Readdressing Network Layers

James McCauley

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-161
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-161.html

August 14, 2020

Copyright © 2020, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Readdressing Network Layers

By

James Ash McCauley

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Professor Sylvia Ratnasamy
Professor Coye Cheshire

Summer 2020

Readdressing Network Layers

Copyright 2020
by
James Ash McCauley

Abstract

Readdressing Network Layers

by

James Ash McCauley

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

It has long been common practice to split the functionality of computer networks into distinct *layers*. In the context of the most common networks today, there are five such layers, bracketed by hardware at the lowest layer and application software at the highest. While this modularization has generally been a huge success story for the field, networks have now been subject to decades of changes in terms of both technologies and use cases. Moreover, we have accumulated significant hindsight.

This dissertation examines the three infrastructural layers that fall between hardware and applications, and in each case suggests an improvement. A new approach to resiliency is explored at the link and network layers. In the former case, the result eliminates the need for the typical spanning tree based approach. In the latter, it can radically improve performance during times of routing protocol convergence, eliminating the need for things like precomputed backup paths. At the transport layer, an approach to server load balancing is discussed, which avoids the fate-sharing problem of stateful load balancers as well as the consistency problems of stateless ones. Finally, this dissertation argues that there is a layer "missing" from our common layered model, the inclusion of which is a vital enabler for the continued evolution of the Internet.

Contents

List of Figures

Acknowledgments

As discussed in this dissertation, network layers work by building atop other layers. So it is with graduate students too: my work and my success is built atop the work, success, and support of others. I will fall short in fully expressing how true this is for me personally, but I will make an attempt.

First, I need to thank my advisor, Scott Shenker. I have worked longer and more closely with Scott than anyone else in my life, and the positive impact he has had on my life in that time really can't be overstated. He has consistently found ways for me to leverage and build on my strengths. He has consistently assisted me in my own research, and listened carefully when I thought I had something to say. He has consistently given me insight into what makes good research and what makes a good researcher. He has put a lot of effort into teaching me, helping me grow, and helping me succeed… all of the above applied to research, teaching, and life in general. In short, having him as a mentor has been an incredible benefit and an incredible privilege. That all makes leaving UC Berkeley and ICSI seem like a pretty questionable decision, though I am hoping to retain his good counsel and friendship indefinitely through some combination of chocolate, LaTeX assistance, and my high esteem for him.

While not technically my advisor, Sylvia Ratnasamy has also never failed to enlighten, inspire, and educate me. Many times when I am out in the weeds, I try to channel her gift for zeroing in on the essential. I've witnessed her do it so well so many times that I do believe some of it has indeed rubbed off on me, for which I will be forever thankful. For many years now, I have relied on and enjoyed Sylvia's additional perspective, and am incredibly thankful to have had the opportunity to teach with her as well. She had not yet rejoined Berkeley when I first started hanging around Soda Hall; her arrival was an unexpected and immeasurable bonus.

I have benefited greatly from all the Berkeley faculty that I've taken classes from or otherwise interacted with as well. I am especially grateful to Coye Cheshire, John DeNero, and Ion Stoica, who have been particularly helpful to me individually. I'll pay it forward.

I am also thankful to all of the Berkeley staff. In particular, those that have served as PhD program advisors have assisted me greatly: Audrey Sillers, Angela Waxman, and Jean Nguyen. The people that fill this role are universally good, and for that, I believe I must also thank Shirley Salanio. Also on the staff side, the NetSys admins have always done their best for me, and Jon Kuroda has always tried to keep my LEDs blinking and my passwords working.

While at Berkeley, I have had the opportunity to work closely with a number of people who were undergraduates at the time. I am truly grateful for all of them, but in particular, I should thank the ones that directly contributed to material in this dissertation: Alice Sheng, Brian Kim, David Cheng, Ian Rodney, Jack Zhao, Michael Dong, and Neil Agarwal.

I also owe a great debt to various others that I've been fortunate enough to work with closely, co-author material with (both included in this dissertation and not), or both. Among others, these include Amin Tootoonchian, Arvind Krishnamurthy, Ethan J. Jackson, Justine Sherry, Mohammad Alizadeh, Radhika Mittal, Teemu Koponen, Yotam Harchol, and probably my three most frequent collaborators: Aisha Mushtaq, Aurojit Panda, and Barath Raghavan. Working with each of them has been an honor. Some of them have been (and continue to be) great advisors to me. Some of them I am fortunate enough to count as friends.

As a member of the NetSys Lab, I have benefited greatly from the extended NetSys family. Their ideas, enthusiasm, support, friendship, and shared snacks have been a consistent positive in my life for many years now. Many of them appear elsewhere in these acknowledgements, but I will also call out my other fellow PhD students: Chang Lan, Colin Scott, Ed Oakes, Emmanuel Amaro, Kay Ousterhout, Lloyd Brown, Michael Chang, Peter Gao, Qifan Pu, Sangjin Han, Silvery Fu, Vivian Fang, Wenting Zheng, Zhang Wen, and Zhihong Luo. I have worked with some fairly closely; I would be lucky to work with any in the future. And the same goes for everyone else I was lucky enough to cross paths with in NetSys: undergraduate and graduate students, visitors, and industry partners. One couldn't ask to work with a better group of people. A few of these which don't show up elsewhere in these acknowledgements but that took time out of their lives for me for one reason or another include Amy Ousterhout, Anwar Hithnawi, Chris Branner-Augmon, and Kyriakos Zarifis.

My PhD brings to a close a really significant arc of my life, and two people had a really significant effect on that trajectory. Without them, I would not have ended up doing networking at Berkeley – I would have ended up somewhere else doing something else, and I think the chances are it wouldn't have been as good. Working backwards, one of these was Martin Casado, who introduced me to networking and what it meant to be a graduate student, and who was never sparing with his good advice. The other is Abe Pralle, who is a cornerstone of my entire adult academic career, and whose continued friendship (and collaboration and insight) along the way has been no small thing either.

Without the International Computer Science Institute and everyone there, my graduate career would no doubt have been more difficult. It may not even have been possible. UC Berkeley and the UC system are also due significant credit. These are all institutions I have been proud and thankful to be affiliated with.

Chapter 4 owes thanks to a great many people for direct contributions to its preceding work or just for sharing their valuable thoughts. Along with a number already mentioned above, this includes Ali Ghodsi, Ankit Singla, Dirk Hasselbalch, Hari Balakrishnan, Igor Ganichev, James Wilcox, Jennifer Rexford, Nick Feamster, Nick McKeown, P. Brighten Godfrey, Vjekoslav Brajkovic, and others. Portions of material in this dissertation also owe thanks to conference/journal reviewers and to shepherds Brad Karp and Marco Canini, all of whom helped make it better.

This material is based upon work supported by the National Science Foundation under Grant No. 1139158, 1216073, 1420064, and 1817115.[1] Thanks are also due to AT&T, Ericsson, Huawei, Intel, and VMware, who all provided additional support. Thank you.

There are countless others that I haven't named individually, including a lot of people at Berkeley from my earlier years here and (unfortunately) some that I am probably forgetting at the moment. I haven't *actually* forgotten you. Thank you. I mean it. Next time you see me, the drinks and/or Skittles are on me. Take me up on this. Please.

Finally, there is my inner circle (especially my family, A.A., and A.T., who have been along for the whole ride). Their support has been invaluable, and their companionship has made this all so much more enjoyable. I'm so thankful they put up with me.

[1] Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Chapter 1

Introduction

Perhaps it is a measure of its wild success that networking research seems to be entirely without glamour today. Indeed, its glamour days seem to have been rather short. The Internet (and computer networks generally) went from beneath the notice of most people, to a curiosity, to an entrenched fact of everyday life in a remarkably small period of time. Somewhere between the curiosity and the entrenchment, perhaps there was glamour. But in the modern era, networks appear rather mundane to most people. On multiple occasions over the course of the author's time at Berkeley, a revelation that said author was a networking researcher received a response of bafflement, as if the confused party had been told that researching the slicing of bread were still an active concern. Networking research? Aren't we done with that?

Of course, what they overlook is that the demands on networks are ever-increasing and that keeping up with those demands requires effort. As has been noted, at any particular time, the Internet only just barely works [46], and this is compounded by the fact that expectations grow not only more and more extreme, but less and less charitable. But if we networking researchers and engineers are doing our jobs right, it *does* keep working, and it *does* keep meeting increasing expectations. Alas, success is marked by being taken for granted, not by glamorous rockstar status. Said another way: it is our job to ensure that the *network* is never the bottleneck for any endeavor. This dissertation is the result of several years of work pursuing that most noble goal of having our field remain largely unnoticed.

It has long been common practice to split the functionality of computer networks into distinct *layers*, and the composition of these layers forms a greater whole. In keeping with this practice, the four main contributions of this dissertation are split similarly; indeed, the contributions are conceptually divided along the exact lines defined by network layers (three of them along widely-recognized layers, and the fourth along the lines of a network layer that it will be argued is "missing" from common layered models). The intention is that these contributions can also compose to form a greater whole: a more effective, more reliable, more dynamic Internet. In the remainder of this chapter, we first provide some background on network layers, and then synopsize the four contributions and place them into context.

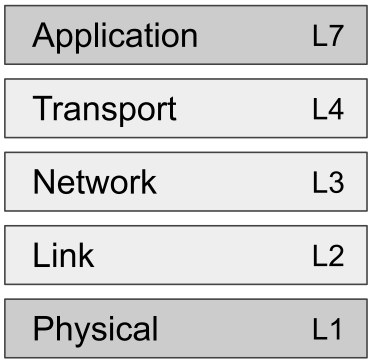


Figure 1.1: *A common conceptual representation of the layers composing the Internet including the typical layer numbers.*

1.1 Background on Networking Layers

For decades, networking functionality has been almost universally considered to be broken down into a number of distinct *layers*. Exactly which functionality is to be divided and how to divide it is less universally agreed upon, and prevailing wisdom is not stable across different networks or across time. Indeed, even the purpose of such decomposition (e.g., whether it is purely descriptive, prescriptive, or proscriptive) is variable. All that said, Figure 1.1 represents an outlook on the layers of the Internet which is likely familiar to many readers – a similar diagram appears in many writings about the Internet. Imperfect as it is, it may represent the most widespread understanding of the layering of the Internet. The following paragraphs elaborate on each of the layers in a way consistent with this understanding.

In explaining this view of the network, it helps to begin by looking at the extremes. The highest layer (the application layer) tells us what we are aiming for. The lowest layer (the physical layer) tells us what our starting point is. The layers in between bridge the gap between the latter and the former. We proceed with this in mind.

The purpose of having a network, of course, is to apply it to some particular problem which benefits from (or is defined by) communication between nodes. These problems likely have their own associated software packages, and likely have different needs from the network. While applications are myriad today, three "principal services" stood out in the early Internet: remote terminal, file transfer, and email [99]. The requirements for these applications differ in many dimensions. Remote terminal requires a high degree of interactivity and a relatively small amount of data, and it must concern itself with issues particular to terminals (*e.g.,* the ability to interrupt a remote process). File transfer needs less interactivity, but must deal with larger amounts of data, and must allow the exchange of various sorts of metadata irrelevant to remote terminals (*e.g.,* filenames and directory listings). Email must cope with communications between systems which are not all available simultaneously. Thus, each of these applications benefit from unique *application layer* protocols, each capable of meeting the individual needs of a particular application, and embedding specific decisions about how to accomplish its task (*e.g.,* How should the recipient of an email be addressed? Should a file be transmitted starting with the last byte or the first byte?). These are very high-level concerns.

At the opposite end of the layering diagram lies the *physical layer*. This layer differs from the application layer, and indeed from all the others, in that it is the one which *actually connects* nodes in the network together, allowing a logical symbol (*e.g.,* a bit) at one node to be transformed into a physical signal that can be sent to one or more other nodes which can then transform the signal back into a logical symbol. Exactly how this symbol/signal transformation is done is the domain of the physical layer, and again in contrast to the application layer, this is a very low-level concern. Different methods for this symbol/signal transformation result in a variety of different instantiations of the physical layer (*e.g.,* wired, wireless, optical) with different properties (*e.g.,* symbol rate, signal-to-noise, point-to-point vs. point-to-multipoint). In some sense, all the other layers are optional and their functions and usage arguable, but the existence and usage of the physical layer is non-negotiable; a networked system must participate in some physical layer to actually move information from one place to another.

While the physical layer defines how symbols (*e.g.,* bits) are actually converted to and from some transmissible signal, more functionality is needed to build a useful network. The *link layer* provides some of this functionality, typically providing a number of services. For example, while the physical layer is concerned with individual symbols, the link layer abstracts groups of symbols into larger units: packets (or as they are sometimes called at the link level, frames). Additionally, the physical layer may have particular properties which are dealt with at the link layer. For example, signals from different nodes in some physical media may interfere with each other, and this interference may be dealt with by the link layer (*e.g.,* by coordination or detection and retransmission); thus, it is typical to find specific link layer implementations closely tied to specific physical link layer implementations (Ethernet, for example, defines both physical and link layers). A final example of link layer functionality is that many link layer protocols provide a multiplexing service, allowing multiple different layers above to utilize the same physical layer more or less simultaneously (*e.g.,* at the granularity of individual link layer packets). Ultimately, the link layer is concerned with the delivery of packets across some subset of all the nodes in the network (indeed, in many cases, a particular instance of the link layer is responsible for communication between exactly two neighboring nodes). The nodes involved are generally contiguous and generally quite homogeneous (*e.g.,* using the same type of physical layer technology if not the exact same physical medium), and such a subset can be thought of as its own link layer network (the term *subnet* is sometimes used here).

Atop the link layer is the *network* layer, most commonly associated with the Internet Protocol or IP. While the link layer is concerned with the communication of subsets of nodes, the purpose of the network layer is to bind subsets together into a greater whole, allowing communication across *all* nodes and creating a *lingua franca* for potentially diverse link layers. Although this same set of technologies can be used to create private networks, we focus here on the case when this greater whole is the public Internet. Two common and important aspects of creating this united whole are network-wide addressing and network-wide routing (where, in the case of the Internet, *network-wide* means globally). The former refers to the notion of giving each node a unique address so that a packet can be directed specifically to that node. Network-wide routing refers to the ability for a node to determine how to forward a packet such that it eventually reaches its intended destination (*i.e.,* which of the link layer networks the packet must pass through). Additionally, much like the link layer, network layer protocols often provide multiplexing such that they can support the more or less simultaneous use of more than one high-layer protocol.

These higher layer protocols are *transport layer* protocols. Transport layer protocols provide so called "end-to-end" communication services for the applications built atop them, where the "ends" on the Internet are typically understood to be "hosts" – most prototypically, computers (or other devices) with which users directly interact (as opposed to intermediate network nodes which serve primarily to deliver data between such hosts). Once again, a primary functionality these protocols support is multiplexing, allowing the same network to be used more or less simultaneously by multiple application layer protocols and indeed by multiple instances of the related applications on a single host. Transport protocols also often provide functionality which only makes sense between the terminal points of communication. A classic example of this is flow control, which accounts for the fact that the receiving host may be limited in how quickly it can receive and process data. For many transport layer protocols, an equally important aspect is the assurance of reliability in a broad sense. The lower layers of the Internet do not have strict rules for their reliability. Data at any lower layer may be corrupted, lost, duplicated, or reordered. At the application layer above, such unreliability is undesirable; transport layer protocols are often tasked with rectifying this situation. At the minimum, this means that transport protocols often include facilities for the detection of data corruption, which *may* also be done at other layers (it is a common feature of link layer protocols), but it has been compellingly argued *must* be done end-to-end [107]. In the case of the Internet's most widely known transport protocol – TCP – the detection of corruption is augmented with facilities to detect and retransmit missing data, remove duplicate data, and sequence data; the desired end goal is that the receiver receives exactly the data sent, in exactly the order it was sent, or – should that not be possible – receives it not at all.

1.1.1 Layers with Regard to Packets

Generally speaking, an application produces a packet of application layer data, which is bundled with additional information by protocols corresponding to each layer (typically by prepending several bytes of *header* metadata to the packet). This extra information facilitates the processing of the packet by the corresponding protocols. For example, application data is often bundled with information from TCP (a transport layer protocol) in order to ensure that it can be uniquely identified in the event that it is lost by the network (and eventually be re-sent). This is then bundled with information from IP (a network layer protocol) to specify a global destination address for the packet. This may then be bundled with information for Ethernet (a link layer protocol) to facilitate its traversal through the local Ethernet network. As the packet traverses the network, the protocol implementations on the nodes which process the packet refer to the corresponding packet metadata. Some nodes "terminate" a layer; for example, an Ethernet cable may be terminated by the nodes on either side. As part of processing the packet, these nodes will (at least conceptually) remove the associated Ethernet metadata (and, if the packet has not reached its final destination, would then likely add new metadata corresponding to the next link the packet is to traverse). Eventually, a packet (hopefully) reaches a destination application; at this point, all layers have been terminated except for the application layer, and the destination application is given the application layer data originally produced by the sender.

1.1.2 Limitations of the Common Model

The above description of the layers of the Internet is not altogether wrong, and is quite consistent with many other similar descriptions, but is also quite imperfect. A significant source of its imperfection can be traced to the fact that it is constructed from two separate views of layering. The first of these views I will call the *IEN model*, as its most noteworthy expression is found in the discontinued Internet Experiment Notes (IENs) and the Request For Comment documents (RFCs) which are today maintained by the Internet Society. The second is the view put forth by the Open Systems Interconnection model. The interplay between the parties involved here was both complex and relatively dramatic (see [106] for parts of the history), but a relevant difference is that they had somewhat different intentions. The OSI model was produced by a large body of stakeholders as part of an effort to establish a comprehensive set of standards for interconnecting computation resources. This model contains seven layers, and at one point was expected to supplant the effort described by the IENs and RFCs as the basis for the Internet. The IEN model was (at least in part) a practical effort to allow interoperation between other existing networks – such as the ARPANET, packet radio and packet satellite networks, and Ethernet networks – for the benefit of the U.S. military [23]. This "network of networks" was intentionally non-specific about the lower layers; its efforts were focused on the IP protocol and those protocols built atop it. Its requirements for lower layers were purposefully minimal (essentially just that they could deliver packets). In other words, the intention of the Internet itself does not lead to a complete listing of layers. The IEN model was also somewhat less ambitious than the OSI model with regard to the upper layers. While both assumed similar notions of applications at the top, and both contained end-to-end (transport) layers, the OSI model had two intervening layers which are not explicitly present in the IEN model.

The common model shown in Figure 1.1, then, can be seen as an attempt to "complete the picture" of the IEN model by borrowing and mapping from the OSI model. A lesser effect of this hybridizing is that while few of the OSI protocols themselves have stuck with us, their layer numbering has, which results in the common model having layer numbers which skip from L4 to L7 (L5 and L6 having no suitable mapping to the IEN model). More significantly, the data link layer and physical layers of the OSI model were essentially slid under the IP protocol, which was clearly a network layer protocol. A number of parties have played a part in establishing this hybrid, and various aspects of the OSI model are now routinely applied to the IEN/RFC-derived Internet. Indeed, coinciding with the rise of OSI's visibility and presumed eventual dominance were shifts in RFC language. For example, between 1987 and 1989, the language used in RFCs to describe the layer below IP changed from the generic "local network" (where *local* here means a directly attached network, not to be confused with a network of limited geographical area as in the term *local area network* [98, 23]) to "link layer", which if not directly OSI-derived was at least much more OSI-adjacent [20, 19].

While the result is satisfying in that it shows an apparently complete picture (*e.g.,* it includes the physical layer) with some basis in reality (*e.g.,* it does not include the widely absent OSI layers 5 and 6), it must be taken with a grain – or perhaps a handful – of salt. For a variety of reasons, the actual layering of real networks is often considerably more complex. It is not incredibly unusual, for example, for the same packet (meaning one carrying the same application-layer data unit) to be processed by multiple protocols commonly associated with the same layer, and even for a packet carrying the same application-layer data to be processed multiple times by the same actual layer. Ultimately, no fixed set of layers (and therefore no fixed numbering of layers) is universally applicable even within the confines of the Internet, or even with respect to a particular application-layer unit of data. Recent work by Zave and Rexford [121] goes a long way toward explaining many reasons behind this additional complexity and to building a more flexible model that accounts for it.

Despite its substantial limitations, in this work we largely proceed with the common model, with the justification that packets are typically processed *at least* by this set of layers. Further, the individual aspects of this dissertation generally correspond to protocols which are strongly associated with particular layers and particular layer numbers (irrespective of whether this association has a strong logical basis).

1.2 Contributions

Having discussed layering in general, we can now begin to examine the individual contributions of this dissertation, which essentially relate to the middle layers of the layer model (the physical layer being more the domain of electrical engineers, and the application layer being specific to particular applications).

Chapter 2 (and the work from which it has been adapted [80, 81]) presents a new link layer protocol designed to maximize the delivery of packets even in the wake of disruptions (*e.g.,* link failures). This work can be seen as a radical reimagining of switched Ethernet. While its intention is to maintain many of the strengths and assumptions underlying Ethernet, it discards the Spanning Tree Protocol control plane which has characterized switched Ethernet for decades. Indeed, in contrast to other more recent reimaginings of Ethernet (which build on the IS-IS control plane), this work discards the notion of a traditional control plane altogether. It is instead driven by an iteration of the essential Ethernet flood-and-learn technique, but extended to accommodate (and take advantage of) topologies containing cycles. The resulting design can learn new paths that use all links in the topology, and can do so at packet – rather than control plane – timescales.

Chapter 3 is derived from recent work [76], and has a similar goal to that of the work in the previous chapter; indeed, the two chapters also share a key underlying technique. Despite these similarities, in other ways, the two projects are almost polar opposites. While the work in the previous chapter focuses on an entirely new link layer protocol, the work in this chapter seeks to spur a minimal and backwards-compatible augmentation of existing network layer routing protocols. An unchanged routing protocol continues to focus on achieving desired behavior at steady state (satisfying various optimization criteria, for example), while the new mechanism handles only packets that the routing protocol would otherwise fail to deliver (*e.g.,* due to outdated routing state following a failure). Other work in this area is complex, challenging to deploy, or only handles a subset of failures. This work attempts to rectify this by focusing on a minimal mechanism with simple behavior and attention to issues of incremental roll-out, minimizing the costs and risks while achieving a very high level of packet delivery.

Chapter 4 is also adapted from prior work [77]. To a fair approximation, the Internet of today is, at its core, the Internet of 1983 – when IPv4 became the one true protocol of the network layer. Even with the more optimistic outlook that we are, in 2020, actually making significant progress in our effort to realize the Internet of the late 1990s (when IPv6 was introduced), we are still living in the past. While we have been waiting for the technology of the Internet to catch up, we have been accumulating an increasing number of increasingly severe new challenges. The networking research community is rich in ideas to address these challenges (*e.g.,* information-centric networking, intrinsic security techniques), but it is unclear that we can wait another two decades to see them realized. This chapter argues against the "narrow waist" dogma, and reimagines the architecture of the Internet not as a fixed feature set based on a single protocol that lurches along once per generation, but as a dynamic and evolving protocol ecosystem – a truly agile and extensible Internet that can accommodate solutions to today's problems *today...* not twenty years from now. It presents a design to achieve this goal, arguing that by leveraging modern technologies and a newly elaborated layering modularity, it can realistically be achieved in practice.

Chapter 5 is derived from previously unpublished work [78] and addresses an issue that stems from a significant change that has occurred since the core protocols of the Internet were designed. When the Internet was designed, it was assumed that the remote resource being accessed was another host. While still true in a technical sense, it is very commonly the case today that the remote resource being accessed is more abstract; from the perspective of a user, one does not connect to a specific host to check their email – they connect to a service. Such services may be implemented by many computers acting in concert, and this allows the service to scale to far more than the capacity of a single host. The Internet protocols have no explicit support for this usage. In response, various techniques for *load balancing* (*i.e.,* balancing the full load of an abstract service across a set of concrete hosts) have been introduced. To remain compatible with existing protocols, this has often involved keeping per-connection state. This violates the *fate-sharing principle* [26] and the articulated goals of IP routers (which include minimizing the state required by intermediate nodes for the sake of robustness [20]). This chapter proposes a way of using a feature of the TCP protocol (a transport layer protocol) to address this change in use case *without* violating the fate-sharing principle, while still remaining substantially backwards-compatible.

Chapter 2

Reimagining Ethernet for Resiliency

Ethernet, developed during the 1970s, continues today to be one of the most widespread layer 2 technologies. Part of this success is that it requires so little configuration. Its "plug-and-play" property ensures that when new hosts arrive (or move), there is no need to (re)configure the host or manually (re)configure switches with new routing state. In Ethernet's early history, this property was relatively easy to achieve since all nodes on an Ethernet network shared the same physical medium, but even as Ethernet grew to support multi-link topologies, its designers were careful to maintain its operational simplicity. This is in contrast to IP (at layer 3, or *L3*) where one must assign an IP address to a newly arriving host, and either its address or the routing tables must be updated when it moves to a new subnet. Even though IP has developed various plug-and-play features of its own (*e.g.,* DHCP), Ethernet has traditionally played an important role in situations where the initial configuration, or ongoing reconfiguration due to mobility, would be burdensome. Indeed, Ethernet remains widely used in enterprise networks and a variety of special cases such as temporary networks for events, wireless or virtual server networks with a high degree of host mobility, and small networks without dedicated support staff.

Because it must seamlessly cope with newly arrived (or relocated) hosts, a traditional Ethernet switch uses flooding to reach hosts for which it does not already have forwarding state. When a new host sends traffic, the switches "learn" how to reach the sender by recording the port on which its packets arrived. To make this flood-and-learn approach work, a control plane protocol maintains a spanning tree, which removes links from the network in order to make looping impossible. The lack of loops plays two essential roles: (i) it enables flooding (otherwise looping packets would bring down the network) and (ii) it makes learning simple (because there is only one path to each host).

This approach combining flood-and-learn with a *spanning tree protocol* (or STP) was first developed by Mark Kempf and Radia Perlman at DEC in the early 80s [63, 96], and is the bedrock upon which much of modern networking has been built. Remarkably, it has persisted through major changes in networking technologies (*e.g.,* dramatic increases in speeds, the death of multiple access media) and remains a classic case of elegant design. However, users now demand better performance and availability from their networks, and this approach is widely seen as having two important drawbacks:

- The use of a spanning tree reduces the bisection bandwidth of the network to that of a single link, no matter what the physical topology is.
- When a link on the spanning tree fails, the entire tree must be reconstructed. While modern spanning tree protocol variants are vastly improved over the earlier incarnations, we continue to hear anecdotal reports that spanning tree convergence time is a recurring problem in practice, particularly in high-performance settings.

In this chapter we present a new approach to L2, called the All conneXion Engine or AXE, that discards spanning tree protocols and extends the traditional Ethernet packet format. It retains the crucial plug-and-play property of Ethernet, but can use all network links (and can even support ECMP for multipath) while providing extremely fast recovery from failures (only packets already on the wire or in the queue destined for the failed link are lost when a link goes down). AXE is not a panacea, in that it does not natively support fine-grained traffic engineering, though (as we discuss later) such designs can be implemented on top. However, we see AXE as being a fairly general replacement for current Ethernets and other high-bandwidth networks where traffic engineering for local delivery is less important.

We recognize that there is a vast body of related work in this area, which we elaborate in Section 2.5, but now we merely note that *none* of the other designs combine AXE's features of plug-and-play, near-instantaneous recovery from failures, and ability to work on general topologies. We also recognize that redesigning L2 is not the most pressing problem in networking, especially as reliance on multi-hop Ethernet has diminished as inexpensive "IP switching" has risen in popularity (partially in response to the above-mentioned shortcomings of Ethernet and spanning tree protocols). Nevertheless, switched Ethernet with a spanning tree protocol remains a widely-deployed solution, and its performance is widely viewed as problematic (as evinced by the number of modifications and extensions vendors now deploy). What we present here is the first substantial rethinking of switched Ethernet that not only improves its performance (in terms of available bandwidth and failure recovery), but also entirely removes the need for a traditional control plane at this layer, in stark contrast to the status quo and to many of the redesigns discussed in Section 2.5.

In the next section, we describe AXE's design, starting with a simplified "clean" design with provable correctness properties under ideal conditions, and moving on to a practical version that is more robust under non-ideal conditions. We then describe a P4-based implementation (Section 2.2) and extensions to support multicast (Section 2.3) before evaluating AXE's performance through simulation in Section 2.4. We end the chapter with a discussion of related work in Section 2.5.

2.1 Design

Traditional Ethernet involves two separate processes: (i) creating a tree (via STP or its variants) and (ii) forwarding packets along this tree via a flood-and-learn approach. In AXE, we only use a single mechanism – flood-and-learn – in which flooded packets are prevented from looping not with a spanning tree, but with the use of switch-based packet deduplication. Packet deduplication enables one to safely flood packets on any topology because duplicate packets are detected and dropped by the AXE switches instead of following an endless loop. This allows AXE to utilize all network links, and removes the need for a complicated failure recovery process (like STP) when links go down.

But these advantages come at the cost of a more subtle learning process. Without a spanning tree, packets can arrive along more than one path, so AXE must actively choose which of these options to learn. Furthermore, in the presence of failures, some paths may become obsolete – necessitating they be "unlearned" to make way for better ones.

To give a clearer sense of the ideas underlying AXE, we first present a *clean* version of the algorithm in Section 2.1.1 that has provable properties under ideal conditions. We then present a more practical version in Sections 2.1.2-2.1.4 that better addresses the non-ideal conditions found in real deployments. Both designs use the standard Ethernet *src* and *dst* addresses and additionally employ an AXE packet header with four more fields: the "learnable" flag *L*, the "flooded" flag *F*, a hopcount *HC*, and a *nonce* (used by the deduplication algorithm). We make no strong claims as to the appropriate size of the *HC* and *nonce* fields, but for the sake of rough estimation, note that if the entire additional header were 32 bits, one could allocate two bits for the flags, six for *HC* (allowing up to 64 hops), and the remaining 24 for the *nonce*. In order to maintain compatibility with unmodified hosts, we expect this header to be applied to packets at the first hop switch and removed at the last hop switch (which might be software virtual switches if AXE were to be deployed in, *e.g.,* a virtualized data center). In addition, switches enforce a maximal *HC* (defined by the operator) to prevent unlimited looping in worst-case scenarios.

As we discuss more fully in Section 2.1.3, each switch uses a deduplication filter to detect (and then drop) duplicate packets based on the triplet <*src, nonce, L*>. While there are a number of ways to construct such a filter, here we use a hash-table-like data structure that can experience false negatives but no false positives (*i.e.,* no non-duplicates are dropped by the filter, but occasional duplicates may be forwarded).

The forwarding entries in a switch's learning table are indexed by an Ethernet address and contain the arrival port and the *HC* of the packet from which this state was learned (which are the port one would use to reach the host with the given address and the number of hops to reach it if a packet followed the reverse path).

Finally, note that AXE pushes the envelope for fast failure *response*, but does not make any innovation in the realm of fast failure *detection*. Instead, AXE can leverage any existing detection techniques. The speed of such techniques can vary widely. Early control plane based techniques could take seconds. More modern techniques can be considerably faster. For example, SONET [74] and BFD [62] can achieve detection in tens of milliseconds. Detection based on Ethernet CFM [49] can even be as low as a few milliseconds. While not everyday networking technologies, it is notable that specific Ethernet equipment (and the Ethernet-derived EtherCAT used in automation applications) can achieve detection delays as low as 10µs [29, 59]. Anecdotally, we have been told by some network operators (running networks that connect hyperscaled data centers) that 1ms detection is within reach for emerging mainstream equipment.

2.1.1 Clean Algorithm

Recall that the traditional Ethernet approach involves flooding and learning: (i) a packet is flooded when it arrives at a switch with no forwarding state for its destination address, and (ii) an arriving packet *from* a host establishes a forwarding entry *toward* that host. The approach can be so simple due to the presence of a nontrivial control plane algorithm – STP – which prunes the effective topology to a tree. Because it operates on a general topology without any control plane protocol, the clean version of AXE is slightly more complicated and can be summarized as follows:

Header insertion and host discovery: *When a packet arrives without an AXE header, a header is attached with* HC=1*, the* L *flag set, and the* F *flag unset. If there is no forwarding state for the source, an entry is created and the* F *flag is set.* The first step merely initializes the header; the second step is how switches learn about their attached hosts (and the subsequent flood informs the rest of the network how to reach this host).

Flooding: *When a packet with the* F *flag set arrives, it is flooded out all other ports. When a packet with the* F *flag unset arrives at a switch with no forwarding state for its destination or for which the forwarding state is invalid (e.g., its link has failed), the* F *flag is set, and the packet is flooded. The flooded packets have the* L *flag set only if the flood originated at the first hop (i.e.,* HC=1*).* The flooding behavior is similar to traditional learning algorithms, with the addition of the explicit flooding and learning flags.

Learning and unlearning: *Switches learn how to reach the source from flooded packets with the* L *flag set, and unlearn (erase) state for the destination whenever they receive a flood packet with the* L *flag unset.* While traditional learning approaches learn how to reach the source host from all incoming packets, in AXE we can only reliably learn from packets flooded from the first hop (since packets flooded from elsewhere might have taken circuitous paths, as we discuss below). Moreover, when switches learn from flooded packets, they choose the incoming copy that has the smallest *HC*. When a flooded packet arrives with the *L* flag unset, it indicates that there is a problem reaching the destination (because the flood originated somewhere besides the first hop, as might happen with a failed link); this is why switches unlearn forwarding state when such packets arrive.

Wandering packets: *When the* HC *of a non-flooded packet reaches the limit, the packet is flooded (with the* F *flag set and the* L *flag unset) and local state for the destination is erased.* If the forwarding state has somehow created a loop, erasing the state locally ensures that the loop is broken. Flooding the packet (with the *L* flag unset) will cause all forwarding state to the destination host to be erased (so the next packet sent *by* that host will be flooded from the first hop, and the correct forwarding state learned).

Algorithm 2.1 shows pseudocode of this clean algorithm, which processes a single packet *p* at a time and consults the learning table *Table* by calling a *Lookup()* method with the desired Ethernet address. *Lookup()* returns *False* if there is no corresponding table entry. *Table.Learn()* inserts/updates state in the table, and *Table.Unlearn()* removes state. *IsPortDown()* returns *True* if the given port is unavailable (*e.g.,* the link has failed). The *IsDuplicate* value (obtained from the deduplication filter) indicates whether the switch has already seen a copy of that packet (as duplicates of a packet may arrive on multiple ports if the topology contains cycles). *Output()*

sends a packet via a specified port, and *Flood()* sends a packet via all ports except the ingress port (*p.InPort*).

