the performance change was real and long-lasting. We evaluate the effects of the *path evaluation timeout* in Section 4.4.2.

Application Monitoring Different applications have different traffic patterns and hence, have their own understanding of what constitutes a "good" network path. One of the benefits of a path bitsinterface is that it frees the application to make its own decisions about the network path that its traffic takes without relying on some predefined metrics which may not take into account the specific needs of the application. In such situations, path bits provides the ability for the application to perform its own monitoring and make path selection decisions based on what the application deems to be a "working" or "non-working" path.

We use VoIP as an example application, where the notion of a "good" path is not easily captured by the standard network metrics of delay and loss rate. Tao *et al.* [85] show that the relation between path latency and loss rate for VoIP quality is complex and also

depends on other factors like the audio codec used by the VoIP application. We implement a VoIP quality algorithm, that measures the loss rate and the latency on a set of paths and determines the VoIP quality metric called *Mean Opinion Score (MOS)*. The MOS metric is subjective and depends on the quality of the voice signal as perceived by a human and ranges from 1 to 5 (best quality). The ITU-T E-model [74] provides a way to approximate MOS by combining different elements that contribute towards reducing the voice quality. We use the non-linear mapping from path loss rate, latency and the codec features to MOS as described in previous work [85] as our path quality metric.

Transport Monitoring Finally, we implement a path selector in the network stack that runs as a Click kernel module. The module observes traffic on an interface and when it sees that for a flow, the same TCP packet (with the same sequence number) is retransmitted without receiving any acknowledgement from the receiver, then the current path to the destination is deemed as not working after it exceeds a configurable threshold. The agent then communicates with the path bits manager to modify the path bits corresponding to that flow in the path bits manager. For applications such as ssh, a simple recovery approach such as this may be sufficient to recover from prolonged network path failures. In Section 4.4, we evaluate the recovery benefits of this technique and also the tradeoffs between the number of switches and the switching threshold.

We define a configurable *retransmission timeout*, and evaluate the effectiveness of application monitoring under different settings. This approach works well if the path is unusable either because of link failure on the path or extremely high loss rate; it works less well for detecting paths with high latency or jitter. In Section 4.4, we evaluate the recovery benefits of this technique and also the tradeoffs between the number of switches and retransmission timeout.

4.4 Does "Blind" Path Selection Work?

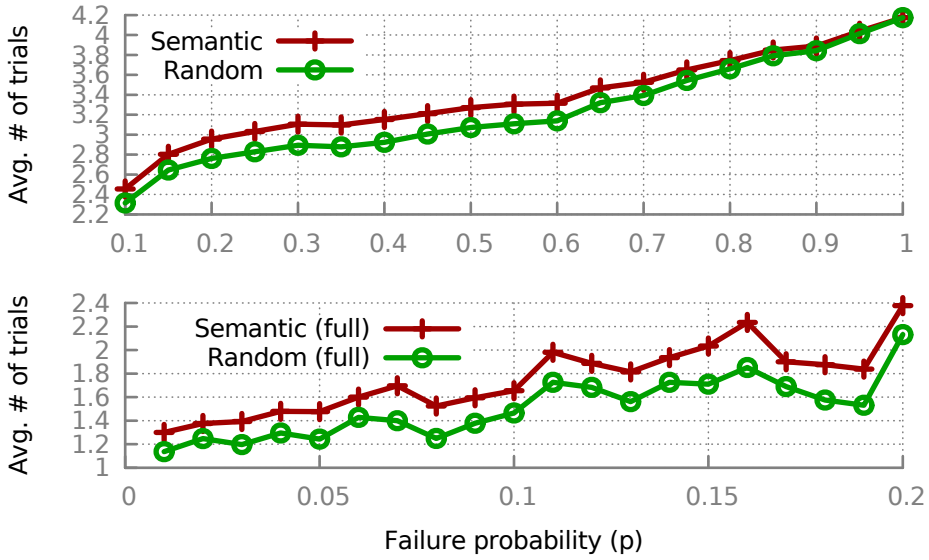
In this section, we focus on whether the path-bits interface can help applications respond more quickly to failure or take advantage of diverse paths in the network, even with an opaque, semantic-free interface. We focus on the following questions:

1. How quickly can the path-bits interface find a better end-to-end path in the network, as compared to an interface that has semantics? (Section 4.4.1)
2. What monitoring strategies work well with the path-bits interface? (Section 4.4.2)
3. Can applications use path bits to increase throughput, by using multiple paths simultaneously? (Section 4.4.3)
4. Can path bits be used to discover diverse paths in the wide area? (Section 4.4.4)

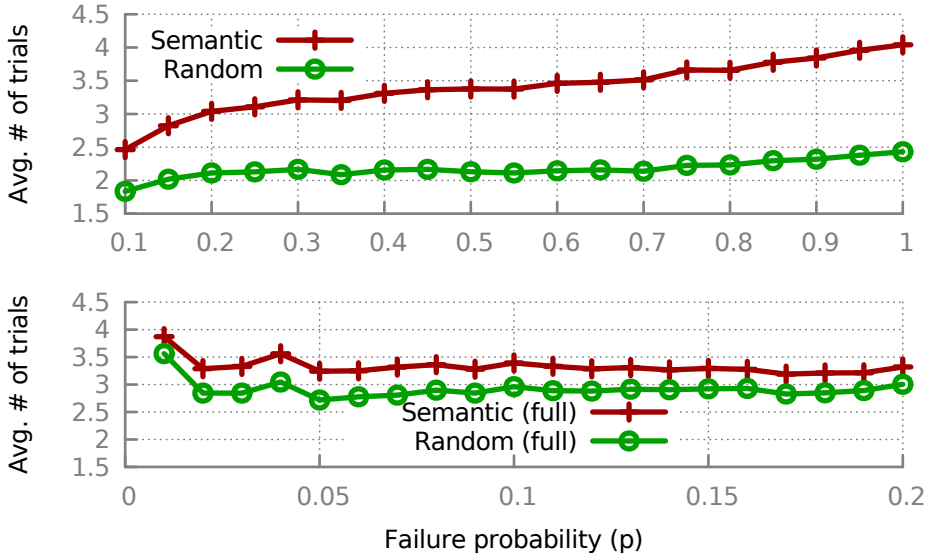
4.4.1 How many trials to find a path?

The first and perhaps most important question is whether an interface like path bits could still allow end systems to find working paths without having to explore too many alternatives. To evaluate this question, we performed an experiment where we fail links in the Rocketfuel intradomain ISP topologies [83] and compare two different path recovery approaches: one where the end system explicitly signals to the network to select a different forwarding tree ("slice") at a specific place along the path (as in path splicing [60]), and another where the end system selects bits completely at random. In the experiment, we create multiple paths in the network (forwarding trees or "slices") using the random perturbation method used by path splicing. We perform two experiments: one where we fail each link in the topology with some probability, p, and second, where we fail only links that belong to highly connected nodes. For each setup, we measure the number of trials for the end system to find a new working path after the links are failed. We average our results over 1,000 runs for each failure probability.

Figure 4.9 shows the results for this experiment, for Sprint and Level 3. The result is a pleasant surprise: Random selection performs better than a method that can explicitly avoid



(a) Level 3



(b) Sprint

Figure 4.9: Number of trials to recover from path failures for Sprint and Level 3 topologies. The top plots are when links are failed from a selected set of most connected links. The bottom plots are when all links in the network are failed with equal probability.

a specific node along the path! In the case of Level 3 (Figure 9(a)), the average number of trials when 5% of the links are failed is 1.3 when using path bits which is about 30% higher than the best-case scenario, where the end-system "knows" exactly which link has failed and explicitly tries to avoid that link.

Recovering from failures in the Sprint topology requires more trials (Figure 9(b)) because the Rocketfuel ISP topology is sparse, and many nodes have low degree. Random path selection works better because attempts to avoid a single link or node failure with coarse bit settings might ultimately not avoid the failed link or node, depending on how the underlying multipath routing protocol is implemented. (For example, a link failure might occur in multiple "slices" in path splicing, along multiple equal-cost paths in ECMP, and so forth.)

We aimed to determine how sensitive our results were to the location of failures in the underlying network topology. For example, because path splicing constructs additional paths in the network by trying to avoid highly connected nodes. To study the sensitivity of our results, we repeated the previous experiment but failed only a set of links that belong to highly connected nodes. The top plots in Figure 4.9 shows the result for this experiment. As before, random selection requires fewer trials to find a working path (if one exists) as compared to the approach used by path splicing.

This experiment demonstrates that random selection of path bits performs better than a method that selects path bits based on slices. However, there could possibly be some other path bits selection mechanism that is aware of the underlying multipath scheme and uses that information to recover from failures better than the random selection. But, still it is important to note that random selection does not perform much worse than the best possible and only requires 30% additional trials.

4.4.2 Which monitoring works well?

We evaluate whether path bits can be integrated with the different monitoring approaches that we described in Section 4.3.3 to recover from failures. Our intention is not to design the best monitoring approach for failure recovery, as there is a significant amount previous work in path monitoring and recovery (*e.g.*, for overlay networks and intelligent routing). Rather, our goal is to evaluate whether the path-bits interface can be integrated with a variety of path-monitoring algorithms for failure recovery.

Experiment Setup We performed our experiments on Emulab; the end system is a Linux host running the host instrumentation described in Section 4.3. We used two different experimental setups: (1) A small topology, as shown in Figure 4.2, where the Emulab nodes (acting as routers) are running the Click software router implementations of Path Splicing, Routing Deflections, and ECMP++ (to emulate intradomain path properties); and (2) Using the Linux `tc` utility to emulate four paths with different round-trip times, jitter, and loss rates between the *src* and *dst* nodes (to emulate wide-area path properties).

This path emulator selects a path based on the last two bits in the packet's IP ID field. The emulator has three different sets of paths: two synthetic path sets, and one based on data obtained from wide-area measurements where we used path bit-like path selection to explore alternative wide-area paths (using the BGP poisoning experiment described in Section 4.4.4). We chose the synthetic paths in a way that illustrates various properties of the monitoring algorithms: Because the shared monitoring algorithms can monitor paths based on round-trip time and loss rate, we selected values for the synthetic experiment where two paths have high round-trip times and two paths have high loss rates. Table 4.1 shows the details of these paths. We use iperf to create the traffic flows and record the changes to the instantaneous throughput of the traffic while introducing network events. We evaluate active monitoring, passive monitoring, and transport monitoring using path emulation because it enabled us to quickly emulate different path characteristics. Active

Table 4.1: Path characteristics of emulated paths on Emulab. For the wide-area experiment setup we also had jitter on the path based on what was measured in the wide-area BGP poisoning experiment.

Experiment	Path Properties			Threshold	
	RTT (ms)	Loss	Tput (Mbps)	RTT (ms)	Loss
Synthetic 1 (Figure 4.11)	30	1%	3.86	50	4%
	30	7%	0.87		
	130	3%	0.49		
	150	2%	0.56		
Wide-area (Figures 4.10 and 4.12)	235	0%	37.40	260	3%
	315	0%	9.67		
	276	3%	0.22		
	258	2%	36.51		
Synthetic 2 (Figure 4.13)	30	1%	3.86	-	-
	30	15%	0.37		
	130	30%	0.06		
	150	20%	0.08		

monitoring works with TCP or UDP, but we only show the results for TCP. Because we designed passive monitoring and the transport monitoring to operate with TCP, we evaluate them using TCP traffic only.

We also evaluated the monitoring algorithms using the setup described in Figure 4.2 to demonstrate integration with our multipath implementations from Section 4.2. The results are similar to the emulated setup. The path monitoring algorithms assume anything about the underlying multipath routing protocol. Our evaluation demonstrates that even simple monitoring algorithms can work well with path bits.

Shared Monitoring We present the resuts for the shared monitoring implementations here:

1. **Active Monitoring**

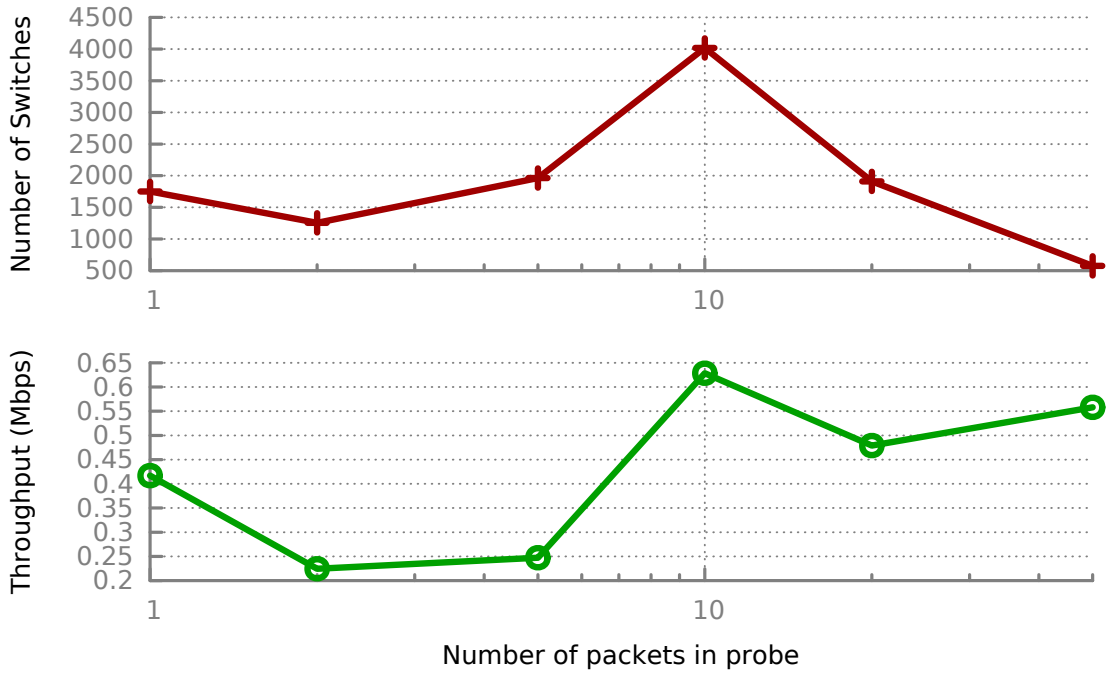


Figure 4.10: Active Monitoring: Failure recovery using active monitoring and TCP traffic.

We present results using the wide-area emulated path setup (Table 4.1). The monitoring agent triggered a path change if the round-trip latency exceeded 260 milliseconds or the loss rate exceeded 3%. The monitoring agent monitored two paths and varied the number of packets to send in a probe on the paths being monitored. We sent a two-minute TCP flow from the source to destination and record the throughput and the number of times the monitoring agent switches the path. We perform ten experiment trials for each value of the number of packets in a probe.

Figure 4.10 shows the median throughput and the average number of path switches as we vary the number of packets in the packet-probing train. Sending too few probes results in poor performance because the measurement quality is poor if there are too few packets in a probe, but sending too many probes can interfere with the application traffic. Even in the best case, the throughput is considerably less than the maximum possible because the monitoring agent sends the probes continuously, which interferes with the performance of the iperf traffic. Also, because the probing interferes

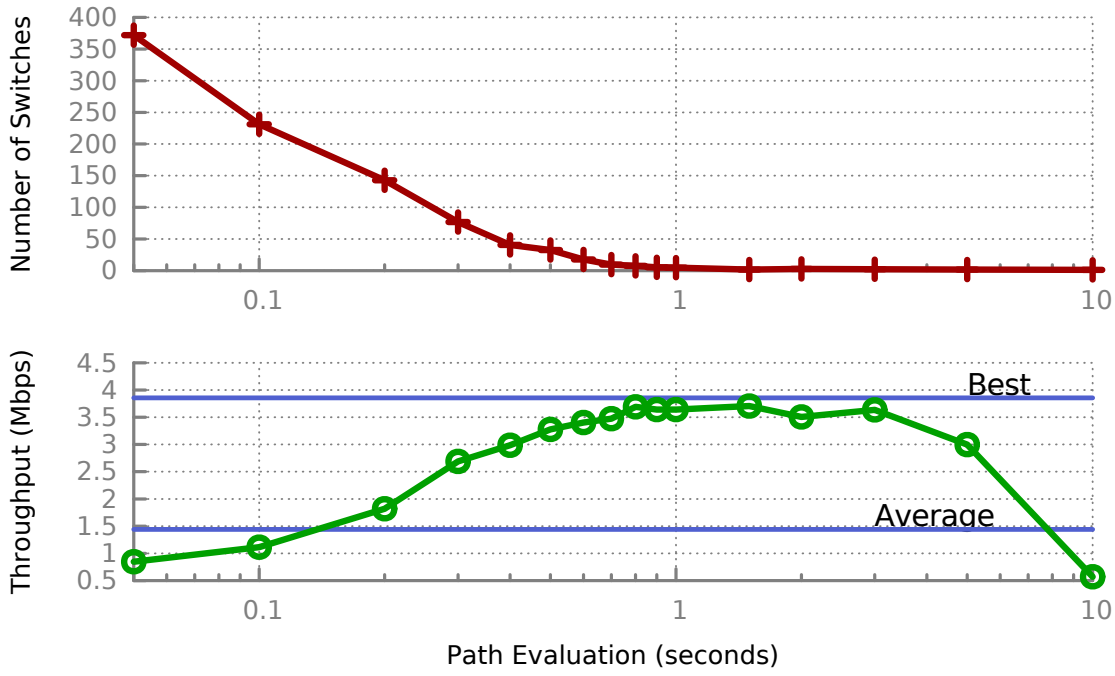


Figure 4.11: Passive Monitoring: Average number of switches and median throughput for Synthetic 1 from Table 4.1. We set the threshold to 50 ms and 4% loss rate.

with TCP traffic, it doesn't accurately measure the path metrics and makes a large number of path switches. A better active-monitoring implementation could mitigate this interference.

2. **Passive Monitoring**

   We evaluate the passive monitoring agent by sending a two-minute TCP flow using iperf from the *src* to the *dst* node through the path emulator, which emulates four different wide-area Internet paths, for the three scenarios in Table 4.1. We show results from the "synthetic 1" and "wide-area" paths. The utility of the passive monitoring scheme is in situations where the network has paths all of which are usable. In the "synthetic 2" setup, where the network has very poor quality paths, a much simpler scheme like the transport monitoring works equally well. The effectiveness of this approach depends on the value of path evaluation timeout, and that the right value for this parameter depends on the application requirements and the round-trip latency.

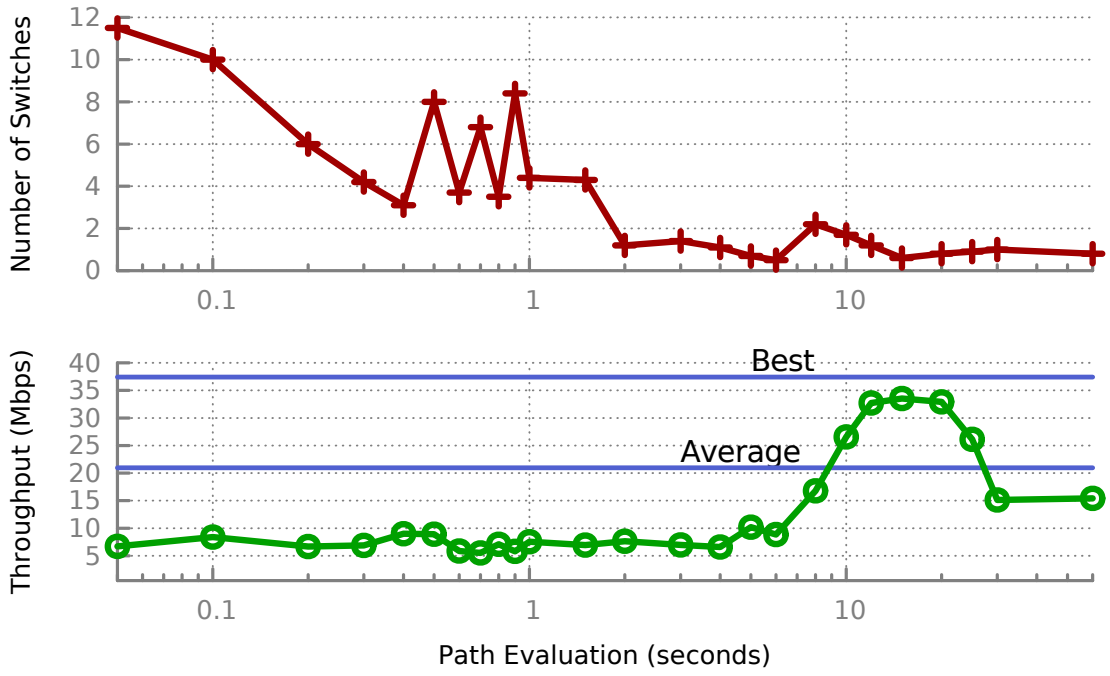


Figure 4.12: Passive Monitoring: Average number of switches and median throughput for the wide-area paths from Table 4.1.

Figure 4.11 shows the results for the first setup, where we vary the path evaluation timeout and measure the average number of switches and the median throughput for ten trials. When the value of the path evaluation timeout is small relative to the round-trip time, the round-trip and loss-rate measurements are inaccurate, causing many path switches and degrading throughput. As seen in the plot, for the given path setup, there is a "sweet spot" for the value of the path evaluation timeout around 1–2 seconds that achieves a low number of switches and a throughput close to the best possible.

Figure 4.12 performs the same experiment with different paths and thresholds; the result shows that the optimal setting of the path evaluation timeout depends on the round-trip time of the path. Selecting a smaller value for the path evaluation timeout causes the monitoring agent can quickly move away from a bad path, but it also causes higher noise in the measured path properties, resulting in unnecessary path switches. A higher timeout value allows for better estimate of the path properties

but less chance of switching paths. Because only one of the paths matches both the round-trip latency and loss rate threshold (the path with a 30-millisecond round-trip latency and 1% loss rate), there is only a 25% chance of a path switch resulting in the selection of the best path, which increases the recovery time when the timeout value is high.

The performance of a monitoring agent also depends on the threshold for switching paths. We performed a second experiment with the same setup as in Figure 4.12, but where the monitoring agent triggered a path only if current path round-trip time exceeded 300 milliseconds. In this case, there was no benefit over the average throughput because the monitoring agent would only attempt to switch paths if it were selecting the path with the highest round-trip time.

Application Monitoring Application-specific metrics can also trigger changes to path bits. Tao *et al.* [85] previously showed that the ability for a VoIP application to rapidly switch between multiple paths can greatly improve VoIP quality. We verified that a similar application could realize similar gains using path bits. We developed an application monitoring agent that used a VoIP path-quality algorithm to monitor a set of paths in Figure 4.2 with varying packet-loss rates, calculated the mean opinion score (MOS) for the resulting stream based on the observed RTT and loss rates, and set path bits accordingly. We found that selecting the best path using MOS works well.

Transport monitoring To evaluate the effectiveness of transport monitoring, we used the same Emulab setup as for passive monitoring. We found that for those path setups, the loss rates are not significant enough to trigger a path switch. (Our transport monitor counts retransmissions but does not measure round-trip latency.)

We show results here from a path setup with high loss rates, as shown in the "synthetic 2" row in Table 4.1, where only one path has a low enough loss rate to sustain TCP traffic. In the experiment, we ran 100 trials, where each trial had a 20-second TCP transmission.

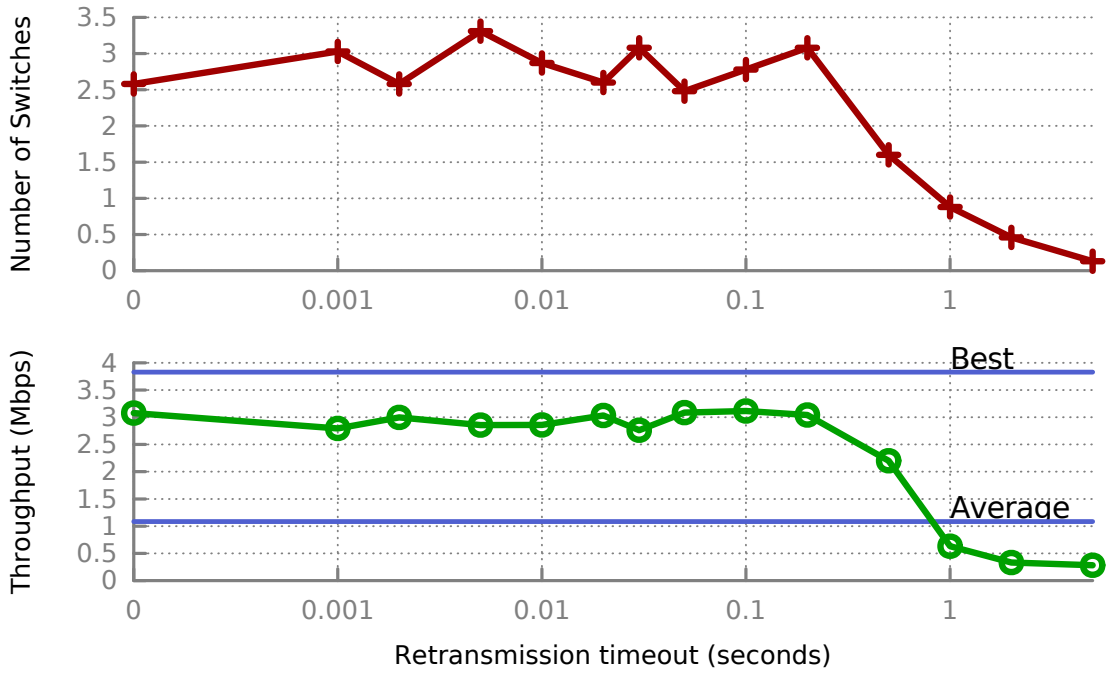


Figure 4.13: Transport Monitoring: Average number of switches and median throughput for Synthetic 2 from Table 4.1.

We varied the *retransmission timeout* parameter for the path failure detection and measured its effect on throughput and the number of paths switches. Figure 4.13 shows the average number of switches and median throughput achieved when using the transport-based monitoring technique. The average number of switches is constant, as is the achieved throughput for low values of the retransmission timeout ($<$ 200ms); higher retransmission timeouts reduce both switched and average throughput. Interestingly, the algorithm performs well even if we let it switch paths whenever it sees a retransmission; more extensive evaluation could uncover whether this works well in all scenarios.

4.4.3 Can path bits increase throughput?

Bulk transfer applications can benefit from simultaneous use of multiple paths. The path bits manager provides an interface to dynamically turn on or off the simultaneous use of multiple paths on a particular flow; in this mode, it keeps multiple entries corresponding to each flow in its path bits table. To enable the use of multiple paths, the path bits manager

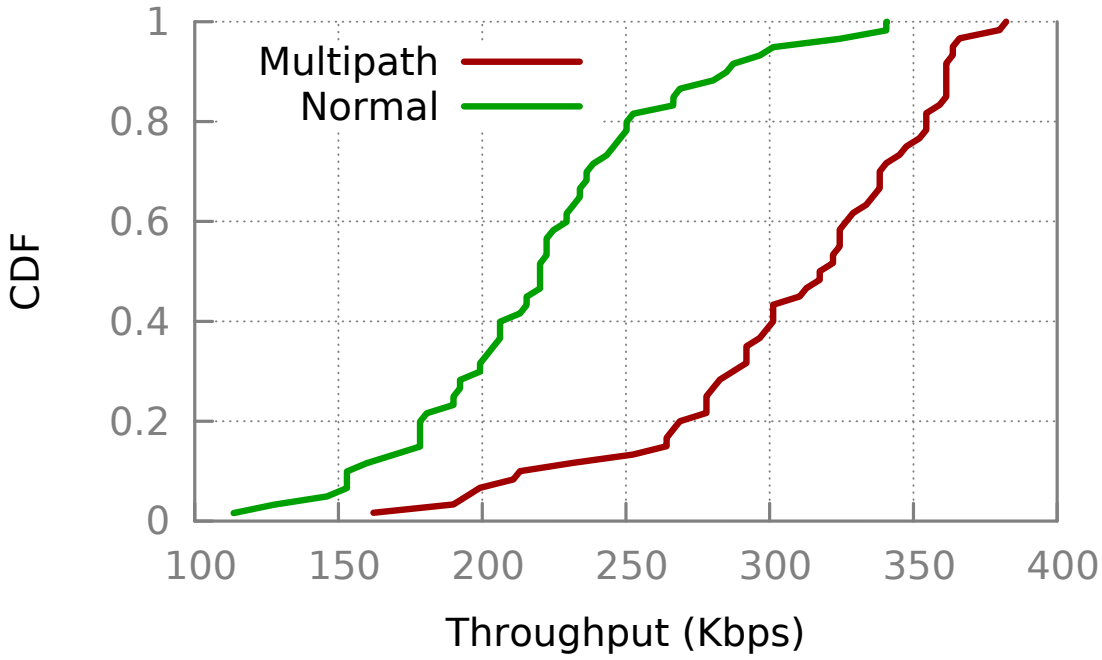


Figure 4.14: Higher TCP throughput by simultaneously using multiple paths in parallel for a single TCP connection.

simply multiplexes packets from a flow on each of the multiple paths in a simple round-robin fashion.

Although this type of splitting across paths may induce reordering, and many multipath approaches that achieve flow affinity do exist (*e.g.*, [82]), our goal is merely to demonstrate blind path selection of the nature that path bits enables can increase throughput. We use the setup in Figure 4.2, where there are four available paths between the source and destination nodes. We ran the experiment for ten minutes, intermittently enabling and disabling the use of multiple paths at the source. Figure 4.14 shows the CDF of the TCP throughput of the flow for the normal and the multipath settings. The plot demonstrates that, even using a naïve multipath routing approach, applications can achieve significantly higher throughput with path bits. An interesting question is the interaction of multipath routing with congestion control; this approach could be integrated with a multipath congestion control algorithm such as mptcp [94], which may yield even better performance.

Table 4.2: The fraction of destinations for which the end-to-end round-trip latency differed by a certain amount after BGP poisoning.

RTT diff. (ms)	Pct. of destinations	# paths
10	16.34%	2.43
30	3.23%	2.44
50	1.26%	2.14
70	0.54%	2
100	0.18%	2

4.4.4 Can path bits find wide-area paths?

We study how likely path bits would be to discover diverse paths in the wide-area. To do so, we used data from a BGP poisoning experiment performed using the Transit Portal [88] to evaluate the available path diversity in interdomain routing that could be discovered using a path bits interface. To explore a random set of interdomain paths to a destination, we "poisoned" the BGP announcement [16] with a specific AS number for a prefix allocated to Transit Portal. Our goal is to discover alternate BGP paths, which networks will select if the poisoned AS was on the default path for the Transit Portal prefix. After each poisoning announcement, and letting the BGP announcement propagate throughout the Internet, ping probes were sent to a set of 650 PlanetLab nodes from one of the Transit Portal locations in Atlanta. We poisoned each route announcement with 30 distinct AS numbers. Table 4.2 shows the percentage of destinations whose round-trip times differed by more than a certain amount after poisoning. From our results we found for example, if we select 30 ms as the minimum round-trip time difference then $> 3\%$ destinations have on average 2.44 such paths. This may not be enough path diversity but we may also underestimate it: we poison only a fraction of ASes, and some paths may have similar round-trip times but have different capacity or loss rate.

4.5 Summary

Many networked applications benefit from access to multiple paths for improved performance and rapid failure recovery. Despite the benefits of a unifying interface, none has yet emerged.

We presented the implementation of the "narrow waist" architecture for path selection, in both hardware and software. The implementations are based on the design of the path bits interface described in Chapter 2. We have successfully demonstrated that the path-bits interface leads to simple and efficient network implementations of multipath routing protocols, in hardware and software. It is also general as demonstrated by our implementations of three different multipath mechanisms.