We define *ideal* conditions as when all hosts are stationary, there are no packet losses, there are no link or router failures, there are no deduplication mistakes (our mechanism ensures that there are no false positives, so this requires that there also be no false negatives), the maximal *HC* is larger than the diameter of the network, and no switch mistakenly thinks a host is directly attached when it is not. Under such conditions, we can make the following two statements about the behavior of the algorithm which hold regardless of the forwarding state currently in the switches (subject to the constraint about directly attached hosts):

Delivery: *Packets will be delivered to their destination.* This holds because there are only three possibilities: (i) the packet reaches the intended host following existing forwarding state (*i.e.,* it is not flooded), (ii) the packet reaches a switch without valid forwarding state and then is flooded and therefore reaches its destination, or (iii) the hopcount eventually reaches the maximal value causing the packet to be flooded which therefore reaches its destination. What cannot happen under our assumption of ideal conditions is that forwarding state on a switch delivers the packet to the wrong host (except in the case of flooding, where it reaches all hosts). Note that this line of reasoning guarantees delivery even in the non-ideal case when there are link/router failures, as long as they do not cause a partition during the time the packet and its copies are in flight.□

Eventually shortest path routes: *Forwarding state that is learned from undisturbed floods will route packets along shortest paths.* We call a flood from source *A* with *L* set an *undisturbed flood* if no unlearning of *A* takes place during the time the flood has packets in transit (*e.g.,* due to other floods with destination *A* which have the *L* flag *un*set). New state is installed if and only if packets have both the *F* and *L* flags set, which happens only when packets are flooded from the first-hop switch. When intermediate switches receive multiple copies of the same packet, the ultimate state reflects the lowest hopcount needed to travel from the first-hop switch to the intermediate switch. Thus, as long as no state is erased during this process, when all copies of the flood from source *A* with *L* set have left the network, every switch ends up with state pointing toward an output port that has a shortest path to the destination. Any packet following this state will take a shortest path to *A*. The reason we require the flood to be undisturbed is because if some state is erased during the original flood, then the last state written may not point towards a shortest path (*i.e.,* the state that was erased may have been the state reflecting the shortest path). Note that this statement of correctness applies even if two or more flooded packets from *A* with the *L* flag set were in flight at the same time: since the network topology is constant under ideal conditions, all last-written state will point towards a shortest path.□

Thus, the clean design under ideal conditions, but with arbitrary initial forwarding state (subject to the constraint on attached hosts), will deliver all packets, and undisturbed floods will install shortest-path forwarding state. However, under non-ideal conditions we can make no such guarantees. Packets can be lost and routes can be far from shortest-path. Indeed, one can find examples where routing loops can be established under non-ideal conditions (though these loops will be transient, as a packet caught in such a loop will reach the maximal hopcount value, be flooded, and cause the incorrect forwarding state to be erased).

Before turning to our practical algorithm, we now explain in more detail why we need both

```
1: if p has no AXE header then
2:     Add AXE header
3:     $p.Nonce \leftarrow new\_nonce()$
4:     $p.HC \leftarrow 1$, $p.L \leftarrow True$
5:     if ! $Table.Lookup(p.EthSrc)$ then
6:         $p.F \leftarrow True$
7:         $Table.Learn(p.EthSrc,\ p.InPort,\ p.HC)$
8:     else
9:         $p.F \leftarrow False$
10:    end if
11: else
12:    $p.HC \leftarrow p.HC + 1$
13: end if
14: if $p.F$ then
15:    if ! $IsDuplicate$ then
16:        $Flood(p)$
17:    end if
18:    if ! $p.L$ then
19:        $Table.Unlearn(p.EthDst)$
20:    else if ! $IsDuplicate$ then
21:        $Table.Learn(p.EthSrc,\ p.InPort,\ p.HC)$
22:    else if ! $Table.Lookup(p.EthSrc)$ then
23:        $Table.Learn(p.EthSrc,\ p.InPort,\ p.HC)$
24:    else if $IsPortDown(Table.Lookup(p.EthSrc))$ then
25:        $Table.Learn(p.EthSrc,\ p.InPort,\ p.HC)$
26:    else if $Table.Lookup(p.EthSrc).HC \geq p.HC$ then
27:        $Table.Learn(p.EthSrc,\ p.InPort,\ p.HC)$
28:    end if
29: else if ! $Table.Lookup(p.EthDst)$ then
30:    $p.F \leftarrow True$, $p.L \leftarrow (HC = 1)$
31:    $Flood(p)$
32:    $Output(p, p.InPort)$
33: else if $IsPortDown(Table.Lookup(p.EthDst))$ then
34:    $p.F \leftarrow True$, $p.L \leftarrow (HC = 1)$
35:    $Flood(p)$
36:    $Output(p, p.InPort)$
37: else if $p.HC > MAX\_HOP\_COUNT$ then
38:    $p.F \leftarrow True$, $p.L \leftarrow False$
39:    $Flood(p)$
40:    $Output(p, p.InPort)$
41:    $Table.Unlearn(p.EthDst)$
42: else
43:    $Output(p, Table.Lookup(p.EthDst).Port)$
44: end if
```

Algorithm 2.1: *The clean AXE algorithm.*

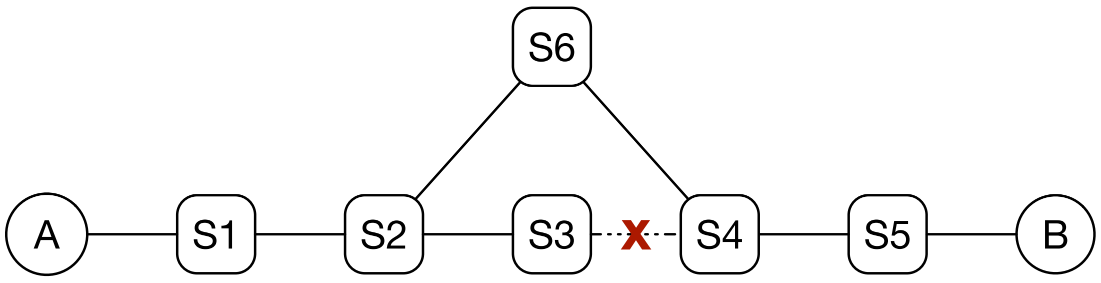


Figure 2.1: *A network with two hosts (A and B).*

the L and the F flags. Note that in traditional L2 learning, flooding and learning are completely local decisions: a switch floods a packet if and only if that switch has no forwarding state for the destination, and it learns from all packets about how to reach the source. This works because packets are either constrained to a well-formed tree (which is established via STP) or dropped (while STP is converging). In contrast, AXE switches set the F flag the first time the packet arrives at a switch that has no forwarding state for the destination (or has forwarding state pointing to a dead link), and then the packet is flooded globally regardless of whether subsequent switches have forwarding state for the destination. This allows for delivery even when the forwarding state is corrupted (*e.g.,* by failures or unlearning) and there is no guarantee that following the remaining forwarding entries will deliver the packet.

While flooding is more prevalent in AXE than in traditional L2, learning is more restrictive: The clean AXE algorithm only learns from flooded packets with the L flag set. This is because when packets are flooded from arbitrary locations, the resulting learned state might be misleading. Consider the network depicted in Figure 2.1, and imagine packets flowing from A to B along the path $S1$–$S2$–$S3$–$S4$–$S5$. If there is a disruption in the path, say the link $S3$–$S4$ is broken, AXE will flood packets arriving at $S3$ instead of attempting to send them down the failed link toward $S4$. Packets flooded from a failure, such as those handled by $S3$, must necessarily go *backwards* (in addition to going out all other ports), as that may be the only remaining path to the destination (as is the case in Figure 2.1, where after the failure of $S3$–$S4$, the only valid path to B for packets at $S3$ is backward through the path $S2$–$S6$–$S4$–$S5$). One certainly does not want to learn from packets that have traveled backwards, as one could potentially be learning the *reverse* of the actual path to the destination. In this example, $S2$ would learn that A is towards $S3$, which is clearly incorrect. Thus, when packets are flooded after reaching a failure, the L flag is switched off, indicating that they are unlikely to be suitable for learning.

2.1.2 Practical Algorithm

We presented the clean algorithm to illustrate the basic ideas and show how they lead to two correctness results under ideal conditions. These ideal conditions do not hold if there is congestion, since packet losses can occur; in the clean design we liberally use packet floods when problems are encountered, which only exacerbates congestion. Thus, for our more practical approach we modify some aspects of the algorithm to reduce the number of floods, to enable learning from non-flood packets, and to give priority to flood packets. Unfortunately, our correctness results no longer hold with these modifications in place. However, simulations suggest that both the clean and the practical designs perform well under reasonable loads and failure rates, but that the practical algorithm is significantly better at dealing with and recovering from overloads or networks with high rates of failure.

The main changes from the clean design are as follows:

- When a packet exhausts its *HC*, we merely drop the packet and erase the local forwarding state (rather than flooding the packet). This reduces the number of floods under severe overloads, though the packet in question is not delivered (which violates the first correctness condition under ideal conditions).

- Switches learn from all packets with the *L* flag set, not just flooded packets. This also reduces the number of floods, though the resulting paths are not always the shortest paths (which violates the second correctness condition under ideal conditions).

- Switches have one queue for flooded packets and another for non-flooded packets, and the flood queue is given higher priority. Because floods occur in the absence of any state or the presence of bad state, and because floods trigger learning, accelerating the delivery of floods enhances the learning of good state.

We also introduced various other wrinkles into the practical algorithm that improved its performance in simulations, such as only unlearning at the first hop and dealing with hairpinning (discussed later). While the old correctness conditions no longer hold with these changes, we *can* say that under ideal conditions (i) unless the state for an address contains a loop or is longer than the maximal HC, packets sent to it will be delivered; and (ii) the forwarding state established by undisturbed learning will enable packets to reach the intended destination, but the paths are not guaranteed to be shortest. We feel that the loosening of the correctness results is a good trade-off for the improved behavior under overload.

We later extend AXE to handle both multipath and multicast delivery, but in the next two subsections we discuss the implementation of the deduplication filter and then examine the pseudocode for the practical unipath algorithm.

2.1.3 The Deduplication Filter

Our deduplication filter provides approximate set membership with false negatives – the opposite of a Bloom filter's approximate set membership with false positives. While there are many ways to build such a filter; the approach we use is essentially a hash set with a fixed size and no collision resolution (that is, you hash an entry to find a position and then just overwrite whatever older entry may be there). Each entry contains a <*src, nonce, L*> tuple. On reception, these packet fields are hashed along with an arbitrary per-switch *salt* (*e.g.,* the Ethernet address of one of its interfaces), and the hash value is used to look up an entry in the filter's table. If the *src*, *nonce*, and

L in the table entry match the packet, the packet is a duplicate and the filter returns *True*. If the values stored in the table entry do not match the packet, the values in the table entry are overwritten with the current packet's values, and the filter returns *False*.

Note that the response that a packet is a duplicate can only be wrong if the nonce has been repeated. We implement the nonce using a counter (thus, it is essentially a sequence number in our implementation), and given that the field has a finite size, it must eventually wrap. Thus, it is conceivable that a switch might produce a nonce such that a filter somewhere in the network still contains an entry for an older packet with the same nonce, and the possibility of this increases as the filter size increases (which is otherwise a good thing). Fortunately, we can compute the minimal amount of time required for this to occur. For example, with a 24 bit nonce space (as put forth as a possibility earlier), a 10Gbit network transmitting min-sized packets would require around 1.16 seconds to wrap the counter (which is a relatively long time considering that the two-queue design ensures that flooded packets are delivered quickly). By timestamping entries in the filter, any entry older than this can be invalidated.

The negative response, however, can happen simply when two packets hash to the same value: the second would overwrite the first, and if another copy of the first arrived later, it would not be detected as a duplicate. We lower the probability of these false negatives by only applying packet deduplication to flooded packets (since flooded packets always "try" to loop, while non-floods only loop in the rarer case that bad state has been established), and the per-switch salt value decreases the chance that the same false negative will happen at two different switches.

2.1.4 Details of Practical Algorithm

We can now present the pseudocode for a more practical version of the AXE unipath design (Algorithm 2.2), in which we explicitly include invocations of the deduplication mechanism, but assume that the AXE header has already been added if not present and that *HC* has been incremented. The code has two phases: the first largely involves deduplication and learning/unlearning, while the second is responsible for forwarding the packet.

Some changes relative to the clean algorithm are fairly straightforward; notably the decision to drop rather than flood packets where the *HC* exceeds the maximum is embodied in lines 3-10, and the decision to learn from all packets with the *L* flag set (rather than just flooded packets) is captured in line 31. Other changes are relatively minor, such as learning even from packets with the *L* flag not set if they have a smaller *HC* (line 30), and unlearning only at first hops (lines 22-27).

A more complicated change is that concerning "hairpin turns" (line 63) where a switch has a forwarding entry pointing back the way the packet came. For an example of this, return to Figure 2.1 and imagine that a packet from *A* arriving at *S*3 encounters state pointing back towards *S*2. Before forwarding the packet back to *S*2, the *L* flag is unset (line 65) as *S*2 learning that the path to *A* is via *S*3 is clearly ludicrous. When the packet arrives back at *S*2, *S*2 may still have state pointing towards *S*3. While *S*2 and *S*3 could simply hairpin the packet back and forth until the hopcount reaches the maximum and the loop is resolved the same as any other loop, the combination of the forwarding state and the unset *L* flag are used to infer the presence of this special case length-two cycle and remove the looping state immediately (line 68). Note that we again make the practical

```
▷ * Start of first phase. *

if $p.HC > MAX\_HOP\_COUNT$ then
    ▷ Either the forwarding state loops or this is an old flood which
    ▷ the deduplication filter has never caught.
    if $!p.F$ then
        Table.Unlearn(p.EthDst)                          ▷ Break looping forwarding state.
    end if
    return                                               ▷ Drop the packet.
end if

▷ Check and update the deduplication filter.
if $p.F$ then
    IsDuplicate ← Filter.Contains( <p.EthSrc, p.Nonce, p.L> )
    Filter.Insert( <p.EthSrc, p.Nonce, p.L> )
else
    ▷ Non-floods aren't deduped; assume it's not a duplicate.
    IsDuplicate ← False
end if

SrcEntry ← Table.Lookup(p.EthSrc)
if $!IsDuplicate$ and $!p.L$ and SrcEntry and $SrcEntry.HC = 1$ then
    ▷ We're seeing (for the first time) a packet which probably
    ▷ originated from this switch and then hit a failure. Since our
    ▷ forwarding state apparently points to a failure, unlearn it.
    $Table.Unlearn(p.EthDst)$
end if

if $!SrcEntry$                                           ▷ No table entry, may as well learn.
  or $p.HC < SrcEntry.HC$                                ▷ Always learn a better hopcount.
  or $(p.L$ and $!IsDuplicate)$                          ▷ Common case, learnable non-duplicate.
 then
    Table.Learn(p.EthSrc, p.InPort, p.HC)                ▷ Update learning table.
end if
```

Algorithm 2.2: *The practical AXE algorithm.*

```
35: ▷ * Start of second phase. *
36:
37: if IsDuplicate then
38:     return                          ▷ We've already dealt with this packet; drop the duplicate.
39: end if
40:
41: if p.F then
42:     ▷ Flooded packets just keep flooding.
43:     Flood(p)                                            ▷ Send out all ports except InPort.
44:     return                                                              ▷ And we're done.
45: end if
46:
47: DstEntry ← Table.Lookup(p.EthDst)                                ▷ Look up the output port.
48: if !DstEntry or IsPortDown(DstEntry.Port) then                             ▷ No valid entry.
49:     if !p.L then
50:         return                      ▷ Packet hairpinned but is now lost. Drop and give up.
51:     end if
52:
53:     p.F ← True                                                  ▷ About to flood the packet.
54:     if p.HC = 1 then
55:         ▷ This is the packet's first hop. L is already set.
56:         Flood(p)                                ▷ Flood learnably out all ports except InPort.
57:     else
58:         p.L ← False                         ▷ Not the first hop; don't learn from the flood.
59:         Filter.Insert( <p.EthSrc, p.Nonce, p.L> )                             ▷ Update filter.
60:         Flood(p)                                         ▷ Sends out all ports except InPort.
61:         Output(p, p.InPort)                                           ▷ Send backwards too.
62:     end if
63: else if DstEntry.Port = p.InPort then                              ▷ Packet wants to hairpin.
64:     if p.L then                                     ▷ If learnable, try once to send it back.
65:         p.L ← False                                                 ▷ No longer learnable.
66:         Output(p, p.InPort)
67:     else                                                       ▷ Packet trying to hairpin twice
68:         Table.Unlearn(p.EthDst)                             ▷ Break looping forwarding state
69:     end if
70: else
71:     Output(p, DstEntry.Port)                                  ▷ Output in the common case.
72: end if
```

Algorithm 2.2: *The practical AXE algorithm (continued).*

decision to drop the packet and not convert it to a flood, as the existence of such a cycle is generally indicative of already adverse conditions. The other possibility is that the packet arrives back at $S2$ and $S2$ now has state which does *not* point to $S3$. Such hairpinning can arise, for example, due to multiple deduplication failures. More commonly, it is caused by our use of two forwarding queues. With two queues, a flooded packet can "pass" an already queued non-flood packet on a switch; when the non-flood one reaches the next switch, the flooded one has already changed the switch's state. For example, imagine a non-flood packet to B queued on $S2$ with $S2$ not yet aware that the $S3 - S4$ link has failed. A learnable flood packet from B arrives at $S2$ (via $S6$), is placed in the high priority flood queue, and is immediately forwarded to $S3$: $S3$ learns that the path to B is back towards $S2$. By the time the non-flood packet finally leaves the queue on $S2$, the state on $S2$ *already* reflects the correct new path to B via $S6$ – as does the state on $S3$ (thus requiring the packet to take a hairpin turn).

2.1.5 Enhancements to the Design

AXE can trivially accommodate various features that Ethernet operators have come to rely on (*e.g.,* VLAN tagging); here we discuss three more significant ways AXE can be extended.

Periodic optimization

In order to make sure that non-optimal paths do not persist, switches will periodically flood packets from directly attached hosts, allowing all switches to learn new entries for it (a switch knows that it is a host's first hop because of the hopcount in its forwarding entry).

Traffic engineering

The approach AXE takes to ensure L2 connectivity is designed to be orthogonal to potential traffic engineering approaches. While some approaches aim to carefully schedule each flow in order to avoid congestion and meet other policy goals, work such as Hedera [3] showed that identifying and scheduling only elephant flows can provide substantial benefit. AXE and Hedera are complementary, and using them together only requires one extra bit in the header – a flag to indicate whether the packet should follow AXE paths or Hedera paths. In a network using both approaches, we use Hedera to compute paths for elephant flows, while mice use AXE paths. When a packet on a Hedera-scheduled flow encounters a failure, we set the extra "AXE path" flag and then route the packet with AXE. The flag is required to ensure that the packet continues to be forwarded to its destination using only AXE, as a combination of Hedera and AXE paths could produce a loop. In this way, traffic can be scheduled for efficiency, but scheduled traffic always has a guaranteed fallback as AXE ensures connectivity whenever the physical network is not partitioned.

ECMP

While the discussion thus far has been about unipath delivery, extending AXE to support ECMP requires only three changes: modifying the table structure, enabling the learning of multiple ports, and encouraging the learning of multiple ports. We extend the table by switching to a bitmap of learned ports (rather than a single number), and by keeping track of the nonce of the packet from which the entry was learned (*i.e.,* the nonce is now stored in the learning table itself in addition to being stored in the deduplication filter). Upon receiving a packet with the L and F flags set, if the hopcount and nonce are the same as in the table, we add the ingress port to the learned ports bitmap. If these two fields do not match, we replace (or don't replace) the entry based on much the same criteria as for the unipath algorithm. If L is set and F is not, we check that the hopcount and port are consistent with the current entry. If not, we replace the entry and flood the packet (to encourage a first-hop flood).

A problem with this multipath approach is that while it is easy to learn multiple paths in one direction – the originator must flood to find the recipient, and this flood allows learning multiple paths – it is not as easy to learn multiple paths in the reverse direction, as individual packets back to the originator will follow *one* of the equal cost paths and therefore only establish state along that single path. To address this, we need to flood in the reverse direction as well, encouraging multipath learning in both directions. This is, in fact, similar to the behavior of the "clean" algorithm discussed in Section 2.1.1, though our implementation here is slightly more subtle in order to integrate with the rest of the practical algorithm and to provide multiple chances to learn multiple paths given that we do not expect it to operate under ideal conditions. The key is adding another port bitmap to each table entry – a "flooded" bitmap. When a packet is going to be forwarded using an entry, if the bit corresponding to the ingress port is 0 ("hasn't yet been flooded") and the packet's hopcount is 1 (this is its first hop), we set the flooded bit for the port, and perform a flood. This is a first-hop flood, so L is set, and it therefore allows learning multiple paths. The obvious downside here is some additional flooding, but the upside is that equal cost paths are discovered quickly.

2.1.6 Reasoning About Scale

In this section so far, we have focused on the details of the AXE algorithm, its properties, and what functionality it can support. Here we address another rather basic question at a conceptual (but not rigorous) level: How well does it scale?

AXE is an L2 technology that adopts the flood-and-learn paradigm. Our question is not how well a flood-and-learn paradigm can scale, because that will depend in detail on the degree of host mobility, the traffic matrix, and the bandwidth of links. Rather, our question is whether AXE's design hinders its ability to scale to large networks beyond the baseline inherent to any flood-and-learn approach. Moreover, we focus on bandwidth usage, and do not consider the impact of AXE's additional memory requirements because hardware/memory constraints will change over time. For bandwidth, the main factor that differentiates AXE from traditional Ethernet is the use of flooding when failures occur.

We can estimate the impact of this design choice as follows. Consider a fully utilized link of bandwidth B that suddenly fails. If the average round-trip-time of traffic on the link is RTT, then roughly $RTT * B$ worth of traffic will be flooded before congestion control will throttle the flows. If we want to keep the overhead of failure-flooding below 1% of the total traffic, that means we can tolerate no more than f failures per second, where $f = \frac{0.01}{RTT}$. If the RTT is 1ms, then the network-wide failure rate would need to be less than 10 per second. Assuming that links have MTBFs greater than 10^6 seconds, then the traffic due to failures is less than 1% of the link for networks with less than 10^7 links. Thus, in terms of bandwidth, AXE can scale roughly as well as any learn-and-flood approach.

2.2 P4 Implementation

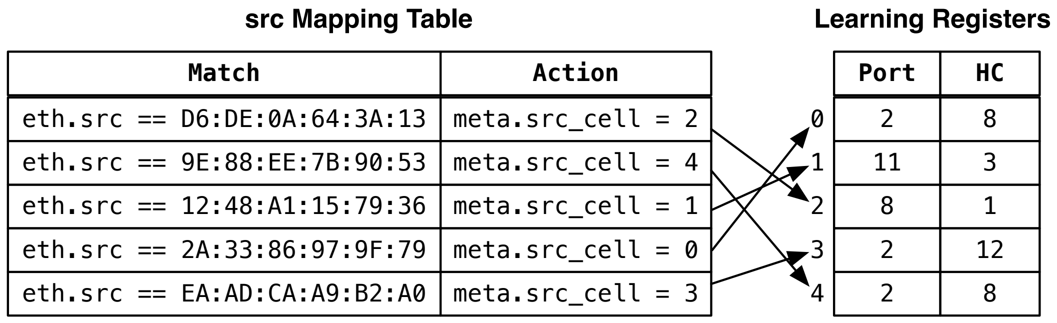
We have argued – and in Section 2.4 we show through simulation – that AXE provides a unique combination of features. However, if vendors were required to create a new generation of ASICs to support it, then AXE would likely be no more than an intriguing academic idea. We think AXE can avoid this unfortunate fate because of the rise of highly reconfigurable hardware with an open-source specification language, and here we are thinking primarily of RMT [18] and P4 [61], but other such efforts may arise. In this section we discuss our implementation of AXE in P4, which we have tested in Mininet [85] using the bmv2 P4 software switch [15]. This testing verified that our implementation does, indeed, implement the protocol as expected.

While the rest of this section delves into sometimes arcane detail, our point here is simple: it may be possible to deploy AXE by simply loading a P4 implementation into a compatible off-the-shelf switch. While this does not assure deployment, it does radically reduce the barriers.

Unlike a traditional general-purpose programming language, P4 closely aligns with the architecture of network forwarding ASICs. Programs consist of three core components: a packet parser specified by a finite state machine, a series of match-action tables similar to (but more general than) OpenFlow [82], and a *control flow function* which defines a processing pipeline (describing which tables a packet is processed by and in which order). The parser is quite general and easily implements AXE's modified Ethernet header. Thus, our primary concern was how to implement the AXE forwarding algorithm as a pipeline of matches and actions.

Putting aside the nonce, deduplication filter, and learning (discussed below), the AXE algorithm is simply a series of *if* statements checking for various special conditions. Such *if* statements can be implemented in two ways in P4: either as tables which match on various values (with the default fall-through entry acting as an *else* clause), or as actual *if* statements in the control flow function. The "bodies" of the *if* statements are implemented as P4 *compound actions*. We were able to use the slightly more straightforward latter method almost exclusively, which allowed us to structure our P4 code very similarly to the pseudocode shown above. This approach, however, is not without its caveats.

As control flow functions cannot directly execute actions, we currently have a relatively large number of "dummy" tables that merely execute a default action; the control flow function invokes these tables simply to execute the associated action (that is, the tables are always empty). If hardware performance is related to the length of the forwarding pipeline, or if there are hard limits on the number of tables (less than the 26 that we currently require), the code may need to be reorganized. Specifically, we can take the Cartesian product of nested conditionals to collapse several of them into a single table lookup. This approach can likely reduce the pipeline length dramati-



Match	Action
eth.src == D6:DE:0A:64:3A:13	meta.src_cell = 2
eth.src == 9E:88:EE:7B:90:53	meta.src_cell = 4
eth.src == 12:48:A1:15:79:36	meta.src_cell = 1
eth.src == 2A:33:86:97:9F:79	meta.src_cell = 0
eth.src == EA:AD:CA:A9:B2:A0	meta.src_cell = 3

	Port	HC
0	2	8
1	11	3
2	8	1
3	2	12
4	2	8

Figure 2.2: src *to register cell mapping, populated by the control plane. A nearly identical table exists for* dst.

cally, though the resulting code will surely become less readable. Whether such an optimization is necessary depends on the particular features and limitations of the associated ASIC (as well as the optimizations that the compiler backend for the target ASIC applies).

Learning: P4 tables cannot be modified from the data plane – only from the control plane. This may be reasonable for a simple L2 learning switch: when a packet's source address is not in the table, the packet is sent to the switch's control plane, which creates a new table entry. Such a trip from data plane to control plane and back has a latency cost, however, and we would like to avoid it whenever possible, especially considering that AXE table entries contain not only the port, but the hopcount to reach the address' host, and keeping this hopcount information up to date is important to the algorithm. We achieve this by separating learning into two parts, as depicted in Figure 2.2. The first part is a table, which is populated by the control plane the first time a given Ethernet address is seen, much like a conventional L2 learning switch. However, instead of the table simply holding the associated port, it instead contains an index into the second part – an array of P4 registers, which are data plane state that *can* be modified by actions in the data plane. Thus, when processing a packet that requires changing learning state, it can be done entirely within the data plane, with the sole exception of the first packet (for which the control plane must allocate a register cell and add a table entry mapping the Ethernet address to it). As the P4 specification evolves, or hardware implementations support new features, it may be possible to eliminate control plane interaction entirely.

Deduplication Filter: The deduplication filter is a straightforward implementation of the design discussed in Section 2.1.3. For reasons similar to the above, we again use an array of P4 registers to hold the filter entries rather than a P4 table (actually, we use a separate register array for each field, as the struct-like registers described in the P4 spec were not yet supported in bmv2 or its compiler at the time of our implementation). Then a P4 *field list calculation* is used to specify a hash of the appropriate header fields (the source Ethernet address, the nonce, and the *L* flag) along with the per-switch salt value which is stored in a P4 register and populated by the control plane at startup. The computed hash is stored in packet metadata and used to index into the filter arrays.

Nonce: A switch must assign a nonce to each packet it receives from a directly-attached host. A straightforward approach is to use a single P4 register to hold a counter, though this requires that reading and incrementing the register be atomic, which may be problematic for real hardware if different ports essentially operate in parallel. Instead, we can use a nonce register per port instead of a single shared register per switch. As each port would have an independent counter, nonce allocations would no longer be entirely unique. However, this is not problematic because, as discussed in Section 2.1.3, the deduplication key is a tuple of <*src, nonce, L*>. For the typical case when a given host interface is only attached to a single port, the *combination* of the interface's address and a per-port nonce *will* be unique (this may preclude some types/implementations of link aggregation; however, AXE obviates some motivations for link aggregation anyway).

Link status: P4 does not specify a way for the data plane to know about link liveness. Our implementation emulates this functionality by creating "port state" registers, and we manually set their values to simulate port up and down events. In a real hardware implementation, such registers could be manipulated by the control plane as it detects link state changes using whatever mechanisms the switch provides for link failure detection. We also speculate that P4-capable ASICs will have some way to query this information more directly from the data plane (without control plane involvement) for at least some failure detection mechanisms.

Our P4 implementation is not written with an eye towards efficiency on any particular P4 target, as targets are diverse and no P4 hardware target is yet available to us. Nevertheless, we see the existence of a functionally complete P4 implementation as a promising beginning. More broadly, we see the fact that AXE can be implemented in an ASIC-friendly language like P4 as an indicator that it may be suitable for ASIC implementation more generally – reconfigurable or otherwise.

2.3 Multicast

Many L2 networks implement multicast via broadcast, with filtering done by the host NICs (sometimes with the addition of switches implementing "IGMP snooping" [25] wherein an ostensibly L2 device understands enough about L3 to prune some links). We investigated whether we could use AXE ideas to provide native support for multicast in a relatively straightforward way, and found the answer to be *yes*.

We essentially try to emulate a DVMRP-like [115] multicast model, with source-specific trees for each group. While AXE's ability to safely flood makes reliable delivery easy, the design challenge is to enable rapid construction (and reconstruction) of trees in order to avoid the additional traffic overhead of unnecessary flooding. Our approach forms multicast trees by initially sending all packets for the group out all ports. Unnecessary links and switches are then pruned. When the topology changes or when new members are added to a pruned section of the tree, we simply reset the tree and reconstruct it; this avoids *maintaining* a tree as changes occur (which turns out to be quite difficult).

Multicast in AXE has four types of control messages: JOIN, LEAVE, PRUNE, and RESET. The first two are how hosts express interest/disinterest in a group to their connected switches.

PRUNE is much the same as its DVMRP counterpart and is used for removing ports and switches from the tree. RESET enables a switch to indicate that something has changed which necessitates that the current tree be invalidated and rebuilt; we come back to this shortly.

Going into the algorithm in more detail, we retain much of the AXE header, but remove the L flag and add a "multicast generation number" field which is used for coordination. A source-group's root switch (the switch to which the source is attached) dictates a generation number for the source-group pair. Other switches that are part of the group simply track the latest number. All switches stamp all packets for this source-group (including control messages) with the latest generation number of which they are aware. If a non-root switch receives a packet for the current generation, it forwards the packet out all of the currently unpruned ports for the source-group. If the packet is for a new generation, the tree is being rebuilt; the switch moves to the new generation number and un-prunes all ports. If a packet is stamped with an old generation number, the packet is sent over all ports; this is not optimal, but it ensures that outstanding packets from old generations are delivered even while a new generation tree is being established.

As mentioned above, when constructing a new tree, all ports are initially unpruned – this is basically the equivalent of flooding. Switches therefore potentially receive packets from the root on multiple ports, and can decide on an "upstream" port based on the best hopcount. When a switch receives a packet from the group on any port that is not its upstream port, it sends a PRUNE in response. This causes the packet's original sender to stop sending on this port. Pruning is kept consistent by ignoring PRUNEs for any generation except the current one. In this way, the initial flood-like behavior is cut down to a shortest-path tree.

When any switch notices that the tree may need to be rebuilt due to either (i) being invalid (*i.e.,* uses a link that has failed) or (ii) possibly needing to expand or change shape (due to a port going up or down or a new member joining the group), the switch enters *flood mode* for the group, and may send a RESET. While a switch is in flood mode for a group, it sets the F flag and floods all packets it receives for the group, disregarding whether a port has been pruned or not. The switch leaves flood mode when it sees a new, higher generation number, which indicates that the root has begun the process of building a new tree. Being in flood mode has two effects. Firstly, it makes sure that packets for the group continue to be delivered. Secondly, when the root switch sees any packet with the F bit set and the current generation number, it recognizes this as meaning that some switch needs the tree to be reset. The root switch then initiates this by incrementing the generation number. While any packet can be used to reset the tree (by setting F), there are times when the tree should be rebuilt but no packet is immediately available (for example, when a new switch is plugged into the network). It is for these cases that the RESET message exists: they can be used to initiate a reset without needing to wait for a packet from the group to arrive (which, if the network is stable, may not be for some time).

For safety, the root switch periodically floods multicast packets even in the absence of any other indications to do so. This bounds the time that the group suffers with a bad tree if another switch is trying to reset the tree and the flood (or RESET) packets are being lost elsewhere in the network.

2.4 Evaluation

In this section, we evaluate AXE via ns-3 [90] simulations. We ask the following questions: (i) How well does AXE perform on a static network? (ii) How well does AXE perform in the presence of failures? (iii) How well does AXE cope with host migrations? (iv) How many entries are required for the deduplication filter? (v) How well does AXE recover from severe overloads? and (vi) How well does multicast work?

For some of these questions, we compare AXE to "Idealized Routing" which responds to network failures by installing random shortest paths for each destination after a specified delay. The delay is an attempt to simulate the impact of the convergence times which arise in various routing algorithms without having to implement, configure (in terms of the many constants that determine the convergence behavior), and then simulate each algorithm. Note that the time to actually compute the paths is not included in the simulated time – only the arbitrary and adjustable delay. The fact that we compute a separate and random shortest-path tree for each destination is significant: a more straightforward algorithm would overlap paths significantly and not spread traffic across the network (especially in the fat tree scenario described below). This approach is not as good as ECMP, but is certainly better than a non-random approach.

We do not compare directly to spanning tree for two reasons. In terms of effectively using links, spanning tree's limitations are clear (the bisection bandwidth is that of a single link) and, as we will show, AXE is essentially as good as Idealized Routing (where the bisection bandwidth depends in detail on the network topology and link speeds). In terms of failure recovery, spanning tree is strictly worse than Idealized Routing (in that failures in spanning trees impact more flows). Thus, we view Idealized Routing as a more worthy target, providing more ambitious benchmarks.

2.4.1 Simulation Scenarios

We perform minute-long simulations in two different scenarios – one is a fat tree [2] with 128 hosts as might be used in a compute cluster, and the other is based on the Berkeley campus topology. For the former, we assume that links have small propagation delay (0.3μs). For the latter, we assume somewhat longer propagation delays (3.5μs). As we do not have specific host information for the campus topology (and it is likely to be fairly dynamic due to wireless users), we simply assign approximately 2,000 hosts to switches at random. While we would have liked to include more hosts, we limited the number in order to make simulation times manageable *for Idealized Routing* – neither our global path computation nor ns-3's IP forwarding table is optimized for large numbers of unaggregated hosts.

For each topology, we evaluate a UDP traffic load and a TCP traffic load. Although large amounts of UDP may be rare in the wild, using it as a test case helps isolate network properties (whether AXE or Idealized Routing) from the confounding aspects of TCP congestion control with its feedback loop and retransmissions. Our UDP sources merely send max-size packets at a fixed rate. For each UDP packet received, the receiver sends back a short "acknowledgment" packet to create two-way traffic (which is important in any learning scenario). For TCP traffic, we choose flow sizes from an empirical distribution [13]. In terms of UDP sending rates, in the cluster case we use a per-host rate of 100Mbps. In the campus case, we use a per-host rate of 1Mbps. For TCP, we pick the flow arrival rate so as to roughly match the UDP per-host sending rates. We ran simulations using both 1Gbps and 10Gbps links, and we omit the 10Gbps results, which were (unsurprisingly) slightly better.

We generate traffic somewhat differently for each topology. For the cluster case, we model significant "east-west" traffic by choosing half of the hosts at random as senders, and assigning each sender an independent set of hosts as receivers (each set equaling one half of the total hosts). For the campus topology, we believe traffic is concentrated at a small number of Internet gateways and on-campus servers, so all hosts share the same set of about twenty receivers.

2.4.2 Static Networks

Here we show no graphs, but merely summarize the results of our simulations. In terms of setting up routes in static networks, the unipath version of AXE produced shortest path routes equivalent to Idealized Routing in both topologies, and in the cluster topology the multipath version of AXE produced multiple paths that were equivalent to an ECMP-enabled version of Idealized Routing. This is clearly superior to spanning tree, but no better than what typical L3 routing algorithms can do (and L2 protocols like SPB and TRILL that use similar routing algorithms). Thus, AXE is able to produce the desired paths.

2.4.3 Performance with Link Failures

To characterize the behavior of AXE in a network undergoing failures, we "warm up" the simulation for several seconds and then proceed with one minute of failing and recovering links using a randomized failure model based on the "Individual Link Failures" in [75] but scaled to considerably higher failure rates. These failure rates represent extremely poor conditions: 24 failures over one minute for the cluster case and 193 failures over one minute for the campus case.

For simulations using UDP traffic, we looked at the number of undelivered packets, which is shown in Figure 2.3. In the cluster case, AXE incurs *zero* delivery failures, while Idealized Routing incurs increasingly many as the routing delay grows. In the campus case, the high failure rate and the smaller number of redundant paths leads to network partitions, and all packets sent to disconnected destinations are necessarily lost. We ignore these packets in our graph, showing only the "unnecessary" losses (packets sent to connected destinations but which routing could not deliver). We see that AXE suffers a small number of "unnecessary" losses (24), while Idealized Routing has significantly more even when the convergence delay is 0.5ms. AXE's few losses are due to overload: AXE has established valid forwarding state, but the paths simply do not have enough capacity to carry the entire load (since we were not running AXE with ECMP turned on in this experiment, Idealized Routing – which always randomizes per-destination routing – does a better job of spreading the load across all shortest paths, and so does not suffer these losses). Running this same experiment with higher capacity links, AXE achieves zero unnecessary losses.

We performed a similar experiment using TCP. However, as TCP recovers losses through retransmissions, we instead measure the impact of routing on flow completion time. We find that

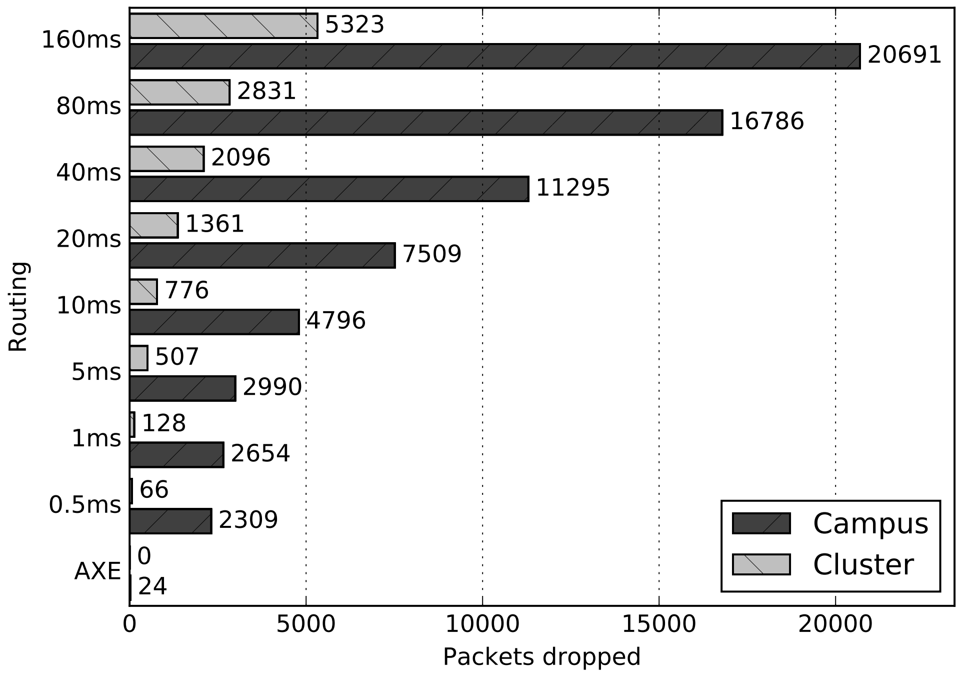


Figure 2.3: *Comparison of unnecessary drops for AXE versus Idealized Routing with various convergence times.*

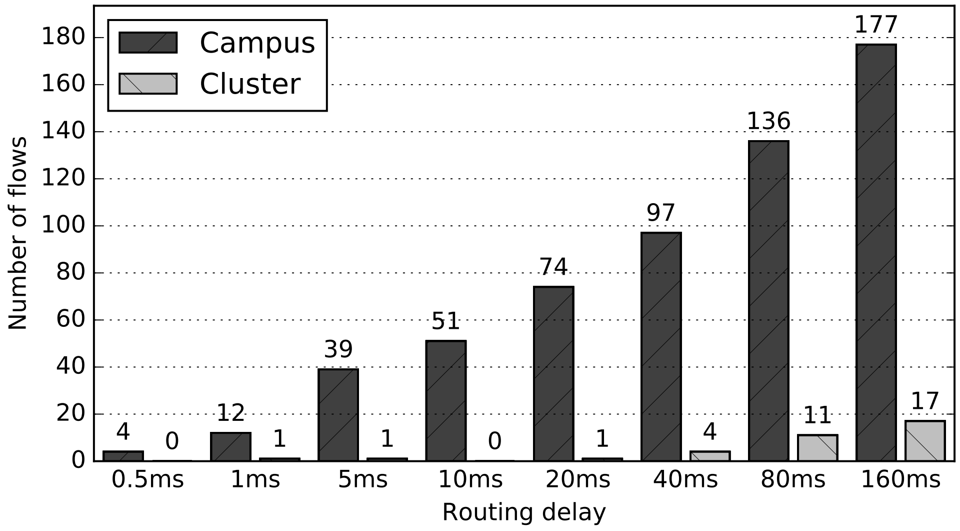


Figure 2.4: *Number of flows where Idealized Routing suffers significantly higher FCT delay than AXE.*

when comparing FCTs under AXE and Idealized Routing, either they are very close, or Idealized Routing is significantly worse due to TCP timeouts. Figure 2.4 shows the number of flows which appear to have suffered TCP timeouts which AXE did not (*i.e.,* have FCTs which are at least two seconds longer); there are *no* cases where the reverse is true.

2.4.4 Performance with Host Migrations

Migration of hosts (*e.g.,* moving a VM from one server to another or a laptop moving from one wireless access point to another) is another condition that requires re-establishing routes. To see how well AXE copes with migration, we run similar experiments to those in the previous section, but migrating hosts at random and with no link failures. Figure 2.5 shows the results of this experiment for various different rates of migration. We find that the increase in total traffic is minimal – even at the ridiculously high migration rates of each host migrating at an average rate of once per minute, the increase in traffic is under 0.44% and 0.02% for the campus and cluster topologies respectively. Note this overhead is just the overhead from AXE flooding and a gratuitous ARP; we did not model, for example, the traffic actually taken to do a virtual machine migration (though at migration rates as high as we have simulated, we would expect the AXE flooding to be vanishingly small in comparison!).

2.4.5 Deduplication Filter Size

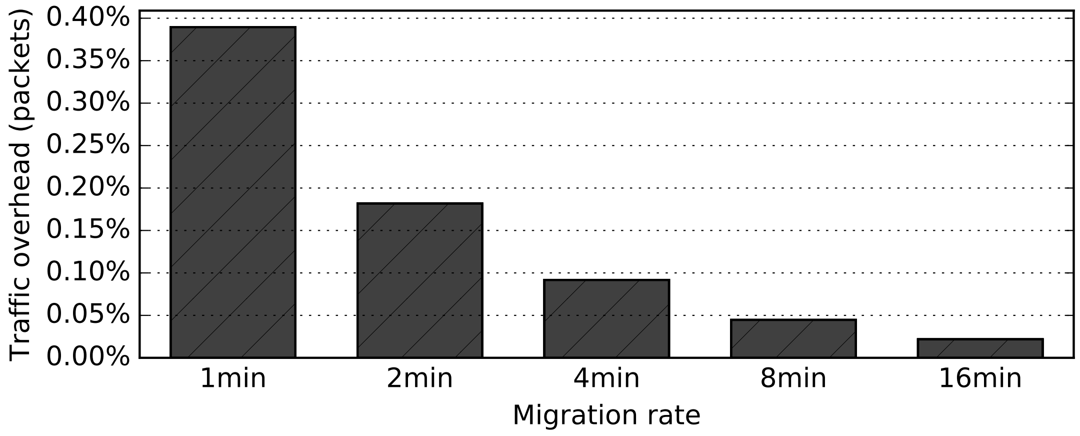
Deduplication using our filter method is subject to false negatives – it may sometimes fail to detect a duplicate. When this happens occasionally, it presents little problem: duplicates are generally detected on neighboring switches, at the same switch the next time it cycles around, or – in the worst case – when they reach the maximum hopcount and are dropped. However, persistent failure to detect duplicates runs the risk of creating a positive feedback loop: the failure to detect duplicates leads to more packets, which further decreases the chance of detecting duplicates.

The false negative rate of the filter is inversely correlated with the filter size, so it is important to run with filter sizes big enough to avoid melting down due to false negatives. To see how large the filter size should be, we ran simulations using filters with sizes ranging between 50 and 1,600 entries. Our simulations were a worst case, as we used the UDP traffic model (which, unlike TCP, does not back off when the network efficiency begins degrading).

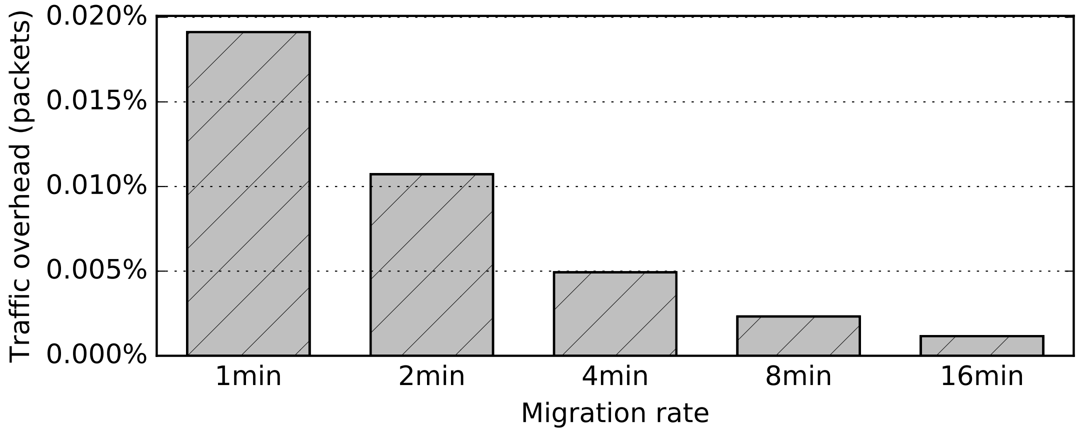
Figure 2.6 shows the number of lost packets (which we use as evidence of harm caused by false negatives) for the cluster network with 1Gbit links. Even under heavy failures, the number of losses goes to zero with very modest sized filters ($\approx$500 – or even fewer for 10Gbit links). We omit the largely similar results for the campus topology.

2.4.6 Behavior under Load

Any learning network can be driven into the ground when faced with a severe overload. Because such overloads cannot always be prevented, it is crucial that the network recovers once the



(a) *Campus topology*



(b) *Cluster topology*

Figure 2.5: *Overhead for host migrations. On average, every host migrates once per the time interval shown on the X axis. The Y axis shows the increase in total traffic.*

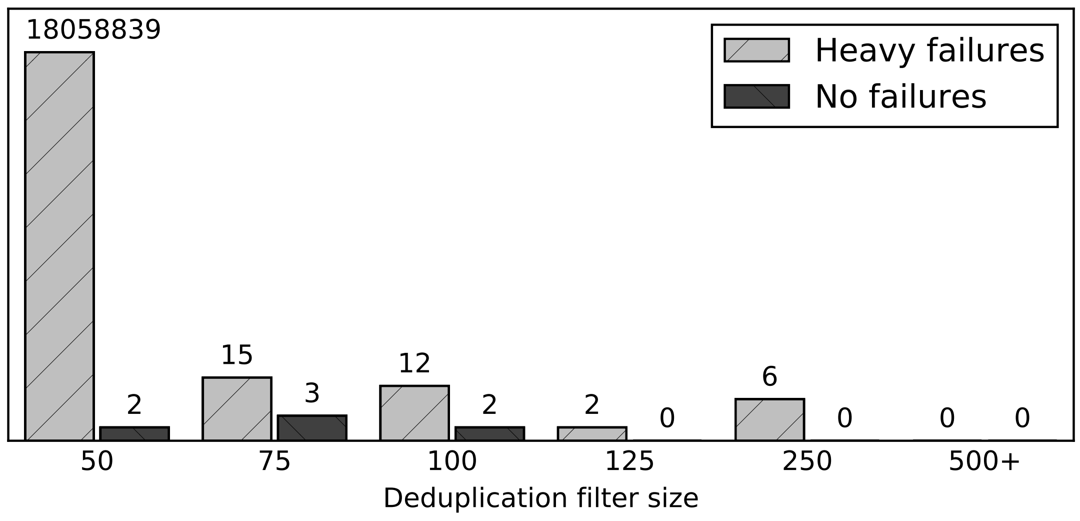


Figure 2.6: *Effect of deduplication filter size on UDP traffic in the cluster with 1 Gbps links.*

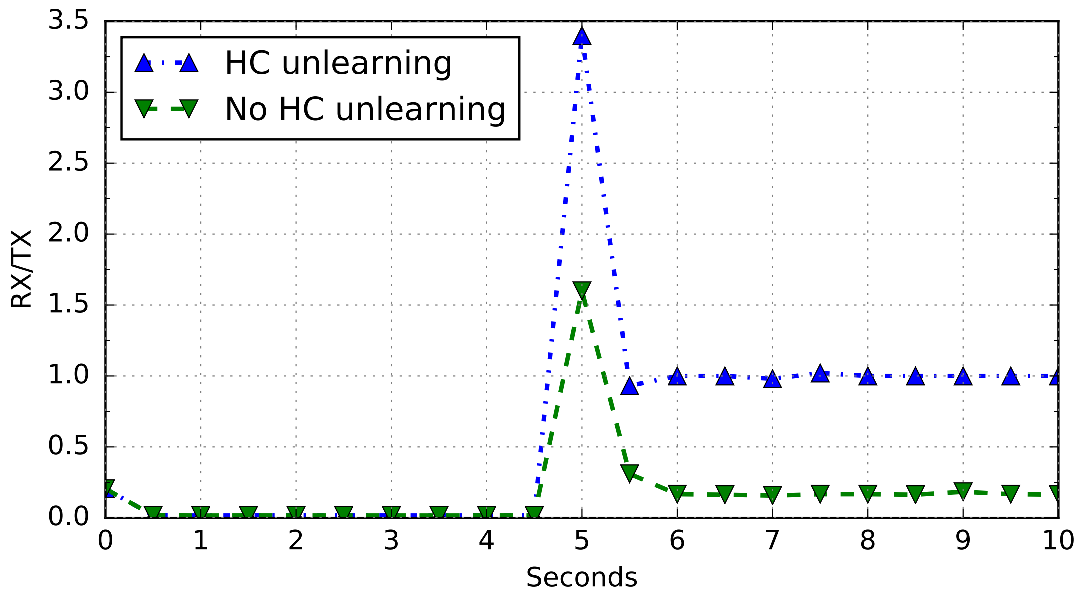


Figure 2.7: *The ratio of received to transmitted packets in an experiment for which the first half is over-driven.*

problematic condition is resolved. This property follows from our design, but to verify it experimentally, we ran a simulation on the cluster topology with highly randomized traffic and a severely undersized deduplication filter. We noted the number of packets transmitted from sending hosts and the number of packets received by receiving hosts in half-second intervals, and the Y axis shows the latter divided by the former: RX/TX. Ideally one would want RX/TX to be 1, and values less than this indicate degradation of network performance. The results are shown in Figure 2.7.

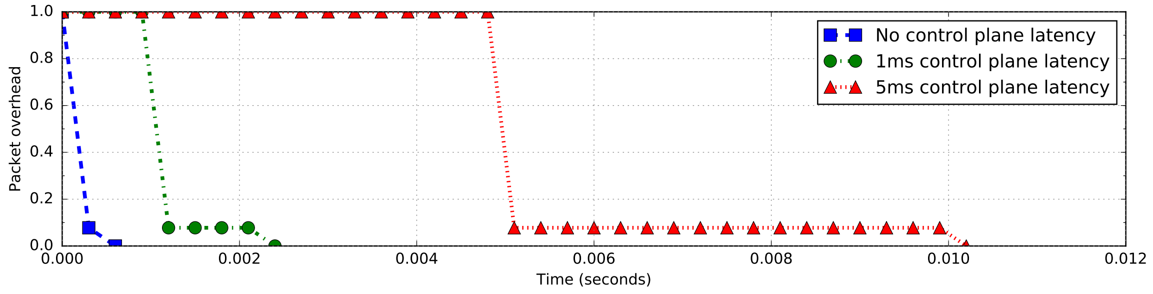
For the first five seconds of the simulation, we generate a large amount of traffic (far more than the links and deduplication filter can accommodate). This drives AXE into a useless state where packets are flooding, dropping, being delayed, and are not reliably deduplicated or learned from. Indeed, the fraction of this traffic that is delivered successfully is negligible. At five seconds, we reduce the traffic to a manageable level. We see that following a spike in delivery (as the queues that built up in the first half of the experiment drain), AXE quickly reaches the expected equilibrium.

We also verified that one of AXE's safety mechanisms has the expected effect. Specifically, when the hopcount reaches its limit for a non-flood packet, the final switch removes (unlearns) state for the packet's destination. In this way, any future packets to that destination will not follow the looping path, but will instead find themselves with no forwarding state and be flooded. To witness this in action, we disabled AXE's hopcount expiration unlearning. As seen in Figure 2.7, compared to the actual algorithm, this results in a dramatically worse RX/TX ratio in the second half of the experiment, because bad (looping) paths established in the first part of the experiment are never repaired.

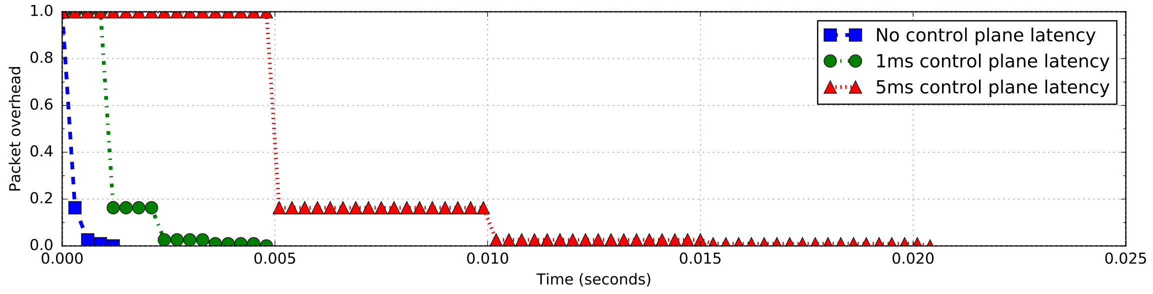
2.4.7 Multicast

While traditional approaches to multicast maintain trees through incremental prunes and grafts, our AXE multicast design rebuilds the tree from scratch every time there is a change. Rebuilding a tree requires flooding packets and then letting the tree be pruned back. Whether this approach is reasonable or not depends on whether periodically switching back to flooding is overly wasteful. While networks can clearly withstand the occasional flooded packet (broadcasts for service discovery, and so on), the danger with multicast is that rebuilding the tree during a high volume multicast transmission (such as a video stream) may result in a large number of packets being flooded. To examine this case, we simulated a transmission of data at 40Mbps (a rather extreme case – equivalent to a high-quality 1080p Blu-ray Disc) and examined the convergence of AXE's multicast after triggering a reset of the multicast tree at $t = 0$. We repeated the experiment several times on both topologies and with group membership between 5% and 40%, and show a representative example of a 5% run in Figure 2.8. The graphs show the overhead of extra traffic, where a value of 1 indicates sending as much traffic as a flood, and an overhead of 0 indicates sending as much traffic as a shortest-path tree.

In the AXE multicast algorithm, the data plane recognizes that a PRUNE should be sent, but the control plane is ultimately responsible for creating the PRUNE message. Interactions between the control and data planes, however, are not especially fast in terms of packet timescales. For example, OpenFlow switches have had latencies measured from 1.72ms up to 8.33ms for expensive



***(a)** Campus topology*



***(b)** Cluster topology*

***Figure 2.8:** Convergence of a multicast tree using AXE on the campus (a) and cluster (b); marks are packets.*

table insertions [47]. As AXE's multicast control plane running directly on the switch would avoid OpenFlow-specific overheads and requires no expensive table update, we modeled 0ms, 1ms and 5ms data-to-control plane latencies, where 0ms represents an unachievable ideal and 5ms represents a reasonable outside estimate. We find that even in the worst case, AXE converges quickly: the overhead has dropped to less than 20% by about 5ms and is either converged (on the campus topology) or negligible (on the cluster topology) by about 10ms – and even at the high rate of 40 Mbps, only 34 packets are sent during this 10ms.

It is worth noting that the convergence time is almost entirely dictated by the control plane latency: with no control plane latency, in the two experiments we show, the trees have converged by the time the third or fifth packet is sent. Indeed, for experiments we ran with a larger number of group members, it had often converged by the second packet. That it converges faster with more members may initially seem surprising, but is a result of the fact that the tree is larger when there are more members, and it therefore takes fewer prunes to converge to it.

2.5 Related Work

Since the introduction of Ethernet, there have been efforts to improve upon it, both in terms of its operational model and its performance. AXE represents a new direction in this body of literature, providing many advantages traditionally associated with routing without sacrificing the simple plug-and-play nature of traditional MAC learning.

There have been a wide variety of enhancements to Spanning Tree Protocol. Rapid Spanning Tree [51] improves upon STP with faster link and failure recoveries. Additionally, there have been numerous enhancements that build multiple spanning trees in a single network to improve utilization and redundancy. These include MSTP, MISTP [53], PVST, PVST+ [52], and others. AXE dispenses with spanning trees altogether, allowing it to achieve instantaneous failure recovery and achieve short paths that utilize all links.

In data center networks, one trend that addresses many concerns surrounding STP is to abandon it and the flood-and-learn approach entirely, instead running IP all the way to the host [2, 41]. This approach performs well (similar to the Idealized Routing we evaluated against in Section 2.4), though not as well as AXE in terms of how quickly it recovers from failures. Work along the same lines has used highly specialized Clos topologies coupled with an SDN control plane to achieve truly impressive results in a data center context [112]. We note that none of these techniques achieve AXE's plug-and-play functionality, and all of them require specially designed network topologies. AXE, on the other hand, works on arbitrary topologies without issue, making it ideally suited for less uniform environments.

Also in the data center context, F10 [73] achieves impressive results by co-designing the topology, routing, and failure detection. While it shares one of AXE's primary goals (fast failure response), the highly-coupled approach is starkly different. VL2 [41] maintains L2 semantics for hosts, but does so using a directory service rather than flood-and-learn, and, again, is designed with particular topologies in mind.

Recently there has been interest in bringing routing techniques traditionally associated with L3 to L2 through technologies like SPB [50] and TRILL [97]. These protocols use link state routing to compute shortest paths between edge switches, thus inheriting the advantages (and limitations) of L3 routing protocols.

In some ways, AXE is similar to Dynamic Source Routing [60], 802.2 Source Route Bridging [54], and AntNet [32]. These schemes all send special "explorer packets" looking for destinations, collecting a list of switches they have passed through. Upon reaching the destination, a report is sent back to the source which can use the accumulated list of switches to build a path. AXE differs most fundamentally in that it does not use or create any special routing messages – everything needed to establish routes is contained in the AXE packet header; rather than have the destination explicitly return a special routing packet, it relies on the destination to create a packet in the return direction in the normal course of communication (*e.g.,* a TCP ACK). This difference also applies to failure recovery: while DSR has another special type of routing message to convey failure information and induce the source to "explore" for new routes again, this too takes place with plain data packets in AXE. An outcome of all this is that AXE is simultaneously routing *and* delivering user data – there is no distinction (or latency) between them. A further difference is that AXE does not keep work-in-progress routing state in special packets, and instead uses per-switch learning, requiring an alternate loop-prevention strategy. While an SRB or an AntNet switch can identify a looped packet because its own identifier already exists in the packet's list of hops, AXE packets contain no such list; thus, the switch must "remember" having seen the packet.

Like AXE, Failure Carrying Packets (FCP) [68] minimizes convergence times by piggybacking information about failures in data packets. In FCP, each switch has a (more-or-less) accurate map of the network, and each packet contains information about failed links in the map that it has encountered. This is sufficient for an FCP switch to forward the packet to the destination if any working path is available. AXE has no such network map and so its strategy for failed packets is simply to flood the packet and remove faulty state so that it can be rebuilt (via learning).

Data-Driven Connectivity (DDC) [72] also uses data packets to minimize convergence times during failure. It uses a global view of the network to construct a DAG for each destination. When failures occur, packets are sent along the DAG in the wrong direction, which signals a failure to the receiving switch which can then (via flipping the directions of adjacent edges) find a new path to the destination. Thus, FCP, DDC, and AXE all use packets to signal failures, but whereas FCP and AXE use actual bits in the header, the signaling in DDC is implicit. Unlike AXE, which generally builds shortest paths, paths in DDC may end up arbitrarily long. And similar to FCP and dissimilar to AXE, it relies on some sort of control plane to build a global view of the network.

The development of All-Path Bridging [103] overlapped with that of AXE, and there is significant crossover in terms of motivation and observations. Indeed, the same basic technique underlying APB was independently developed and explored in the early development of AXE. Ultimately, we set that approach aside because it fundamentally cannot achieve the same near-instantaneous response to topology change in all cases that the approach we pursued further (and have described in this chapter) can.

Despite this large set of related efforts, *none* combine all of AXE's features: plug-and-play, the ability to work on arbitrary topologies, and near-instantaneous recovery from failures (*i.e.,* limited only by failure detection time and the time required for the switch to change forwarding behavior – *not* by network size, communication with other switches, etc.). Thus, we see AXE as occupying a useful and unique niche in the networking ecosystem.

2.6 Conclusion

Ultimately, AXE is intended as a general-purpose replacement for off-the-shelf Ethernet, providing essentially instantaneous failure recovery, unicast that makes efficient use of bandwidth (not just short paths, but also allows ECMP-like behavior), and direct multicast support – while retaining Ethernet's plug-and-play characteristics and topology agnosticism. We are not aware of any other design that strikes this balance. While we do not see AXE as a contender for special-purpose high-performance data center environments (where plug-and-play is largely irrelevant), we see it as a promising alternative to today's designs for many other cases.

Chapter 3

Augmenting Routing for Resiliency

Any network operator will tell you that availability is important, and that one of the important factors in availability is how fast the network can resume delivering packets after a failure. Clearly, a network which recovers faster is preferable to one which recovers more slowly, all other things being equal. In the real world, however, all other things are rarely equal. Operators must balance improvements in recovery time against increased operational costs for faster (but more complex) solutions, the added uncertainty of new technologies (which may introduce new failure modes), and other factors. Such a balance must be found with an organizationally-specific calculus. Organizations that live or die on the performance of their network rightly dedicate significant resources into eking out all possible uptime. For a larger group of organizations, further minimizing their network recovery falls into the "nice to have" category behind other concerns; it is not that they would not benefit from it, nor that it is technically impossible to achieve. It is that there is simply not enough benefit to justify the deployment of sophisticated solutions that involve additional operational risk and expertise. Indeed, even the approach provided in the previous chapter may be unsuitable. While it is designed as a "drop in" replacement for traditional Ethernet that achieves much higher availability, deploying it still may be prohibitive when availability is only a "nice to have" – not an absolute requirement – since it requires replacing all the switches in a network and may require operators to become accustomed to new behavior. Moreover, it is a replacement for layer 2 Ethernet-plus-STP type networks, and as noted, modern networks increasingly use layer 3 IP routing at each hop.

Thus, this chapter proposes a simple L3 routing recovery mechanism intended to cater to the mass of organizations which use IP extensively and for whom better recovery behavior is desirable, but not a "must have". The key factor here is that it be minimally disruptive in every sense. A solution should require no additional expertise, incur little risk, not demand configuration nor a "forklift" upgrade of all the routers in a network, and – in general – shake things up as little as possible.

3.1 Approach

Of course, better availability is not a new goal for networks, whether inside an enterprise or spread across an AS. Early routing protocols (and here we consider only routing at L3, not at L2) and their implementations took seconds or more to converge (*e.g.,* due to the one second timer granularity in OSPF v2 [86]), and during this reconvergence were not able to deliver packets that were supposed to traverse the failed link. However, these long outage periods soon became unacceptable, so three approaches to improving convergence were pursued.

The first of these approaches was to merely reduce the various timers in routing protocols. However, timers can only be reduced so far, as such reductions increase network traffic and router computation (the latter of which can cause CPU contention resulting in other important operations being delayed). In addition, this eliminates much of the stability achieved via hysteresis and damping that some timers were intended to provide (such as mitigating route flaps and coalescing related messages). More fundamentally, as we discuss later, multiple factors go into the reconvergence delay, and not all of them disappear just by tuning timers.

Second, to make further progress, some networks use pre-computed backup paths that can be switched to immediately upon failure. This requires a level of oversight that is appropriate for carefully managed networks (*e.g.,* ISPs), but is probably impractical for most enterprise networks.

Third, others have turned to designs such as Segment Routing for IPv6 (SRv6) [36], Loop Free Alternates (LFA) [10], or MPLS link/node protection [93], but these introduce additional complications, such as using IPv6 or MPLS in the data plane, adding additional control plane mechanisms (MPLS and SRv6), or only handling a subset of failures (LFA). Additional designs exist which can generate a full set of backup paths, but these remain largely untested in practice and require an entirely different routing scheme [72, 71] or highly structured networks [73].

Given the drawbacks of these previous approaches, the question is whether there is an approach that improves convergence time while introducing minimal additional complexity to the network and being realistically implementable. The approach introduced in this chapter is one such possibility, called Ark, that is inspired by the work in chapter 2. The basic conceptual approach follows now-familiar lines: when a packet arrives at a router where the next-hop link (as dictated by the current routing table) is down, then the packet is flooded. If the network remains connected – and uncongested – then at least one copy of the packet should arrive at its destination.

The trick is to make this flooding operation safe, in the sense of (i) not causing loops (which can cause exponential packet replication) and (ii) not causing congestion for the non-flooded traffic. Ark achieves safe flooding largely through two simple measures. The first of these is much the same as in the previous chapter: looping is avoided through probabilistic duplicate suppression at each router (with TTLs backstopping any probabilistic failures). The second aspect is new: the flooded packets are sent at low priority, so they do not interfere with regular traffic. Thus, Ark leaves the basic routing algorithm unchanged, and normal traffic almost entirely unperturbed (it would be completely unperturbed if routers supported preemptive priority forwarding). Ark merely attempts, through safe flooding, to deliver packets (that otherwise would have been dropped) while the routing algorithm reconverges. Note that the use of priority here is inverted relative to that in chapter 2. In that work, floods were sent at high priority in order to speed the learning of new paths.

Here, the establishment of new paths is independent of the flooded packets (it is the responsibility of the unaltered routing protocol), and our goal for flooding is simply to deliver as many additional packets as we can without affecting any other non-flood traffic.