The path-bits interface is also simple and flexible to allow different path selection mechanisms at end systems as demonstrated by our implementations of three monitoring and path selection mechanisms. One likely drawback of a semantic-free interface is that end systems must use trial and error to discover paths. However, we show a surprising result, random path selection performs better than a selection that relies on semantics when using path splicing as the multipath mechanism. This result is encouraging that random path selection can be effective.

We make our implementations available as the first framework that allows both different multipath algorithms and different real-time monitoring and recovery frameworks in a common context [3]. We hope that both researchers and practitioners will extend and evolve this reference implementation to support new multipath routing protocol implementations and new applications that use the interface.

End systems and network operators can have conflicting goals. Giving end systems unrestricted access for selecting paths in the network can be in conflict with the network operator's goal of reducing traffic costs in the network. We explore this question and propose a way of allowing operators to limit the set of paths based on the cost of carrying traffic over those paths to the network in the following chapter.

CHAPTER V

COST-BASED PATH SELECTION

5.1 Introduction

So far the dissertation has focussed on making alternate paths available in the network so as to benefit a variety of applications. The network path selection architecture delegates the responsibility of creating and maintaining a large number of paths to the network. The network can use any number of multipath routing schemes (*e.g.*path splicing 3) to create alternate paths. Multipath schemes (including path splicing) create alternate paths with the goal of providing failure tolerance (in case of link or node failures). However, networks (especially commercial networks) are also business entities and would like to run their network for as low a cost as possible. Multipath schemes do not consider this added cost dimension of carrying traffic in the network when creating alternate paths. In this chapter we argue that this is an important consideration and the adoption of multipath schemes in the real-world is likely hampered due to the lack of this consideration when creating alternate paths in the network.

We then propose a cost-based path selection framework, where the network is aware of the costs of carrying traffic on the alternate paths and can then make a decision on which to expose to end-systems. However, in order to build such a framework, networks must first be able to assign traffic cost to traffic flows. This is the focus of this chapter, to propose a holistic traffic cost model and its applications.

Towards a Traffic Cost Model Carrying traffic in an IP network incurs many costs, including transit fees, port costs, backhaul, and various other personnel and capital costs. How traffic is routed across a network and exchanged with neighboring ISPs can significantly affect the overall costs of routing traffic over the network. For example, the costs of

carrying traffic over trans-oceanic or satellite links is more expensive than routing traffic over underutilized commodity backhaul links; similarly, routing traffic over transit links incurs more cost than routing over settlement-free peering or customer links. Although traffic costs may not be the dominant cost in running a network, they can play a significant role in helping operators make decisions about planning, provisioning, and traffic engineering.

Currently, operators understand how individual elements contribute to operational costs, but they lack a holistic cost model that maps traffic flows to the costs of carrying the traffic. As a result, although business-level decisions about peering, provisioning, and interconnection may consider costs of individual elements (*e.g.*, the cost of peering, interconnection, or a committed rate), these decisions are currently *ad hoc*. For example, a decision about whether an operator should peer at a particular location should not only take into account the cost of that individual peering session, but also potential costs *saved* by sending less traffic over backhaul links. The inability to attribute costs to traffic flows can result in missed opportunities for cost savings and ad hoc decisions about routing and interconnection. Previous work jointly optimized cost and performance in a multihomed stub network [37], but no similar approach exists for transit networks or networks that peer in multiple locations.

Making decisions about traffic based on cost is challenging for two reasons. First, information about traffic costs is relatively inaccessible; if this information is available, it typically comes as individual cost elements, rather than as a holistic model. Further, some aspects of traffic costs are not linear (*e.g.*, commit rates, traffic symmetry constraints, 95^{th} percentile pricing), and these costs do not map naturally to *individual* flows. We solve this problem by developing a holistic cost model that attributes a cost to each traffic flow, which incorporates both interconnection and backhaul costs, as well as non-linear cost elements (like percentile pricing) with approximate functions. Using this model, operators can input values for various aspects of cost that they are likely to know from other sources; the model outputs an overall cost for routing each traffic flow. Second, the number of traffic

flows and the number of possibilities for routing each flow makes it difficult to efficiently find a solution that reduces cost. To solve this problem, we use our cost model to identify the most expensive traffic flows in the network and apply heuristics to move those flows to less expensive links. We also demonstrate how attributing costs to traffic flows can help operators in rationalizing planning decisions like peering location, peer selection and evaluating existing peering arrangements.

Our evaluation shows that network operators can realize significant cost savings by moving only a small fraction of overall traffic flows: For example, we find that, for three realistic cost scenarios, moving 10% of the flows that reduce traffic cost in the network can help operators achieve at least 65% of total possible cost savings.

Many network planning tools and techniques could build on our holistic cost model. We expect that our model might ultimately be coupled with tools that help network operators make the actually configuration changes to reassign these flows. It could also be incorporated with tools that help network operators perform forecasting, to better help make better decisions regarding network upgrades and provisioning. In Section 5.7 we discuss future research directions and issues with our cost model.

Cost-based Path Selection As we described in Chapter 1, network operators and end systems can have conflicting goals. Giving end systems unrestricted access to paths in the network can conflict with the network operator's goal of reducing the cost and uncertainty of traffic in the network. We develop a *cost-based path selection* framework that operators can use to only expose low cost paths to end systems. Network operators can use the cost model and use it in combination with a multipath routing scheme to create paths which have low cost. We present a sketch of how this can be achieved when using path splicing as the scheme for creating alternate paths and our traffic cost model.

Type	Components	Factors influencing	Examples
Usage-based	Transit fees, customer revenue	Dependent on geography, typically 95^{th} percentile pricing	$2-$5 in US to $40 in Asia
Port	Port costs, interface cards, installation fees	Price based on total volume of traffic exchanged *e.g.*1GE v/s 10GE port. Also depends if the peering is public or private	$550 to $2040 at LINX for 1GE v/s 10GE ports[1]
Misc.	Exchange fees, equipment costs	Depends on the Exchange	LINX charges $2500 as annual fees[1]

Table 5.1: Interconnect Traffic Costs.

5.2 Network Traffic Cost: A Model

We develop a model for reasoning about the various costs incurred by a network for carrying IP traffic. To build this model, we first need to understand the various components which contribute towards the cost of carrying traffic in a network. Based on where the cost is incurred, Figure 5.1 shows a breakdown of cost into two components: *interconnect costs* and *backhaul costs*. We further discuss what contributes to these cost components.

5.2.1 Interconnect Costs

We refer to the cost associated with the place where traffic is exchanged with neighboring networks, including providers, peers, or customers, as *interconnect cost*. Depending on the agreement between the two networks, a network might pay for transit based on the volume of traffic exchanged, be paid by the other network, or engage in settlement-free peering. Transit fees vary depending on the geographic location of the interconnect point (*e.g.*, transit fees vary from $\$2-5$ per Mbps in the United States to about $40 in Asia [26, 43]). For transit providers, which charge customers, the interconnect cost would be negative.

A network must also pay recurring *port costs* at a public exchange. These port costs

[1] data from LINX website

include costs associated with buying network interface cards and paying for installation fees for buying ports. The port and installation costs are dependent on the exchange and the transmission medium. At a public exchange, the member network can exchange traffic with other networks present at the exchange using a single port; adding a peer at the same exchange has no incremental cost, as long as the aggregate traffic from all the peers does not exceed the port capacity. If the traffic exchanged exceeds the port capacity then the network can pay for additional ports at the exchange. A private peering (sometimes called a "private interconnect") between two networks requires purchasing a separate port (and interface cards) for every neighbor network. Although private peering is more expensive than public peering, the traffic over the interconnect may be more predictable; this option may be cost-effective if two peers exchange a lot of traffic. In addition, there exist *other fixed costs* at an exchange, such as paying an annual (or monthly) fee for being a member of the exchange and a one-time installation fee.

To simplify reasoning about interconnect costs, we consider all of these cost components as *the cost per neighbor network at a given geographic location (PoP or exchange).* For the interconnect costs that depend on traffic volume, we make the above cost proportional to the rate of traffic (*in Mbps*), whereas for fixed costs, the operator can choose to amortize the cost over a period of, say, a few years. Although we may not know the current or accurate values for each of the contributors to interconnect costs, we expect that network operators can fill in the values for the different cost components.

5.2.2 Backhaul Costs

A network must carry traffic across its own network to reach either a customer network or send it to another neighboring network; a network incurs costs from three components: the circuits themselves, the equipment for the backhaul links (*i.e.*, routers, switches, etc.), and operational costs associated with running the network. Ultimately, our model incorporates these costs into a single quantity that corresponds to the physical distance the traffic must

be carried over its own network. We describe these costs in detail below.

In practice, *circuit costs* may fall into two distinct categories: (1) *Metro-range* costs, which are often negligible for large networks. A network may lease local network connectivity from other networks, however, in which case, the metro-range costs may be significant; and (2) *Regional* costs, which depend on the geographic location and the distance of the regional circuit. Although carrying traffic across backhaul links roughly correlates with the distance of the circuits, some regions are more expensive than others. For example, carrying traffic in the northeast corridor is more expensive than carrying it across other parts of the United States, and carrying traffic across transoceanic links is more expensive than carrying traffic across land. Our model can incorporate these differences with a distance function that depends on the location of the PoPs.

Capital costs, such as routers and switches, as well as the cost associated with their maintenance and upgrades, also contributes to the backhaul costs. Depending on the network, the model might reflect these costs as fixed; alternatively, it could amortize these costs over several years.

Operational costs, such as salaries for network engineers who keep the network running to paying for cooling and power consumption can also be amortized. In our model, we incorporate these costs as fixed costs and include them as a component of the cost of the collection of all backhaul links in the network.

5.2.3 Cost Model

Our cost model is based on the interconnect and backhaul costs that we described in the previous sections. Our model does *not* tell a network operator how to adjust the routing configuration itself to actually move a particular traffic flow from one path to another but can help identify which traffic flows should be moved and could provide input to "what if" configuration analysis tools (*e.g.*, [30]), or even a network designed around central routing control (*e.g.*, [29, 39]).

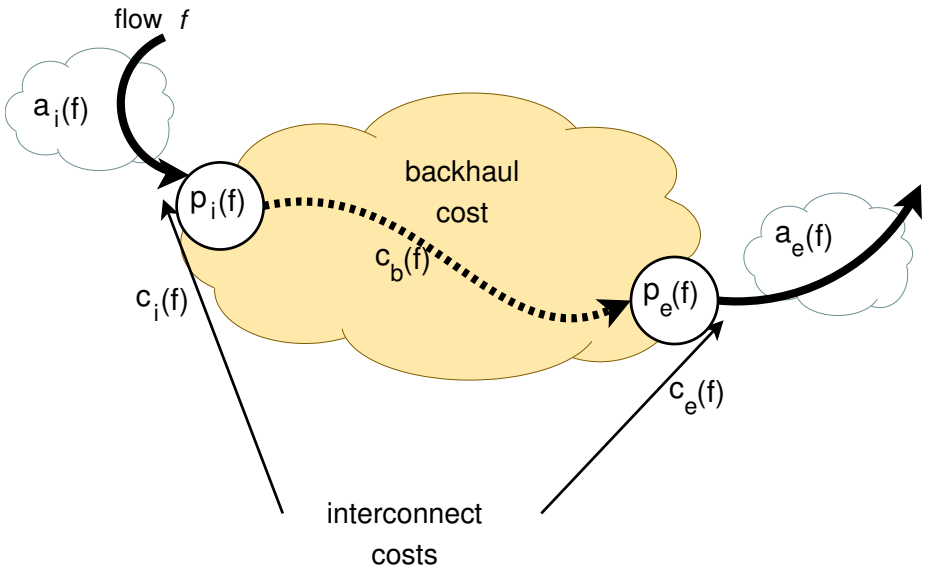


Figure 5.1: Classification of traffic costs for a flow f.

We now describe a formal traffic cost model. We can write the total cost of running a network as a sum of the fixed network costs and the usage-based costs.

Fixed Costs A network's fixed costs ($\mathscr{C}_F$) are defined by the network topology and relationships with the neighboring ASes. Although the backbone and interconnect network topology depend on the traffic the network is designed to carry, in the short term we assume the fixed costs are independent of traffic.

The fixed-cost component incorporates the fixed costs associated with the backbone and interconnect topology of a network. We abstract the backbone costs as the cost for the path between every pair of PoPs that exchange traffic. The cost component $c_{F,b}(p_1, p_2)$ is the fixed-cost component of the backbone path between PoPs p_1 and p_2. An alternate formulation could replace the fixed backbone cost with the cost of each backbone link in the network. The fixed interconnect cost component $c_{F,i}(a, p)$ is the fixed cost for the interconnect between neighbor AS a at PoP p. Because the fixed interconnect cost is summed over all pairs (a, p), it would be set to 0 where the neighbor AS a is not present at a particular PoP p.

$$\mathscr{C}_F = \sum_{p_1,p_2} c_{F,b}(p_1,p_2) + \sum_{a,p} c_{F,i}(a,p)$$

Usage-based Costs The usage-based component of the cost ($\mathscr{C}_U$) depends on the volume or rate of flow f, and the route that f takes in the network. The usage-based component has three sub-components, as shown in Figure 5.1. A flow f enters a network at an interconnect and the cost associated with that is the cost at the ingress interconnect ($c_{u,i}(f)$). The flow is then routed on the backbone with cost $c_{u,b}(f)$ and finally the flow egresses the network at an interconnect with egress interconnect cost of $c_{u,e}(f)$.

$$\mathscr{C}_U = \sum_f (c_{u,i}(f) + c_{u,b}(f) + c_{u,e}(f))$$

We will now describe how to calculate each of these cost components. For the usage-based cost components, each function and term refers to a flow f; thus, we drop f from the notation, and the usage-based cost of a flow is simply:

$$c_u = c_{u,i} + c_{u,b} + c_{u,e} \tag{5.1}$$

Usage-based Interconnect Cost The equations are symmetric for the ingress and egress points, and hence both of those interconnect costs have the same form. For a particular interconnect, the usage-based interconnect cost is: $u_i \cdot R + s_i \cdot R^\alpha$, where R is the volume (or rate) of the total interconnect traffic that is charged, u_i is the charge per volume (or rate). We use a concave function of the form $s_i \cdot R^\alpha$ to approximate certain types of costs like port costs, which are a step function of the traffic rate. Previous work focusing on peering contracts has estimated the value of α to be between 0.4 to 0.75 [20] using market price data. The unit usage-based cost parameter u_i, depends on the neighbor a and PoP p of the interconnect. The concave function parameter s_i is also unit per rate of traffic and depends on the PoP p and in some cases can also depend on the neighbor AS a through which f is

routed. For simplicity here we assume that s_i depends only on the PoP p. Thus, we have $u_i = \mathscr{U}_i(a,p)$ and $s_i = \mathscr{S}_i(p)$, where a is either the ingress or egress AS and p is either ingress or egress PoP. $\mathscr{U}_i(a,p)$ is the price per unit of exchanging traffic with AS a at PoP p. $\mathscr{S}_i(p)$ is also in units of price per unit of traffic volume (or rate) and depends on the PoP (or exchange) p the network is present. This price reflects the port costs, which are dependent on the PoP. The total interconnect cost is thus:

$$\mathscr{U}_i(a,p) \cdot R + \mathscr{S}_i(p) \cdot R^{\alpha}$$

Next, we need to find the contribution of flow f to the total interconnect cost where f is routed on (ingress or egress). Most transit pricing on the Internet is based on the 95^{th} percentile of traffic, where the transit provider charges for the traffic by removing the top 5% of the traffic. Because the customer pays for the 95^{th} percentile of the aggregate traffic at the interconnect, we need some method of calculating the per-flow contribution to that price. We use two techniques to approximate incorporating 95^{th} percentile pricing of interconnect links. Let r be the volume (or rate) of the flow f.