Since flooding occurs immediately upon a packet hitting a failed link, and since the first arriving flooded packet will have taken the lowest delay path, there is no significant increase in latency. Thus, the performance metric of interest here is the number of packets not delivered during reconvergence, which we strive to minimize.

To be clear, we recognize that there are many special-purpose solutions that can achieve high availability, and they are in widespread use in hyperscaled data centers and ISPs. In contrast, our simple design is intended for enterprise networks that do not have a large operational staff to implement backup paths, nor want to modify the normal operation of their network (by deploying MPLS or SRv6), nor want to change the timers significantly (which may increase traffic and computation, or lead to instabilities), but do still care about improving availability. We think this category – which includes a wide range of businesses and academic institutions – probably comprises the vast majority of operational networks in the world. We are not aware of any other low-risk and low-complexity proposals for achieving significantly higher availability in such networks, which typically run some relatively standard interior IP routing protocol, most likely a link-state protocol such as OSPF or IS-IS.

3.2 Reconvergence Delays

While Ark applies to any intradomain routing protocol, we will compare it to a link-state protocol in order to place its performance in context. For a link-state algorithm, the time between when a failure occurs and communication is fully restored is comprised of several delays. Primarily, these are: **detection** of the failure, **generating** a link update, **flooding** that update throughout the network, **recomputing** the route upon receipt of the update, and **updating** the routing table (and then the forwarding table) once the routes have been recomputed.[1] In an ideal world, all of these delays are zero, but this is not possible in practice.

The detection delay – in many ways the most fundamental, as no recovery will be attempted until the failure is detected – was once on the order of seconds, but with more modern techniques can be considerably lower. As discussed in the previous chapter (at the end of Section 2.1), we are given to believe that 1ms detection is a reasonable near-term goal, and we use this estimate in our analysis.

The other delays are highly implementation-specific, but are generally hard to minimize further (*e.g.,* route computations are already highly optimized, link-state flooding must be handled by the router's CPU and is bound by propagation delays, and FIB updates are stubbornly slow). Moreover, routing algorithms typically induce delays in the form of timers. As mentioned above, reducing these timers can create problems for convergence and communication or computational load. Indeed, a recent RFC [30] produced by major routing vendors standardizes an algorithm for computing dynamic timers for link-state routing algorithms, and specifies a minimum recomputing

[1]For a more complete discussion of routing delays, see [37, 69, 1].

delay timer of 50ms (and we expect most operators of the networks we are targeting will use this default setting rather invest time and risk negative consequences attempting to tune it).

Thus, rather than suggesting that existing techniques simply be made faster and timer intervals simply be set smaller, Ark is designed to run in concert with an existing routing protocol configured to favor stability (*i.e.,* with conservative timers). As we show in our evaluation, preliminary results indicate that when appropriately configured, *Ark can achieve loss rates comparable with an optimal recovery strategy*. Ark is notable not because it can perform so well, but that it can do so without adding much risk or complexity.

3.3 Design

Ark leaves the existing routing control plane entirely unchanged, only altering the data plane to provide alternate behavior in cases when a packet is destined for an outgoing link which has failed (so the packet would otherwise be dropped). We sketch a simple version of the Ark data plane in Algorithm 3.1. Under non-failure conditions, this is much like the algorithm implemented by the data plane in a conventional router: the TTL is decremented and checked (line 1), the packet's destination address is looked up in a table (line 2), and the packet is sent over the link specified by the retrieved entry (line 23). When the link associated with the looked-up route has failed, the packet is flooded. Ark handles these flooded packets at all subsequent routers to ensure that the flooding is safe.

In more detail, we can divide the lifetime of a packet which has encountered a failure into three phases:

1. *Transitioning*: When an ordinary IP packet first encounters a failure, we say that it *transitions* from being an ordinary IP packet to being an Ark flooded packet (practically speaking, this entails altering the header; we discuss this further in Section 3.4.1).
2. *Flooding*: Ark flooded packets are flooded (*i.e.,* duplicated and sent out multiple ports) throughout the network by other routers they encounter using a safe flooding algorithm described below.
3. *Delivering*: When an Ark flooded packet arrives at some router local to its destination (*i.e.,* the packet matches a route for a directly attached subnet), the packet ceases flooding and is sent out the appropriate port.

The additions to basic IP forwarding in Algorithm 3.1 correspond to these different phases. Note that the condition on line 18 would, for plain IP forwarding, simply result in dropping the packet. In Ark, it leads to *transitioning* and beginning the process of *flooding*. Once a packet has transitioned, subsequent routers that receive it will continue to flood it (line 14) subject to a constraint of the safe flooding algorithm (line 7; described in more detail below), or – if it has reached its last hop – *deliver* it (line 11).

Note that the flooding behavior in the flooding phase (line 14) differs from the flooding behavior in the transitioning phase (line 21). While there is no purpose in sending a packet back to a node that already flooded it, when a packet first encounters a failure, the previous node *had not* flooded it – and it is possible that the only remaining path is back the way the packet came (requiring it to make a "hairpin turn").

```
1:  Decrement TTL; Drop packet if expired
2:  route ← route_lookup(packet.destination)
3:  if route is null then
4:      Drop packet
5:  else if packet has Ark header then
6:      ▷ Packet encountered a failure and is flooding
7:      if router has seen packet before then
8:          Drop packet
9:      else if route is local subnet and port not failed then
10:         ▷ Deliver
11:         Remove Ark header
12:         Forward packet according to route
13:     else
14:         Flood packet out all ports but ingress port
15:     end if
16: else
17:     ▷ Packet is a plain IP packet
18:     if failure on port specified by route then
19:         ▷ Need to transition
20:         Add Ark header
21:         Flood packet out all ports including ingress port
22:     else
23:         Forward packet according to route
24:     end if
25: end if
```

Algorithm 3.1: *The Ark algorithm.*

As mentioned in the introduction of this chapter, for packets that would otherwise have been dropped (due to a failure), Ark utilizes safe flooding which involves (i) preventing loops via probabilistic deduplication and (ii) sending the packets at lower priority than regular data packets. For the deduplication, we reuse the basic idea underlying the probabilistic duplicate packet detection mechanism discussed in chapter 2. Before flooding, packets are stamped with a nonce – a unique value which allows packets to be uniquely identified. As in the previous chapter, this is a sequence number in our implementation (and we discuss the issue of wrapping again below). The combination of the nonce and a unique ID for the router which did the stamping should be unique for as long as the packet can be in the network. Routers keep a simple and efficient data structure (a *deduplication filter*) to track and discard packets they have already seen: they hash the ID and the nonce (along with a per-router salt value to prevent identical collisions on many routers) to determine a table slot. If the ID and nonce in the slot match those in the packet, the packet is a duplicate and is dropped. Otherwise, the values in the slot are updated with those in the current packet.

This mechanism may fail to detect a duplicate if the original entry has been "bumped" from the filter (by a different packet which just happens to hash to the same slot) by the time the duplicate arrives. In this case, the packet will be flooded when it should be dropped. In the worst case, when all the routers in a loop have failed to detect the duplicate, the IP TTL field eventually ends the cycle. The use of a lower priority for flood packets prevents them from impacting normal traffic in any case. While it is conceivable that failed deduplication may lead to delivering a packet to a host more than once, we assume that this is already accommodated by transport and application layer protocols.

This mechanism may also mistakenly identify a non-duplicate packet as a duplicate due to the transitioning router running out of unique nonce values (*e.g.,* the underlying sequence number wrapping). We address this in Section 3.4.1, but note that even if one did nothing to address it, the worst case here is that an occasional packet is dropped – a packet which would have been dropped anyway in the absence of Ark.

3.4 Practical Issues

We now dig into a few details we have so far ignored.

3.4.1 Partial Deployment and Packet Details

Packets flooded by Ark must be distinguishable from ordinary packets and, as described in Section 3.3, they must contain a nonce and a unique ID for the router which first flooded them. To that end, these packets are sent using a custom value for the EtherType field and have a small Ark-specific header before the IP header. Rather than include an ID within the Ark header itself, we reuse the source Ethernet address field to serve this purpose – when a router transitions a packet, it will insert its own Ethernet address (this is, of course, typical for IP packet processing anyway; the only difference is that *subsequent* routers flooding the packet should leave the source Ethernet address untouched). Thus, the Ark header need only contain the nonce.

The primary question with regard to the nonce is how many bits it must be to avoid duplicate values within the lifetime of a packet. Chapter 2 poses a similar question, and suggests that 24 bits of a larger 32 bit header be used, as this is more than adequate for 10Gbps networks. The other fields of that header are not required in Ark, however: the flags are unnecessary and the IP TTL can be used instead of a separate hopcount field. Thus, we could maintain the same 32 bit overhead, but allocate it entirely to the nonce/sequence number. This allows enough unique values to avoid producing duplicates for around 22 seconds even with minimum-sized packets produced at 100Gbps. This seems likely to prove sufficient for the near term, especially as a packet's lifetime is bounded by its TTL (assuming 64 hops maximum, a packet would need to incur an average delay of over 300ms per hop). This also provides a bound for the lifetime of a particular filter entry: any entry older than 22 seconds may refer to a previous packet with the same nonce, and should therefore be ignored. A reasonable approximation can be achieved by simply clearing the filter every 22 seconds.

Additionally, we propose Ark packets use a registered multicast Ethernet destination address. In combination with the custom EtherType, this facilitates interoperation with existing equipment. Ark-unaware IP routers will simply drop the packets because they do not appear to be IP. Ark-unaware Ethernet switches simply see a multicast packet of an unknown type and flood it (along their spanning tree if it exists).[2] These are safe behaviors which allow for incremental roll-out of Ark with it enabled by default: when partially deployed, Ark floods reach across L2 domains and contiguous Ark-aware routers (providing improved behavior in these regions), and are dropped by Ark-unaware routers.

As a final comment on backwards compatibility, Ark flood packets should include an "802.1p" [52] header. Such headers are supported by commodity switches and allow the encoding of a class of service. In this way, flood packets can be tagged as low priority (and queued accordingly) even on Ark-unaware L2 hardware.

3.4.2 Implementability

Fundamentally, Ark does not do anything extremely unusual for a router or switch. It adjusts TTLs and forwards or floods packets (as in any router or switch), utilizes priority-scheduled queues (as in many switches with QoS/CoS support), and hashes packet headers (similar to any switch supporting ECMP). The only novel requirements for Ark are its additional header and the deduplication filter. A vendor could clearly extend a fixed-functionality switching ASIC design to handle the Ark packet header in addition to the ones it handles already. For ASICs that can be programmed using P4 [61], the code required for this is trivial and could likely be applied to existing routers as a software update. The primary question is whether the deduplication filter can be implemented, and chapter 2 discusses why we are optimistic on this front.

While we believe Ark is likely feasible on both custom ASICs and fully programmable switch chips, we also wish to sketch a design wherein Ark is implemented on a router with an off-the-shelf

[2]An exception is if an existing L2 device has a security policy to drop traffic of unknown EtherTypes, in which case it – like an Ark-unaware router – will simply (and reasonably) drop the packet.

```
1:  **if** packet is coming from slow path and has Ark header **then**
2:      ▷ Slow path wanted router to flood the packet
3:      ▷ (Utilizes flood_avoid packet metadata)
4:      Flood packet out all ports except packet.flood_avoid
5:  else if packet has Ark header then
6:      ▷ Packet flooded from another router
7:      packet.flood_avoid ← ingress port
8:      Send packet to slow path
9:  else
10:     Decrement TTL; Drop packet if expired
11:     route ← route_lookup(packet destination)
12:     if route is null then
13:         Drop packet
14:     else if failure on port specified by route then
15:         ▷ Need to transition
16:         packet.flood_avoid ← none
17:         Send packet to slow path
18:     else
19:         Forward packet according to route
20:     end if
21: end if
```

Algorithm 3.2: *The split Ark Algorithm (fast path portion).*

```
1: if packet has Ark header then
2:     ▷ Packet encountered a failure and is flooding
3:     if router has seen packet before then
4:         Drop packet
5:     else
6:         route $\leftarrow$ route_lookup(packet destination)
7:         if route is local subnet and port not failed then
8:             ▷ Deliver
9:             Remove Ark header
10:            Send packet back to fast path for delivery
11:        else
12:            ▷ Continue flooding
13:            Decrement TTL; Drop packet if expired
14:            Send packet back to fast path for flooding
15:        end if
16:    end if
17: else
18:    ▷ Do transition
19:    Add Ark header
20:    Send packet back to fast path for flooding
21: end if
```

Algorithm 3.3: *The split Ark algorithm (slow path portion).*

switching chip and a conventional CPU. The pseudocode for this split design – where a "fast path" portion is implemented by a switching chip, and a "slow path" portion by the CPU – is shown in Algorithms 3.2 and 3.3. While we will not belabor the details of the split algorithm, we note that this approach requires only that: (i) the switching chip, in addition to its basic IP forwarding functions, also supports the ability to send packets to the CPU for processing (including the ingress port as metadata) when either it has a particular EtherType or the associated egress port in the forwarding table is down, (ii) the forwarding/routing table can be accessed by the CPU, and (iii) the CPU can send packets (or direct the switching chip to do so).

These required features are not exotic. They are the type of features available in proprietary switch SDKs, and they are all available in OpenFlow 1.5 [92] (which is generally implemented atop such SDKs). This design leverages the fact that the slow path need not process the entire aggregate bandwidth of the router – only the Ark flood packets. While it is still unlikely to be capable of the full performance of a version based on a custom or programmable switching ASIC (and, thus, will be somewhat less capable of preventing losses), this may be entirely acceptable for a design that can be deployed as a software upgrade on existing platforms with fixed-functionality ASICs.

Ultimately, the most problematic aspect of adapting an existing platform to support Ark may be the requirement for failure detection. As touched on in Section 3.2, mechanisms for fast detection exist today, but are not necessarily supported by all existing platforms (in particular, not by platforms that currently only implement protocols that would not greatly benefit from fast detection even were it available). However, Ark may provide sufficient incentive for implementing fast detection algorithms more widely.

3.5 Evaluation

We evaluated Ark using a discrete event simulator intended for network simulations. We begin with a general description of our experiments, and later note variations used in individual cases.

Topologies. We perform simulations on three router-level topologies, to which we add hosts. The first two are similar to the ones from the previous chapter: one is based on the UC Berkeley campus topology (39 routers, 79 inter-router links, 2024 hosts), and the other is a small fat tree network [2] as might be used in a well-connected cluster (80 routers, 256 inter-router links, 128 hosts). In addition, we evaluate a private WAN topology for which, for lack of a better model (since companies don't typically publish their internal networks), we used a version of the Abilene network which was the precursor to Internet2 [55] (11 routers, 14 links, 1100 hosts). Links in the campus and cluster networks have propagation delays of 3.5µs and 0.3µs respectively. In the Abilene network, links have propagation delays based on their rough geometric distance and the assumption that signal travels at $^2/_3$ the speed of light. In all cases, there are redundant links so that no one inter-router link failure can partition the network.

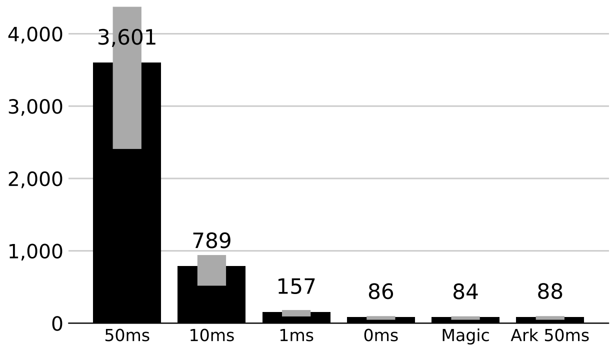
Traffic. Our simulations all use randomized traffic. For the Abilene and cluster topologies, we first pick some hosts as senders (a random sampling of about 90% and 50% of the hosts respectively). For each sender, we choose a set of receivers (a random sampling of approximately 50% of the hosts). We then performed simulations with increasing send rates to find the maximum rate sustainable without incurring drops, and we use half this rate for our standard experiments. This is likely a higher level of utilization than most enterprises operate at (making it a challenge), but is low enough that the network can handle the offered load without the failed link present.

For the campus network we used a more skewed traffic distribution, breaking traffic into two classes. Each host has some amount of *north-south* traffic which traverses one of two gateway routers which connect the campus to the Internet (and where southbound traffic is at a higher rate than northbound). Additionally, approximately 1% of hosts are selected at random to act as "servers", and approximately 40% of hosts participate in bidirectional *east-west* traffic with these servers. We ran experiments with increasing amounts of north-south traffic to find the maximum supported north-south rate. We then added east-west traffic (which naturally avoids the links that the north-south traffic overloads) until we again incurred drops. Again, we then use half these rates for our standard experiments.

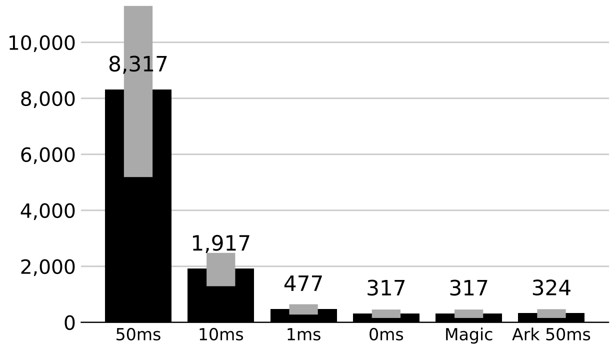
Failures. In our experiments, we are primarily concerned with the cost – in terms of undelivered packets – of a link failing. Of course, this cost may change depending on which link fails, so we individually fail each link in the topology and record the resulting losses.

In all experiments, we start sending traffic (staggering individual flow start times slightly), fail the link half a second later, recover the link one second after that, and finally stop sending another half second later. After letting the network drain, we determine how many packets never reached their destinations. As we set rates to avoid dropping due to lack of bandwidth (as described above), all such undelivered packets are due to failure either directly (*e.g.,* attempting to send over a dead link) or indirectly (*e.g.,* due to congestion caused by flooding or rerouting).

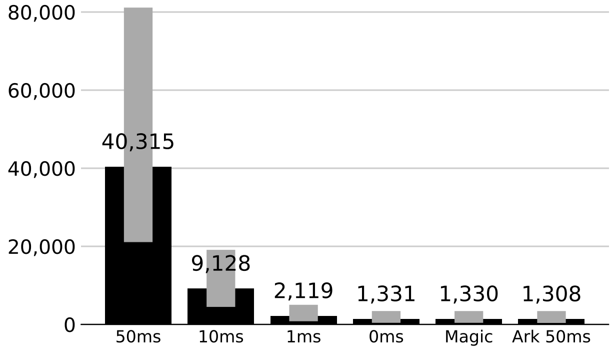
Routing. Ark is meant to augment an existing routing solution in order to provide packet delivery during periods immediately following failures. As link-state routing algorithms are well known and dominate the types of networks we believe are a good fit for Ark (at least as far as open protocols go), we evaluate Ark atop (and in comparison to) a simple link-state protocol that we implemented in our simulator based roughly on single-area OSPF. Routing updates are placed in higher priority queues than regular packets (which themselves are placed in higher priority queues than Ark flood packets), which is common practice and serves to both deliver them in a more timely manner and to prevent them from being dropped when competing against ordinary traffic. As briefly described in Section 3.2, routing is typically subject to a number of delays – both inherent and explicitly introduced – and the reasons and quantities behind these delays can vary across vendors, technologies, and operator choices. As our purpose is to evaluate Ark as a recovery mechanism rather than quibbling about OSPF tuning or particular router implementations, we use a simplified link-state model where, besides packet delivery delays, *routing is only subject to two types of simulation-induced delay*.



***(a)** Campus topology*



***(b)** Cluster topology*



***(c)** Wide area topology*

***Figure 3.1:** The number of undelivered packets following an individual link failure using routers with different routing delays and Ark. The wide, black bars are the average number of losses across all links in the respective topology. The gray bars represent the range of losses between the 10th and 90th percentiles.*

The first of these delays is the detection delay – how long between when a link fails and when the directly connected routers know about it. In all experiments we show, we assume this to be 1ms (which is optimistic but not unreasonable, as discussed in Section 3.2); increases in this delay would impact Ark and the baseline we compare it to equally.

After the detection timer expires, the routers adjacent to the failure will send a new link-state update to their neighbors. Routers immediately flood such updates to their own neighbors. Our simulator models the actual delivery process of these updates, and the packet delivery delays (from when failure is detected until update is received) of course depends on the underlying network topology.

The second delay is simply an additional time between when an update arrives (or a local failure is detected) and when the local forwarding table is updated. This delay amalgamates a variety of other delays including three listed in Section 3.2 (recomputing the routes, updating the RIB/FIB, and explicitly imposed timers) as well as general overheads (*e.g.,* communication overhead between the data and control planes). Rather than trying to model these separately, which would vary from implementation to implementation and configuration to configuration, we merely lump them all into a single artificial delay. Note that this delay is imposed *after* the update packets are forwarded, so that these delays are experienced mostly in parallel among the various routers. As mentioned earlier, a recent RFC suggests an initial induced recomputation delay of 50ms for purposes of stability; we therefore consider this number an optimistic lower bound for link-state convergence delay (that is, if one follows the RFC, 50ms is only achievable in the real world in the impossible case that all other sources of delay are zero), and we use this as the baseline for our simulated link-state algorithm.

Note that this simple link-state algorithm plays two roles in our evaluation. First, it is the routing algorithm deployed alongside Ark, and for this role we only consider the default 50ms delay value. That is, we use this algorithm as the basic underlying routing protocol, which Ark supplements to quickly recover from failures. Second, it can be seen as a competitor to Ark, and for this role we investigate the performance of the simple link-state algorithm with significantly lower delay values. These lower values are unrealistic and/or unwise, as such delays are (i) likely difficult or impossible to achieve due to various factors like FIB update rates, and (ii) quite possibly inadvisable inasmuch as they are below the purposefully imposed delay recommended by the recent RFC and the default settings of routers from major vendors. However, we use these lower values to provide some context for how well Ark performs. At the extreme, we consider two impossible cases. The first is an unachievable ideal case for a baseline link-state algorithm: a routing delay of 0ms. The second is a *magical* algorithm where routing updates are not subject to message propagation time. This represents almost the best that any approach (*e.g.,* failover routing) could possibly achieve (the only difference being that failover might allow packets to hairpin turn). In both of these cases, losses are hugely dominated by the detection delay.

Queue and Filter Sizes. When network delays are smaller, a packet is less likely to be "bumped" from a deduplication filter while it (or a duplicate) is still in flight. Thus, the sizes of the flood queues and deduplication filters are related quantities: larger queues lead to more delay and therefore require larger filters. The ideal/minimal values are dependent on loads, latencies, the router implementation, and so forth. At this preliminary stage, we have not carefully evaluated the full space of these parameters and simply picked values large enough to work well in our main experiments. This meant filters with $8,000$ entries in the campus and cluster topologies, and $80,000$ entries for the WAN topology. The larger filter for the WAN follows from the fact that average latencies in this topology are at least three orders of magnitude higher than in the other topologies. $80,000$ entries seems easily achievable in the split design mentioned in Section 3.4.2, where it can be implemented using cheap DRAM rather than CAM or SRAM, and may well be feasible for native implementations since it is on the same order as, *e.g.,* a MAC learning table.

3.5.1 Effectiveness

The most crucial question, of course, is: when a failure occurs, does Ark help to deliver substantially more packets than if one simply waits for routing to reconverge?

The short answer is an emphatic yes. As we see in Figure 3.1, on average, more than 96% of the packets that the baseline link-state algorithm would drop at the default 50ms delay end up being delivered by Ark. Only at the impossible 0ms delay does the baseline algorithm achieve a comparable degree of delivery, where the lost packets are almost entirely limited to the ones sent before the failure was detected. Notably, in the campus and cluster topologies, Ark only loses two and seven packets more than 0ms link-state (representing respectively only 0.06% or 0.08% of the packets dropped by the baseline 50ms). Perhaps surprisingly, Ark does *better* than 0ms delay on the wide area topology. This is entirely due to cases where the only viable path for some packets after a failure is back the way the packet came. The link-state router disallows such hairpinning packets as they are typically a source of micro-loops. In Ark, these packets are converted to floods and flood back the way they came safely. As a reminder, the WAN experiment used an $80,000$ entry filter. If performed with an $8,000$ entry filter (as with the other topologies), the mean number of undelivered packets raises to $1,765$ – still lower than without Ark and a 1ms delay.

Beyond comparisons to the OSPF-like routing algorithm, we note that our results are also comparable to the *magic* routing algorithm in which routing updates happen immediately upon failure detection; in essence, this is modeling instantaneous recovery. Ark providing comparable results should not be surprising, since as soon as a failure is detected in Ark, all subsequent packets are flooded until the routing state is updated. These packets are delivered unless there is significant congestion so that these flooded packets are dropped. Under the conditions we simulated, where the load on the network was roughly 50% of what it could handle (for a given traffic matrix) and we selected sufficient filter sizes, almost all flooded packets are delivered.

3.5.2 Under Other Loads

In previous experiments, we focused on our default traffic matrix based on halving the maximum supportable rate. Here we simply report that (unsurprisingly), Ark performs even better on experiments with lighter loads, and (more significantly) that it also performs reasonably under heavier loads. For example, in experiments with 85% of the maximum load on the campus and cluster topologies, we found that Ark still performs comparably to link-state with 0ms delay (though in some cases – especially on the cluster topology, where traffic is evenly distributed – this is not particularly better than link-state with 50ms because at high load, a failed link is likely to reduce the capacity of the network such that losses due to overloaded links dominate losses due to convergence).

3.6 Conclusion

Ark is a simple approach to routing resilience that works in conjunction with existing routing algorithms by utilizing safe flooding to deliver packets while the underlying algorithm is converging. Our preliminary evaluation shows that Ark coupled with a link-state algorithm using reasonable delays comes within a hair's breadth of the performance of a hypothetical ideal link-state algorithm with no delays other than the propagation time of routing messages, which in turn closely matches a magical algorithm that updates routing tables immediately upon failure detection.

Our point here is not that such results are not available by using highly sophisticated approaches such as the extensive use of backup paths and topology-specific routing techniques. Instead, our point is that these significant improvements are available via an algorithm that (i) leaves the current routing algorithm untouched, with no changes in configuration or behavior, (ii) poses no additional risk to the network operation, as the only packets being handled differently would have been otherwise dropped and are sent at lower priority than other packets, (iii) can be implemented on many existing routers which are either programmable or have a reasonably performant CPU, and (iv) does not require any additional operational expertise to manage.

Thus, we think Ark provides, for the first time, a resilient routing solution *for the rest of us* – that is, one suitable for the large class of networks where extreme availability is appreciated but not a first-order concern. Work remains, such as understanding the queue and filter sizes that would be needed for operational networks, a detailed evaluation of the split design's performance, and further investigating of the possibility of porting Ark to commodity routers. All of these will be the subject of future work. However, given that Ark's performance promises to rival that of instantaneous recovery, the most fundamental limiting factor in routing resilience may be the time taken to detect a failure – an issue which has not seen substantial attention outside of highly engineered networks. This is where the field's attention should now be focused.

Chapter 4

Internet Architecture Revolution

In discussions about changes to the Internet architecture, there is widespread agreement on two points. The first is that the Internet architecture is seriously deficient along one or more dimensions. The most frequently cited architectural flaw is the lack of a coherent security design that has left the Internet vulnerable to various attacks such as DDoS and route spoofing. Many also question whether the basic service model of the Internet (point-to-point packet delivery) is appropriate now that the current usage model is so heavily dominated by content-oriented activities (where the content, not the location, is what matters). These and many other critiques reflect a broad agreement that despite the Internet's great success, it has significant architectural problems.

However, the second widely-accepted observation is that the current Internet architecture is firmly entrenched, so attempts to significantly change it are unlikely to succeed, no matter how well motivated these changes are by the aforementioned architectural flaws. More specifically, the IP protocol is deeply embedded in host networking and application software, as well as in router hardware and software. As a result, architectural changes face extremely high deployment barriers (as evinced by the decades-long effort to move from IPv4 to IPv6).[1]

These two beliefs have moved the research community in contradictory directions. Spurred by the first – that the Internet architecture is deeply flawed – projects like NewArch [27] and others began (in the late '90s) to look at fairly fundamental redesigns of the Internet (which would later become popularly known as *clean-slate* designs following Stanford's Clean Slate program). In subsequent years, there were large NSF research (FIND, FIA) and infrastructure (GENI) programs (along with their numerous EU counterparts) devoted to this clean-slate quest. Many good ideas emerged from these efforts, ranging from new security approaches [122, 7] to mobility-oriented architectures [110] to service-oriented architectures [89] to new interdomain designs [119, 40] to information-centric networking (ICN) [65, 58] to entirely new conceptions of the Internet [44, 87, 28, 8, 118, 104].

Despite their motivating needs and useful insights, and the long history of sizable research funding, none of these clean-slate designs had significant impact on the commercial Internet. This

[1]There has been great progress in various network technologies, such as transport protocols (*e.g.,* QUIC), data center congestion control, reconfigurable data center topologies, and host networking stacks (*e.g.,* DPDK and RDMA); however, as we will see later when we define the term "architecture", these changes are not architectural in nature.

eventually led to a backlash against clean-slate designs, with many in the field viewing them as useless exercises that are hopelessly detached from reality. As a result of this backlash, the second observation – that the Internet architecture is, and will remain, difficult to change – then became the dominant attitude, and there has been a marked decrease in clean-slate research. The community's emphasis is now on backwards-compatible designs that have been tested in large-scale operational networks, or at least evaluated with data drawn from such networks. Here, we try to reconcile these two perspectives by proposing an *architectural framework* called Trotsky that provides a *backwards-compatible* path to an *extensible* Internet. There are five key words in our mission statement that we now clarify.

We delve deeper into this definition in Section 4.2.3, but in brief we define an Internet *architecture* to be the specification of the hop-by-hop handling of packets as they travel between sender and receiver(s). By architectural *framework* we mean a minimal set of design components that enable the deployment and coexistence of new architectures. Saying a new design is *backwards-compatible* or *incrementally deployable* means that one can deploy it in a way that does not prevent legacy applications or hosts from functioning, and that one can reap some benefits while only partially deployed. One may require that systems be upgraded to take advantage of the new capabilities of this design, but legacy systems can continue to function unchanged.

By an *extensible* Internet we mean an Internet where new architectures can be deployed in a backwards-compatible manner, and multiple of these architectures can exist side-by-side. This latter property directly contradicts our current notion of IP as the Internet's narrow waist. We chose architectural extensibility as our central goal because it: (i) allows us to easily deploy radical new architectures without breaking legacy hosts or domains, (ii) allows these new architectures to "evolve" over time (new versions are just another architecture, and hosts/domains can smoothly transition from one to another), and (iii) allows applications to avail themselves of multiple architectures, so they can leverage the diversity of functionality to achieve their communication goals.

This last point is crucial. The ability for an application to simultaneously use multiple architectures changes the nature of an architecture from something that must try to meet all the needs of all applications, to something that more fully meets some of the needs of some applications. Thus, architectural extensibility would hopefully lead to a broader and ever-evolving ecosystem of architectures which, *in their union*, can better meet all the needs of all applications. Our hope is that by creating an extensible Internet, the Trotsky architectural framework will be true to its name by enabling a *permanent revolution* in architectural design.

4.1 Contribution

Trotsky has two key properties. The first is that the framework itself is backwards-compatible; nothing breaks while it is being deployed, and those adopting it accrue benefits even while it is only partially deployed. Second, once deployed, Trotsky enables radical new architectures to be deployed in a backwards-compatible fashion and to coexist with other architectures. To our knowledge, Trotsky is the first proposal that can simultaneously achieve both of these goals. *As such, it is the first proposal that provides a backwards-compatible path to an extensible Internet.*

This work is unabashedly architectural, and thus our contribution is conceptual rather than mechanistic. When designing systems to be extensible, the key questions are typically about the modularity of the design (*i.e., are there aspects that break legacy systems, or limit the scope of new extensions?*), not about the novelty of algorithms or protocols. The same is true here. The basic mechanisms we use within Trotsky are extraordinarily simple; what is novel about Trotsky is the way in which these mechanisms are applied. Trotsky is based on a better understanding of the modularity needed to achieve architectural extensibility, not on a better design of the necessary mechanisms. In fact, to the extent possible, we try to reuse current mechanisms, and our resulting design strongly resembles the current Internet in all but one important aspect: how the Internet handles interdomain traffic.

After providing some background in the next section, we make our conceptual case for Trotsky in three steps. First, we analyze why architectural evolution is currently so hard. We find (in Section 4.3) that rather than this difficulty being inherent, it is due to a missed opportunity when ASes became a part of the Internet's structure, and argue that this mistake can be corrected by treating interdomain delivery as an overlay on intradomain delivery. Trotsky, the design of which is presented in Section 4.4, *is merely the set of infrastructural pieces needed to support this overlay.*

Second, we have implemented Trotsky (see Section 4.5), producing a prototype of a Trotsky-Processor (Trotsky's version of a router) that supports Trotsky and some selected architectures using a mixture of hardware and software forwarding.

Third, in Section 4.6 we use this prototype to demonstrate that with Trotsky: (i) one can incrementally deploy a variety of new architectures, and (ii) applications can use multiple of these architectures simultaneously.

Note that our focus here is not on what future Internet architectures should be, or what problems they should solve, but rather on how we can change the Internet to the point where we can more easily deploy such architectures. Of course, we are trusting in the premise that architectural extensibility is an important goal. While we firmly believe that to be true, we are not saying that there is a specific new architecture that needs to be deployed immediately. Rather, our contention is merely that the Internet would be far better if it could easily adopt new architectures and if applications could make use of multiple architectures. Such a development would turn the Internet into a more dynamic and diverse infrastructure.

There are essentially only two alternatives to architectural extensibility: (i) forever using ad hoc architectural workarounds for security, resilience, and other shortcomings of the current Internet, or (ii) eventually replacing the Internet entirely. Given that architectural extensibility is reachable in a backwards compatible manner, it seems preferable than either of these two alternatives. And if we think we will need to eventually evolve the Internet's architecture, then we should start laying the intellectual foundation now. This foundation, rather than immediate adoption and deployment, is the primary goal of this work.

4.2 Clarifications, Assumptions, and a Question

In the remainder of this chapter, we will proceed with some assumptions and terminology that should first be elaborated upon; we do so in this section.