- **Linear function -** We assume that the 95^{th} percentile is a linear function of the average or the peak traffic rate at the interconnect, as has been empirically observed for different types of networks [27]. In this case, we calculate the per flow contribution by replacing R with some constant times the volume (or rate) of the flow. Thus,

$$c_{u,i} = \mathscr{U}_i(a,p) \cdot r + \mathscr{S}_i(p) \cdot r^{\alpha}$$

- **Shapley Values -** The drawback of the linear function approach is that it ignores the distribution of the flow across different time intervals which can influence the 95^{th} percentile price at the interconnect. Stanojevic et al. [84] propose the use of Shapley value [81] for computing the contribution of each flow to the 95^{th} percentile price of interconnect links. Because computing Shapley value is computationally infeasible for even a small number of flows, the paper proposes an approximation technique to estimate the Shapley values of the flows.

Usage-based Backhaul Cost The backhaul cost $c_{u,b}$ is: $c_{u,b} = r \cdot u_b$ and has two factors: the geographic location of the ingress PoP, p_i, and egress PoP, p_e, and the distance between them. The unit usage-based cost u_b depends on properties of the ingress and egress PoPs, and the distance between the ingress and egress. We model this as $u_b = \mathscr{U}_b(p_i, p_e) = \mathscr{R}(p_i, p_e) \cdot \mathscr{D}(p_i, p_e)$, where $\mathscr{U}_b(p_i, p_e)$ is the unit cost per traffic rate unit, $\mathscr{D}(p_i, p_e)$ is the distance between p_i and p_e, and $\mathscr{R}(p_i, p_e)$ accounts for the dependence of the usage-based backhaul cost on the ingress and egress PoP. For a flow f with rate r the usage-based backhaul cost is:

$$c_{u,b} = r \cdot \mathscr{R}(p_i, p_e) \cdot \mathscr{D}(p_i, p_e)$$

5.3 Applications of the Traffic Cost Model

In this section, we present examples to motivate and demonstrate the utility of understanding the traffic cost of flows in a network. These examples are empirical and have been developed after talking with network managers at different ISPs around the world. A number of decisions about how to route traffic in the network to reduce cost, and also long term planning decisions like which networks to select for peering, the location of peering can be rationalized and made with much more ease if cost can be attributed to traffic flows in a network. We classify these applications into two categories, based on whether the decision can be implemented with changes to existing routing configurations, or whether the changes require more fundamental modifications to existing peering relations. We present these from the perspective of a network referred to as "network X".

5.3.1 Routing Decisions - Cost Optimization

As shown in Figure 5.2, network *X* can route a flow arriving at PoP *S* via either PoP *A* or *B*. This situation could arise if network *X* peers with a particular neighbor at two locations, *A* and *B* for instance, and can choose to route traffic via either PoP. Further, as shown in Figure 5.3, network *X* may be able to route traffic to a particular destination via multiple neighbor ASes. The operator of network *X* thus has various choices for the egress AS

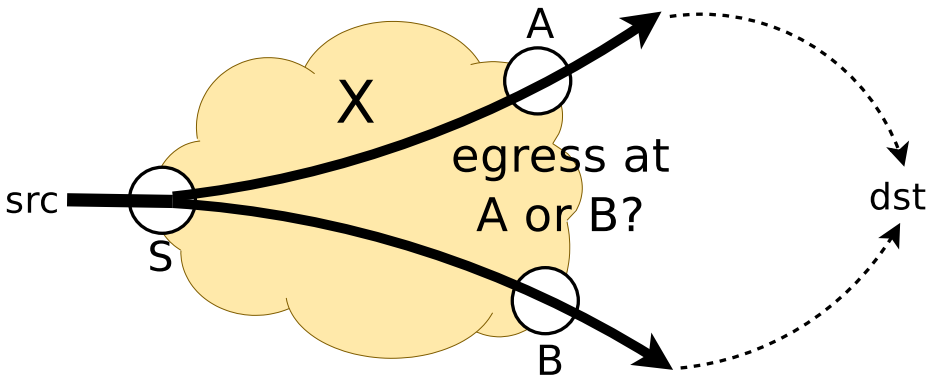


Figure 5.2: Which PoP to egress traffic to a prefix from?

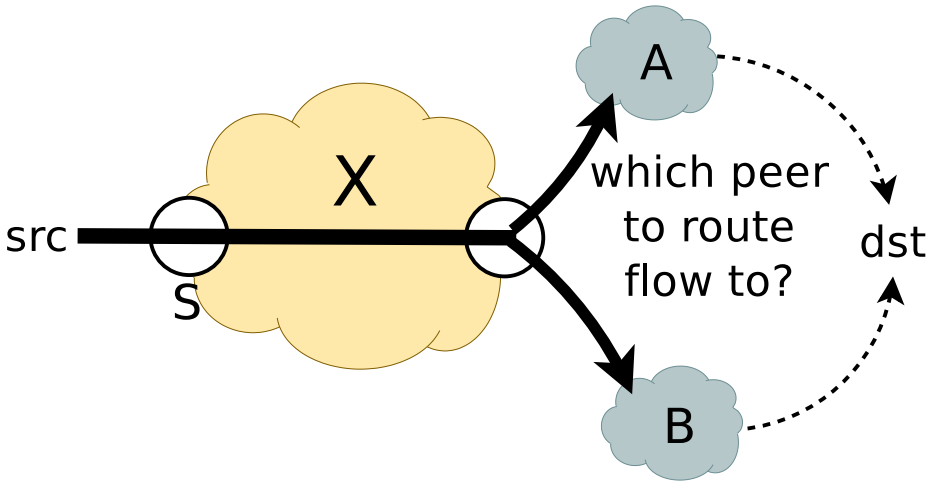


Figure 5.3: Which peer to send traffic on?

and egress PoP over which to route a given flow, from which *X* would prefer to use the cheapest *(egress AS, egress PoP) pair* for routing a particular flow. Network *X* can use the traffic costs, current routing, and topology information to to select the cheapest (egress AS, egress PoP) pair for each flow, thus minimizing the total cost. For each flow, the operator must account for the total cost (interconnect and backhaul) for routing that flow via each of the (egress AS, egress PoP) pairs. It is not enough to simply use the egress PoP that incurs the lowest backhaul cost for a flow (if, for instance, that egress PoP is closest to the ingress PoP), because the interconnect costs of exchanging traffic with ASes at that egress PoP may be high. If appropriate, the operator can also introduce capacity and performance constraints to avoid rerouting traffic in ways that might create congestion or introduce high

symbol	meaning
Inputs	
f	Set of s,d traffic flows
$\mathscr{V}(a,p)$	Availability of AS a at PoP p
$\mathscr{C}_i(a,p)$	Capacity of interconnect link with AS a at PoP p
$\mathscr{F}_b(p_1,p_2,l)$	Fraction of traffic between PoP p_1 and PoP p_2 sent over link l
$\mathscr{C}_b(l)$	Capacity of backhaul link l
$route(p,f)$	Route for flow f from PoP p, returns a PoP, AS pair
Outputs	
a_i	Ingress AS to which flow f is mapped
a_e	Egress AS to which flow f is mapped
p_i	Ingress PoP to which flow f is mapped
p_e	Egress PoP to which flow f is mapped

Table 5.2: Notation for optimization problem formulation.

performance penalty. We now describe this example and its formulation in more detail, and evaluate simple greedy heuristics to solve the problem.

Formulation Our formulation uses the traffic cost model from Section 5.2 as an input, along with additional routing information from the network. Given a network topology, routing information, and the set of s,d flows, our goal is to reduce the total cost of routing the flows while satisfying constraints on backhaul and interconnect links. Table 5.2 defines the new notation we introduce in formalizing the problem. Note that the optimization assumes that the network topology and neighbor AS relationships are fixed, hence the optimization only deals with optimizing the usage-based cost ($\mathscr{C}_U$) of carrying traffic flows in the network.

Inputs The input to the optimization problem is the complete set of s,d flows routed on the network, and the fully parameterized cost model that determines the usage-based interconnect and backhaul cost for routing each flow (as defined in Section 5.2). In addition, the optimization requires information about the capacity of the interconnect links and backhaul paths in the network. The optimization also takes as input information about availability

of a neighbor at the different PoPs in the network. We obtain the (egress PoP, egress AS) pair for each flow f at PoP p based on the destination d of the flow from the routing table dumps at each PoP.

Output The desired output is the routing configuration that minimizes the total cost of routing every flow. This takes the form of a *mapping*, which defines the ingress AS, ingress PoP, egress AS and egress PoP for every flow f. The output variables are listed in Table 5.2. The output of the optimization configures routing in the network at the granularity of prefixes. The realization of the routing decisions may be complicated, depending on how the network is configured, but, fortunately we find that that most of the cost benefits can be achieved by routing only a small fraction of the flows. We discuss a few possible ways of making it easier to implement in Section 5.5.

Even though the formulation we have described can determine each of the ingress interconnect and egress interconnect, there are important differences between the ingress and egress mappings. Changing the ingress AS and/or PoP for a flow depends on neighboring and remote networks. For instance, attempting to change the ingress AS for a flow f using AS path prepending requires remote ASes to prefer short AS paths. Changing the ingress PoP for a flow involves negotiating hot/cold-potato routing with neighboring ASes. On the other hand, given a destination prefix, the network has complete flexibility in choosing to route traffic towards that prefix via any neighbor AS that advertises that prefix. Similarly, the network can choose among multiple PoPs where a particular neighbor AS may be peering. Given that the network cannot deterministically control the ingress mapping for a flow, we choose to retain the ingress mapping. We assume that the network can only control the egress mapping for a flow f, *i.e.*, the network can route the traffic internally, and choose the appropriate egress AS/PoP to reduce overall cost of routing that traffic.

Objective Function The objective is to minimize the total cost of network traffic. From the cost model developed in Section 5.2, there are two types of costs associated with each

flow: interconnect and backhaul costs. Thus, the *objective* is to minimize the total usage-based cost over all the flows in the network.

Constraints The routing configuration that minimizes costs must also satisfy capacity constraints in the network. We consider two categories of capacity constraints: interconnect link capacities and backhaul link capacities. Operators can add other constraints; for example they might restrict the set of available paths to achieve certain performance guarantees. These can also be modelled as linear constraints but would require additional information about performance of various paths. We give examples of how to model such constraints:

- **Interconnect link capacity constraints**

  If $\mathscr{C}_i(a,p)$ is the capacity of an interconnect link with AS a at PoP p, then the total rate of flows that map to the AS a and PoP p should be less than the capacity $\mathscr{C}_i(a,p)$. Because the optimization does not change the ingress mappings of the flows, the constraint only applies to the egress interconnect links. Formally:

  $$\sum_{f:a_e,p_e=a,p} r \leq \mathscr{C}_i(a,p) \qquad \forall a,p$$

  To obtain $\mathscr{C}_i(a,p)$ for peering links that have strict requirements for traffic ratios, we sum the total ingress traffic from the particular AS a and make $\mathscr{C}_i(a,p)$ satisfy the traffic ratio guarantee that the egress traffic would not exceed $x\%$ more than the ingress traffic from the neighbor. Thus, we can write:

  $$\mathscr{C}_i(a,p) = k \cdot \sum_{f:a_i,p_i=a,p} r$$

  Because we do not modify the ingress mapping of traffic, the right side is a constant for a given traffic matrix.

- **Interconnect Traffic Ratio constraints**

  We show here an example of how to incorporate real-world agreements between a

network and its neighboring ASes as constraints for the cost optimization. There are a number of agreements that require neighboring ASes to maintain a certain traffic ratio to each other. Such constraints are of the form, that the egress traffic from a network to its neighbor would not exceed $x\%$ more than the ingress traffic from the neighbor. Such constraints can be mapped as:

$$\sum_{a_e, p_e = a, p} r \;\leq\; \text{const} \cdot \sum_{a_i, p_i = a, p} r \qquad \forall a, p$$

Because we do not modify the ingress mapping of traffic, the right side of the above constraint is a constant for a given traffic matrix.

- **Backhaul link capacity constraints**

  If the complete internal network topology and routing configuration of the network is known, then it is possible to infer the complete path taken by a flow f in the network. The backhaul link constraints should capture the fact that the total rate of all flows that are routed over a backhaul link should not exceed the capacity of that link. Let $\mathscr{F}_b(p_1, p_2, l)$ be the fraction of traffic between PoP p_1 and p_2 sent over backhaul link l. In case the network is using only single-path shortest path routing, then $\mathscr{F}_b(p_1, p_2, l)$ will be zero or one. However, networks often deploy MPLS or ECMP to split traffic between PoPs over a number of paths. If $\mathscr{C}_b(l)$ is the capacity limit for the backhaul link l, then the capacity constraint is:

$$\sum_{p_1, p_2} \sum_{\substack{f: \\ p_i, p_e = p_1, p_2}} r \cdot \mathscr{F}_b(p_1, p_2, l) \;\leq\; \mathscr{C}_b(l) \qquad \forall l$$

- **Performance constraints**

  The assignment of traffic flows in the network which only optimizes the cost of the traffic in the network could in cases lead to traffic flowing on paths which are not equivalent in performance. Depending on the type of network, performance can be either be the most important metric for assigning flows or could be a desirable

objective to achieve. In networks where performance is absolutely critical, the performance constraints can be introduced as hard constraints in the formulation which must be met, for example, if $lat(dst, pop, as)$ is the latency of sending traffic to prefix dst via egress PoP pop and peer as, and $max\_lat(dst)$ is the max latency for the prefix, then the constraint can be modelled as:

$$\sum_{pop} \sum_{as} map(dst, as, pop) \leq max\_lat(dst) \quad \forall dst$$

There could be other constraints which the operator may wish to incorporate in the cost optimization which restricts the traffic to certain paths in the network.

Solving the Optimization Solving the optimization involves determining the egress mappings for every flow so as to minimize the total cost of all the flows, subject to the various capacity (link and PoP) constraints. The capacity constraints restrict the amount of traffic that can be routed on a particular interconnect link or backhaul path in the network. This is similar to the bin packing problem where objects (flows in this case) are assigned to bins (links in this case) and the bin has a fixed capacity and each object has a fixed size (rate of the flow), which is NP-hard. Simple greedy approaches yield good approximate solutions for the bin-packing problem (*e.g.*first-fit, best fit decreasing and first fit decreasing). We develop a simple heuristic to find a good approximate solution. We present a simple greedy assignment that respects the interconnect and backhaul capacity constraints. The greedy assignment assigns a flow to the lowest cost path on which it can be routed while respecting the interconnect and backhaul capacity constraints.

The traffic flow assignment is not a direct mapping to bin packing, so we use the following variation of the first fit decreasing strategy. We consider flows in decreasing order of their cost and assign each flow to a path that has enough backhaul and interconnect capacity, and has the least cost among all such paths. This is different from bin-packing because a flow can only be assigned to a limited set of paths, and there is a different cost associated

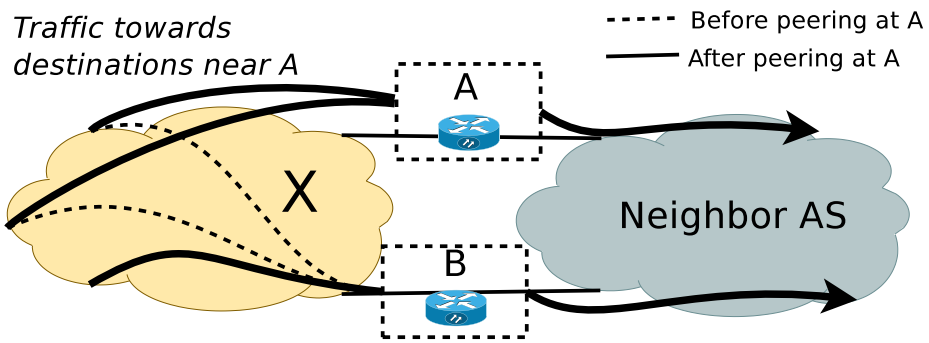


Figure 5.4: Peering Location Decision.

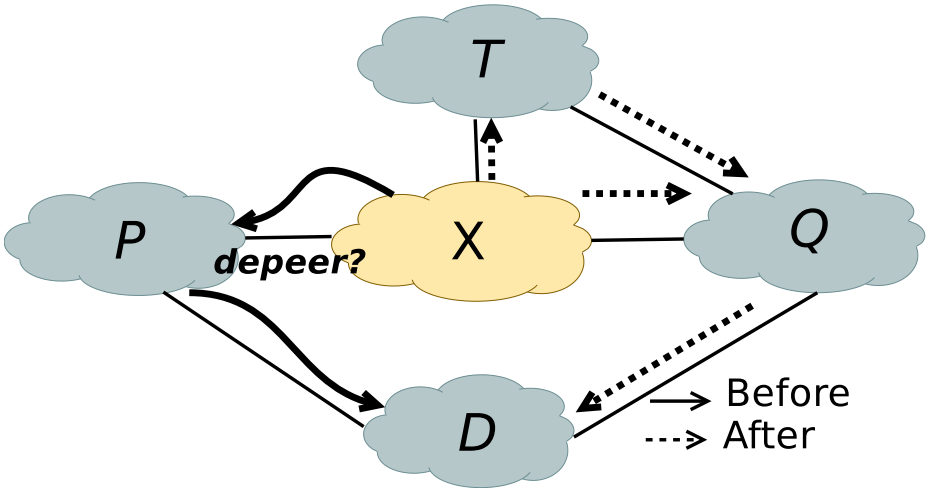


Figure 5.5: Existing Peering Contracts.

of being assigned to any particular path. We show the results from our greedy assignment, using two different methods of attributing interconnect costs to flows, in Section 5.4.3.

5.3.2 Planning Decisions

Another useful application of mapping traffic flows to their associated costs is to to identify potential opportunities for reducing cost or increasing revenue by re-evaluating existing interconnections. A network operator or planner may wish to evaluate the locations where the network is peering with a particular AS, or it may wish to evaluate the profitability of peering with that AS at all. We present two examples here:

Determining Peering Locations Network X can use the available cost information and, based on its current traffic demands, estimate how peering with a neighboring network at additional locations might affect the overall cost of carrying traffic. Such a decision will depend on the costs of transporting traffic over various backhaul links, as well as the costs of various interconnection and peering points along the path. For example, as shown in Figure 5.4, network X might have a significant amount of ingress traffic near a certain location, A, that is also destined for locations near A in a neighboring network. Depending on the cost of interconnecting at A relative to backhaul costs (*i.e.*, if interconnection is less expensive than carrying the traffic to B via backhaul links), it may make sense for network X to also peer with this neighboring AS at location A. If, on the other hand, a second peering location B offers more attractive pricing (*e.g.*, port costs and exchange fees could be lower at B), it may be more profitable to simply send all of the traffic to the neighbor through a peering location at B.

Evaluating Existing Peering Contracts The peering relationships of network X are beneficial to X *when they are created.* Over time, network X may connect to additional peers, or the traffic flow and interconnection costs may change sufficiently for the peering link to no longer be beneficial to network X. An operator or planner at network X may want to periodically re-evaluate the value of peering with a certain AS. Figure 5.5 shows network X and its peer P. When X created a peering relationship with P, it may have been less expensive to route traffic destined to D via P, as opposed to using a transit provider, T. Over time, however, transit provider T might offer a better price, or the backhaul cost of routing traffic to T might decrease; X might add another peer Q that can route traffic to the same destination D. The operator of network X must continually re-evaluate whether there is value in continuing to peer with P. For example, the operator may wish to compute the cost for routing traffic towards a customer AS, D, if it depeered P and instead routed this traffic over either T and/or Q. In Section 5.4.4, we describe a method that X can use to

evaluate the value of an existing peering contract with peer P.

Evaluating potential peering contracts Networks periodically engage in peering trials to gauge the effect of the new peering relation because they cannot evaluate both the performance and cost effects of a potential contract. Lack of this knowledge makes evaluating potential peering contracts cumbersome, thus making it difficult (if not impossible) to identify these new peering opportunities. Such data data is usually gathered by the engineering team, which compiles reports and sends it to the sales and marketing teams, who intern negotiate contracts with the particular networks. This *ad hoc* approach can result in many missed opportunities for beneficial peering contracts and an inability to evaluate the real value of certain peering contracts. In Section 5.4.4, we demonstrate how network X can use the traffic cost model to discover new peering opportunities.

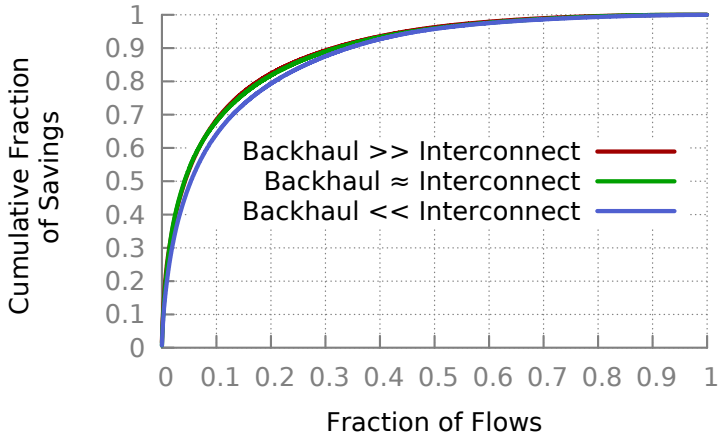
5.4 Evaluation

In this section, we evaluate the different applications of the traffic cost model as described in Section 5.3. We evaluate the greedy algorithm to reduce cost of routing traffic in the network and the two planning decision examples described in the previous section.

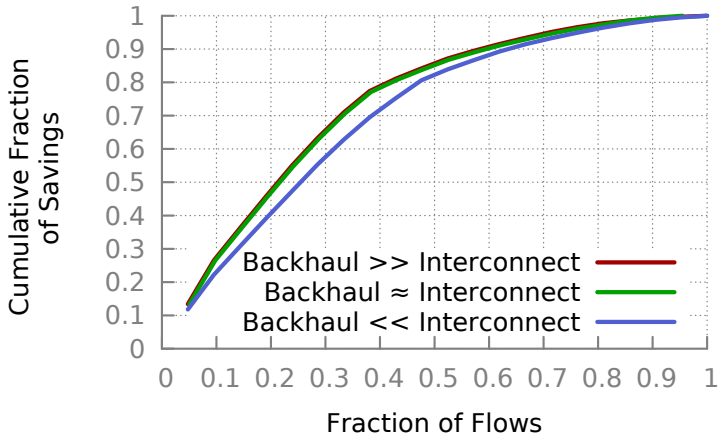
5.4.1 Setup

We use traffic flow statistics, routing data, and topology data from a large access provider in the UK. The traffic statistics consist of packet sampled (1 in 1000) NetFlow data from a weekday in July '09. The routing data consists of full BGP routing table dumps from the edge routers and the complete ISIS topology for the network.

We extract flow-level statistics from the NetFlow data that gives us the traffic (in bytes) between every s,d pair, where s is the source prefix and d is the destination prefix. The s,d pair defines a flow f; we compute its rate r by dividing the total bytes transferred by the duration of our measurement. Combining this flow-level data with available BGP and IGP routing data, we obtain the path in the network for each flow f. We draw the unit traffic



(a) Linear Function



(b) Shapley Value

Figure 5.6: Fractional savings for greedy heuristic with capacity constraints. These plots show the cumulative fraction of savings for the fraction of flows that are reassigned.

costs from the range $[1, 10]$ per Mbps of traffic; this range corresponds to publicly available pricing data. We evaluate three different scenarios for different relative prices of backhaul and interconnect cost:

- **Backhaul $\approx$ Interconnect:** We scale the unit backhaul cost $\mathscr{U}_b(p_i, p_e)$, to be in the same range as the unit interconnect cost, *i.e.*$[1, 10]$ per Mbps of traffic.

- **Backhaul $\gg$ Interconnect:** This simulates the scenario where transit prices and peering costs are very low, due, possibly, to competition in the transit market or the presence of Internet Exchange Points (IXPs). We model this scenario by keeping the

Table 5.3: Most flows have two available points of egress that allows for potential cost savings by moving traffic from one egress to another.

# of egress points	% of flows
1	**25.74**
2	**73.43**
3	0.36
4	0.03
5	0.03
6	0.36
7	0.03
8	0.027

backhaul costs in the range $[1,10]$, but draw the unit interconnect cost from the range $[0.1,1]$ per Mbps of traffic.

- **Backhaul $\ll$ Interconnect:** Represents the case where transit prices and peering costs are much higher than backhaul costs. This could be the case in regions where certain ISPs have monopolies in the transit market, and peering opportunities are limited. We model this scenario by keeping the backhaul costs in the range $[1,10]$, but draw the unit interconnect cost from the range $[10,100]$ per Mbps of traffic.

These three scenarios can represent the cost structure for links in different types of networks. For example, transit providers may have high backhaul but lower interconnect costs; on the other hand, large content providers might have relatively higher interconnect costs.

5.4.2 Shapley Value Computation

We estimate the Shapley values for a subset of flows at every interconnect link in the network. The computation of Shapley values quickly becomes computationally infeasible even for a small number of flows. Hence, we use the approximation technique described in [84]. The complexity of estimating Shapley values for a given interconnect is $\mathcal{O}(|flows|^2 * K)$, where K is the number of permutations used. Thus, for a fixed number

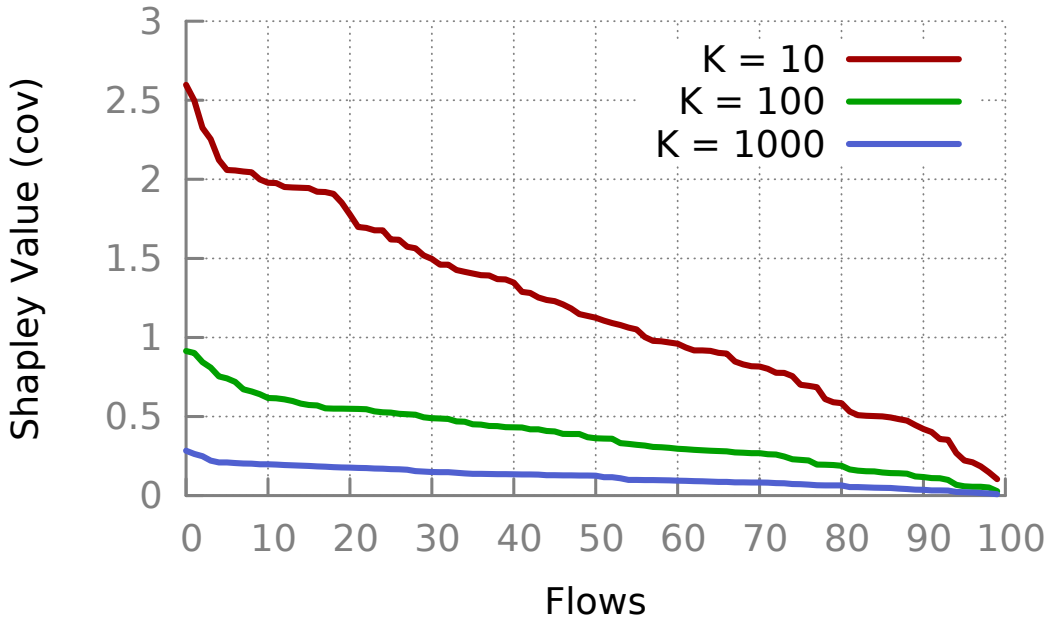


Figure 5.7: Co-efficient of variation for estimated shapley values for flows at a particular interconnect link. The values are computed over 100 runs.

of flows the smaller the value of K, the faster the computation but it also implies higher variation from the true Shapley values. We computed the coefficient of variation (CV) for the Shapley values for a fixed set of flows at a particular interconnect for $K = 10, 100, 1000$. We found that the CV is $> 100\%$ for most flows for $K = 10$, between 50% and 100% for most flows for $K = 100$ and $< 30\%$ for all flows for $K = 1000$, with a median of 11%. Even though, it is computationally infeasible to calculate the ground truth, the above results show that for $K = 1000$, the exact permutation used has little effect on the estimated Shapley values. We use this value of K in our evaluation.

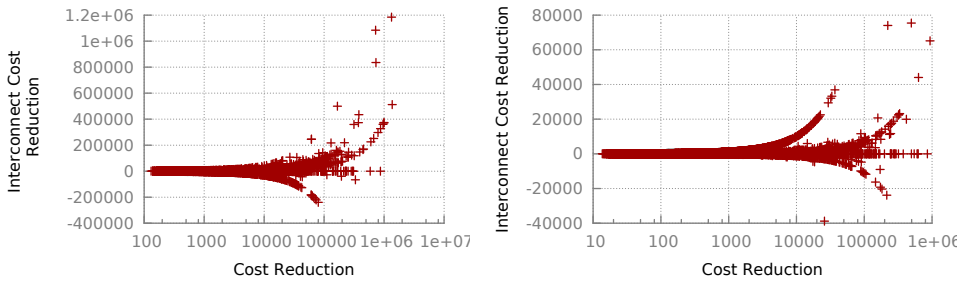
Figure 5.7 shows the co-efficient of variation for the shapley values computed for a fixed set of flows at a particular interconnect for three different values of K. We find that the for $K = 1000$ for all flows, the estimated shapley values deviate by $< 30\%$ from the mean. We use this value of K for computing the shapley values of flows in our evaluation.

5.4.3 Greedy Cost Reduction

We evaluate the greedy cost optimization described in Section 5.3.1. We aim to demonstrate the benefits that can be obtained by using a simple greedy strategy. Table 5.3 shows that for around **73**% of flows, an alternate path is available, but a very small fraction of flows have more than two alternate paths. Here, we assume that the network operator has a target utilization of 30%, i.e., the capacity for interconnect and backbone links.

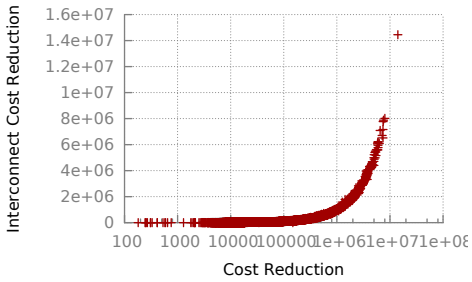
We evaluate cost savings using two different techniques for calculating the flow contribution to the interconnect costs as described in Section 5.2.3. The cost saving results for the three different scenarios of backhaul and interconnect costs are shown in Figure 5.6. When using *linear function* (Figure 6(a)), moving only the most expensive 10% of flows that have alternate cheaper paths achieves 68% of the maximum possible saving (in the case of Backhaul $\approx$ Interconnect). This result is significant, because the network operator may not wish to reassign many flows, since doing so might require large changes in routing configuration or entail disrupting a large fraction of traffic. When using *Shapley values* (Figure 6(b)), moving the most expensive 30% of the flows achieves 65% of the maximum possible saving (in the case of Backhaul $\approx$ Interconnect). Since the greedy strategy assigns flows in the order of their original cost and also obeys the capacity constraints, it can lead to some flows flowing on routes which are more expensive than the original route, but significant cost savings are possible regardless.

Next, we study the relative contribution of interconnect and backhaul cost savings to the total cost savings, for each cost scenario. Figure 5.8 shows, for each reassigned flow, the cost savings for that flow due to reduction in the interconnect cost (y-axis) as a function of the total cost saving (x-axis). As expected, we find that when Backhaul $\ll$ Interconnect (Figure 5.8(c)), almost all the savings are due to reduction in the interconnect cost for the reassigned flows. On the other hand when Backhaul $\gg$ Interconnect (Figure 5.8(b)), there are a number of reassigned flows for which the interconnect cost actually *increases* (negative interconnect cost saving). For these flows, however, the backhaul cost savings are



(a) Backhaul≈Interconnect

(b) Backhaul≫Interconnect



(c) Backhaul≪Interconnect

Figure 5.8: Contribution of interconnect cost savings to the total cost savings for the three different cost scenarios of cost.

sufficient to give a positive total saving. This finding highlights the importance of optimizing *both interconnect and backhaul costs* for flows. If the operator considered interconnect costs in isolation, he would miss certain cost saving opportunities.