4.2.1 Two Clarifications

First, in what follows we typically refer to packets as the means of communication, with the packet header containing crucial signalling information (source, destination, QoS, etc.). Everything we describe can also be applied to connection-oriented designs where the signalling information is carried out-of-band. While we briefly explain how this can be done within Trotsky in Section 4.6, for the sake of clarity our basic presentation focuses only on packet-based designs.

Second, there are several other works on architectural evolution. We postpone a detailed comparison of our approach with these until after we have more fully explained our design (see Section 4.7), but here we merely note that we have borrowed heavily from the insights in [66, 100, 39] but extended them to provide an incrementally deployable evolutionary process, which is the central contribution of this work.

4.2.2 Three Key Assumptions

First, while we wish to enable architectural innovation, one aspect of the Internet we do not expect to change any time soon is its domain structure (*i.e.,* being organized around ASes). This structure is not a technical design decision but rather a manifestation of infrastructure ownership and the need for autonomous administrative control – factors that we do not see changing in the foreseeable future. Thus, we assume that future architectures involve the cooperative behavior of multiple autonomous domains.

Second, given the momentum behind Network Function Virtualization (NFV) and Edge Computing (as in [16, 48]), our designs assume that domains have software packet processing capabilities at their edges (typically in the form of racks of commodity servers). This assumption has not yet been fully realized, but it is where NFV and Edge Computing are rapidly moving, so it seems a reasonable assumption when considering a future Internet.

Third, we assume that support for IPv4 (or IPv6) remains in place indefinitely (perhaps eventually replaced by a similar successor) so that hosts and applications that are Trotsky-unaware can continue to function without change. We will refer to this as the *default* architecture.

4.2.3 What Is an Architecture?

In a paper about architectural extensibility, we should clearly define the term "architecture". Informally, people often say an Internet architecture is what entities must agree on in order to interoperate, but what does this really mean? While there are multiple L1, L2, and L4 designs, the crucial aspect of today's Internet is that it has a single design for L3. This "narrow waist" is the component that different networks must agree on in order to interoperate, and the necessity of such a narrow waist is one of the central dogmas of the Internet. Thus, when we talk about the current Internet architecture we usually mean its L3 design, IP, which dictates the data plane behavior at both hosts and routers. In addition, in order for different domains to interoperate, they must also agree on the interdomain routing protocol (currently BGP).

The literature about new architectures is then mostly about alternative designs for L3 (along with interdomain routing). When looking only at L3 in the current architecture, a packet's trajectory starts at the sending host, then traverses some number of routers (perhaps zero), and then arrives at the destination host (or hosts).

Generalizing this to not focus solely on L3 (for reasons that will soon be clear), we will define an Internet *architecture* to be a design for handling packets that travel from an originating host, through a series of intermediate network nodes that process them according to the architecture's data plane (which, in turn, is guided by its control plane), to eventually arrive at one or more receiving hosts anywhere else in the Internet (*i.e.,* to be an architecture, these designs cannot be limited to local networks or particular domains, but must be capable of spanning the entire Internet).

As noted earlier, Trotsky is not an architecture but an architectural *framework*. This is because Trotsky does not encompass all the features needed to handle packets end-to-end; instead, it defines only those features necessary to create an extensible Internet capable of deploying new architectures in a backwards-compatible manner. While this distinction between architecture and framework is new to the networking community, the operating systems community has long understood that these concepts can be quite different; microkernels taught us that, rather than having a rigid monolithic OS, one can enable innovation by standardizing only a small subset of OS functionality and allowing the rest of the functionality to be supplied at user-level. While the microkernel is not enough, by itself, to support applications, it represents the key portion that enables extensibility. Similarly, Trotsky is not enough, by itself, to handle packets end-to-end, but it represents the key components that enable architectural extensibility.

4.3 Motivating the Design

We noted earlier, without explanation, that the key to achieving extensibility is to treat interdomain delivery as an overlay on intradomain delivery. Here we provide the motivation for this statement.

4.3.1 The Internet Is a Series of Intrinsic Overlays

Today when we hear the term "overlay" we typically think of an ad hoc approach for deploying one protocol on top of another (as in IPv6 over IPv4), or a virtual infrastructure over a physical one (as in RON [6]). However, the Internet's layering is essentially the inherent and recursive use of overlays [116]. In these overlays, which are built into the architecture, nodes in each layer *logically* connect directly with their peers, but *physically* are communicating using technologies from the layer below (which are, recursively, built out of technologies below them). For instance, as shown in Figure 4.1: neighboring L2 endpoints directly communicate with each other at a logical level,

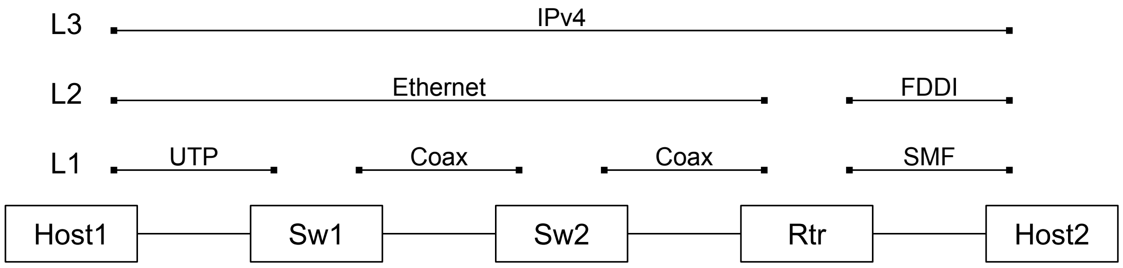


Figure 4.1: *Simplified hybrid layer diagram of a modern network with two L2 switches and one L3 router. L1 depicts the physical media (the lowest sublayer of L1 in the OSI model). L2 depicts the media access layer (the lowest sublayer of L2). L3 is the Internet layer (described in RFC 1122).*

but are connected via physical L1 links; neighboring L3 endpoints directly communicate with each other at a logical level, but are physically connected via "logical links" provided by L2 networks. This extends to L4, where two L4 endpoints directly communicate at the logical level with L3 providing "logical pipes" connecting the two endpoints. Thus, L2 is an inherent overlay on L1, L3 is an inherent overlay on L2, and L4 is an inherent overlay on L3.[2]

The intrinsic use of overlays allowed the Internet to accommodate heterogeneity – and thereby spurred innovation at L1, L2 and L4 – through the presence of three factors. First, these layers provided relatively clean interfaces so, for instance, an L2 network could use multiple L1 technologies, as long as each supported the required L2 interface. Second, there was a unifying point of control to manage the heterogeneity. Each L2 network provided the unifying point for multiple L1 technologies, and L3 provided the unifying point for multiple L2 technologies. Applications were the unifying point for L4, as each application typically uses a default L4 protocol (so the two ends of an application would automatically use the same transport protocol). Third, the underlying layers L2 and L3 provide the equivalent of a next-header field, so that the packet could be handed to the appropriate next-layer protocol during processing.

If we consider the Internet before the advent of Autonomous Systems (or domains), then the one layer where there was no way of managing heterogeneity was L3. If there were multiple L3 protocols then (i) two L2 networks that only supported disjoint subsets of these L3 protocols could not interconnect and (ii) even if all adjacent L2 networks shared a common L3 design, there would be no way to guarantee end-to-end connectivity for any particular L3 design. Thus, because there was no way to manage heterogeneity at L3 before domains, we assumed that there could only be one L3 protocol (the "narrow waist" of the architecture). In short, in the absence of domains, a narrow waist at L3 was needed to enable arbitrary sets of networks to easily connect and provide end-to-end connectivity, while at the other layers the overlay approach could support diversity.

This made architectural evolution extremely difficult because, when there is a single L3 protocol that must be deployed everywhere, converting to a new L3 design requires (eventually) changing the L3 implementation in all routers. Moreover, if the new L3 design is not compatible with

[2]The discussion here and what follows is not a summary of what the original Internet designers thought at the time, but is a description of how we might think about the Internet's design in hindsight.

current router hardware, then this is not just a configuration or firmware change in routers, but a hardware one as well. In the case of the IPv4 to IPv6 transition, we have had decades to prepare all routers to support IPv6, and yet the transition still requires many ad hoc measures such as building temporary overlays (so that IPv6 traffic could be carried over IPv4 networks) and deploying special-purpose translators (to translate between IPv4 and IPv6 packets). These techniques have been effective, but it required decades of preparatory work; this is not an example we can follow for more general architectural change.

When domains arose, they presented a unique opportunity to manage the heterogeneity at L3. A domain could ensure *internal* connectivity (by carefully managing which L3 designs their internal networks supported) and also manage *external* connections to ensure global end-to-end delivery (using remote peering, which we describe later in this section). Managing internal and external heterogeneity independently – *i.e.,* decoupling the internal choices from the external ones – requires making a clean architectural distinction between interdomain and intradomain data planes (just as we make a distinction between L1, L2, and L3 data planes). The natural way to do this, following what had been successfully done at each of the other layers, would have been to make interdomain connectivity an intrinsic overlay on intradomain connectivity. But this path was not chosen, and the opportunity to support heterogeneity at L3 was squandered.

Instead, it was decided to connect domains by using the same data plane L3 protocol as used within domains, and to devise a new control plane protocol to handle interdomain routing. This allowed domains to choose their own intradomain routing protocols, but there was still a single universal L3 data plane. It is this L3 universality in today's architecture – an architecture which in so many other ways is designed to support heterogeneity – that makes the current architecture so hard to change.

4.3.2 Two Design Decisions

Our goal for Trotsky is to make the mechanisms needed to incrementally deploy new architectures a fundamental part of the infrastructure, rather than something deployed for a specific architectural transition (such as IPv4 to IPv6). This involves two major decisions.

First, to use domains as a way of managing heterogeneity at the L3 layer, we reverse the decision to use the same data plane for both intradomain and interdomain delivery. Instead, we decouple the distinct tasks of interconnecting networks within a domain (which we leave to L3) and interconnecting different domains, for which we introduce a new layer (which we will call L3.5) which is an intrinsic overlay on L3. This overlay approach – which allows the Internet to (as we argue later) support multiple L3.5 designs and rely on domains to manage the resulting heterogeneity – is the key to creating an architecturally extensible Internet. *The role of Trotsky is little more than providing the intrinsic support for this overlay.*

Second, we initially deploy any new L3.5 design only at domain edges (in entities we will call "Trotsky-Processors" or TPs) and at hosts. While the term *edge* is increasingly overloaded, what we mean here are all the ingress/egress points in a domain, including peering points (bilateral and IXPs) and customer attachment points.

This decision has two implications: (i) it greatly limits the required scope of deployment for any new L3.5 design, and (ii) the aforementioned presence of software packet processing at the edge (due to NFV and Edge Computing) means that the required deployments can be in software rather than hardware (and we later show in Section 4.6 that this is feasible). Thus, this second decision means that supporting a new L3.5 protocol requires only a software modification at domain edges, rather than (as today) hardware modifications in all routers.

How can we limit these changes only to the edge? Just as L3 abstracts L2 connectivity using logical links, L3.5 abstracts L3 connectivity as "logical pipes". That is, when looking only at L3.5, TPs are connected (to each other and to hosts) by logical pipes which use technologies from the layers below to deliver packets from one end of the logical pipe to the other (see Figure 4.2). Thus, even when only deployed at the edge, packets still travel from a host through a series of L3.5-supporting nodes to the destination host(s), all connected by logical pipes.[3]

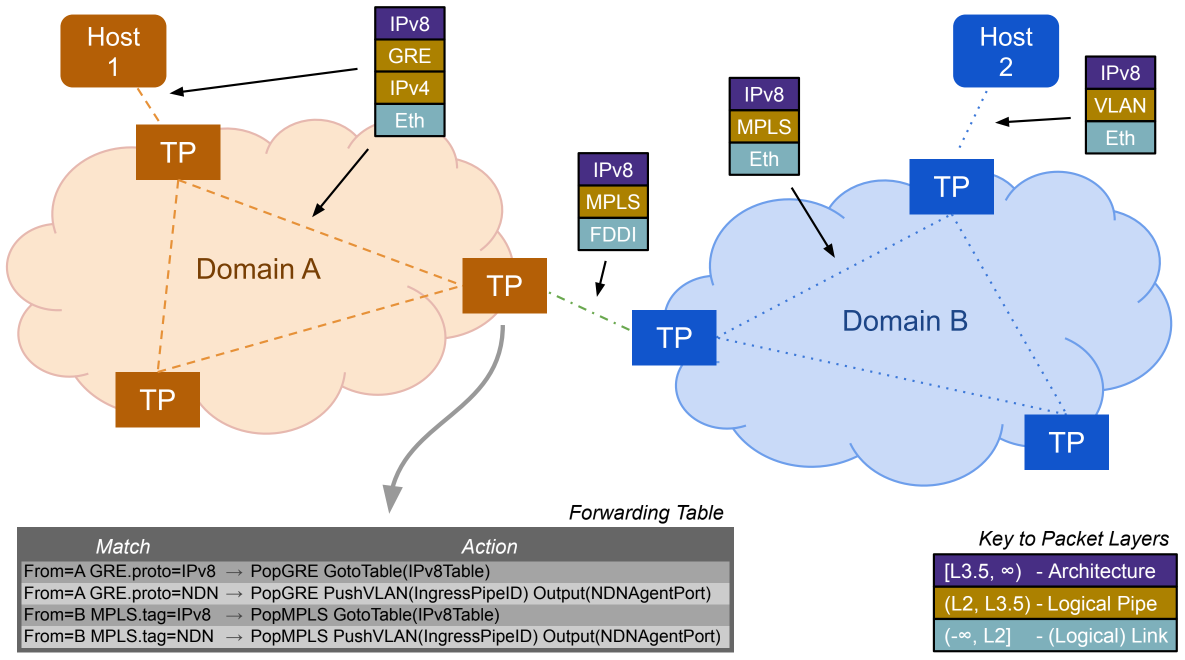
4.3.3 Making Interdomain an Inherent Overlay

For L3.5 designs to be overlaid on L3 (and layers below), L3 must provide logical pipes to connect pairs of L3.5-aware nodes (*i.e.,* TPs and hosts). These pipes must support, at the very least, best-effort packet delivery (below we describe optional enhancements to the pipe model) and also provide the equivalent of a *next-header* field (as Ethernet does at L2, and IP does at L3), so that a receiving TP knows what L3.5 protocol should be applied to an arriving packet. Note that a logical pipe is nothing more than an intradomain packet delivery service (like IP) that delivers packets from one location to another, and need not entail establishing n^2 tunnels. As shown in Figure 4.2, logical pipes can be constructed in various ways, and encapsulate packets as they travel between TPs.

An L3.5 design defines the data plane behavior (*e.g.,* packet format, forwarding rules, etc.) executed in the TPs, along with whatever interdomain control plane protocol is needed to support the L3.5 design. This is no different than what a new L3 design would define in the current Internet; thus, the Trotsky approach imposes no additional architectural restrictions. Of particular note, this control plane may be fully distributed among the TPs (similar to the familiar BGP mesh), may involve additional special-purpose nodes (similar to BGP route reflectors or logically centralized SDN control planes), or be of any other conceivable design (*e.g.,* securely outsourced [42]).

An L3.5 design's data and control plane determines an end-to-end service model. This service model applies even when the communicating nodes are in the same domain: just as IP determines (via host L3 code) the service model when two hosts in the same subnet (L2 network) communicate, the host L3.5 implementation determines the service model even when two hosts in the same domain connect. Thus, one should not think of an L3.5 design as *only* an interdomain data plane, but as the only data plane that can span domains and which therefore determines the nature of the end-to-end service. In contrast, L3 protocols in our approach only provide logical pipes within domains, and do not determine the Internet's service model.

[3]One might ask whether putting functionality only at the edge gives adequate performance for content-oriented architectures (which rely on caching). This has been studied, and it was shown that supporting ICN designs only at the edge provided a very high fraction of the performance benefit [35].



Match		Action
From=A GRE.proto=IPv8	→	PopGRE GotoTable(IPv8Table)
From=A GRE.proto=NDN	→	PopGRE PushVLAN(IngressPipeID) Output(NDNAgentPort)
From=B MPLS.tag=IPv8	→	PopMPLS GotoTable(IPv8Table)
From=B MPLS.tag=NDN	→	PopMPLS PushVLAN(IngressPipeID) Output(NDNAgentPort)

[L3.5, ∞)	- Architecture
(L2, L3.5)	- Logical Pipe
(-∞, L2]	- (Logical) Link

Figure 4.2: *Illustration of a possible deployment, with headers for a hypothetical IPv8 L3.5 packet traveling between two hosts. Different logical pipes are implemented differently (using GRE+IPv4, MPLS, and VLANs). A partial and abstract forwarding table for one TP's Pipe Terminus (see Section 5.1) describes how different packets are handled: IPv8 packets are sent to a hardware IPv8 forwarding table, and NDN packets (another L3.5 supported by the domain) are handled by a software NDN agent attached to a physical port (the ingress pipe ID being encoded in a VLAN tag so the agent is aware of it).*

The service model defined by an L3.5 design is not restricted to basic best-effort packet delivery; instead, it could be service-oriented, or information-centric, or mobility-oriented, or something entirely different. In addition, these L3.5 designs need not support all applications, but instead could focus on more specific use cases (such as supporting in-network aggregation for IoT, or providing a particular security model). Because of such restricted use cases, L3.5 designs can impose restrictions on the underlying logical pipes. For instance, an L3.5 design could require the underlying technologies to be all-optical, or provide end-to-end encryption, or various multipoint delivery functions (such as broadcast or multicast). While these restrictions would limit the applicability of such designs, this is appropriate for L3.5 designs that are intended for specific use cases.

Thus, we expect that L3.5 designs might support a wide variety of service models, and therefore hosts must have a network programming API (or *NetAPI*) that can support a range of communication semantics. Fortunately, this is already provided by existing mechanisms for adding support for new protocols to the socket API (*e.g.,* protocol families and the flexible *sendmsg()* function), which is supported by virtually all current operating systems. Applications need to be aware of the new NetAPI semantics in order to use the associated L3.5 designs (*e.g.,* the service model of ICN designs is different from that of IP, and this must be reflected in the NetAPI; an application must be aware of these different semantics in order to invoke them).

4.3.4 Implications for Extensibility

Before delving into the design of Trotsky in the next section, we should clarify why inserting a new interdomain layer (L3.5) addresses the issues of extensibility. There are three factors here: independence, deployment, and coexistence. First, by decoupling interdomain (L3.5) from intradomain (L3) designs via a clean "logical pipes" interface, one can evolve one without changing the other. Second, the deployment of new L3.5 designs is made easier by our only requiring that they be deployed at domain edges, and initially in software. Third, multiple L3.5 designs can coexist, and this is made possible by the existence of an extensible NetAPI (which is already in use) and the fact that the L3 logical pipes must support a next-header field that identifies which L3.5 design should handle the packet. Because of these three factors, one can deploy a new L3.5 design without changing any L3 designs, this deployment can initially be limited to only software at domain edges, and these L3.5 designs can operate in parallel. But three questions remain:

How general is the Trotsky approach? The definition of L3.5 involves the forwarding and receiving behavior of hosts and TPs (all connected by logical pipes), along with an interdomain control plane. Note that this picture is directly analogous to the current picture of L3 except with TPs replacing routers, and with logical pipes replacing logical links (we use different terms – pipes vs. links – because the pipe abstraction is global as in L3, whereas the link abstraction is local as in L2). Thus, any design that could be seen as a candidate for an L3 architecture in today's Internet could also be seen as a candidate for L3.5 in our approach. The reverse is not true, because (as mentioned above) we allow L3.5 designs to put restrictions on the nature of the logical pipes. However, one could imagine designs that require specifying more than a per-hop data plane and a control plane (*e.g.,* designs requiring some large-scale physical-layer coordination) that would not be implementable within Trotsky.

How does Trotsky handle partial deployment of an L3.5 design? When a new L3.5 design arises, it obviously won't be deployed by all domains simultaneously. Trotsky deals with partial deployment by the well-known technique of *remote peering*. Two domains that are not directly physically connected but wish to use the new L3.5 can peer by (i) building a tunnel between them (using, for example, the default L3.5 design) and then (ii) establishing a logical pipe over this tunnel (which treats the tunnel like a logical link). In this way, L3.5 designs can be initially deployed by a subset of domains which connect via remote peering.

How does Trotsky handle partial deployment of Trotsky itself? In terms of functionality, domains that are not Trotsky-aware are roughly equivalent to Trotsky-aware domains that support only IP at L3.5.[4] To peer with a domain that has not adopted Trotsky, Trotsky-aware domains use special logical pipes that strip off all but the IP headers.

4.4 Design

We now turn to the design of Trotsky. This is anticlimactic, as the design is minimal and straightforward. However, since Trotsky is the framework that remains constant, while various L3.5 architectures come and go, it is important that we properly identify the crucial functionality and leave everything else to other (more dynamic) components in the ecosystem. Thus, in this section, we start by describing the necessary pieces of Trotsky (whose implementation we describe in the next section), and then discuss issues that are left to either L3.5 designs or to the domains. We then end this section with a walk-through of what happens when a laptop arrives in a domain to illustrate how the various pieces fit together.

4.4.1 Design Overview

Trotsky must provide intrinsic support for overlaying L3.5 designs on L3, which involves two key components.

Logical Pipes. The lower layers must provide logical pipes (*i.e.,* connectivity) between every two L3.5 nodes. Constructing these pipes requires data plane mechanisms (to support basic delivery and a next-header field) and control plane mechanisms (to support functions such as MTU negotiation).

Host Bootstrapping. The bootstrapping of a host requires some amount of basic information – most crucially, which specific protocols the domain uses for logical pipes and which L3.5 designs the domain supports. While one could imagine an alternative where each L3.5 design defines its own idiosyncratic discovery procedures and protocols for this purpose, we prefer an explicit and unified approach. To this end, we propose a single protocol conceptually akin to today's DHCP.

[4]Note that in Trotsky, IP can be used both at the L3 layer (to support logical pipes) and at the L3.5 layer (to provide end-to-end connectivity); the address spaces for these two uses are separate.

Trotsky is nothing more than the union of these two components, with everything else left to L3.5 designs and the domains. We now list some of those other functions.

4.4.2 Functions Left to L3.5 Designs

Naming and Addressing. In the simplest case, all L3.5 designs would have their own set of names, name resolution services, and addresses. However, we expect that names and name resolution systems might be shared between L3.5 designs; similarly, addressing schemes might be used by more than one L3.5 design (*e.g.,* IPv4 addresses might be used for an end-to-end optical design). To accommodate this, we assume that each L3.5 design identifies the naming and addressing schemes it is compatible with, and the host OS and domain bootstrapping mechanisms can direct resolution requests to the appropriate infrastructure. We assume that the responsibility for establishing a new name resolution infrastructure does not lie with individual domains, but with third-parties (such as ICANN).

Interlayer Mapping. In the current Internet, ARP is commonly used to map between L3 addresses and L2 addresses. Similarly, in Trotsky, L3.5 designs must provide a mechanism to map between L3.5 addresses (which determine end-to-end-delivery, but are not necessarily traditional host locators and could instead refer to content or a service) and the pipe addresses (addresses used by logical pipes to deliver packets within a domain). This can be done by the L3.5 packet itself (by having a field in the packet to carry the pipe address) or with some ARP-like mechanism defined by the L3.5 design. One might question why this is being left to L3.5 designs rather than via a generic Trotsky functionality. In fact, early versions of Trotsky included a generic ARP-like mechanism that could be used by all L3.5 protocols. This approach shared a weakness with ARP: the required state scales with the number of endpoints (e.g., host addresses). Unfortunately, this is potentially more problematic in Trotsky than with ARP because the required state for a domain is much larger than a single L2 network. While we believe the state requirements are currently feasible, we think it unwise to assume this remains true in a design that should underlie the Internet for the foreseeable future. Accordingly, we moved the responsibility for this task from the Trotsky framework to the individual L3.5 protocols. This allows for L3.5s to perform optimizations that are specific to their designs and even allows building solutions to the mapping problem into their design from the outset (for example, designing addresses to allow for the mappings to operate on aggregations, assigning addresses to facilitate a programmatic rather than state-based mapping, or embedding L3 addresses within their own L3.5 protocol).

Reachability. With multiple L3.5 designs coexisting, and with not all domains supporting all L3.5 designs, how does a host decide which to use to reach a particular destination or content or service? There are several answers to this. First, if the name of the destination/object/service is tied to a specific L3.5, then the sending/requesting host knows which L3.5 to use (assuming that L3.5 is supported by its own domain). Second, L3.5 designs could (but need not) support an interface that allows hosts to query reachability (a negative result returning something like the ICMP Host

Unreachable message). Third, names and/or the name resolution service might contain hints about which L3.5 designs are supported. Ultimately, however, applications can simply attempt using the name with multiple L3.5 designs.

Congestion Control Support. Today, congestion control is implemented in L4. We assume this general approach will largely be followed, where congestion control is implemented somewhere above L3.5. However, there are various proposals (*e.g.,* RCP [21] and FQ [31]) in which routers provide support for congestion control, and individual L3.5 designs can specify that the TPs and logical pipes that carry it support such congestion control support mechanisms.

Security. In discussing security, it is important to make two distinctions. First, we must distinguish between the broad concerns of operating systems and applications, and the more narrow responsibilities of the network itself. Second, for those concerns that are the responsibility of the network, we must distinguish between those which must be addressed within the Trotsky framework itself, and those which can be left to L3.5 designs.

Turning to the first question, as has been observed by many (see [12, 66]), the goal of network security, very narrowly construed, is to provide communicating parties with availability (ensuring that they can reach each other), identity (knowing whom they are communicating with), provenance (knowing the source of data they have received), authenticity (knowing that this data has not been tampered with), and privacy (ensuring that data is accessible only to the desired parties). All but availability can be handled by cryptographic means at the endpoints (which may be made easier by the choice of naming system and other aspects of an L3.5 design). But availability – in particular the ability to withstand DDoS attacks – requires a network-level solution. In addition, there are a broader set of network security concerns such as anonymity and accountability, as discussed in [66, 88, 7, 70] among other places, and these too require network-level solutions.

The question, then, is whether issues not handled solely by endpoints should be left to individual L3.5 designs, or incorporated into Trotsky itself. Here we firmly believe that these security concerns should be addressed within individual L3.5 designs for the simple reason that, as security threats change and security approaches improve, we should allow these security solutions to evolve over time. For instance, for DDoS, there are a range of approaches, ranging from capabilities [120] to filters [9] to shut-up-messages [7, 66], and we think it presumptuous to bake a single approach into Trotsky itself. Moreover, it is not even clear how effectively Trotsky *could* lay out a security framework sufficient to address the needs of all potential L3.5 designs (with their own potentially idiosyncratic principals, communication patterns, and so on).

All the above said, the protocols of Trotsky itself (*e.g.,* the bootstrapping protocol) must be designed to ensure that they do not create additional security vulnerabilities. To that end, we have designed Trotsky so that it can be implemented in a secure fashion.

4.4.3 Functions Left to Domains

Resource Sharing. When multiple L3.5 designs are supported by a domain, the domain can control how its internal bandwidth is shared between these designs using standard methods available in current routers.

Internal L3.5 Deployment. While we envision that initially L3.5 designs are deployed only at the edge, domains can decide to provide additional internal support later if desired (*e.g.,* to provide closer-by caches if the L3.5 design involves content caching). This can be done by adding additional TPs, and whatever internal control plane mechanisms are needed. This decision can be taken on a domain-by-domain basis.

4.4.4 How Does This All Fit Together?

Consider a Trotsky-aware laptop that arrives in a domain, and proceeds to start a game application. The laptop connects, as it does today, to an L2 technology (*e.g.,* plugging into an Ethernet port, or attaching wirelessly) and uses the Trotsky bootstrap mechanism to determine which L3.5 designs are available, as well as what protocols are used for logical pipes. It then uses the bootstrapping mechanisms relevant to the internal domain protocols (*e.g.,* DHCP if IP is being used for logical pipes), and whatever L3.5-specific bootstrapping mechanisms are available for the L3.5 designs it wants to leverage. The game invokes the NetAPI for content-oriented retrieval (to download graphical and other game assets) and also invokes the NetAPI for a special-purpose multiagent-consistency L3.5 design ideal for multiplayer gaming.

Each packet leaves the host in a logical pipe that uses its *next-header* field to identify which L3.5 the packet is associated with. If the destination for the packet is within the origin domain, then the L3.5 design describes how to reach it using an internal logical pipe without having to send packets to an intermediate TP. If the destination for the packet is in a different domain, the initial logical pipe takes the packet to a TP on the edge of the origin domain, where it is processed by the appropriate L3.5 design and then sent over a logical pipe to a TP running in a peer domain. The peer domain sends the packet through an appropriate logical pipe either to the destination (if the destination is in that domain), or to another peer domain (if the destination is elsewhere).

Note that if the user starts a legacy application (*e.g.,* an application that is not Trotsky-aware), the OS automatically invokes the default L3.5 design (*e.g.,* IPv4) and the application proceeds as today. Similarly, when a host arrives in a domain and does not use the Trotsky bootstrap mechanisms, the domain automatically assumes all of its traffic is using the default L3.5 design. Thus, Trotsky is incrementally deployable since it does not break hosts or applications that are not Trotsky-aware.

4.5 Implementation

In the previous section we identified Trotsky's two key technical tasks: providing logical pipes and host bootstrapping. Here, we describe how these tasks are accomplished in the two entities where Trotsky is implemented: Trotsky-Processors (TPs) and Trotsky-capable hosts. We do so by first describing the implementation challenges in general, and then how they are met in our prototype implementation. Before starting, however, recall that logical pipes essentially encapsulate L3.5 packets as they traverse between two L3.5-aware nodes (just like L2's logical links do for

L3), with each endpoint serving as a terminus for these logical pipes. As was shown in Figure 4.2, these logical pipes – which must support best-effort packet delivery and the equivalent of a next-header field – can be constructed in many ways using protocols at or below L3 (such as using IP at the L3 level for intradomain connectivity, and VLAN IDs as the next-header field). In addition, different domains can use different pipe constructions, and a single domain can use different pipe constructions for different contexts (*i.e.,* the logical pipe used to connect to one neighboring domain may be different than the one used to connect to another domain, and those may both be different than the logical pipes that connect internal hosts to TPs).

4.5.1 Trotsky-Processors

The essential functional units of a Trotsky-Processor are illustrated in Figure 4.3. We now discuss them in turn.

Pipe Terminus. Packets carried by logical pipes arrive at physical interfaces of a TP, enter the pipe terminus (which removes the logical pipe encapsulation), and are then handed to the appropriate L3.5 implementation based on the next-header field. Essentially the opposite operations are performed for packets exiting the TP. In addition, the TP has a unique ID for each logical pipe it terminates, and the pipe terminus provides a mechanism to map each packet with the logical pipe ID, which can be implemented by either storing this ID in the packet header or through other means.

L3.5 Implementations. Every TP supports the default L3.5 design along with possibly several other L3.5 designs. Such support includes an L3.5 data plane and any associated control plane protocols. While control protocols are typically implemented in software, data planes for new protocols are likely to begin in software but can eventually transition to hardware for better performance and power usage. As we discuss further in Section 4.6.1, in addition to a default IPv4 L3.5 implementation, we integrated existing software codebases for two different L3.5 designs into our prototype TP, wrote code for a third, and investigated how one might support several other L3.5 designs. Should hardware support for any of these designs eventually arise, it would be simple to migrate to it. Moreover, existing hardware for the efficient handling of IP already exists, and – as we describe later in this section – it can be directly used in support of interdomain (L3.5) IP. Thus, Trotsky can accommodate both hardware and software data plane implementations.

The Trotsky Control Plane. TPs also contain control/management infrastructure that primarily consists of an extensible store of configuration information which is used for internal purposes (*e.g.,* information about the logical pipes – how many there are, what protocols they use, whether they are intradomain or peering, etc.), and can also be used by L3.5 implementations via an API. Configuration data can be marked for sharing with other TPs via the Trotsky Neighbor Protocol. This protocol also allows for sanity-checking (*e.g.,* that logical pipes are connected to their intended peers), and can probe logical pipes to maintain an accurate view of the pipe bottleneck MTU. This

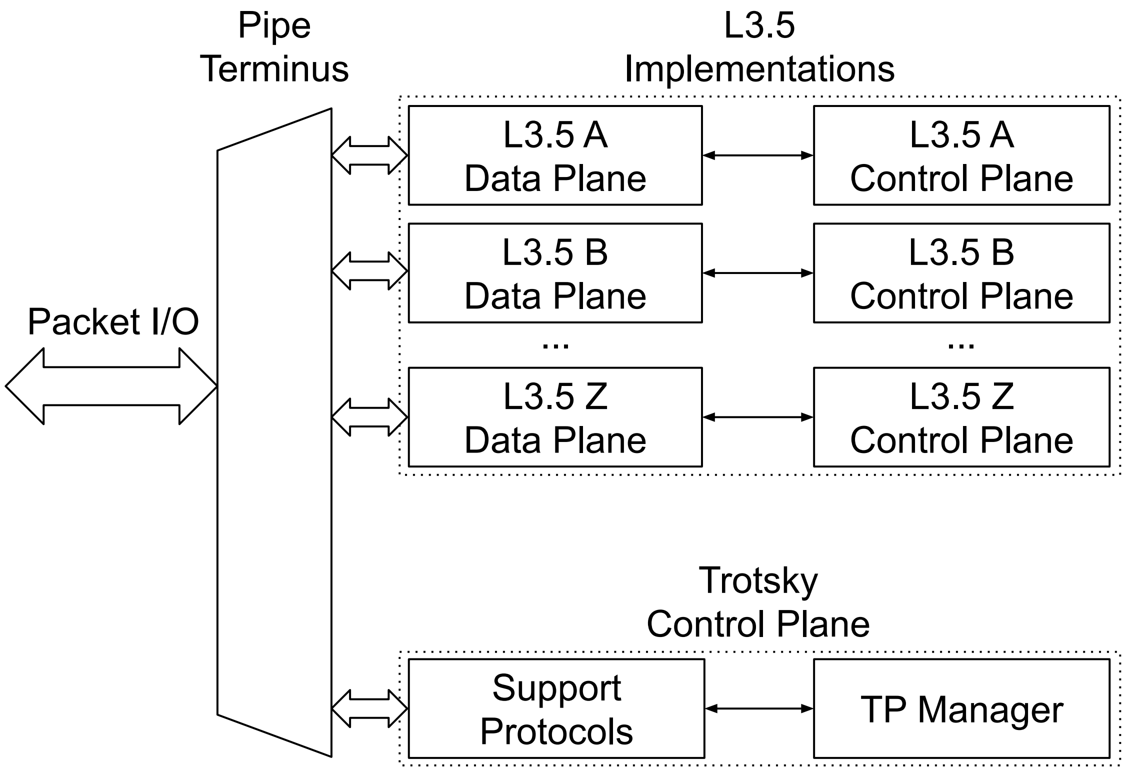


Figure 4.3: *Trotsky-Processor System Diagram, supporting multiple L3.5 designs.*

last aspect becomes necessary when logical pipes span multiple logical links with different MTUs, as the bottleneck may not be directly observable from the pipe endpoints.

Our Implementation. We have implemented the pipe terminus in hardware by using an OpenFlow switch. Arriving packets are de-encapsulated (using any encapsulation formats supported by the switch), and a next-header field extracted to determine the appropriate L3.5 protocol. In the case of the default L3.5 (IPv4), OpenFlow also performs IP prefix matching. For other L3.5s, the packet is forwarded to an attached server where the software implementation of the L3.5 protocol is running, but first, an OpenFlow action encodes the input pipe ID into the packet as a VLAN tag so that the L3.5 implementation knows which pipe the packet arrived on. Packets arriving *from* an attached server have the VLAN tag set (by the server) to the desired *output* pipe ID; OpenFlow rules match on the pipe ID, perform the appropriate logical pipe encapsulation, and forward the packet out the appropriate physical port. All of our control code – which controls the OpenFlow switch, implements the Trotsky Neighbor Protocol, and automates configuration of the integrated L3.5 protocols – is written in Python and runs in user mode on a server attached to the switch.

4.5.2 Trotsky-Capable Hosts

Our prototype host code converts a standard Linux distribution into a Trotsky-capable one. This is done almost entirely via user mode code, some of which (re)configures standard Linux networking features (such as tap interfaces). Host support for Trotsky includes a pipe terminus and implementations of L3.5 protocols, which we do not describe because they are so conceptually similar to the case for TPs. Unlike TPs, hosts incorporate bootstrapping and a NetAPI, which we now discuss.

Bootstrapping. When a new host arrives on a domain, it needs answers to the following three questions: (i) Is this domain Trotsky-capable? (ii) What type of logical pipes does this domain support and what configuration parameters (if any) are required to use them? and (iii) What L3.5 protocols does this domain support? The Trotsky Bootstrap Protocol provides answers to these questions. The server side is implemented with a small, stateless server that must run on each L2 segment and responds to requests sent by hosts to a multicast Ethernet address. If the server answers with empty responses (*i.e.,* no pipe types and no L3.5 protocols supported), or if repeated requests produce no response, the host can conclude that the domain does not support Trotsky and can fall back on legacy IP bootstrapping (*e.g.,* DHCP). Otherwise, two things occur. First, the local pipe terminus is configured. In our prototype, the terminus is a userspace process that implements two types of logical pipe based on IP encapsulation as well as a VLAN-based one, and pipes show up as tap interfaces. Secondly, the supported L3.5s are configured, *i.e.,* DHCP is run over the logical pipe to acquire an L3.5 IP address, and the userspace L3.5 daemons are (re)started attached to the logical pipes.

As running a new bootstrap server on every L2 network may be impractical in the short term, a simplified version of the bootstrap protocol can be encapsulated in a DHCP option. On some

DHCP servers, this can be implemented with only a configuration change; if the server code must be changed, the change is minimal. A modified DHCP client on the host can then send a DHCP request and use legacy IP or Trotsky configuration depending on whether the response contains the Trotsky bootstrap option.

The NetAPI. Within hosts, an API is needed to allow applications to utilize L3.5 protocols. Rather than design a new API, our prototype uses the standard Berkeley sockets interface for this. This choice is based on two factors.

First, several existing prototypes for new architectures (*i.e.,* L3.5 protocols, from a Trotsky perspective) already use APIs closely modeled on the sockets API. Second, as we imagine an ecosystem of L3.5 protocols, it makes sense to – as much as practical – share an API between them, such that software can be written that is agnostic to which L3.5 it is run on as long as it shares common semantics. This is, in fact, already an idea embraced by the sockets API; for example, much code can be written to be agnostic to whether it is run on a unix socket or an IP socket as long as it properly implements the semantics appropriate to its type (e.g., SOCK_STREAM). Moreover, architectures with different primitives can often be usefully mapped to common semantics (in Section 4.6.1, we briefly discuss how we mapped a common content-centric networking pattern – where a sequence of named chunks constitute a larger unit of data – as a streaming socket).[5]

To achieve this in our prototype, we undertook a FUSE-like [102] approach, by developing a kernel module which can redirect API calls on sockets of specified protocol families to a userspace backend. The backend then performs the actual operation. As mentioned above, it is typical for existing interdomain protocol prototypes to already offer a sockets-like API (*e.g.,* XIA's Xsocket interface), so the backend in such cases is straightforward glue code. This approach has a performance hit relative to kernel-based networking, but we use it only for protocols that do not have a kernel implementation (*e.g.,* there is no performance hit for IP).

4.6 Evaluation

In this section we substantiate our architectural claims by: (i) describing how we were able to deploy a variety of architectures within Trotsky, (ii) how within Trotsky, applications can seamlessly use more than one architecture, and (iii) illustrating how a domain can deploy a new L3.5 design. We then provide performance numbers that show that (iv) Trotsky itself does not introduce significant performance overheads and (v) software forwarding is sufficient for simple L3.5s.

4.6.1 Implementing L3.5 Designs within Trotsky

As described above, we implemented a hardware-based IPv4 L3.5 via OpenFlow. In addition, as touched on in Section 4.5.1, we also integrated two existing software implementations of new

[5]While we focus on the sockets API here, we admit that some L3.5s may well benefit from more than trivial additions to it. We have no objection to this if done with a focus on general semantics rather than low-level details, and we note that the IETF TAPS Working Group is already working along these lines (*e.g.,* in [114]).

architectures, prototyped a third L3.5 from scratch, and investigated the implementability of several others. We discuss these here.

NDN. Integrating the existing NDN codebase as an L3.5 protocol was largely a matter of installing the codebase adjacent to the pipe terminus (as in Figure 4.3). The pipe terminus is then directed to forward packets to the NDN forwarding daemon (*i.e.,* the NDN data plane component) when the next-header field corresponds to NDN. Additional "glue code" translates from the TP configuration to an NDN configuration file (crucially, this configures an NDN "face" for each logical pipe).

NDN usually provides its own unique API to applications. As discussed in Section 4.5.2, we favor the reuse of the sockets API when possible. To demonstrate this, we used our FUSE-like mechanism to map requests for an NDN *stream* socket to the NDN *chunks* protocol as implemented by the *ndncatchunks* and *ndnputchunks* tools maintained by the NDN project. The *chunks* protocol already functions much like a stream transport, providing a mechanism to download more data than fits in a single named packet while adding reliability and congestion control. Exposing this through the sockets API allows applications to simply open an NDN streaming socket, connect it to an address which is just the root name of some NDN chunks, and read data from the socket. Section 4.6.2 below describes an example use case.

XIA. XIA is a flexible architecture that we discuss in Section 4.7. Much like the NDN use case, we incorporated XIA as an L3.5 design in Trotsky with minimal modification to the public XIA source code. XIA packets arriving at the TP pipe terminus are forwarded to the XIA daemons and from the XIA daemons back through the terminus to pipes. We made minor changes to the XIA code to account for running on logical pipes rather than directly on Ethernet (essentially disabling the XARP protocol). The "glue code" for our XIA L3.5 implementation leverages the Trotsky neighbor protocol to propagate XIA configuration information between TPs within a domain.

Trotsky Optical Protocol. To illustrate how Trotsky can support non-packet L3.5 designs, we devised the Trotsky Optical Protocol (TOP) which supports the establishment and management of end-to-end (interdomain) optical paths. We assume that each domain participating in TOP has at least one *optical controller* (OC) that manages the optical switches of the domain. In addition, we assume there is a global *path computation engine* (PCE) that keeps track of all optical links, their wavelength (lambda) usage, etc., and computes paths in response to requests.

The TOP L3.5 implementation in TPs is simple. When a host requests the establishment of an optical path to a host in another domain, the host sends a TOP request, which reaches the TOP L3.5 implementation in a TP of the originating domain. The TOP L3.5 code contacts the domain's OC, which requests a path from the PCE. When the OC receives the path, it communicates the relevant portions to the domain's optical network, and also sends the path information to the TP that handles the peering relationship with the next hop domain. The TOP L3.5 implementation in that TP hands the path to the peer's TP, which requests a path from the OC in its domain. The process recurses until the entire path is established.

Other potential L3.5 Designs. For the aforementioned L3.5 designs, we either used existing codebases or (for TOP) wrote our own L3.5 design. To broaden our perspective, we also looked carefully at several other potential L3.5 designs, including IPv6, AIP [7], SCION [122], and Pathlets [40]. In each case, the porting of the design to Trotsky appeared to be straightforward. We did not actually do the porting because in some cases little was to be learned (IPv6), or codebases were not available (Pathlets and AIP), or lower layers in the current codebase were tightly coupled such that porting the code to use pipes would have been overly time-consuming (SCION).

4.6.2 Diversity

We modified the GNOME Web (aka Epiphany) browser to allow it to use both IP and NDN L3.5 designs simultaneously. This required adding an `ndnchunks:` URL scheme, which involved 156 lines of code, and which allows HTML documents to contain URLs of the form `ndnchunks:<ndn name>` (in addition to the usual `http://` and `https://` URLs). Such URLs can read NDN *chunks* published via the *ndnputchunks* tool (see Section 4.6.1). One immediate benefit of this is the ability to leverage NDN's caching benefits when retrieving static objects that today are often served via CDNs. Thus, a single application can utilize multiple L3.5 designs, leveraging the entire suite of available L3.5 functionality rather than being restricted to a single design.

4.6.3 Deploying a New L3.5

We performed functional testing using a software networking tool roughly similar to the well-known Mininet [85]. With this tool, we could create virtual networking environments composed of Linux network namespaces and Docker/LXC containers ([83, 43]) connected in arbitrary topologies by virtual network links. Within the nodes (*i.e.,* namespaces/containers), we can run software TPs, routers, the XIA and NDN protocol stacks, and so on, or simply emulate hosts. While the tool has low performance fidelity, it provided a flexible way to experiment with our prototype and its administration.

For example, by treating a group of nodes as a domain and configuring things accordingly (*e.g.,* choosing a common L3 protocol, configuring the TP nodes within the domain appropriately), we could role-play as the administrators of two domains deploying a new L3.5 design. Consider two peering domains $AS1$ and $AS2$ that both support the IPv4 L3.5 design, but $AS2$ is also running the NDN L3.5 design. $AS1$ wants to add support for NDN and peer NDN with $AS2$ in addition to IPv4. Here are the steps we take in our test environment to instantiate this support, which mirrors reality.

First, acting as the administrators of $AS2$, we alter the configuration of the $AS2$ TP adjacent to the peering logical pipe to allow NDN on it; this is the only action that must be taken for $AS2$. Acting as administrators for $AS1$, we then enable NDN for its end of the peering pipe as well as for other logical pipes internal to $AS1$. Following this, we install the NDN software on the $AS1$ TPs and reconfigure the pipe terminus to send NDN packets to it (in our prototype, software L3.5 code runs in containers, and we envision a standard container-based "package format" for L3.5 processing code in the future).

The final step is to update $AS1$'s bootstrap daemons to advertise NDN as an available L3.5. After this, Trotsky-capable hosts attaching to $AS1$ see NDN as available and are able to make use of it (provided they also have NDN installed).

This is all it takes to deploy new L3.5s. In a large domain, changing configurations obviously poses additional challenges (*e.g.,* testing, training, etc.), but our point is that Trotsky removes all the *architectural* barriers to extensibility.

4.6.4 Overhead Microbenchmarks

We evaluated the performance overhead of the TP design presented in Section 4.5 for two different L3.5 designs: IPv4 (in hardware), and NDN (in software). We used another server as a traffic generator and compared two setups for each L3.5 design: with and without Trotsky. For Trotsky, we encoded the logical pipe next-header field using a VLAN tag.

Our experiments show no measurable difference in the throughput and latency with and without Trotsky for both L3.5 designs. For IPv4, in hardware, the roundtrip latency is 2µs in both cases and the link became saturated (10Gbps throughput). Using the NDN software, roundtrip latency is around 1.2-1.8ms and the median goodput is 49Mbps in both cases. Thus, the presence of Trotsky does not impose significant performance overheads.

4.6.5 Viability of Software L3.5 Implementations

One might worry that software L3.5 implementations will always be too slow. To provide some context for this issue, we ran some software IP packet forwarding experiments using BESS [45] on an Intel® Xeon® E5-2690 v4 (Broadwell) CPU running at 2.60GHz. We achieved roughly 35Mpps on a single core; roughly 18Gbps for minimal-sized packets. The server has 28 cores and cost roughly $10,000, so this is roughly 50Gbps per $1,000. While this is clearly less than one can achieve with a networking ASIC, the question is whether it is fast enough to be viable.

In terms of handling the load at a single interdomain connection point, the measurements above show that it would only take six cores to provide 100Gbps IP connectivity for min-sized packets. This is clearly feasible, but is the overall cost of provisioning cores at the edge too much? While there is little public data on the load entering a domain, the best information we can collect results in an estimate on the order of 10Tbps. To handle this load with minimum-sized packets, we would need to spend roughly $200,000 on servers to handle the *entire* load entering the domain. This number contains many approximations (*e.g.,* moving from min-sized packets to average sized packets would reduce the cost by an order of magnitude). But with the capital budgets of large carriers running into the billions of dollars (up to tens of billions), this number will likely be a tiny fraction of their overall costs.

Thus, it seems eminently viable for a domain to use software packet processing to handle their incoming load for L3.5 designs with computational requirements on par with IP (which we used as a baseline because it is well-known and relatively – but not trivially – simple). Performance numbers would obviously be significantly worse for L3.5 designs that require extensive cryptography or other computation. However, we note that even for IPsec, one can achieve roughly 20Gbps per core with MTU-sized packets. Moreover, we envision that it is only the *initial* deployment of an L3.5 design that is in software; once it becomes widely and intensely used, we assume hardware support will become available if needed.

4.7 Related Work

While this chapter has referred to the Internet architecture as having cleanly defined layers, the actual usage patterns of the Internet are far more complicated, as pointed out in chapter 1. As that chapter pointed out, a recent paper [121] examines the actual usage, and it observes that it might be best to think of the Internet as an exercise in composing networks rather than being a static set of layers. We agree with this sentiment, but think that making the distinction between intradomain and interdomain data planes remains crucially important in enabling an extensible Internet. Domains will presumably continue to use complicated sets of overlays to, for instance, carry wireless traffic over the public Internet, but this does not change our basic observation that separating intradomain from interdomain (*i.e.,* differentiation between an L3 and an L3.5) will allow new interdomain designs to be incrementally deployed and for multiple of them to coexist.

There have been several other approaches to architectural innovation. The earliest, and most incrementally deployable, is that of overlay networks. However, in most overlay approaches the process of constructing the overlay network (and eventually transitioning from overlay to "native") is typically ad hoc. By contrast, in Trotsky the overlay is fully intrinsic. More precisely, of course one can use a purpose-built overlay to deploy a new clean-slate architecture. Trotsky is a framework in which the interdomain delivery is an intrinsic overlay that allows all such clean-slate designs to be deployed straightforwardly and simultaneously.

MPLS can be considered an *underlay*, which for some domains handles most intradomain delivery but leaves interdomain delivery to IP. However, in contrast to Trotsky, MPLS is strictly a mechanistic separation, with IP remaining as the universal L3 design.

Beyond overlays and underlays there are several interesting clean-slate proposals for architectural change. Active networking [113] innovated on the data plane, but did not address the issue of the NetAPI or interdomain interactions. Nebula [8] offers a great deal of extensibility in network paths and services, which is an important dimension. However, the core of the architecture (*i.e.,* the data plane) is universal within and across domains and therefore hard to change.

Plutarch [28] represents an entirely different approach to evolution, stitching together architectural contexts, which are sets of network elements that share the same architecture in terms of naming, addressing, packet formats and transport protocols. These contexts communicate through interstitial functions that translate different architectures. It is interesting to contrast this with the current Internet, which was founded on two principles. The first is that there should be a universal connectivity layer, so that in order to support n different internal network designs (what we call L2 today) we would not need n^2 translators. The second is the end-to-end principle, which pushes (to the extent possible) intelligence to the edge. We feel that Plutarch runs counter to these two principles, requiring architectural translations within the network to achieve evolvability.

XIA [44, 87] enables the introduction of new service models by defining *principals*, and XIA itself defines a number of broadly-useful classes of identifiers. To cope with partial deployment, XIA relies on a directed acyclic graph in the packet header that allows the packet to "fall back" to other services that will (when composed) provide the same service. For instance, a DAG can have paths for CCN [58] and a source route via IP, with edges permitting intermediate routers to select either. Thus, XIA's approach to partial deployment (which is a key step in enabling evolution), much like Plutarch before it, is to require translations between architectures at network elements that understand both. In this respect, both Plutarch and XIA deploy new architectures "in series", and any heterogeneity along the path is dealt with by having the network explicitly translate between architectures. In contrast, in Trotsky, one simply uses any of the L3.5 designs mutually supported by both endhosts and their domains, which is an end-to-end approach. Naming systems can provide hints about how hosts and objects can be reached (*e.g.,* a name might be tied to one or more L3.5 designs), but the network does not try to translate between designs.

In terms of the many proposed architectures arising from clean-slate research (such as NDN or SCION), none are incrementally-deployable general frameworks for deploying arbitrary architectures, so our goals are different from theirs. However, Trotsky could serve as a general deployment vehicle for all of them, so they need not explore deployment mechanisms on a design-by-design basis.

Sambasivan *et al.* observe that were it possible to make two clean-slate changes to BGP, it would then be possible to evolve routing thereafter [108]. Their work highlights the value of enabling architectural evolution, and presents an alternative path to achieving it in the context of interdomain routing. ChoiceNet [118, 104] aims to improve the Internet's architecture by fostering a "network services economy", focusing on service offerings in a competitive marketplace; this could be complementary to Trotsky.

The work in this chapter follows from a line of research begun by the FII [66] proposal, which was also concerned with architectural evolution, though through a non-backwards-compatible clean-slate approach. In addition, FII focused largely on specific solutions to routing (specifically Pathlet routing [40]) and security, rather than general interdomain services. A similar but more general approach was taken in later work [39, 100, 101], which this work continued to expand on (while also shifting to embrace backwards compatibility).

Thus, to our knowledge, Trotsky is the first attempt to make a backwards-compatible change in the current architecture which then enables incremental deployment of radically different architectures (which need not be backwards compatible with any current architecture). Whatever one thinks about clean-slate design, the ability to make a single backwards-compatible change in the infrastructure (migrating to Trotsky) – one that is conceptually simple and mechanistically mundane – that then unleashes our ability to incrementally deploy new architectures is a significant step forward.

4.8 Conclusion

In our design and the implementation of our prototype, we were guided by six axioms, which we now review.

First: Trotsky's basic goal of architectural extensibility is *desirable*. This derives from (i) our belief that despite innovation at higher layers (*e.g.,* recent developments such as SPDY and QUIC) there are more fundamental architectural improvements that we would like to deploy – ranging from security mechanisms [122, 7] to ICN designs [58, 65] to mobility enhancements [110] to service-centric architectures [89] to more readily available Edge Computing [94] – and (ii) that there are currently high barriers to the deployment of these new architectures.

Second: Trotsky's approach is *novel*. In particular, we are the first to observe that making the interdomain data plane an inherent overlay over the intradomain data plane would both (i) be incrementally deployable and (ii) enable the incremental deployment of radical new architectures. As such, Trotsky is not just a typical ad hoc overlay, but a structural rearranging of the Internet architecture to create an intrinsic overlay.

Third: Trotsky is *effective*. We have shown via implementation how Trotsky can seamlessly deploy a wide range of architectures and demonstrated that applications can leverage the resulting architectural diversity. Moreover, Trotsky can support any L3.5 design that could have been deployed as a clean-slate L3 design on routers, so there is no significant limit to its generality. While we cannot eliminate all operational barriers, Trotsky does eliminate the architectural barriers to extensibility.

Fourth: software packet processing, as currently being deployed in NFV and Edge Computing, is a key enabler for easy initial deployment. There is no question that software processing is far slower than hardware forwarding, but there is also no question that software packet processing is sufficient to handle the load at the edge for simple L3.5 designs, particularly when they are first deployed.

Fifth: *enabling architectural extensibility could greatly change the nature of the Internet*. Trotsky can transform the Internet not just because it can deploy multiple coexisting architectures, but also because it allows applications to use a diversity of architectures. This means that individual L3.5 designs need not meet the needs of all applications, only that the union of L3.5 designs can meet those needs in a way that is superior to today's Internet. This greatly lowers the bar on what new architectures must do in order to be a valuable part of the overall ecosystem.

Sixth: Trotsky may change the *incentives* surrounding architectural change. Our motivation in designing Trotsky is to show that some long-held beliefs about Internet architecture, such as the necessity of a narrow waist and the inherent difficulty of architectural evolution, are wrong. But Trotsky will remain merely an academic exercise if there are no incentives to deploy it, or to adopt new L3.5 designs once Trotsky itself is deployed. While the long history of architectural stagnation and lack of carrier innovation dampens our optimism, we do believe that the incentives now may be more favorable than in the past. The ISP business is rapidly becoming commodity, and carriers are desperate to offer new revenue-generating services. However, such services are of limited value if they can only be offered by a single carrier. Because L3.5 designs can initially be deployed in software running in the TPs, Trotsky provides a way for carriers to offer new services – in the form of new L3.5 designs – without endless standards battles and long hardware replacement cycles. Moreover, these carriers can easily peer with other carriers supporting those services. Only if such a design attracts significant traffic does the carrier need to contemplate more extensive support (in the form of hardware or internal deployment). Thus, Trotsky may play a crucial role creating the incentives for deploying new cross-domain services (in the form of L3.5 designs). In so doing, it would enable a permanent revolution in Internet architecture.

Chapter 5

Packet State Load Balancing

Preamble

This chapter warrants some additional context. The work described here began in 2017 shortly after the presentation of SilkRoad [84] at SIGCOMM. SilkRoad is a stateful load balancer implemented in P4 and meant to run on programmable switching hardware. It is a marvelous piece of engineering, but while reading it, this author was struck by just how much good work (on load balancing generally – extending to things like Ananta and Maglev [95, 34]) was being expended to address a symptom without addressing the underlying cause.

As touched on in chapter 1 (and elaborated on in other work [89]), a core problem in server load balancing stems from the fact that many services in use are no longer implemented by *a* server, but by a pool of servers – a stark difference from the dominant view at the time when the Internet protocols were created (indeed, at the time, the server – or host – itself was often considered *to be the service*). Thus, the Internet protocols have no good mechanism for dealing with this issue. Considering the problem in the context of a connection-oriented service, we can see that it actually runs fairly deep. The association of a connection to a concrete server must be made (i) at the service side, (ii) at the time of connection. Once this association is made at the service side, packets from the client side should honor it for the rest of the connection. There are three main classes of solution to this problem. First, the association could be hidden from the client entirely, leaving it up to the service to maintain. This has been the typical solution and has a number of problems, as described later in the chapter. Second, during connection establishment, the service could inform the client of a new address (of a concrete server) to use for the connection. Third, the service could inform the client of an *additional* address that corresponds to the concrete server, creating a hierarchical address containing a service portion and a server portion, where the latter is provided by the service during connection establishment. These latter options have advantages, but would

– at first glance – seem to require additions to the Internet protocols which may be difficult to deploy (*e.g.,* because no suitable facilities for switching the address of an existing connection exist).

However, the third option asks very little of the client side of the connection – simply that with every packet after the first, it should include an additional address given by the service. I realized that we could repurpose an existing mechanism to accomplish this. My colleagues and I pursued this idea, which we called PSLB, and this chapter is a description of this work at a relatively early state, much of the text originally having been intended as a first conference submission. That initial submission was rejected and we opted not to pursue the matter further for two reasons. First, we discovered that the the core of the idea had already appeared in a workshop paper [91], but we had missed it in our original literature review because – while the technique is more general – the prior work was aimed specifically at Multipath TCP. Secondly, it became clear that a similar idea was being explored simultaneously in the context of the emerging QUIC protocol (indeed an Internet draft specifically on this subject followed shortly later [33]). In brief, we no longer felt that the main technical contribution was sufficiently novel to warrant continued effort.

After discontinuing work along these lines as an end in themselves, we did use our familiarity with the subject as part of a larger argument about the proper place of programmable forwarding hardware, which was published as an editorial [79]. In the editorial, we attempted to stoke discussion by taking a good-natured but fairly extreme position against the use of programmable switching hardware for doing much more than switching whenever alternatives existed (as discussed below, PSLB may leverage P4, but only for basic switching; this is a far cry from the masterful, but – we argued – unnecessarily complex use of P4 by SilkRoad). Whether we were successful in stoking that particular discussion is up for debate, but our brief mention of the PSLB approach *was* successful in providing another group with inspiration to pursue the approach further, which resulted in a publication [11]. Ultimately, their work is more complete and more polished, but PSLB is included here – in its relatively early form – as it does contain some discussion not found elsewhere.

Server load balancing is a cornerstone of the modern Internet. Roughly defined, it is when the connections to a service are spread across multiple backend servers in order to provide a service that scales larger than a single server could handle. While this seems simple enough, it represents a rather significant departure from the expected usage of the protocols underlying the Internet. These protocols were intended to allow access to *hosts*; the concept of an *abstract serivce* larger than a single host is not reflected in their design. Thus, server load balancing can be seen as an attempt to shoehorn the modern usage into the existing protocols by providing the illusion that an abstract service is just one big server – when in fact it may be implemented by a large and dynamic pool of servers.

An ecosystem of load balancing solutions has long existed which spans a spectrum from simple open source solutions to expensive and featureful proprietary load balancing appliances. In recent years, there has also been a growing literature of solutions from large web service providers (*e.g.,* [95, 38, 34, 84]) who have found that existing solutions do not meet their needs for scale, cost, reliability, and performance. Load balancers may operate at different network layers. One common type are application layer load balancers, which typically operate on HTTP. We set aside that case, and focus instead on lower layers in this chapter – in particular, we focus on the transport layer (L4), and give particular attention to the dominant case, which is load balancing of the TCP transport layer protocol.

One key measure of such load balancers is the degree to which they maintain the property of *Per Connection Consistency* (PCC). When a connection arrives, it is assigned to a backend server based on some policy (which may be a complicated load-balancing algorithm or simple hashing). Once this decision has been made, *each packet from the connection should consistently be sent to that server*; otherwise the connection will break. The traditional approach to ensure this property is to record where the connection was sent in a *connection table*, and use that state to forward all subsequent packets in the connection. This approach worked well for many years, but at the scale of a modern web service, it suffers from two main problems.

First, because all packets must go through a load balancer to reach the correct backend server, the bandwidth and state requirements for load balancing are substantial. In fact, the first version of one of the aforementioned solutions from the literature required an infrastructure investment of over 10 million dollars to support its traffic – and this was a purpose-built solution meant to improve upon off-the-shelf solutions! (A later revision to this approach reduced the cost substantially [38].)

Second, one can violate the PCC requirement when a load balancer fails. Consider a connection that was initially handled by a particular load balancer, but then that load balancer fails while the connection is still active. When the next packet of the flow arrives, it will be sent to a different load balancer that has no prior state for it, which will then direct the flow to the backend server dictated by the current policy. If the policy decision has changed since the connection started, due to various changes in load or infrastructure, then a PCC violation will occur.

The root of these problems is the lack of *fate sharing*. The state that determines where a connection should go resides not with the entities directly involved (*e.g.,* the connection's endpoints), but in intermediate nodes in the network itself. To address this, we have devised a load balancing strategy where the state is carried within the packets of the connection itself, not retained in a load balancer. The design is quite simple, and (as we describe later) achieves better performance and correctness than traditional approaches. However, the real challenge is retrofitting the approach into today's protocols.

In the remainder of this chapter, we provide further background and discuss related work (in Section 5.1), describe and analyze our approach at the abstract level (in Section 5.2), dive deep into the details of realizing our approach today with unmodified client software (in Section 5.3), examine security implications of the approach (Section 5.4), describe a prototype implementation (in Section 5.5), and finally compare to two other recent approaches via simulation (in Section 5.6).

5.1 Background and Related Work

A service to be load balanced is often referred to as a *VIP*. Although this stands for Virtual IP, it typically also incorporates the port number on which the service is running. A *DIP* is the Direct IP of a backend server actually implementing the service, and the term is often used as a synonym for a server. Each VIP has a *DIP pool* of servers implementing the service. A *policy* for the VIP determines the method of assigning new connections to a particular DIP. Classical policies include, for example, round robin, weighted round robin, least connections, and least response time. For reasons we discuss later, modern load balancing systems operating at scale typically implement a policy based on unweighted hashing. Temporary adjustments to policy may also be desirable. For example, if a rack is due for maintenance, it is desirable to implement a "draining" policy, where new connections are not assigned to DIPs in the rack and existing ones are drained out of the system before the rack is shut down. *Health checks* evaluate the health of the DIPs, removing them from DIP pools when they are dead, and possibly influencing policy (*e.g.,* reducing the weight of an unhealthy DIP).

An individual load balancer typically works as sketched in the introduction to this chapter. First, packets to a VIP arrive at the load balancer. If the load balancer has no state for the connection in its *connection table*, then the policy is applied to choose a DIP (we call this sending the packet to the *policy engine*). The result of the policy is stored in the connection table (*e.g.,* associating the connection's 5-tuple with the selected DIP). The packet's destination is either rewritten to the DIP's address or the packet is encapsulated in a new IP header containing the DIP's address, and the packet is then forwarded to the DIP. When packets arrive and their state is found in the table, the process is the same except that the policy engine portion is skipped. In many modern systems, packets sent back to the client employ *direct server return*, meaning that they need not traverse the load balancer on the way back, allowing possibly fewer hops or less load on the load balancer; this typically requires that the DIP sends packets where the source is the VIP address (rather than it being the DIP address as it would usually be, and relying on the load balancer to rewrite it).

While load balancers were traditionally deployed as a single appliance (or an appliance with a hot standby), this approach only scales up so far, and more recent work in the literature either examines *scale-out* approaches, or approaches with intrinsic scaling properties. Ananta [95], Maglev [34], and Facebook's L4 load balancing infrastructure [111] are examples of the former. Rather than have fixed load balancers, they implement the load balancing functionality in software running on servers from the same pool of servers running the services being load balanced. Typically, ECMP hashing is used at the gateway router to spread traffic into the load balancers. Duet [38] and SilkRoad [84] are examples with intrinsic scaling. That is, these designs reuse switches that already exist in the data center (Duet using them in conjunction with servers in a hybrid design); the number of switches has a relationship with the amount of servers and traffic being load balanced, so as the network is scaled, so is the load balancing capability.

In the chapter introduction above, we mentioned the issue of how a failed load balancer will cause existing connections to be handled by another load balancer which does not have a connection table entry for the connection (connection tables are rarely synchronized between load balancers, *e.g.,* due to performance issues this would cause [34]). When ECMP is functioning as a “pre-load balancer load balancer”, one may suffer a similar problem any time the ECMP is disturbed, *e.g.,* by changing link state or even *adding* a load balancer. Since the policy must be reapplied when this occurs, these systems all limit themselves to a policy based on some form of consistent hashing of the connection’s packet headers (e.g., its 5-tuple). The benefit of such policies is that they increase the odds (but do not ensure) that reapplying the policy will yield the same result and thus maintain per connection consistency.

While any hashing would be sufficient if the DIP pools were static, consistent hashing improves the behavior even when DIP pools are dynamic – which they are. For example, one service provider found that almost a third of their clusters experience at least 10 updates per minute 1% of the time, and some even experience 10 or more updates *half* the time [84].

5.2 PSLB in the Abstract

At first glance, it would seem that the connection table is the solution to the problem of per-connection consistency: even when the policy (or hash) would fail to assign all the packets of a connection to the same DIP, the connection table ensures that they are. However, the cracks in this line of thought show as soon as one considers more than one load balancer: now one needs to ensure that packets reach the correct load balancer (and thus, the correct connection table). What provides *this* consistency? Another connection table? And the consistency for *that* one? Perhaps the solution is to replicate the connection table across all load balancers. This – if one can even justify the additional traffic for replication – only serves to call further into question an issue that arises from even a single connection table: can you afford the cost of state which scales with the number of connections?

In truth (and despite decades of load balancing practice), the connection table is the problem and not the solution. In his seminal retrospective on the design philosophy of the Internet, Dave Clark elaborates on exactly the issues at play here [26], and coins the term “fate sharing” to describe the property of systems that avoid them. Designs which exhibit good fate sharing avoid crucial state being collected at arbitrary intermediate nodes, instead coupling the state directly to the interested entities, *i.e.,* to the communication endpoints and possibly the packets flowing between them. In Clark’s words, “[t]he fate-sharing model suggests that it is acceptable to lose the state information associated with an entity if, at the same time, the entity itself is lost”. A corollary which generally holds is that if the entity itself is not lost, then the state should not be lost. It is this principle which underlies our approach, which we term Packet-State Load Balancing or PSLB, as the state is carried in each packet.

With the basic direction of our design now clear, we can begin to sketch out its details.