Using path bits to select paths of similar cost As shown in Table 5.3, a large number of flows have an alternate egress location available. In our dataset, each flow has on average 1.77 alternate paths. Using our cost model, we evaluate the cost of each of these alternate paths for a flow. Our goal is to identify paths of similar cost, so that network operators can use path bits to expose these paths to the end system (or customers). We fix a threshold for how much the cost of an alternate path can exceed the cost of the cheapest path for a flow and we measure the average number of alternate paths available for different values of the allowed threshold. Our result is summarized in Table 5.4. The results vary based on the particular cost scenario and also the allowed threshold. We see that if we allow the cost of the flow to vary by upto 10% from the cheapest path, then for Backhaul ≈ Interconnect,

Table 5.4: Number of paths with similar cost, depending on the difference between the path cost and that of the cheapest path for a particular flow.

Type	Threshold	Avg. # of similar paths
Backhaul $\gg$ Interconnect	5%	1.053
	10%	1.126
	20%	1.287
	30%	1.335
	50%	1.409
Backhaul $\ll$ Interconnect	5%	1.009
	10%	1.051
	20%	1.160
	30%	1.296
	50%	1.491
Backhaul $\approx$ Interconnect	5%	1.096
	10%	1.240
	20%	1.317
	30%	1.473
	50%	1.608

we get about 1.24 paths on average which is about 30% fewer than the best possible.

5.4.4 Peering Decisions

An important application of having a holistic traffic cost model is that it can enable networks to perform "what-if" scenario evaluations. Now, we describe the evaluation of two "what-if" scenarios (described in Section 5.3.2) as a way of demonstrating the utility of our traffic cost model. We skip the results of how the cost model might help operators identify potential peers. We use the linear function for calculating interconnect costs for these examples.

Peering Location Evaluation For an existing peer A, we consider each PoP where the network does not currently peer with A, and try to route existing flows (which use A as the egress AS) via the new PoP. We calculate the total cost of routing flows after adding the new PoP and pick the additional PoP which gives the maximum cost savings for the peer A. For our analysis, we ignore any capacity constraints while reassigning flows and assume

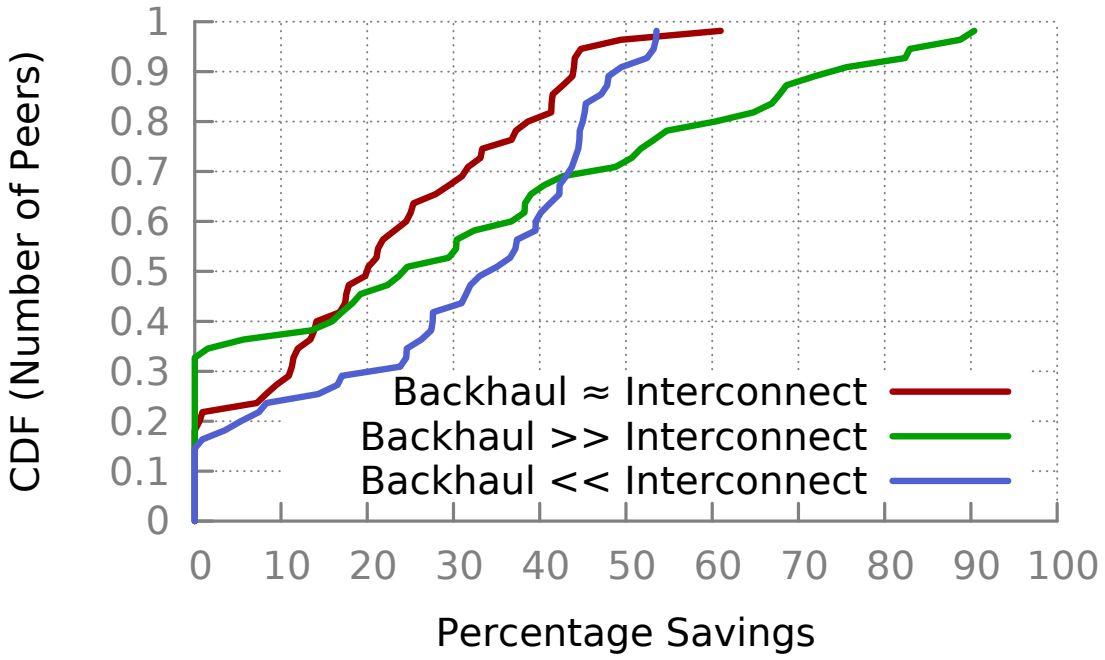


Figure 5.9: CDF of benefits (% savings) from selecting a new PoP for a neighbor done for different scenarios of the cost function.

that A is available for peering at each additional location. It is easy to extend our method to include capacity constraints and the availability of peer A at the new peering location.

Figure 5.9 shows the CDF of savings by selecting one additional peering location for each existing peer. We find that when Backhaul $\gg$ Interconnect, the benefit of adding a peering location with an existing peer depends on the peer. For about 35% of existing peers, there is no benefit from adding an additional peering location, perhaps because the network already connects to certain peers at the best possible PoP. On the other hand, for some peers, adding an additional peering location saves $> 80\%$ of the current cost of routing traffic via that peer. The following example explains how this could happen. Suppose that most of the traffic that X routes via A enters X at a certain PoP p_i. If there is an egress PoP p_e close to p_i, then adding a peering location with A at p_e will yield significant backhaul cost savings.

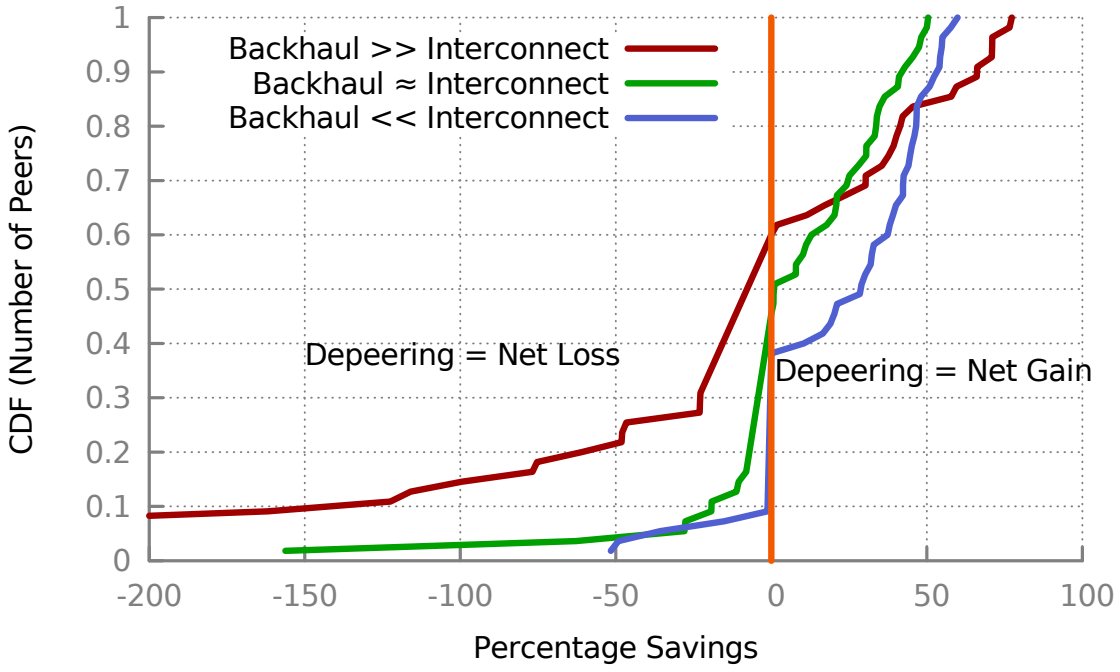


Figure 5.10: CDF of benefits (% savings) from depeering for different scenarios of the cost function.

Existing Peer Evaluation There are a number of reasons why networks peer with each other, such as to save costs for traffic which would otherwise be routed via a transit provider. As we described in Section 5.3.2, network X may wish to periodically re-evaluate the value of the peering link with an existing peer A. We describe a method using which network X can estimate the value of a peering link with a neighbor. For a neighbor AS A, we try to reassign each flow that was routed via A to some other AS. If a flow cannot be routed via any other AS, then we assume that that flow must be routed via a transit provider, and charge it by the maximum rate. We then calculate the difference in total cost by reassigning the flows which used A as egress. This is the net saving for network X by depeering network A. If the net saving is negative, then it makes sense for network X to keep peering with A, while if the net saving is positive, then network X would benefit from depeering A.

Figure 5.10 shows the CDF of the net saving from depeering each neighbor. The CDF is skewed, showing that some networks are extremely beneficial. We find, however, some peers for which the net saving is positive, i.e., X would benefit by depeering these peers.

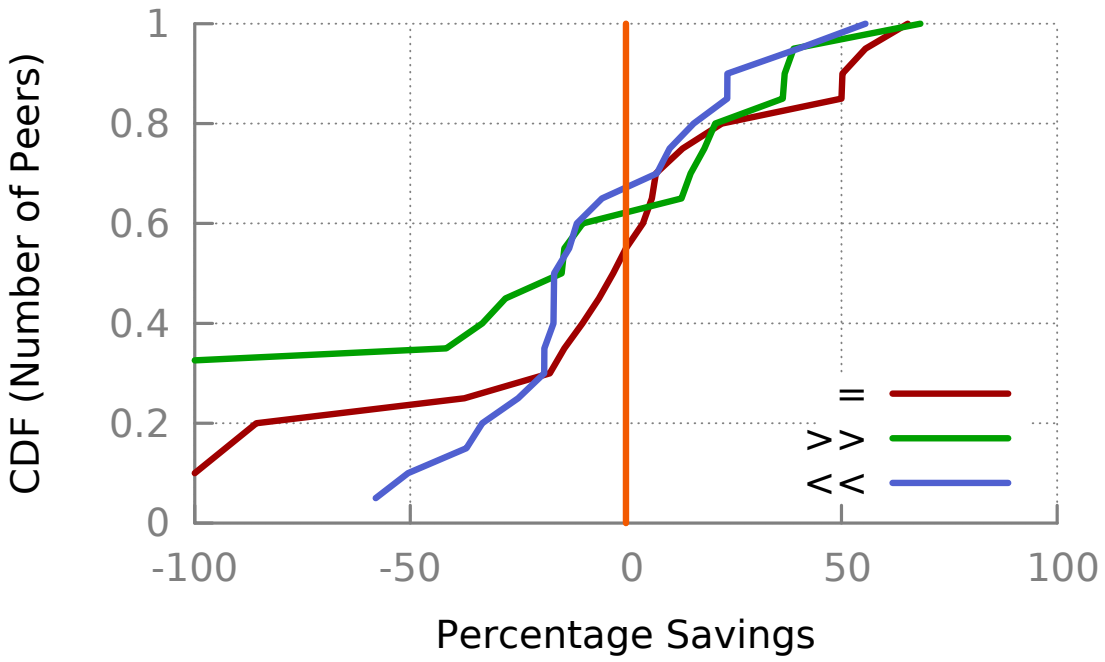


Figure 5.11: CDF of benefits (% savings) from selecting a new network to peer with done for different scenarios of the cost function.

When Backhaul $\gg$ Interconnect, a smaller fraction of peers gives a net saving. The intuition is that moving traffic from an existing peer to another peer (which may offer cheaper interconnect) may lead to a large increase in backhaul cost, and no net saving. But, when Backhaul $\ll$ Interconnect, we see a larger number of peers which X can benefit by de-peering. This is because moving flows from an existing peer to other peers offering less expensive interconnect is beneficial, even if it involves carrying the traffic for longer distances on less expensive backhaul paths.

New peering opportunities As we described in Section 5.3.2, the operator of network X may wish to look for new peering opportunities, based on the current traffic and cost profile. The key question that the operator would like to answer is *"What is the potential benefit of peering with a network that is currently not a peer?"*. We describe how the operator of network X can use the cost model to determine the potential benefit of adding a new peer. To find potential peers, we look at the routing announcements from the different

peers of network X and find the set of networks which are currently not peers, but are on the path from X to remote destinations. Let R be such a remote network. We "simulate" peering with R, by assigning the traffic flows traversing R (via current peers) directly to R, and calculate the cost of those flows. The cost saving by peering with R is the total savings for all such flows. For each remote network R, we can then calculate the cost saving of adding a peering link to R.

Figure 5.11 shows a CDF of potential savings from new peerings. We find that peering with only about 40% of remote networks yields cost savings. We conjecture that this is due to the nature of network X (a large access provider) that we use in this evaluation. Network X has an almost open peering policy, and consequently, not many potential cost saving peering opportunities are available. However, we expect our technique to be much more useful in cases of networks with restrictive or limited peering. We plan to apply our technique to such networks in future work.

5.5 Cost-based Path Selection

The path selection architecture can be successfully deployed if networks (or network operators) have sufficient incentive for exposing more than one path to end systems. Indiscriminately exposing paths in the network to end systems is not optimal for operators because end systems might favor paths which are expensive. Hence, we develop a cost-based path selection framework in this section. In this section, we focus on describing how a network can implement a cost-based paths selection architecture using path splicing as an example of a scheme used to create multiple paths in the network. However, the approach is general enough to replace path splicing with a different multipath generation scheme with only little changes.

We also show how to add feedback in the model, where traffic characteristics could change, pricing by neighbors change, etc which can affect the cost of the traffic and hence, would require operators to re-optimize the network.

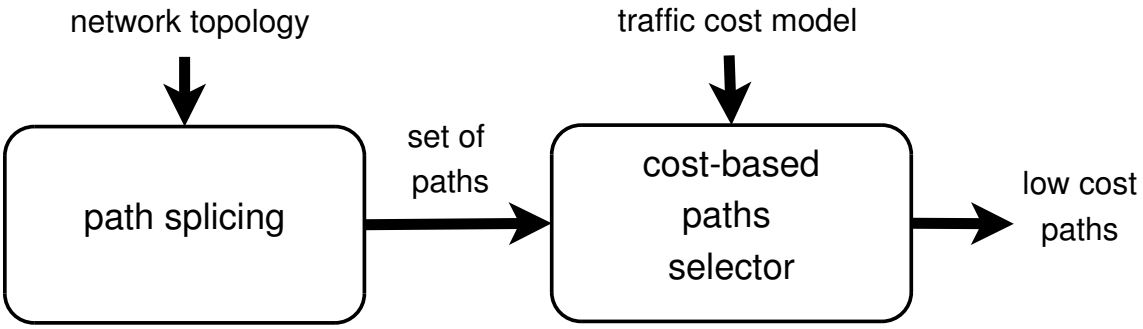


Figure 5.12: Cost-based path selection framework

5.5.1 Centralized Controller

We leverage recently proposed centralized network control plane architectures like RCP [17], 4D [97, 39]. OpenFlow [64] also utilizes a centralized controller [18] for making changes in the OpenFlow network devices deployed in the network. This shows that the use of centralized control plane architectures or centralized decision making is becoming popular among networks.

Figure 5.12 shows how the operator can use a centralized controller as a "low cost paths selector". The centralized controller can take as input the network topology information and use a multipath scheme like path splicing (as described in Chapter 3) to compute multiple paths in the network. These multiple paths are then sent to another module, *cost-based path selector* that takes as input the multiple paths computed by path splicing, traffic cost model and the observed traffic matrix to compute a set of *low cost paths*. These paths can then be configured by the controller in the network devices. For example, using OpenFlow protocol, the controller can translate the computed paths into forwarding rules which can then be installed in the network devices, similar to the solution for implementing splicing in a centralized fashion in a network as shown in Figure 3.5. A number of proposals utilize this feature of OpenFlow to control paths in the network for performance [51] or security reasons [18].

5.5.2 Feedback mechanism

Routing protocols are dynamic in nature and respond to changing network conditions. For example, in the event of a link or node failure, the routing protocols compute an alternate path to avoid using the failed link or node if such a path exists. A practical cost-based path selection framework must similarly be dynamic and respond not just to network failures but also changing traffic patterns in the network. Thus, there is a need to incorporate feedback in the path selection procedure. Failures can be dealt with using the multipath mechanism or using an active failure recovery mechanism like fast reroute [79]. We focus here on how to respond to changing traffic patterns in the network and provide a sketch of how a centralized path selection controller can incorporate feedback.

As shown in our cost model, interconnect and backhaul costs are determined based on the traffic flowing over the interconnect links and backhaul paths. Usage-based interconnect and backhaul costs are based on the amount of traffic flowing over the particular interconnect and backhaul links. If a number of end systems start preferring paths going through a particular interconnect link that can increase the cost of the traffic cost at that link. In this case, the controller must compute new set of paths based on the changing traffic patterns. There are a number of practical concerns like how often should the controller recompute the low cost paths which we do not address in this dissertation.

5.6 Related Work

In this section, we describe the large body of related work in the area of improving routing in a network to achieve goals like better performance, lower traffic costs or both. For presentation convenience, we split the related work in three broad categories.

Optimizing cost for multihomed stub networks A number of recent studies has focussed on multihomed stub networks to formulate problems for optimizing the cost and performance in such networks. Akella et al [4] showed the benefits of multihoming and

how to select ISP providers to maximize performance gains. Goldenberg et al [37] studied the problem of optimizing cost and performance in a multihomed network and proposed an optimization based approach for solving the problem. Wang et al [91] generalize the previous work to include in the optimization formulation, the set of all available ISPs to the multihomed network and propose a dynamic programming approach to solving the cost optimization. These studies are applicable only for the set of multihomed stub networks and the approach described do not generalize to the problem of a general network, which has a large number of potential ASes, and the relation between the neighboring AS can be of a customer-provider or that of settlement-free peering. Also, the previous works use a very simple cost model which is just based on the unit rate pricing for purchasing bandwidth from the providers. Finally, the formulations do not consider any backhaul costs associated with carrying traffic within the network nor do they consider the possibility of traffic ingressing (or egressing) from the network at different locations.

Reducing traffic costs in data center environments Although performance is the most important metric for optimization in datacenter environments, cost considerations have become significant due to the increase in the size and footprint of datacenters. Qureshi *et al.* [73] propose an approach to routing in a data center environment to exploit the disparity in energy prices at different geographic locations and demonstrate savings in energy costs. Zhang *et al.* [101] perform an optimization based on performance and cost for datacenter applications where performance is critical.

Optimizing path performance A number of overlay routing schemes [9, 41] suggest building overlays to find better performing paths. A number of commercial products use Intelligent Route Control to select among one of the egress providers for selecting paths with good performance [46, 76]. Our work considers traffic costs for selecting paths in the network. Also, our work is not restricted to only stub networks but is more generally applicable for transit, access and content providers as well.

5.7 Discussion and Summary

In this chapter, we developed a holistic cost model for associating costs for individual traffic flows in the network and showed how a network operator could use such a cost model to reduce the cost of forwarding traffic both across backhaul links in a network and at interconnection points. Although network operators currently apply some heuristics to control the cost of network traffic, they lack a holistic cost model that incorporates all contributors to the cost of forwarding individual traffic flows. This paper presents the first such cost model for network traffic, which we believe could serve as the foundation for many applications that could help network operators control network costs. For example, with knowledge of *future* backhaul and interconnection costs, network operators could evaluate the benefits of establishing peering connections versus provisioning more capacity on backhaul links.

Operators could also use our cost model to jointly optimize cost and performance, as previous work has done for stub networks [37]. The cost model could also be integrated with a configuration tool that helps an operator determine a set of configuration changes that could achieve the appropriate re-mapping of traffic flows; alternatively, a controller (as in RCP [17] or 4D [39]-like networks) could directly map flows onto the appropriate paths.

There are a number of directions that require additional work. Instead of random interconnect costs we used in our evaluation, it would be useful to work with real cost data. We used Shapley values for distributing interconnect costs among flows; if the number of flows is large, however, computing these values is expensive. Another important avenue of exploration is to incorporate feedback in this model. When a network reroutes flows to reduce cost, it can potentially affect incoming traffic patterns, making the resulting cost sub-optimal. This would require further routing changes, which could again change the traffic patterns. We plan to explore the conditions under which our cost-based routing optimization converges to a stable routing configuration. We are exploring the feasibility of a tool that continuously monitors traffic patterns and cost information and re-optimizes the

routing to reduce traffic costs in the network.

We also showed how the traffic cost model can be used together with a scheme for computing multiple paths in the network like path splicing to allow operators to expose only low cost paths to end systems. Such a cost-based path selection architecture can help resolve the tension between network operators goal of reducing the traffic costs in a network and end systems requirement of being able to influence path selection decisions in the network. With the advent of new routing architectures that aim to centralize the control-plane in the network can aid the deployment of a cost-based path selection framework in networks.

CHAPTER VI

CONCLUDING REMARKS

The Internet was an academic experiment and for a long time remained the domain of researchers. As the Internet has become more mainstream there has been an explosion of rich applications deployed at the edge, aided primarily by the end-to-end focus of the Internet architecture. The adoption of real-time applications like VoIP, gaming, video streaming have made the case for allowing applications some control over selecting their paths stronger. This has led to a number of research proposals of providing multiple paths in the network and allowing applications (or end systems) some control over path selection.

6.1 Towards a Path Selection Architecture

Traditional shortest-path routing protocols compute paths in the network based on a single metric (*e.g.*link weights). Also, the notion of a path failure is a physical failure: link or router failure. However, applications perceive failures in different manner. Even moderate amounts of packet drops on a path could make it unusable for an application relying on TCP and a high jitter path would render real-time communication applications ineffective. Thus, as the diversity of applications at the edge has exploded, the application-specific definition of path failures has become important. Hence, networks must compute multiple paths and allow some control for applications to influence the path their traffic can take in the network.

Multipath routing proposals vary in how much control do they provide end systems to influence the path selection. They range from giving end-systems complete control over path selection [69], to no control [54, 96, 52] and a wide spectrum of distributing control between the end systems and the network [99, 60]. Each of this multipath schemes design their unique interface for incorporating application choice in path selection. For example,

path splicing allows applications to include a list of forwarding trees at each intermediate hop in a separate header in the packet (splicing bits), routing deflections uses the IP-ID and TTL field of the IP header (that can be modified by the end system) to determine if the packet is to be "deflected" to a non-default path and ECMP uses a hash of the source and destination identifiers to select amoung two equal-cost paths in the network.

In this dissertation, we hypothesize that the lack of a standard interface separating the end systems method of influencing path selection and the actual multipath routing scheme implementation in the network is hampering real deployment of multipath protocols and path selection frameworks on the Internet. Hence, we propose a "narrow-waist" interface for path selection on the Internet. The interface is simply a bit-string called *path bits*, that an end-system includes in packets. Network (or routers) use the path bits to select an end-to-end path. The bits have no semantics for the end-system, except that when they are modified, the end-system expects, with a high probability to get a different path to the destination. We believe that the path-bits interface, which acts as a "narrow-waist" between the applications running on end-systems and the multipath routing schemes deployed in the network, will foster innovation. Both "above the waist" in the path selection and monitoring schemes at the end-systems and "below the waist" in the multipath routing schemes to compute alternate paths in the network. This is in similar spirit to the IP/TCP "narrow-waist" of the Internet [5].

6.2 Summary of Contributions

The central contribution of this thesis is the design, implementation and evaluation of **path bits**, a "narrow-waist" path selection architecture for the Internet. The thesis also addresses the tussle between network operators and end-systems, and develops a framework for cost-based path selection. The specific contributions of this dissertation are as follows:

- **A Narrow-waist Interface for Path Selection** We present the motivation and design of *path bits*, which is a "narrow-waist" interface for enabling path selection in

the Internet, in Chapter 2. We also showed how to map a large number of existing multipath schemes to the path-bits interface.

- **Creating Network Paths using Path Splicing** We present, path splicing, a multipath routing primitive that creates alternate paths in the network by peturbing the default shortest path. We show how to construct paths by splicing together segments of the different perturbed paths. We present both intradomain and interdomain versions of path splicing.

- **Addressing the tussle between End-systems and Network Operators with Cost-based Paths Selection** We present a holistic traffic cost model that can help attribute costs to traffic flows in a network. We show how the cost model can be used by network operators to present a choice of low cost paths to the end-system, thus, mitigating the tussle between end-systems goal of being able to make path selection decisions and network operator's objective of reducing traffic costs. We also show other applications of the cost model in modifying routing to reduce traffic carrying costs and rationalize planning decisions like selection of new peers, peering locations and evaluating existing peering relationships.

- **Prototype Implementations publicly available** We present prototype implementations of three multipath routing schemes, on hardware and software platforms, to demonstrate the path-bits interface permits simple and efficient implemetations of multipath forwarding schemes. We also implement a number of path selection and monitoring mechanisms that exploit the path-bits interface to gain access to alternate paths in the network. We make these implementations available for the research community.

6.3 Future Directions

This dissertation has explored building an architecture for network path selection on the Internet. This is definitely not the final word for this highly dense research topic. We present three future directions in which this work can be extended in the future.

6.3.1 Narrow Waist in Datacenter Networks

A datacenter networking environment is very different from a traditional network and presents its own set of challenges. In a datacenter, the end-systems (or servers) and the network devices (switches, routers and firewalls) are under the same administrative control. Hence, there is an opportunity for co-operation between the end-systems and the network to achieve common goals. This has in fact led to a number of interesting research proposals where the co-operation between the edge and the network helps solve a number of problems [93, 6] in the datacenter environment. It would be interesting to explore how a path selection framework fits in such an environment. Because a "narrow-waist" interface like path-bits decouples the underlying implementation of multiple paths in the network from the particular mechanism used by the end-systems to discover, monitor and select alternate paths; it can be of utility in a datacenter network.

A datacenter environment is also rapidly evolving with newer protocols finding quicker adoption. This implies that a decoupling would be of even higher utility as it can lead to rapid deployment of new network protocols. End-system protocol stacks can also be independently modified to add new and better features if the path selection interface is fixed. Because all the end-systems in a datacenter are under a common administrative control, protocol stack updates can be pushed by the datacenter operator. In a shared cloud environment, where each physical end host is shared among multiple cloud tenants, each running inside their own virtual machines [68]. The cloud provider controls the hypervisor that implements policies to ensure fair sharing of network resources among the cloud tenants.

These environments already use multipath routing schemes in the network instead of relying on layer-2 shortest path algorithms to utilize the network resources. Path-bits interface can help in evolving both the hypervisor-controlled end-system path selection implementations and the network-based multipath schemes independently.

6.3.2 Alternate slice generation schemes in path splicing

Path splicing uses random perturbations of the link weights in the network graph to generate alternate paths in the network. Perturbing the paths randomly has several nice properties like low path stretch and high path novelty. Unfortunately, the slices created with such method cannot guarantee to cover the complete underlying network graph. In other words, there is no hard guarantee that if the underlying graph does not get disconnected then there would exist a spliced path. It would be interesting to explore more deterministic slice generation techniques that can guarantee failure recovery. Discovering such schemes could also help compare paths computed using path splicing versus such deterministic techniques.

Commercial networks offer different quality of service to traffic based on the amount charged to the customer. This is typically handled by using MPLS and creating label switched paths (LSPs) in the network with different priorities so that the highest priority traffic gets better quality of service as compared to traffic with lower priority. A slice generation scheme that is aware of such preconfigured priorities in the network could create a different slice based on the type of traffic and a particular traffic can be routed on the corresponding slice or set of slices.

Alternately, slices could be created such that each slice is optimized for a particular network metric. For example, the network could create a slice that has paths which are low latency, another slice which has paths that have high bandwidth. The network can then route traffic corresponding to the metric which is more suitable for that traffic on the appropriate slice. End-systems could use path bits to *signal* to the network the particular slice their traffic needs to be routed on.

6.3.3 Comparison of multipath routing algorithms

There are a number of multipath, fast rerouting schemes proposed in the literature. Each multipath scheme evaluates a specific failure scenario. However, there is no well defined methodology when evaluating a multipath routing scheme. There is no well-defined metric in the research community that can be used to evaluate the multipath routing schemes. For example, path splicing uses a metric called *reliability curve* to show the effectiveness of path splicing from recovering from failures in a network. While, failure carrying packets [54] uses a failure model in which links fail and recover continously in the simulated network, and use that to evaluate how well the routing scheme can recover from transient network failures. It is not clear which is a better metric when comparing two different multipath schemes. Defining such metric would be a valuable addition to the research community in being able to compare different multipath schemes.

A "narrow-waist" can also help in evaluating different multipath schemes. The evaluation criteria could be how fast can applications find *good* paths using the particular multipath scheme. With the release of our prototype implementations of the end-system support for path bits, we hope researchers will use it to evaluate new and existing multipath routing proposals.

REFERENCES

[1] "IETF transparent interconnection of lots of links (TRILL) working group." `http://datatracker.ietf.org/wg/trill/charter/`.

[2] "NetFPGA." `http://www.netfpga.org`.

[3] "Path Splicing." `http://www.gtnoise.net/splicing/`, Sept. 2009.

[4] AKELLA, A., MAGGS, B., SESHAN, S., SHAIKH, A., and SITARAMAN, R., "A measurement-based analysis of multihoming," in *Proc. ACM SIGCOMM*, (Karlsruhe, Germany), Aug. 2003.

[5] AKSHABI, S. and DOVROLIS, C., "The Evolution of Layered Protocol Stacks leads to an Hourglass-shaped architecture," in *Proc. ACM SIGCOMM*, (Toronto, Canada), Aug. 2011.

[6] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., and SRIDHARAN, M., "Data Center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, (New Delhi, India), Aug. 2010.

[7] ALLMAN, M., EDDY, W., and OSTERMANN, S., "Estimating loss rates with TCP," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 3, pp. 12–24, 2003.

[8] ALLMAN, M. and PAXSON, V., "On Estimating End-to-End Network Path Properties," *ACM Computer Communications Review*, vol. 31, no. 2 supplement, 2001.

[9] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., and MORRIS, R., "Resilient Overlay Networks," in *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, (Banff, Canada), pp. 131–145, Oct. 2001.

[10] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., and RAO, R., "Improving Web availability for clients with MONET," in *Proc. 2nd USENIX NSDI*, (Boston, MA), May 2005.

[11] ANDERSEN, D. G., SNOEREN, A. C., and BALAKRISHNAN, H., "Best-path vs. multi-path overlay routing," in *Proc. ACM SIGCOMM Internet Measurement Conference*, (Miami, FL), Oct. 2003.

[12] ANWER, B., MOTIWALA, M., BIN TARIQ, M., and FEAMSTER, N., "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware," in *Proc. ACM SIGCOMM*, (New Delhi, India), Aug. 2010.

[13] APOSTOLOPOULOS, G., "Using multiple topologies for ip-only protection against network failures: A routing performance perspective," Tech. Rep. 377, ICS-FORTH, Apr. 2006.

[14] ATLAS, A. and ZININ, A., "Basic Specification for IP Fast-Reroute: Loop-free Alternates." http://tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-spec-base-10, Nov. 2007.

[15] BROIDO, A. and KC CLAFFY, "Topological Resilience in IP and AS Graphs." http://www.caida.org/analysis/topology/resilience/, 2006.

[16] BUSH, R., MAENNEL, O., ROUGHAN, M., and UHLIG, S., "Internet optometry: Assessing the broken glasses in internet reachability," in *Proc. Internet Measurement Conference*, (Chicago, Illinois), Oct. 2009.

[17] CAESAR, M., FEAMSTER, N., REXFORD, J., SHAIKH, A., and VAN DER MERWE, J., "Design and implementation of a routing control platform," in *Proc. 2nd USENIX NSDI*, (Boston, MA), May 2005.