5.2.1 Design Sketch

In PSLB, the per-connection state is not stored in a connection table on an intermediate node, but is carried within every packet of that connection. When the state within the packet is lost, it is because the packet itself has been lost (that is, dropped) – and the loss of state is moot (a

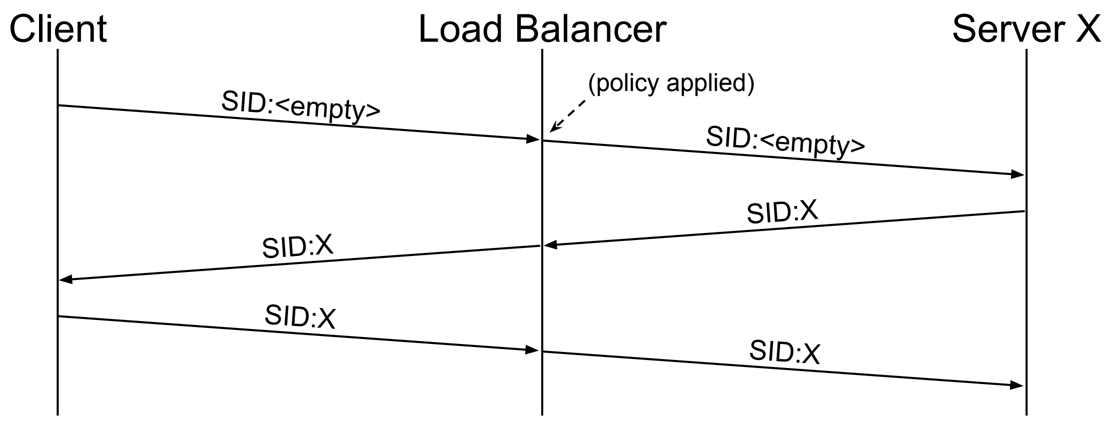


Figure 5.1: *The initial packet from the client contains no Server ID, so the Load Balancer applies a policy to select a server. In the first reply from the server, the server inserts its own ID, which the client then echoes in subsequent packets. When packets with a SID arrive at the Load Balancer, they are sent directly to the appropriate server without ever invoking the policy engine.*

corner case which invalidates this assertion and its remediation is discussed in Section 5.3.4). Additionally, wherever the packet goes – the state is there with it. In such a system *perfect per-connection consistency is trivial*, because the packets making up the connection and the state with which they must be consistent *are one and the same*.

To be somewhat more concrete, we propose the addition of a new packet field, the SID or Server ID. This field simply refers to the DIP handling the connection. Indeed, one might imagine simply placing the server's IP (the DIP itself) into the packet. This is conceptually equivalent, though we do not do so for reasons discussed in Section 5.2.3. Thus, we use SID as a more general term for something which refers to a backend server, but is not necessarily the server's actual direct IP address.

When a client first connects to a service, the SID is empty. The load balancer can choose a DIP using whatever policy it likes. When the first response from the service is sent back to the client, the response includes the DIP's SID. The client stores this state locally as part of its connection information, and in every subsequent packet, the client includes the SID. When these packets – with a non-empty SID – are seen by the load balancer, the load balancer does not invoke the policy engine again, but simply forwards the packets to the appropriate DIP based on the SID field. This approach is depicted in Figure 5.1.

5.2.2 Positive Implications

A first implication which falls out of this design is simply that the load balancers require much less memory, as load balancer memory requirements are dominated by the connection table. Moreover, the memory requirements are much easier to understand and provision, as they are no longer a function of the traffic they are handling.

Secondly, note that since the state tying a connection to a DIP is now stored at the client and in the packet instead of a connection table, even if there are multiple load balancers, it no longer matters to which a particular packet goes, so long as they share the same interpretation of the SID.

Continuing that argument, note that there is no longer a need to constrain oneself to hash-based policies. The benefit of hash-based policies is that any packet in the connection can be used to (with some probability) reconstruct the state should the state have been lost. In PSLB, the state can never be lost except in cases which render the loss moot anyway due to fate sharing. Thus, PSLB achieves *policy generality*. This means, for example, that PSLB can retain perfect PCC while executing valuable policies such as draining DIPs (so that a service can be scaled in) or dynamic DIP weighting (particularly valuable for services in which, *e.g.,* the CPU load varies greatly across requests), which is not possible with a simple consistent hash based policy.

5.2.3 Negative Implications

We began this section with a principle-based argument in favor of PSLB, and we wish to finish it by beginning to address a principle-based argument that might be made against it. Specifically, Bestavros et al. [14] articulate the principle of transparency with respect to load balancing:

> *Transparency:* Clients should not be exposed to design internals. For example, a solution that allows a client to distinguish between the various servers in the cluster – and hence target servers individually – is hard to control.

If the SID were indeed just the backend server's direct IP address or something with a clear mapping to a particular DIP, then PSLB would clearly be in direct violation of transparency – a violation which would come with implications for security. In a case of terminological irony, however, we argue that by sufficiently obscuring the meaning of the SID, we ultimately can achieve transparency of the system as a whole. In other words, if a client sees the SID but cannot derive meaning from it, the system remains transparent. We describe our approach to doing so in Section 5.4.

5.3 PSLB and TCP

In this section we look at making the abstract design described in Section 5.2.1 practical by mapping it to TCP. Crucially, we desire to do this without requiring changes to client operating systems.

The semantics of the Server ID or SID field are simple: the server returns its value in its first reply (the SYN+ACK in the context of TCP), and the client inserts the value in every subsequent packet it sends. TCP once had TCP options which achieve similar semantics: the TCP echo and echo reply options from RFC 1072 [57]. These allow one side of a connection to place a 32 bit value in an option, and the other side will then echo it back in another option. These options were later obsoleted by RFC 1323 [56] (later updated by RFC 7323 [17]), which introduced the TCP timestamp option.

The TCP timestamp option is much like an echo request and an echo response stuck together: if one assumes that both sides want to use the options, this saves between two and four bytes (depending on alignment). The primary use of these options is the same: to “allow every segment, including retransmissions, to be timed at negligible computational cost” in order to improve and simplify RTT estimation. In short, a timestamp is put in a sent packet, the recipient echoes it back, and the initial sender subtracts the echo from the current time to yield the RTT.

To implement PSLB with TCP, we propose re-purposing this field (or part of it) to carry the SID rather than timing information. In the rest of this section, we detail our proposal and examine its implications and potential pitfalls.

5.3.1 Basic Approach

Our approach to TCP-based PSLB involves two major parts. The first part, as mentioned, is to use the TCP timestamp to carry the SID. If packets leaving a data center contain the SID in the timestamp value field, packets returning from unmodified clients should contain the SID in the timestamp echo field. The second part is that the timestamp must determine the server to which packets are sent. The TCP timestamp option, of course, is not generally a routable field in switches, of course, but we assume that either minor changes can be made to switches, or that switches contain programmable data planes which facilitate this (and in Section 5.5, we describe a prototype implementation for such programmable switching hardware).

Minimally, the requirements from the switch are:

1) A load balancing policy engine is run on the switch’s control plane to choose the DIP for each new connection.
2) Switches identify ingress SYN packets, which do not yet have a SID, and forward them to the policy engine.
3) The SID is inserted into the TCP timestamp option of egress traffic based on the source address (*i.e.,* the DIP address).
4) The SID is read from the TCP timestamp field of ingress traffic and the packet is forwarded to the corresponding DIP.
5) The timestamp echo field is set to zero before forwarding a packet to the DIP.

Although we refine this later, it may be useful at this point to think of the SID as a field of some number of bits which maps to the IP address of a specific DIP in a relatively static set of *potential* DIPs. That is, individual DIPs can be enabled or disabled within this set (*i.e.,* the DIP pool is dynamic) but the set of possible DIPs is relatively stable. A trivial way to implement this is that the SID is the suffix of the DIP’s IP address, and the load balancer adds the correct prefix. A more flexible approach is that it is an index into a table of IP addresses held by the load balancers (this is the basic technique used by our prototype modulo changes made for security, described later). This allows one to alter the set of possible DIPs by having the policy engine “drain” one (stop assigining new connections to it and wait for existing ones to terminate) and then replacing it. The timescale involved is such that maintaining a consistent set of active DIPs across multiple load balancers is not difficult.

The first item in the list above assumes the policy engine is run on the switch's CPU. We believe this general approach to be desirable, as it allows the policy to be fully general. The policy engine could, for example, actively monitor DIP health in arbitrarily complex ways, or coordinate with other load balancers. This assumes that the switch CPU is capable of keeping up with incoming requests since the policy engine only need be invoked for the first packet. Other designs are possible; for example, the policy engine could be on a separate server or distributed on several servers. Of course, if the policy requirements are simple (*e.g.,* just a statistical spreading that can be achieved with hashing), it could be implemented entirely within the switch data plane.

One might note that the formulation above varies slightly from that depicted in Figure 5.1. In the figure, it was the server which inserted the SID into the outgoing packet, whereas in the formulation above, it is the load balancer. While either is certainly possible, the advantage of the latter is that not only are no changes to the client TCP stack required, but *no changes to the server stack are required either*, at least assuming Linux servers.

One might wonder how this last bit is possible. Given that we are interfering with the server's timestamp (though, note, not the client's), surely the server will in some way object. Indeed, were we to simply pass an arbitrary, modified TCP timestamp back to the server, it could certainly interfere with the server's RTT estimation. Fortunately, we can resolve this in the load balancer. After the load balancer uses the SID encoded in the timestamp option to determine the DIP, the load balancer can simply zero the echoed timestamp field. As it happens, Linux ignores the field when its value is zero, and it will go back to pre-timestamp RTT estimation. While this may not seem ideal (and, indeed, it may not *be* ideal), there exists (possibly dated) research that calls into question the benefit of the timestamp approach [5], and it turns out that a fair portion of current traffic does not use the timestamp approach anyway (we discuss the reason for this in Section 5.3.4). Moreover, in Section 5.3.3, we discuss the possibility of reducing timestamp granularity rather than eliminating timestamp data altogether.

A reader might be concerned that this approach – where the switch rewrites the timestamp – means that one cannot employ direct server return. We do not see this as problematic, as the switches are on-path to begin with; it does not require extra hops for a packet to return through the switch, nor are returning packets likely to be a performance problem for a hardware switch (as opposed to a software load balancer). Of course, having the servers insert their own SIDs is a viable alternative, but we saw no advantage, and doing so would require modifying the kernel on the servers, which we saw as a disadvantage.

While the approach here is fairly straightforward, this is not to say that it is entirely without some complexity. We now turn to a more detailed look at the implications.

5.3.2 PAWS

While the server may not actually need its timestamp echoed for the purpose of RTT estimation, the timestamp option has other uses, and we must ensure that our manipulation of the timestamp does not cause irreparable harm with these other uses either. Perhaps the most significant of these is the PAWS or Protect Against Wrapped Sequences mechanism, also defined in RFC 1323. The primary purpose of this mechanism is to allow TCP to function adequately when used with much faster links than when TCP was designed. The fundamental issue is that the TCP sequence space is only 32 bits. At 10Gbps, this space can be wrapped in a few seconds. At 40Gbps, it can be done in less than a second. Should the sequence space wrap, a delayed duplicate from before the wrap could corrupt the stream. With the assumption that a sender's timestamp value is nondecreasing (within its own 32 bit space), it can essentially be used as an extension of the sequence space. Note that for this to work, each side is watching not the echo of its own timestamp, but the timestamp of the *other* side. This prevents one side from putting truly arbitrary values in their timestamp; since it functions like additional sequence number bits, it must be non-decreasing or a packet may look like a delayed duplicate and be rejected.

Ultimately, our questions are: how can we encode the SID into the timestamp field while keeping the PAWS mechanism happy, and how many SID values can we encode (*i.e.,* how large of a DIP pool can we support)? The answer is that it depends. To understand on what it depends, we need to understand the details of PAWS's requirements on the timestamp.

It turns out the requirements are rather lax. The value must be non-decreasing (within its 32 bit modulo space). It must not increment "too slow": it must change value quickly enough that different wraps of the sequence space are differentiable by different timestamp values (thus, a 1 second timestamp frequency would work for links up to around 8Gbps, and a 1ms frequency could support links up to 8Tbps). It also must not increment "too fast". As the receiver does not know the timestamp frequency, it must "see" the timestamp make forward progress; that is, two subsequent timestamps must not be separated by more than half the timestamp space. One case where this might happen is if the connection is idle for a long period. After such an idle period, the receiver – ignorant of the timestamp frequency – has no way of determining whether the value has wrapped, and how many times it may have wrapped if so. To resolve this, the RFC authors assume that the fastest timestamp increment is once per millisecond. Under this assumption, it would take approximately 50 days for the timestamp to wrap, so the shortest time after which timestamp ordering is ambiguous is about 25 days. Therefore, PAWS will never reject a packet based on its timestamp if it has been idle for more than approximately 25 days.

Given the above, it is clear that we should encode the SID into the low bits of the timestamp, and leave the top bits to represent time. Doing this means the field still "looks like" a 32 bit timestamp in that it can be made to be non-decreasing (modulo wrap), and when it wraps, it has the expected behavior (*i.e.,* the top bit changes) – it is just that the bottom bits will remain the same; we have lost granularity. By way of example, Linux currently has a timestamp frequency of 1ms. If we simply replaced the bottom 13 bits with the SIDs (allowing 2^{13} or 8,192 DIPs), then the timestamp portion would change approximately once every eight seconds, allowing for links up to approximately 1Gbps. The timestamp would still wrap after about 50 days, so the PAWS "idle safety net" would still work as expected.

If we wanted a 14th bit for SIDs, we could simply drop another timestamp bit, though this would change the maximum supported link speed to approximately 500Mbps. If this were unacceptable, we could shift the time bits left one place. This would ensure that the timestamp changed once every 8 seconds again, thus allowing 1Gbps links again. However, it would now wrap in approximately 25 days. Shifting the 1ms clock makes it appear like a 0.5ms clock. This breaks the idle safety net mechanism and it would now be possible for a connection to get "stuck" until the timestamp progressed far enough again. The remedy here is simply to not let connections go idle for so long that this becomes problematic. In the example here, it is unlikely it ever would be – few connections last (much less are idle for) 12.5 days. However, if one needed to support faster links or more DIPs, this may become a concern. The most straightforward solution is to simply turn on TCP keep-alives (on the servers in the case of PSLB) and set the interval such that it would send packets often enough that the timestamp wrap would never be ambiguous.

5.3.3 RTT Estimation

Earlier in this section, we explained that the load balancer could prevent Linux servers from interpreting the (altered) timestamp echo by setting it to zero. With the more detailed discussion of timestamp manipulation in the previous subsection now covered, it is worth pointing out that one does not necessarily need to prevent the server from interpreting the field as a time. If the load balancer sets the SID bits to zero – leaving the upper bits alone – it still functions as a timestamp of lower granularity. Subtracting it from the current timestamp will produce a conservative (longer) measurement of the RTT. Depending on how the bits are allocated, this may still provide sufficient signal to the RTT estimator to be worthwhile. As RFC 1323 says, the timestamp "does not need much more resolution than the granularity of the retransmit timer, e.g., tens or hundreds of milliseconds".

5.3.4 Missing Timestamps

PSLB with TCP relies on TCP timestamps. Unfortunately, not every packet has a timestamp. In this subsection, we look at some of the cases where this is so in order of increasing severity, and include discussion on how these cases can be addressed.

The BSD/Apple Bug

According to the TCP timestamp RFC, when the timestamp option has been negotiated, all packets in the connection (with the possible exception of RST packets) should have timestamps. In practice, a bug in the BSD network stack (a bug which Apple seems to have inherited) prevents it from including the timestamp in TCP keep-alive messages. This was fixed in FreeBSD in March of 2016, but had not yet been fixed in macOS as of the middle of 2017.[1]

Until the bug is fixed, this will prevent TCP keep-alives sent from macOS clients from reaching a DIP, which *could* in turn cause servers to terminate connections which appear idle. However, the more common use for TCP keep-alives is to keep state alive in middleboxes like firewalls, NATs, and load balancers which evict state corresponding to idle connections. Firewalls and NATs, however, will typically be between the client and load balancer. Thus, dropping the keep-alive at the load balancer will not have a negative impact (and the PSLB load balancer, of course, has no state to keep alive in the first place). This would, however, be problematic if macOS were being used for the server, and if keep-alives were being used for the purpose described in Section 5.3.2.

[1] Apple was aware of this bug in 2017, so it may be fixed now; we have not checked more recently.

Fragments

When TCP packets are fragmented, the IP header is duplicated in each fragment, but the TCP header is not. Thus, when a fragment without the timestamp arrives at a PSLB load balancer, the state required to forward the fragment is not immediately available.

As one cannot tell whether a connection will contain fragmented packets, one cannot simply use a policy like "for all connections with fragments, fall back to hashing based on source IP and port". Rather, in order to perform correctly, a load balancer must identify fragments, read the timestamp/SID from the fragment that contains it, and then re-associate each fragment with that value and forward it accordingly. This re-association process is not terribly difficult, and the state required to re-associate the fragments is extremely short-lived (indeed, it is even simpler and requires fewer resources than packet reassembly). We suggest that when a packet with the More Fragments bit set or a nonzero Fragment Offset field arrives in the load balancer data plane, the packet be forwarded to the control plane where the re-association process be done in software. If the load balancer control plane is insufficiently provisioned to reassemble the number of fragments received, an alternate design could forward all fragments to one or more "re-association servers" instead.

No timestamps and Windows

Unfortunately, not all TCP connections carry the TCP timestamp option. There are various reasons for this: middleboxes may strip or zero them, hosts may have been specifically configured to disable them, embedded TCP stacks may not have them or may not have them enabled by default, and so on. However, the most likely reason (especially from the perspective of modern web services) is simply that the client OS is Windows.

Whether TCP timestamps will be used for a connection is determined during the SYN and SYN+ACK of a connection. If the connection originator supports and desires to use them, it inserts the option into the SYN. If the recipient supports and desires to use them as well, it includes the option in the SYN+ACK. Significantly, the recipient is prohibited from sending the option in the SYN+ACK if it was not received in the SYN.

The current versions of macOS, iOS, Linux, and Android all send the timestamps option by default when initiating a connection, however, current versions of Windows do not. To be clear, current versions of Windows *support* the timestamp option, and servers on Windows will, by default, happily enable the option if the originator requests it – Windows just will not request it by default when originating connections (nor will current versions respond to a timestamp option that is received unsolicited in the SYN+ACK, although past versions did [64]).

There are several possible explanations why Windows' behavior might stray from that of other operating systems. One is that it is in an abundance of caution. Historically, timestamps on some systems leaked information which might pose a security risk. While these issues have largely been addressed, it is possible that Microsoft simply does not see it as being worth the marginal risk – perhaps particularly in conjunction with a second possibility. It may be that Windows does not use the timestamp, and therefore does not bother to enable it; why waste the ten bytes? This logic, however, is faulty. Just because the originator would not benefit from the option does not imply that the recipient would not. Unfortunately, the RFC provides the originator no way to express that it does not actively desire the option itself but is *willing* to enable it if the recipient wants it. In fairness to the RFC, the migration from independent TCP echo and echo response options to a single timestamp option as touched on in Section 5.3 seems to indicate that the prevailing assumption was that both sides would want the option if it was supported.

It is worth noting that a simple and backwards-compatible update to RFC 1323 could resolve this issue. One way this might be done is simply to relax the requirement that the recipient only insert the option in the SYN+ACK if it was received in the SYN. An originator that is willing but does not actively desire the timestamp option would not send it in the SYN, but would enable it if received in the SYN+ACK. This approach would likely work fine in practice, but is inconsistent with the typical TCP option negotiation pattern where the option appears in both SYN and SYN+ACK, and, in the case of an outdated originator and an updated recipient, causes the recipient to engage in behavior which violates the RFC from the originator's perspective. We favor an alternate approach. When the originator sends the timestamp option, it sets its own timestamp value, but it has no value for the echo portion yet; the RFC specifies that it SHOULD set the echo field to 0, and that the recipient MUST ignore the echo value in the SYN. We would alter this so that the originator – if it is willing but not actively desiring – would set the echo field to 1. A non-updated recipient MUST ignore this; it would still either enable the option or not according to its own local configuration and capability as if nothing were different. An updated recipient receiving a 0 would interpret the 0 as meaning the originator wants the timestamp option, and in this case, the recipient would also enable the option or not as if nothing were different. However, when an updated recipient receives an echo value of 1 in the SYN, this is interpreted as the originator being willing but not actively desiring. Should the recipient also not be actively desiring, it can simply fail to send the timestamp option back. Otherwise, it enables the option. This alteration is highly backwards compatible. If neither originator nor recipient is updated, nothing changes (of course). If only the recipient is updated, nothing changes unless the non-updated originator violates the original RFC's suggestion that it SHOULD set the initial echo to 0 and sets it to 1 instead (and one must imagine such cases to be vanishingly rare). If only the originator is updated and the originator is *willing* (but does not actively desire) to do the timestamp, then the timestamp will be enabled (it may be that this is a waste since neither side may actually care, but it should be relatively harmless). And if both are updated, we get the improved negotiation process which disables the timestamp if neither host cares, but enables it if *either* cares.

In lieu of the type of general fix that an updated RFC would provide, the lack of timestamps on Windows can be resolved on an application-by-application basis with minimal effort – enabling timestamps on new connections is as simple as setting a socket option. A trivial patch to Chrome and Firefox would enable timestamps for a large fraction of the currently un-timestamped traffic. Such a patch would bring the behavior of Windows hosts in line with that of the behavior of Chrome and Firefox on other platforms. Of course, Windows clients are not even important for a growing category of services (*i.e.,* those accessed entirely though mobile iOS/Android apps).

Ultimately, however, some traffic may simply not have timestamps. For these cases, one can consider several workarounds: (i) Send all non-timestamped traffic to the same DIP. (ii) Simply use a hash-based policy for all such traffic and accept that its PCC will suffer. (iii) Redirect such traffic to a more traditional stateful load balancer; as this is only necessary for the portion of traffic without timestamps, even this option potentially entails a significant reduction in cost.

5.3.5 SYN Cookies

TCP SYN cookies are a technique used to prevent a SYN flood from exhausting server state by generating a large number of embryonic connections. When SYN cookies are active, the server does not actually save state for the embryonic connection. Instead, it replies with a SYN+ACK where the sequence number contains a specially crafted value – the SYN cookie. For SYN flood attack traffic, things end here – the server has expended no state on this connection which will never complete. For legitimate traffic, when the client completes the three-way handshake, it echoes the server's initial sequence number (plus one) in the ACK field. The cookie contains a field which is generated by a cryptographic hash, which allows the server to validate that the cookie is legitimate (one that it sent itself). The cookie also contains an encoded version of the MSS to use, since the client may have originally sent the value in an MSS option in the SYN, but the server did not store the value. There are two things to notice immediately. First, there is a conceptual similarity to the SYN cookie use of the sequence and acknowledgement fields and PSLB's use of the TCP timestamp option: in both cases, they are used simply to get the other side of the connection to echo a value back. Secondly, while the SYN cookie encodes the MSS, it does not encode any other TCP options, many of which are crucially negotiated only in the SYN packet and therefore the state for which is lost when SYN cookies are used.

Historically, this second aspect simply meant that when SYN cookies are active, TCP options beside MSS negotiation are disabled – the SYN+ACK sent by the server disallows them. While reading the Linux kernel code as part of our investigation for PSLB, however, we found something of which none of the authors were previously aware: the current implementation of SYN cookies can support several TCP options by encoding the option data in the lower bits of the TCP timestamp, very much like how PSLB uses the timestamp. Again, the server simply uses the field to cause the client to echo this back at the end of the three-way handshake so that the server need not have "remembered" the options from the SYN.

This usage of the timestamp is problematic for PSLB: our placement of the SID within the timestamp means we lose the option information that the SYN cookies implementation has stored there. When the modified timestamp reaches the server, it will be misinterpreted as option settings which are unlikely to be the correct ones and will lead the connection to be aborted. There are two solutions to this. The first is simply disabling SYN cookies (on many distributions, the default setting is such that SYN cookies activate when the number of embryonic connections is large). The second is to modify the kernel such that the timestamp aspect of SYN cookies can be disabled without disabling SYN cookies altogether, bringing us back to the original option-less behavior with the exception that the presence of a timestamp option at the end of the three-way handshake would indicate that timestamps should be enabled. This would likely be done by introducing a fourth value for the *tcp_syncookies* kernel variable and requiring manual configuration (the existing options are for disabling, enabling when under load, and enabling at all times). While this necessitates the loss of options, it (i) is only when SYN cookies actually activate due to an attack,

(ii) returns to behavior that was seen as acceptable for years (before the new timestamp-using SYN cookies variant was introduced), and (iii) is likely the behavior most Windows users see anyway, as timestamps are disabled by default on current versions of Windows (as discussed in Section 5.3.4).

5.3.6 Non-Problems

To close out this section, we discuss two cases which might seem to present a challenge for PSLB, but, in actuality, do not. These two are duplicate SYNs and TCP Fast Open.

Duplicate SYNs

If the SYN from a client were duplicated and arrived at a load balancer twice (or the duplicates took different paths and arrived at two load balancers), the load balancer(s) might independently choose different DIPs (while a hashing policy would make this unlikely, PSLB's policy generality means that the policy could be, for example, to simply choose a random DIP). Each of the DIPs will respond *with a completely valid SYN+ACK*.

One of these will arrive back at the client first, and the client connection becomes synchronized. When the second SYN+ACK arrives from the second server, the acknowledgement number will be correct (for a duplicate SYN+ACK), but the sequence number is very likely to be different from the first SYN+ACK. Per RFC 5961, the most likely response from the host will be to send an ACK (re-acknowledging the last valid sequence number – the one from the original SYN+ACK). The crucial question here is whether the client will use the timestamp from the second server's SYN+ACK to send its next packets, or whether it will stick with the original one. If it is the former case, we would have a problem: the acknowledgement numbers will never be correct for the second server, and no packets will make it back to the first server. Fortunately, RFC 7323 indicates that in the case of a hole in the sequence space (which is the likely case here, as the sequence number from the second server has no relation to that of the first), the client should use the timestamp of the most recent segment that advanced RCV.NXT. In this case, that would be the one which set it in the first place – the SYN+ACK from the first server. The rationale is that a hole is likely due to congestion, and therefore the more conservative estimate (using the earlier timestamp) is more advisable, but the practical upshot from the point of view of PSLB is that the timestamp from the first server continues to be used, and thus the connection between the client and the first server remains undisturbed.

TCP Fast Open

TCP Fast Open [24] allows a client to send data before receiving the SYN+ACK. This may seem problematic for PSLB: if the client could send multiple packets before receiving a response from the server, it would be impossible to place the SID in them and each might wind up at a different DIP. Fortunately, TCP Fast Open's ability to send data is not general. It allows the sending of data in the SYN packet, but can send no more until the three-way handshake completes. Thus, PSLB is entirely compatible with TCP Fast Open. The initial packet may contain data, which the load balancer will send to a DIP per policy. The SYN+ACK will carry the SID as usual, and only after receiving it will the client send further data.

5.4 Security

In this section, we look at the security implications of PSLB, beginning with the the issue of attacks on its transparency which was foreshadowed in Section 5.2.3, attacks to the infrastructure required to address some of the corner cases discussed in the previous section, and ending with a quick note about a type of load balancing attack that PSLB obviates entirely.

5.4.1 Transparency Attacks

In an attack on PSLB's transparency, an attacker would generate traffic while setting the SID to a value of their choosing. The concern is that this would allow the attacker to focus the attack on a specific server and degrade service for legitimate users with connections to the server, even in cases when the attacker would not have been able to significantly impact service for anyone had their attack not been focused. We can make two observations about such an attack:

1) PSLB only introduces the possibility of an attacker "steering" non-SYN packets. Thus, any new attack exposed by PSLB is not about, *e.g.,* overloading a server with actual TCP connections, embryonic connections (as in a SYN flood), server sessions, or the like. New attacks must essentially be volume-based attacks on something "south" of the load balancers, such as saturating the link between a server and its first hop switch, overwhelming the server's vswitch, or the like.
2) Such an attack must come from many sources (*i.e.,* must be a DDoS), as a volume-based attack on something south of the load balancers from a fixed source can be detected and mitigated using existing techniques.

In a naive implementation, the SID allows an attacker to focus traffic creating an attack as per observation 1 simply by sending many packets to the same SID. The key to mitigating this is to leverage observation 2. Rather than using a "literal" SID, we use an "encoded SID" or ESID. The ESID is based on the SID, but permuted by the client's IP address and port along with a secret salt. In this way, the same ESID value on connections with different source IPs/ports results in a different actual SID. This prevents an attacker from targeting specific servers unless they fix the the source information of the attack traffic... which would create the easily-mitigated situation in observation 2. This type of encoding/permutation can be implemented efficiently on the data plane using bit manipulation, built-in hash support, and table lookups in tables of reasonable size.

If servers are not distinguishable, an attacker now has no way of knowing whether different connections are reaching the same server (aside, perhaps, from statistical analysis of timing correlations and the like) unless they know the salt (we assume a strong attacker already knows the algorithm used to construct the ESID). If servers *are* distinguishable (*e.g.,* responses leak server-specific data in an informational HTTP header), an attacker could potentially make many connections, and – using the known source IP/port, server, and algorithm – manage to find the unknown salt value. This would again allow an attacker to target a single server. Note that finding the salt value necessarily includes an online attack; the need to make many connections to map the ESIDs of different sources to each other is inescapable.

If the connections can be limited to some maximum duration, then we can both frustrate attempts to find the salt as well as minimize the impact of having found it by periodically remapping individual SID values and by doing a rolling replacement of multiple salt values. This frustrates attempts to find the salt because an attacker will not know whether a particular connection is being mapped using the same salt or not. It minimizes the impact because even a discovered salt is only valid for just so long. This rolling replacement could be done without affecting active connections, even across multiple load balancers (so long as they permute using the same random seed and at roughly the same time).

While even a trivial encoding of the SID would likely make an attack much more difficult (especially considering the rolling salt procedure described above and that the attacker has no direct way to observe the decoded SID), one might argue that this technique is fundamentally flawed, as it is essentially attempting to encrypt the SID using a presumably insecure and ad hoc encryption algorithm. Indeed, based on our description above, one might have imagined the simplest case of simply applying the salt value to the SID using a bitwise exclusive OR. While we believe one can do considerably better than this by, *e.g.,* utilizing hardware-based hashing functions, the hashing functions meant for ECMP are of course still not cryptographic, so the objection stands. While our work in this regard is very preliminary, we did begin to attempt to address this criticism. The two constraints for the SID encoding process are that it must encode a small field (13 bits is far smaller than the block size of common encryption schemes), and it must be performed at line rate on every packet. These constraints rule out a very large number of possible encryption algorithms immediately. The Hasty Pudding Cipher or HPC [109], however, seems to possibly fit the bill. HPC is a block cipher with several variants, where the "tiny" variant allows 13 bit blocks, suitable for a SID. HPC was a submission for the AES contest; it was not selected and certainly has not been as rigorously analyzed as the eventual AES winner, but – as far as we know – has not yet been broken. Moreover, the limited operations required for HPC encryption/decryption suggest it may be possible to realize reasonably in high speed packet switching hardware, and possibly even in existing/emerging programmable switching hardware.[2]

Finally, it is worth noting that variants of these attacks already exist with conventional load balancers. In the case of load balancers which assign a connection to a DIP based on hashing the connection's 5-tuple, the attacker can, for example, manipulate the source port number to steer connections to a particular DIP if they know (or reverse) the details of the hashing. Even without reversing the hashing, if servers are distinguishable, an attacker can simply establish connections until reaching a critical mass connected to the same server and then begin the attack in earnest against only that server. Ultimately, PSLB does not actually lower the security bar at all, and – using the techniques outlined above – may raise it.

[2]To convince ourselves of the potential for this approach, we implemented the "tiny" variant of HPC in the P4 domain specific programming language; as far as we know, this is the first block cipher implemented in P4.

5.4.2 Corner-Case Attacks

Section 5.3.4 discusses solutions to two corner-case issues where packets do not have timestamps. The first is the case of fragmented packets. One possibility put forth is that all fragments are re-associated on one or more servers dedicated to the task. An attacker could potentially exploit this by creating many fragments, overwhelming the re-association servers. It should be noted that one can scale out this approach by simply adding more such servers. Moreover, in a good implementation, such an attack would only have an ill effect on presumably rare legitimate fragments. Indeed, given that legitimate fragments are presumably rare in the age of Path MTU Discovery, one might ignore the issue altogether and simply drop all fragments.

The second case is when packets do not have timestamps due to, *e.g.,* poor OS support for the timestamp option. One of the solutions put forth is that all non-timestamped packets be sent to the same DIP. This certainly creates the possibility that an attacker could deny service to legitimate users with no timestamp support. The other two options are far more robust. The first of those is to rely on a simple hashing policy for packets without timestamps, which will have worse PCC but is not so easily targetable. The other is to fall back to a more traditional load balancer, in which case PSLB makes it no more or less attackable than it already was.

5.4.3 State Exhaustion Attacks

In closing this section, we wish to point out a type of attack that PSLB simply eliminates. With a conventional load balancer, the connection tables are subject to state exhaustion attacks, which limit their effectiveness and thus reduce PCC. PSLB has no such state to exhaust.

5.5 P4 Implementation

To verify that PSLB could be deployed in the real world, we implemented a basic version (along the lines described in Section 5.3.1) in P4 [61], which is a domain specific language for packet processing that can be compiled for execution on high performance reconfigurable switching hardware; specifically, we used the P4_16 version. While we lack P4-capable hardware and production traffic on which to truly experiment, we were able to run our implementation on the P4 bmv2 software switch [15] in small-scale test scenarios to verify that it functioned as expected.

All told, our implementation is less than 500 lines of P4 code, and much of this is dedicated simply to rather boilerplate code, such as the definition of the IPv4 and TCP header structures and basic IP forwarding. The two major points of interest with our implementation are the TCP option/timesamp parsing and the TCP checksum update.

Parsing TLV-type data such as TCP options was challenging in earlier versions of P4. Most typically, the option values are not actually of interest to switches, and thus can simply be read as a single "blob", but this approach does not work for PSLB, as we are interested in (and may wish to modify) the value of one of the options (the TCP timestamp). To parse individual options, one can create a P4 header type for each option type and parse them in the order in which they occur in a packet. However, one must explicitly specify the order in which the headers are deparsed to reform a packet. The result is that simply processing the packet – without making any *intentional* changes – is likely to reorder the options and change the checksum. We deemed this undesirable. P4_16 introduces what should be an ideal mechanism for such TLV parsing: *header_union* stacks. One defines the structure of each option individually, and then creates a union of all of them. Then the TCP packet simply includes a stack of these unions. Each header is parsed individually and added to the stack, and deparsing puts them back together in the same order. Unfortunately, stacks of *header_unions* were not yet supported in bmv2 as we were doing our implementation. Our final solution leverages the fact that we are actually only interested in a single option. We define a TCP timestamp option structure and a generic TCP option structure. Our TCP packet definition ends with a (possibly empty) stack of generic options, then a (possibly empty) TCP timestamp option, followed by a second (possibly empty) stack of generic options. The order of deparsing this structure is unambiguous, and the timestamp option (if present) can be read and written.