[18] CASADO, M., FREEDMAN, M. J., PETTIT, J., LUO, J., MCKEOWN, N., and SHENKER, S., "Ethane : Taking control of the enterprise," in *Proc. ACM SIGCOMM*, (Kyoto, Japan), Aug. 2007.

[19] CHA, M., MOON, S., PARK, C.-D., , and SHAIKH, A., "Placing Relay Nodes for Intra-Domain Path Diversity," in *Proc. IEEE INFOCOM*, (Barcelona, Spain), Mar. 2006.

[20] CHANG, H., JAMIN, S., and WILLINGER, W., "To peer or not to peer: Modeling the evolution of the Internet's AS-level topology," in *Proc. IEEE INFOCOM*, (Barcelona, Spain), Mar. 2006.

[21] "MPLS Traffic Engineering Fast Reroute – Link Protection." http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/120newft/120limit/120st/120st16/frr.htm.

[22] "Cisco Multi-Topology Routing." http://www.cisco.com/en/US/products/ps6922/products_feature_guide09186a00807c64b8.html.

[23] "Cisco Optimized Edge Routing (OER)." http://www.cisco.com/en/US/products/ps6628/products_ios_protocol_option_home.html, 2010.

[24] CLARK, D., WROCLAWSKI, J., SOLLINS, K., and BRADEN, B., "Tussle in cyberspace: Defining tomorrow's Internet," in *Proc. ACM SIGCOMM*, (Pittsburgh, PA), pp. 347–256, Aug. 2002.

[25] DIMITROPOULOS, X. A., KRIOUKOV, D. V., VAHDAT, A., and RILEY, G. F., "Graph Annotations in Modeling Complex Network Topologies," *CoRR*, vol. abs/0708.3879, 2007.

[26] "Why care about Transit Pricing?." http://drpeering.net/a/Peering_vs_Transit___The_Business_Case_for_Peering.html, 2009.

[27] "Internet Video Traffic." http://tinyurl.com/2777mhl.

[28] ERMOLINSKIY, A. and SHENKER, S., "Reducing Transient Disconnectivity Using Anomaly-Cognizant Forwarding," in *Proc. 7th ACM Workshop on Hot Topics in Networks (Hotnets-VII)*, (Calgary, Alberta. Canada.), Oct. 2008.

[29] FEAMSTER, N., BALAKRISHNAN, H., REXFORD, J., SHAIKH, A., and VAN DER MERWE, K., "The case for separating routing from routers," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, (Portland, OR), Sept. 2004.

[30] FEAMSTER, N. and REXFORD, J., "Network-Wide Prediction of BGP Routes," *IEEE/ACM Transactions on Networking*, pp. 253–266, Apr. 2007.

[31] FELDMANN, A., MAENNEL, O., MAO, Z. M., BERGER, A., and MAGGS, B., "Locating Internet routing instabilities," in *Proc. ACM SIGCOMM*, (Portland, OR), pp. 205–218, Aug. 2004.

[32] FORTZ, B. and THORUP, M., "Internet traffic engineering by optimizing OSPF weights," in *Proc. IEEE INFOCOM*, (Tel-Aviv, Israel), Mar. 2000.

[33] FORTZ, B. and THORUP, M., "Optimizing OSPF/IS-IS weights in a changing world," *IEEE Journal on Selected Areas in Communications (J-SAC)*, vol. 20, pp. 756–767, May 2002.

[34] GAO, L. and REXFORD, J., "Stable Internet routing without global coordination," *IEEE/ACM Transactions on Networking*, pp. 681–692, Dec. 2001.

[35] GJESSING, S., "Implementation of two Resilience Mechanisms using Multi Topology Routing and Stub Routers," in *International Conference on Internet and Web Applications and Services/Advanced*, Feb. 2006.

[36] GODFREY, B., GANICHEV, I., SHENKER, S., and STOICA, I., "Pathlet routing," in *Proc. ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[37] GOLDENBERG, D. K., QIU, L., XIE, H., YANG, Y. R., and ZHANG, Y., "Optimizing cost and performance for multihoming," in *Proc. ACM SIGCOMM*, (Portland, OR), pp. 79–92, Aug. 2004.

[38] GREENBERG, A., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., and SENGUPTA, S., "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[39] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., and ZHANG, H., "A clean slate 4D approach to network control and management," *ACM Computer Communications Review*, vol. 35, no. 5, pp. 41–54, 2005.

[40] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., and SHENKER, S., "NOX: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 105–110, July 2008.

[41] GUMMADI, K. P., MADHYASTHA, H. V., GRIBBLE, S. D., LEVY, H. M., and WETHERALL, D., "Improving the reliability of Internet paths with one-hop source routing," in *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, (Banff, Canada), Oct. 2001.

[42] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., and LU, S., "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[43] "Wholesale Internet Bandwidth Prices." `http://www.circleid.com/posts/wholesale_internet_bandwidth_prices/`, 2008.

[44] HOPPS, C., *Analysis of an Equal-cost Multi-Path algorithm*. IETF, Nov. 2000. RFC 2992.

[45] HSIEH, H.-Y. and SIVAKUMAR, R., "pTCP: An end-to-end transport layer protocol for striped connections," in *IEEE International Conference on Network Protocols (ICNP)*, (Paris, France), Nov. 2002.

[46] "Internap." `http://www.internap.com/`, 2009.

[47] "Global internet phenomena report." `http://www.sandvine.com/downloads/documents/2010GlobalInternetPhenomenaReport.pdf`.

[48] JIANG, H. and DOVROLIS, C., "Passive Estimation of TCP round-trip times," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, pp. 75–88, 2002.

[49] KAUR, H. T., KALYANARAMAN, S., WEISS, A., KANWAR, S., and GANDHI, A., "BANANAS: an evolutionary framework for explicit and multipath routing in the internet," in *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, 2003.

[50] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., and KAASHOEK, M. F., "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263–297, Aug. 2000.

[51] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., and OTHERS, "Onix: A distributed control platform for large-scale production networks," in *Proc. 9th USENIX OSDI*, (Vancouver, Canada), Oct. 2010.

[52] KUSHMAN, N., KANDULA, S., KATABI, D., and MAGGS, B. M., "R-BGP: Staying connected in a connected world," in *Proc. 4th USENIX NSDI*, (Cambridge, MA), Apr. 2007.

[53] KVALBEIN, A., HANSEN, A. F., CICIC, T., GJESSING, S., and LYSNE, O., "Fast IP Network Recovery using Multiple Routing Configurations," in *Proc. IEEE INFOCOM*, (Barcelona, Spain), pp. 23–26, Mar. 2006.

[54] LAKSHMINARAYANAN, K., CAESAR, M., RANGAN, M., ANDERSON, T., SHENKER, S., and STOICA, I., "Achieving Convergence-Free Routing with Failure-Carrying packets," in *Proc. ACM SIGCOMM*, (Kyoto, Japan), Aug. 2007.

[55] LESKOVEC, J., KLEINBERG, J., and FALOUTSOS, C., "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (Chicago, IL), Aug. 2005.

[56] "Juniper Networks: Intelligent Logical Router Service." `http://www.juniper.net/solutions/literature/white_papers/200097.pdf`.

[57] MOTIWALA, M., ANWER, B., FEAMSTER, N., and ANDERSEN, D., "A Narrow Waist for Multipath Routing," in *Technical Report*, Oct. 2011.

[58] MOTIWALA, M., BAVIER, A., and FEAMSTER, N., "Inband Network Path Diagnosis," Tech. Rep. GT-CS-07-07, Georgia Institute of Technology, 2007.

[59] MOTIWALA, M., DHAMDHERE, A., FEAMSTER, N., and LAKHINA, A., "Towards a cost model for network traffic," in *To Appear in ACM SIGCOMM Computer Communications Review*, Jan. 2012.

[60] MOTIWALA, M., ELMORE, M., FEAMSTER, N., and VEMPALA, S., "Path Splicing," in *Proc. ACM SIGCOMM*, (Seattle, WA), Aug. 2008.

[61] MOTIWALA, M., FEAMSTER, N., and VEMPALA, S., "Improving Interdomain Routing Security with BGP Path Splicing," in *Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, (Princeton, NJ), May 2007.

[62] MOTIWALA, M., FEAMSTER, N., and VEMPALA, S., "Path Splicing: Reliable Connectivity with Rapid Recovery," in *Proc. 6th ACM Workshop on Hot Topics in Networks (Hotnets-VI)*, (Atlanta, GA), Nov. 2007.

[63] MOY, J., *OSPF Version 2*, Mar. 1994. RFC 1583.

[64] "OpenFlow Switch Consortium." `http://www.openflowswitch.org/`, 2008.

[65] ORAN, D., *OSI IS-IS intra-domain routing protocol*. IETF, Feb. 1990. RFC 1142.

[66] PAXSON, V., "End-to-End Routing Behavior in the Internet," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 601–615, 1997.

[67] PERLMAN, R., *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Oct. 1988. MIT-LCS-TR-429. `http://www.lcs.mit.edu/publications/specpub.php?id=997`.

[68] POPA, L. AND KRISHNAMURTHY, A. AND RATNASAMY, S. AND STOICA, I., "FairCloud: Sharing The Network In Cloud Computing," in *Proc. ACM HotNets*, 2011.

[69] POSTEL, J., *Internet Control Message Protocol*. IETF, Sept. 1981. RFC 792.

[70] PSENAK, P., MIRTORABI, S., ROY, A., NGUYEN, L., and PILLAY-ESNAULT, P., *Multi-Topology Routing in OSPF*. IETF, June 2007. RFC 4915.

[71] QIU, L., YANG, Y. R., ZHANG, Y., and SHENKER, S., "On selfish routing in Internet-like environments," in *Proc. ACM SIGCOMM*, (Karlsruhe, Germany), Aug. 2003.

[72] QUOITIN, B. and UHLIG, S., "Modeling the routing of an autonomous system with C-BGP," *Network, IEEE*, vol. 19, no. 6, pp. 12–19, 2005.

[73] QURESHI, A., WEBER, R., BALAKRISHNAN, H., GUTTAG, J., and MAGGS, B., "Cutting the electric bill for internet-scale systems," in *Proc. ACM SIGCOMM*, (Barcelona, Spain), Aug. 2009.

[74] REC, I., "G. 107-The E Model, a computational model for use in transmission planning," *International Telecommunication Union*, 2003.

[75] REKHTER, Y., LI, T., and HARES, S., *A Border Gateway Protocol 4 (BGP-4)*. IETF, Jan. 2006. RFC 4271.

[76] "RouteScience." Whitepaper available from `http://www.routescience.com/technology/tec_whitepaper.html`.

[77] "Routing Area Working Group (rtgwg)." `http://www.ietf.org/html.charters/rtgwg-charter.html`.

[78] SESHAN, S., STEMM, M., and KATZ, R. H., "SPAND: Shared Passive Network Performance Discovery," in *Proc. 1st USENIX Symposium on Internet Technologies and Systems (USITS)*, (Monterey, CA), Dec. 1997.

[79] SHAND, M. and BRYANT, S., "IP Fast Re-route framework." `http://www3.tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-framework-07`, June 2007.

[80] SHAND, M. and BRYANT, S., "IP Fast Reroute Using Not-via Addresses." `http://www3.tools.ietf.org/html/draft-ietf-rtgwg-ipfrr-notvia-addresses-01`, July 2007.

[81] SHAPLEY, L., "A Value for n-Person Games," *Classics in Game Theory*, p. 69, 1997.

[82] SINHA, S., KANDULA, S., and KATABI, D., "Harnessing TCP's burstiness with flowlet switching," in *Proc. 3nd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, (San Diego, CA), Nov. 2004.

[83] SPRING, N., MAHAJAN, R., and WETHERALL, D., "Measuring ISP topologies with Rocketfuel," in *Proc. ACM SIGCOMM*, (Pittsburgh, PA), Aug. 2002.

[84] STANOJEVIC, R., LAOTARIS, N., and RODRIGUEZ, P., "On economic heavy hitters: Shapley value analysis of the 95th-percentile pricing," in *Proc. ACM SIGCOMM Internet Measurement Conference*, (Melbourne, Australia), Nov. 2010.

[85] TAO, S., XU, K., ESTEPA, A., GAO, T., GUERIN, R., KUROSE, J., TOWSLEY, D., and ZHANG, Z., "Improving VoIP Quality Through Path Switching," in *Proc. IEEE INFOCOM*, (Miami, FL), Mar. 2005.

[86] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., and PATIL, S., "An architecture for Internet data transfer," in *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, (San Jose, CA), May 2006.

[87] TURNER, J., CROWLEY, P., DEHART, J., FREESTONE, A., HELLER, B., KUHNS, F., KUMAR, S., LOCKWOOD, J., LU, J., WILSON, M., and OTHERS, "Supercharging planetlab: a high performance, multi-application, overlay network platform," in *Proc. ACM SIGCOMM*, (Kyoto, Japan), Aug. 2007.

[88] VALANCIUS, V., FEAMSTER, N., REXFORD, J., and NAKAO, A., "Wide-Area Route Control for Distributed Services," in *Proc. USENIX Annual Technical Conference*, (Boston, MA), June 2010.

[89] "Using network intelligence to provide carrier-grade voip." `http://www.sandvine.com/general/getfile.asp?FILEID=31`.

[90] WALFISH, M., STRIBLING, J., KROHN, M., BALAKRISHNAN, H., MORRIS, R., and SHENKER, S., "Middleboxes no longer considered harmful," in *Proc. 6th USENIX OSDI*, (San Francisco, CA), Dec. 2004.

[91] WANG, H., XIE, H., QIU, L., SILBERSCHATZ, A., and YANG, Y., "Optimal isp subscription for internet multihoming: Algorithm design and implication analysis," in *Proc. IEEE INFOCOM*, (Miami, FL), Mar. 2005.

[92] WHITE, R. and AKYOL, B., "Considerations in Validating the Path in BGP." IETF Draft, 2007.

[93] WILSON, C., BALLANI, H., KARAGIANNIS, T., and ROWSTRON, A., "Better Never than Late: Meeting Deadlines in Datacenters," in *Proc. ACM SIGCOMM*, (Toronto, Canada), Aug. 2011.

[94] WISCHIK, D., RAICIU, C., GREENHALGH, A., and HANDLEY, M., "Design, implementation and evaluation of congestion control for multipath tcp," in *Proc. 8th USENIX NSDI*, (Boston, MA), Apr. 2011.

[95] WISEMAN, C. and OTHERS, "A Remotely Accessible Network Processor-Based Router for Network Experimentation," in *ANCS*, 2008.

[96] XU, W. and REXFORD, J., "MIRO: Multi-path Interdomain ROuting," in *Proc. ACM SIGCOMM*, (Pisa, Italy), Aug. 2006.

[97] YAN, H., MALTZ, D. A., NG, T. S. E., GOGINENI, H., ZHANG, H., and CAI, Z., "Tesseract: A 4D Network Control Plane," in *Proc. 4th USENIX NSDI*, (Cambridge, MA), Apr. 2007.

[98] YANG, X., "NIRA: A New Internet Routing Architecture," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture*, (Karlsruhe, Germany), Aug. 2003.

[99] YANG, X., WETHERALL, D., and ANDERSON, T., "Source selectable path diversity via routing deflections," in *Proc. ACM SIGCOMM*, (Pisa, Italy), Aug. 2006.

[100] ZHANG, M., LAI, J., KRISHNAMURTHY, A., PETERSON, L., and WANG, R., "A transport layer approach for improving end-to-end performance and robustness using redundant paths," in *Proc. USENIX Annual Technical Conference*, (Boston, MA), pp. 99–112, June 2004.

[101] ZHANG, Z., ZHANG, M., GREENBERG, A., HU, Y. C., MAHAJAN, R., and CHRISTIAN, B., "Optimizing Cost and Performance in Online Service Provider Networks," in *Proc. 7th USENIX NSDI*, (San Jose, CA), Apr. 2010.