While our approach makes changing the timestamp possible, doing so invalidates the TCP checksum. Unfortunately, the TCP checksum calculation function provided by the switch did not yet support header stacks such as our stacks of generic TCP options. To account for this, we used P4 to implement an action which “manually” performs an incremental checksum update (this is relatively cheap and easy, as it must only consider the old and new timestamp values – not the whole packet payload). We then defined a P4 table which simply applied this action to all packets.

While our current implementation is basic, it convinces us that a P4 implementation of PSLB is feasible, even when it must confront the idiosyncrasies of particular switch targets.

5.6 Evaluation

In this section, we use simulations to compare PSLB to Maglev and SilkRoad in the context of a data center. We look at two aspects of the systems: the load on the network fabric, and the number of PCC violations which occur under changing conditions.

5.6.1 Experimental Setup

Topology: We set up a “spine leaf” data center topology similar to that described in other work [4], with 4 spine/core switches and 9 leaf/ToR switches which form a complete bipartite graph connected by 40Gbps links with 2µs latency. We replace one leaf ToR switch with a gateway router to connect the the data center to the rest of the world. Each ToR is connected to 16 servers with 10Gbps/2µs links resulting in a total of 128 servers.

The gateway’s ingress link has a capacity of 50Gbps and there is a delay of 5ms between it and our simulated clients. The gateway uses flow-based ECMP to balance traffic across the four spine switches. Since the ingress capacity is less than the capacity of the data center network there is no significant congestion in the system.

Comparing Load Balancers: We evaluate the following Load Balancing approaches in our simulations:

- *Maglev:* In Maglev load balancing, the load balancers reside in a subset of the servers, and we simply choose five random servers to act as load balancers. We use Maglev hashing [34] to balance connections across DIPs.
- *SilkRoad:* In SilkRoad load balancing, the load balancers are implemented by switches; we use the four spine switches. Consistent hashing is used to balance connections across DIPs.
- *PSLB:* In PSLB, the load balancers reside in the four spine switches as well. We evaluate PSLB with both Maglev hashing and consistent hashing schemes to balance connections across DIPs, and refer to these as PSLB (Maglev) and PSLB (Consistent) respectively.

For all load balancers, we host 5 VIPs which are stored in all the load balancers. Each VIP is initially assigned 20 DIPs from the server pool. For Maglev and SilkRoad, we assume an infinitely large connection table, so no entries ever need be evicted. Further, we assume that the DIP pools are modified simultaneously in each load balancer.

Note that these conditions are favorable for Maglev and SilkRoad: in the real world, each VIP may only be assigned to a subset of load balancers in order to reduce the connection table memory on each one. This is especially true in the case of SilkRoad, where load balancer memory is at a premium. Indeed, SilkRoad provides an approach for determining VIP placement wherein some VIPs are handled in the (more numerous) leaf switches (where there is more total memory) – an approach which improves scalability in the face of constrained memory at the cost of additional complexity and additional hops to reach a load balancer.

Evaluation metrics: We look at two metrics in our simulations:

- *Fabric load:* We look at load on the fabric by counting the total bytes received by all of the spine and leaf switches in our topology.
- *Percentage of PCC violations:* We look at the percentage of PCC violations when we update the DIP pool and fail a load balancer at roughly the same time.

When examining PCC violations, we randomly choose a VIP, either add a DIP to its pool or remove one, and then kill a randomly chosen load balancer instance mid-simulation. We use the killing of a load balancer to model any sort of change that would lead to the loss of connection table state. This could, indeed, be the loss of a load balancer, but since traffic to load balancers is divided using ECMP, it could also be any event that perturbs the ECMP (*e.g.,* a link state change or the *addition* of a load balancer). When connections are rerouted to one of the remaining load balancers, there is no entry in the connection table at the new load balancer; if the load balancing state is not captured in the packet, the balancing decision must fall back on a re-application of the hash, which may no longer provide the same result as when the connection began, potentially leading to a PCC violation.

We calculate the percentage of PCC violations by dividing the number of violations by the number of active connections at the time of load balancer failure. When a DIP is removed, connections that were assigned to it are *not* counted towards PCC violations as there is no way to ensure consistency for those connections – their DIP is gone. We limit our analysis to a single DIP update, rather than a continuous churn, in order to understand the effect of a single update in isolation.

Simulation Setup: We run simulations in a packet-based network simulator for 15 minutes of simulated time. Each experiment is run with 16 different random seeds. We look at two traffic workloads:

- *Point workload:* All connections have a duration of 60 seconds. This is intended to model the case where HTTP persistent connections are the dominant factor in connection lifetime.
- *Web servers workload:* We use the flow durations from a distribution given for Facebook web server traffic [105].

The total amount of traffic is similar in both workloads, as in both cases, we adjusted the traffic generation such that the aggregate rate would be enough to keep the Maglev boxes operating at close to line rate.

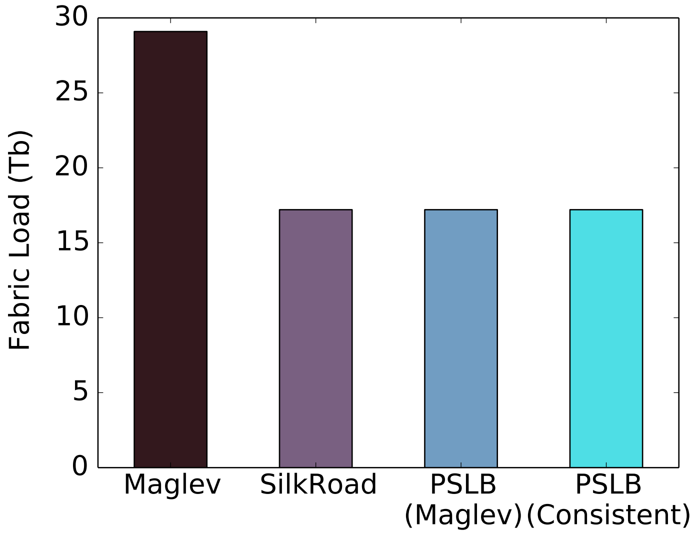
5.6.2 Fabric Traffic

Figure 5.2(a) and 5.2(b) show the fabric load (averaged over 16 runs) for the point and web server workload respectively. It shows that the network fabric in Maglev sees 69% more traffic than SilkRoad or PSLB. This happens because Maglev places the load balancers in servers rather than the spine switches; packets must traverse extra hops to get to the load balancer and then to the destination DIP. While we did not evaluate it, extra traversals would also have a small impact on latency. (Note that we simulate direct server return, so returning packets never take extra hops to go through a load balancer.)

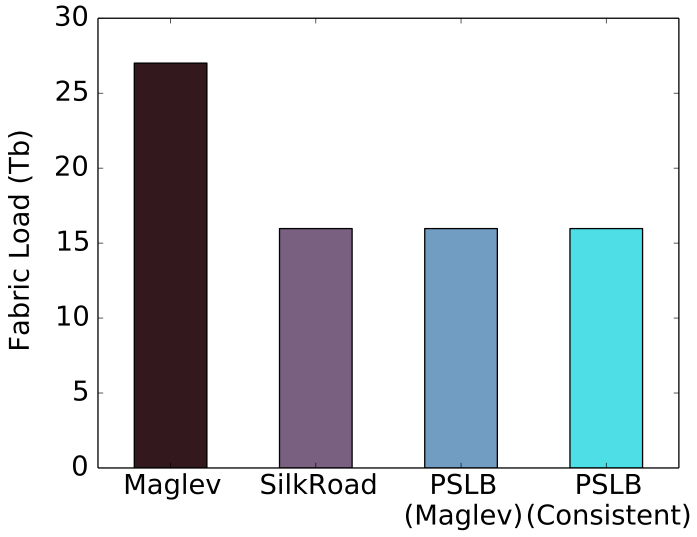
5.6.3 PCC Violations

Figure 5.3(a) shows the PCC violations for adding a DIP right before load balancer failure. The results show that Maglev has a high median, but the variation of violations is low. This is because Maglev uses Maglev hashing which even re-maps some healthy connections in the interest of achieving better load balance. Due to this, the variability decreases at the cost of a high median. For SilkRoad, we see the opposite happen. The median PCC violations are lower, but there is a large spread of violations. SilkRoad uses consistent hashing as its underlying connection-mapping mechanism. As the name suggests, it tries to provide more consistent mapping of existing healthy connections, but might not split load as equally as possible in order to do so. Finally, PSLB has no violations, as the server ID is picked up from the packets arriving at the load balancer. This is true agnostic of hashing algorithm, as the hashing algorithm is not used to determine the destination DIP after the first packet of the connection. Figure 5.3(b) shows results for the more realistic web server workload; they maintain a similar pattern.

Figure 5.4(a) shows the results for randomly removing a DIP just before a load balancer failure. Note that the connections that were destined for the removed DIP are not counted as violations as their failure was inevitable. The figure shows that there are some violations for Maglev. This is

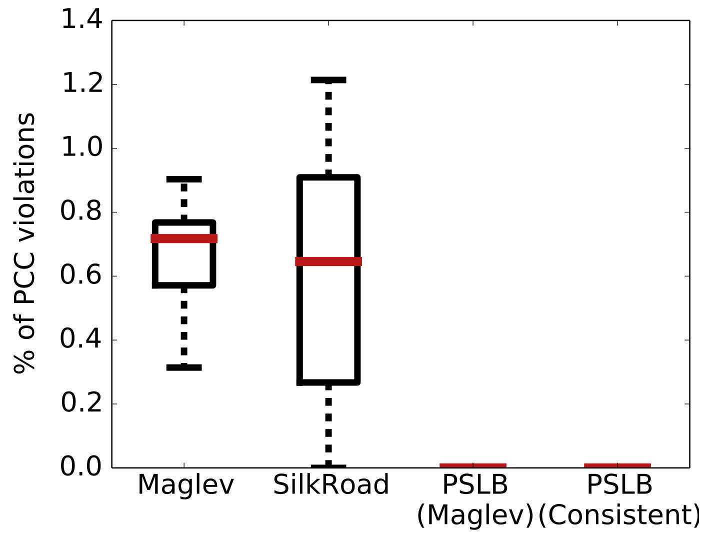


(a) *Point workload*

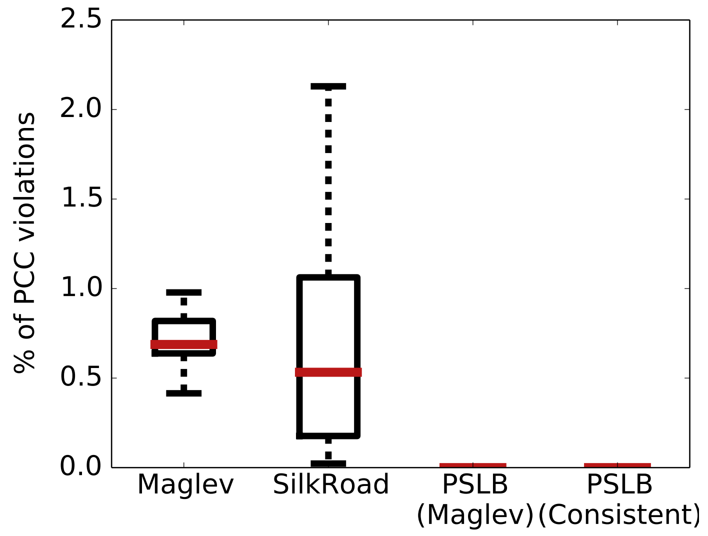


(b) *Web servers workload*

Figure 5.2: *Load on the fabric for 900 seconds (averaged over 16 runs).*

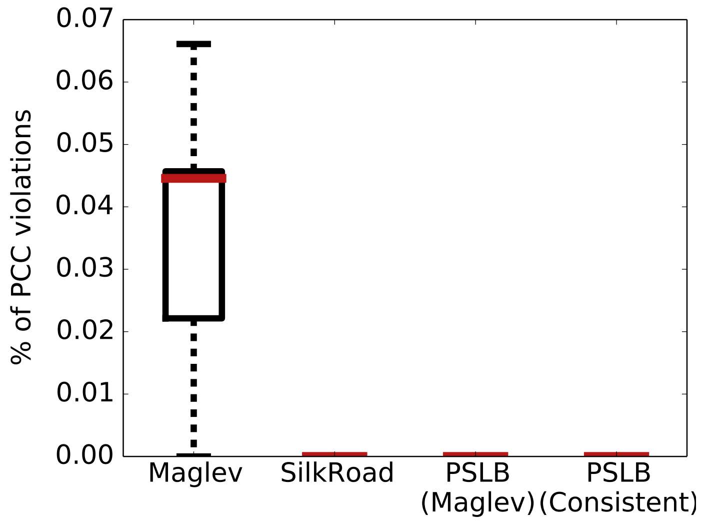


***(a)** Point workload*

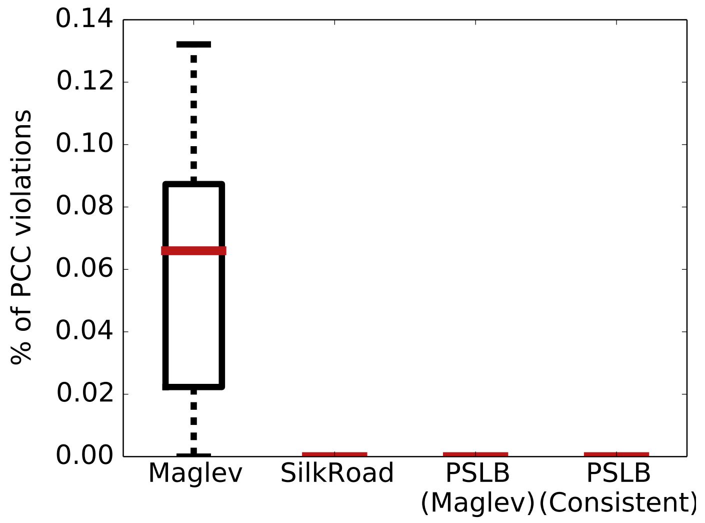


***(b)** Web servers workload*

***Figure 5.3:** Percentage of active connections which result in PCC violations when a DIP is added and a Load Balancer fails at the same time.*



(a) *Point workload*



(b) *Web servers workload*

Figure 5.4: *Percentage of active connections which result in PCC violations when a DIP is removed and a Load Balancer fails at the same time.*

because Maglev hashing recomputes the mapping of connections on DIP update to equalize the load and some of the connections are re-mapped to other DIPs. This causes some connections to be re-directed to incorrect DIPs when their hash is recomputed. On the other hand, SilkRoad uses consistent hashing so no connections are re-mapped to incorrect DIPs and thus it has no violations in the case of DIP removal. PSLB, with either Maglev or consistent hashing, has no violations again, because the state is carried in the packets and is not tied to the load balancer. Figure 5.4(b) shows similar behavior for the web server traffic workload.

We do not pretend to be in a position to say what level of PCC is acceptable. Indeed, no one answer would seem to be applicable in all cases anyway. Google was willing to exchange a few more broken connections in order to achieve better balancing. Meanwhile, Github worked hard to break as few connections as possible in their load balancer because connection failures of long git repository clone operations leads to particularly bad customer experience [117]. Here, we merely point out that PSLB meets or exceeds the PCC of other solutions.

5.7 Discussion

In this section, we briefly discuss some secondary but related issues.

5.7.1 PSLB and QUIC

Concurrent with our work on PSLB, the QUIC working group has apparently been pursuing similar ideas in the context of QUIC: a brief mention of using the QUIC Connection ID field as a SID appears in a QUIC related Internet Draft [67]. The QUIC Connection ID was originally meant to allow client mobility – by selecting a large random ID, a client gains the ability to roam to another network. In the data center, the actual source IP address is ignored, and only the Connection ID is used to determine the source (in conjunction with the source information implicit in the fact that all QUIC communication is encrypted). In many ways, using an ID to enable client mobility is the conceptual mirror of using a server ID for server load balancing: the client ID lets the server side ignore the IP address of the client, and the server ID allows the client to "ignore" the true IP of the server. Following from its initial usage, the QUIC semantics were extended to allow the server side to change the Connection ID, and in subsequent packets, the client echoes this new ID.

QUIC's focus on 0-RTT communication, however, presents a unique challenge that the aforementioned draft does not discuss. As with TCP Fast Open (discussed in Section 5.3.6), QUIC can send data immediately – before it receives a response (and therefore a SID) from the server. Unlike TCP Fast Open, however, this can take the form of multiple packets. To ensure that these are sent to the same DIP, one has essentially two options. First, one might give up policy generality and revert to a hash based policy (likely based on hashing the initial randomized Connection ID). As the hashing is only relied upon for the short duration in which the client is sending without having yet gotten a response from the server, the likelihood of an event which perturbs the hash and causes a PCC violation is small. A second option is to use a similar technique to the one we describe being used for fragments in Section 5.3.4: the initial packets of the connection would be sent to the policy engine, and the policy associated with the initial Connection ID in the switch's control plane or a re-association server. Packets with the initial Connection ID would be sent there, associated with the chosen server, and forwarded. As with fragments, this state would be short-lived. (Our fragmentation reassociation technique could also be applied to resolve fragmentation of QUIC packets.)

A final note on the application of PSLB to QUIC (which the draft touches on) is that in QUIC, the scheme we describe to prevent malicious targeting of individual servers (involving the mutation of the DIP mapping and the rolling salt value) is likely not required due to QUIC's pervasive use of cryptography.

5.7.2 Relationship to L7 Load Balancing

It is worth noting that a similar technique has long existed in the context of application layer load balancing. While L7 web load balancers are inherently stateful in terms of an individual connection, some web applications benefit from broader "stickiness". That is, they benefit from, *e.g.,* multiple requests from the same user being sent to the same DIP – even across connections. The reasons for this are because things like session or user data may end up cached on a particular server: if every connection from the same user went to a different DIP, it would not leverage this locality. To facilitate stickiness, a web load balancer may perform *cookie injection*, which injects an HTTP cookie into a response from the server. The client echoes this back in future requests, and the load balancer directs these requests to the same DIP.

5.7.3 The Utility of Echo

It is interesting to observe that a simple "echo" facility is usable for implementing stateless connections (SYN cookies), L4 load balancing without a connection table, and L7 session "stickiness". In effect, it allows for a small amount of extensibility in behavior without requiring the cooperation of all parties. Given that most network protocols are prone to ossification, one wonders whether this type of facility is a candidate for purposeful inclusion in new L3 or L4 protocols. Especially if programmable data planes become the norm, one can imagine that intermediate nodes may find it advantageous to be able to store (and have echoed) a handful of bits in a packet without waiting for a new revision of the protocol. In the context of intermediate nodes, placement at L3 may be the more useful (as well as the more logical). Indeed, it would have even benefited PSLB if we had been able to place our SID at L3 (where all fragments would carry it, avoiding the fragmentation issue) rather than L4, and it certainly would be cleaner if we hadn't been required to find a way to share bits with an existing feature.

5.8 Conclusion

PSLB provides L4 load balancing with essentially perfect per connection consistency and requires little additional state. In both respects, it is far superior to the current and proposed load balancing approaches. The challenge to PSLB is retrofitting it into today's TCP, and in this we are only partially successful, in that there are cases in which our primary mechanism will not work and it must fall back on a secondary mechanism, though this is still likely a win in terms of both cost and PCC. However, its applicability would be greatly expanded by a simple change in the timestamp option (allowing it to be invoked by either side), a change that may well be feasible given the high degree of interest in better load balancing solutions and the current costs operators pay to deploy them at scale.

Chapter 6

Conclusion

The preceding four chapters have each individually addressed specific networking issues and proposed remedies. This chapter begins by attempting to summarize the contributions that chapters 2, 3, and 5 make at the link, network, and transport layers of the common Internet layering model; this is followed by a summary of chapter 4, which proposed a change to that layering model. The chapter ultimately concludes with a reflection based on the author's experience with the work as a whole.

6.1 Summary of Contributions

The Spanning Tree Protocol was standardized in 1990, and its origin goes back into the 1980s; depending on how one wishes to count it, it has now spanned four or five decades. Despite a number of newer alternatives, STP and its direct descendants are still common enablers for the link layer. This is also despite what may often amount to significant limitations for the networks of today, chiefly its relatively slow convergence and the inability to leverage arbitrary topologies. Chapter 2 presented an alternative approach to the link layer which rectifies this – obviating the spanning tree and leveraging every link in a topology.

Routing protocols commonly associated with the network layer (IS-IS and OSPF, for example) distribute information across the network such that each node can perform a local operation (*e.g.,* select which link to forward a packet to) to achieve a global goal (*e.g.,* have the packet reach its intended destination). When changes occur in the network, such as a link being added or removed, the relevant information must be spread across the network, and nodes may need to alter their behavior in response. This process takes time and is fundamentally messy due to asynchrony. Various techniques exist that attempt to address this problem by planning ahead. When it is suspected or known that the existing local routing state is problematic, a node can fall back to a secondary behavior (*e.g.,* as in MPLS Fast Reroute). However, none of these techniques are universally deployed. Chapter 3 addresses this same problem with an even simpler technique, intended to be safe, easy, effective, and applicable to virtually any routing algorithm: when existing state is questionable, safely flood packets. The result appears highly effective, and the specifics of the proposed design are intended to lower the bar for the approach far enough to make widespread deployment possible so that all networks may reap the benefits of very high availability.

Server load balancing arose out of a need and desire to scale a service "out" across multiple physical servers, rather than scaling a server "up". At the mechanical level, addressing this change seems relatively straightforward, and there have now been decades of work developing stateful and stateless solutions. At a deeper level, this change introduces a major shift, where much of the Internet is today about accessing services and not individual hosts – and it was the assumption of the latter around which the fundamental protocols of the Internet were designed. While there is work which deeply rethinks the Internet protocols to embrace a fully service-centric view, chapter 5 focuses on just this specific subproblem. Working backwards from its principled viewpoint, it then explores how the solution can be applied in today's Internet. The result is a system for load balancing that can be implemented efficiently, used at scale, can use arbitrary load balancing policies, and maintains perfect per connection consistency.

While the works mentioned above are all fairly strongly associated with a particular networking layer, chapter 4 has a somewhat different flavor. Today, the Internet is most clearly associated with the IP protocol. However, many previous research efforts have pointed out significant shortcomings of IP, and proposed various alternatives. A key problem with such proposals is that there is seldom a clear path to get from where we are today to the Internet they foresee, *e.g.,* because of how widely integrated IP is. Moreover, a simple transition from where we are today to one of these alternate futures seems to be risky (the selected alternative may end up being worse than the present) and likely to just lead us back to where we are now eventually (that is, as the world changes, the selected protocol will eventually fall short and require replacing again). The work in chapter 4 attempts to lay the framework to confront this, in part by cleanly separating intradomain protocols from interdomain protocols. By decoupling packet delivery at the domain level from Internet protocols, it becomes radically easier to change the latter (it does not require a change to every router – only ones at domain edges). Moreover, it becomes feasible to support multiple Internet protocols simultaneously. The effect is to transition from a view of the Internet as a specific protocol to a view of the Internet as a framework within which multiple protocols can fit. This would facilitate experimentation and migration, and it would allow for interdomain protocols to be developed to fit different niches, rather than requiring each new iteration of IP to support radically new use cases and capabilities without compromising old ones (a task which will only become harder and harder).[1]

[1]While this project is named in reference to its *revolutionary* possibilities, the author finds this more *evolutionary* conceptualization – where the future Internet is viewed as a rich and dynamic ecosystem where protocols can evolve to fill particular use cases – particularly compelling. This inspired the suggestion that the name be changed to Grinnell, in reference to the person most associated with (though not the inventor of) the term "ecological niche". Alas, the die was already cast.

6.2 Reflection

All areas of computer science are continually called upon to improve and adjust, and each area likely pushes up against its own unique challenges as it strives for better things. Networking is interesting in that its own success creates its most significant challenge. Networks are fundamentally about communication and cooperation. To paraphrase Metcalfe's Law: as you add nodes to a network, the value of the network grows more than linearly. Clearly, that represents success. However, it is also an example of a complex system (in the systems theory sense as well as the quotidian one) becoming more complex – especially when one accounts for not just the narrow technical concerns, but for the *entire* system (including infrastructural stakeholders, "consumer" stakeholders, standards processes, *actual human lives* impacted by the network, and on and on). But even just considering those basic technical concerns, the more parties involved, the harder it is to coordinate change. Updating the protocols on a network with two nodes is easy; updating the protocols on a network with billions is not. To echo the phrasing above: as you add nodes to a network, the challenges to evolving that network grow more than linearly. This fundamental tension leads to a fundamental issue in networking: success very often leads to ossification (with more success generally leading to more ossification).

This dissertation covers some specific contributions I have made to improve the state of networking. Each chapter is an attempt at a reimagining. The changes these reimaginings imply must contend with this ossification, and as a result, this dissertation is – aside from its specific technical contributions – an exploration of creating change in the successful, complex, and densely interconnected system that forms the Internet. Chapter 2 reimagines Ethernet without a spanning tree, and – more broadly – reimagines local area network path selection as a process of controlled chaos with almost instantaneous response to disruption resulting from local decisions based on metadata attached to data-carrying packets… rather than as an orderly process carried out by an independent control plane process with independent control packets. The change is significant, but as with other works in this space (*e.g.,* TRILL and SPB), it is careful to leave the host-facing interface alone, allowing it to act as a drop-in replacement for Ethernet. Chapter 3 reimagines how to deal with routing disruptions at the network layer, eschewing approaches that attempt to rely on (possibly outdated) global information, and again opting for controlled chaos in the form of flooding. It goes even further than the work from chapter 2 in its attempt to meet networks where they currently are, allowing routers with the new functionality to coexist with (and hopefully gradually replace) existing routers. Chapter 5 details an attempt at a principled approach to the problem of server load balancing which recognizes that the fundamental problem is that a *service* must be able to refine the address (in the broad sense) used by a client – a problem for which the existing IP protocols have no well-tailored mechanism, in no small part because the notion of an abstract service implemented across multiple machines was simply not part of the model around which they were designed. Like the previous works, however, it goes to lengths to bend that approach to work with existing protocols such that only the service need be aware of the change.

The work described in chapter 4 continues a line of work which explored its particular topic as if the world were a clean slate – without considering the constraining effect already imposed by successful networked systems. One of my biggest contributions to this line of work was completely reversing this, carefully fitting the earlier ideas into the world in which we actually live. The kicker, of course, is that the entire point of the referenced "earlier ideas" was to enable an extensible Internet. That is to say, one built for change rather than one that will fight tooth and nail against the ever-changing world. A world where the principle behind chapter 5 could more easily be embraced rather than contorted and packed into a TCP-shaped box, for example.

So, in my desire to re-examine networks and change them for the better, I have spent a fair amount of time considering the issue of how they resist change and how they allow it. Perhaps this should be unsurprising, since I cut my networking teeth surrounded by the blooming of Software Defined Networking. While that term carries quite a bit of baggage with it now, before it had been coined, Martin Casado laid out his intended goal to me pretty succinctly – something very much along the lines of "I wanted to write a program to control the network". There are a couple things worth pointing out in this simple statement. First, it focuses on "a program" because it is something fluid, something that could change easily, something unlike the networking status quo. Second, it implies that there is a purpose to controlling the network, recognizing that the way the network operated yesterday should not be the way it operates today (and, likely not the way it operates tomorrow). So I am greatly enthused by SDN – interpreted in this broad sense which encompasses things like software forwarding, programmable NICs, P4, and so forth in addition to things like OpenFlow and logical centralization of the control plane. These all reflect a new outlook on networking which focuses change. I think at its best, it focuses the idea that we can work together to achieve the interoperability that networks require at the speed of a pull request rather than the speed of a standardization process.

Similarly enthusing are developments like those seen in QUIC. In QUIC, encryption is deployed in part for the express purpose of resisting ossification by only allowing QUIC fields to be interpreted by the necessary parties. This is a new way to *enforce* decisions about the "horizontal" applicability of layers, which is a perhaps the single most important tool we have for the prevention of ossification. Attention to where a layer applies is what lets us constrain the parties involved. Where we can limit the number of players, or limit the players to those with aligned interests, we have a hope of being nimble. Where we tie a layer to an excessive number of players, or players with unaligned interests, progress will crawl, lurch, or stumble. This is a principle we know, but maybe we do not know it as deeply or follow it as frequently as we should.

We are very fortunate to have good records of TCP and IP being teased apart during the development of the Internet. This is a perfect example of separating the players – hosts versus hops – in a useful way. But we followed this up with a willingness to compromise on the principle for decades (*e.g.,* for stateful load balancing) without making real strides to truly correct the problem.

Similarly, the Internet layers are often pointed to as a thing of beauty because there is an allowance for a variety of physical and data link layers. This, of course, is a leveraging of this principle. The players involved for the lower layers are wonderfully scoped such that they number few and are well-aligned for a particular concrete instance of the layer; in theory, a particular company or IT department can make their own choices about these layers (*e.g.,* to deploy something like that found in chapter 2) without needing agreement from anyone else besides, perhaps, equipment vendors. But we have seen the importance of the link layer diminished relative to the network layer of the Internet. Even many of the most basic switching devices today have grown an

IP layer, perhaps taking the message of the famous "IP ON EVERYTHING" t-shirt too far. This chips away at this lauded aspect of the layering, and we are the worse for it: embedding IPv4 on such a huge number of intermediate nodes is one of the things which has made the IPv6 transition so difficult, for example.

The OSI standard [22] includes ten well-considered principles behind their layering model, but it is notable that none of them clearly motivates the definitions of layers based on a broad appreciation for where those layers must be implemented (*i.e.,* who the involved players are). The third of the principles suggests that one "create separate layers to handle functions that are manifestly different in the process performed or the involved technology," and while the processes and technologies *may* coincide usefully with the players involved (it was certainly true in the early days of the Internet, when the networks being interconnected were technological one-offs built by different organizations), this is not assured. We might amend this principle to also suggest creating separate layers when the *players* involved are different.

This, of course, is one of the keys to the work in chapter 4, which suggests essentially separating the network layer into separate intradomain and interdomain layers, where the former is implemented on hops within a domain, and the latter on hops between domains. One of the wins from this separation is exactly that it allows limiting the players involved for each of the layers, and it does so along aligned interests. Indeed, this echoes part of the rationale behind the explicit differentiation of the intradomain and interdomain routing control planes which began with EGP; the suggestion here is that a similar explicit separation be made for the data plane. In this arrangement, domains are free to change their intradomain protocols in much the same way that a local network administrator should be able to change their link layer protocols, and an interdomain protocol need only be implemented on participating hosts and where participating domains join. This bit about participation segues to another key argument in the same chapter, which is that we should let go of the idea of a single "narrow waist" network layer protocol. We can again see the same principle at play in this argument. At one point, the players on the whole of the Internet were relatively few and extremely homogeneous. This is no longer the case. Expecting a single protocol to align well with four billion people and all their wildly varying use cases is an unrealistically high bar, and attempting to meet it will only stifle progress. And, indeed, it will do so unnecessarily, since – along with being unrealistic – it is also unnecessary. Finally, the musings on the utility of an "echo" functionality in chapter 5 are also about exactly this principle, positing that there may be a simple mechanism which allows one end of a connection to, in effect, usefully alter the behavior of the other in an ad hoc manner – limiting the players involved in the alteration to a single side of a connection.

Ultimately, each of the works in this dissertation looked at something that existed in the context of networking, tried to imagine a better version of it, and tried to find a way that the better version might be wiggled into networks we use today. The concern expressed above is that this wiggling is already harder than desirable, and may become harder yet. I am obviously not the only person to share this concern, but I am taking this opportunity to add my voice to the choir.

The circumstances of the world at the time of this writing have made it increasingly clear to an increasingly large group of people that we must continue to reimagine and reinvent the world to be a better place. The Internet – fundamentally about communication and cooperation – holds unparalleled *promise* as a tool to that end. *Fulfilling* that promise means we must continually reimagine and reinvent a better Internet. This dissertation marks the end of the beginning of my own humble efforts.

Onward.

Bibliography

[1] S. Abdallah, A. Kayssi, I. H. Elhajj, and A. Chehab. Network Convergence in SDN versus OSPF Networks. In *International Conference on Software Defined Systems (SDS)*, 2018.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of USENIX NSDI*, 2010.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *Proc. of ACM SIGCOMM*, 2013.

[5] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *ACM SIGCOMM Computer Communication Review*, volume 29, 1999.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of ACM SOSP*, 2001.

[7] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. of ACM SIGCOMM*, 2008.

[8] T. Anderson, K. Birman, R. M. Broberg, M. Caesar, D. Comer, C. Cotton, M. J. Freedman, A. Haeberlen, Z. G. Ives, A. Krishnamurthy, W. Lehr, B. T. Loo, D. Mazières, A. Nicolosi, J. M. Smith, I. Stoica, R. van Renesse, M. Walfish, H. Weatherspoon, and C. S. Yoo. The NEBULA Future Internet Architecture. In *The Future Internet - Future Internet Assembly 2013: Validated Results and New Horizons*. Springer, 2013.

[9] K. Argyraki and D. R. Cheriton. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *Proc. of USENIX ATC*, 2005.

[10] A. Atlas and A. Zinin. Basic Specification for IP Fast Reroute: Loop-Free Alternates. RFC 5286, RFC Editor, September 2008.

[11] T. Barbette, C. Tang, H. Yao, D. Kostić, G. Q. Maguire Jr, P. Papadimitratos, and M. Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *Proc. of USENIX NSDI*, 2020.

[12] S. M. Bellovin, D. D. Clark, A. Perrig, and D. Song. A Clean-Slate Design for the Next-Generation Secure Internet. GENI Design Document 05-05, July 2005. Report on NSF workshop.

[13] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM Internet Measurement Conference (IMC)*, 2012.

[14] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its

Application to Scalable Server Architectures. In *Proc. of IEEE International Conference on Network Protocols (ICNP)*, 1998.

[15] P4 Behavioral Model. `https://github.com/p4lang/behavioral-model`.

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog Computing and its Role in the Internet of Things. In *Proc. of the MCC Workshop on Mobile Cloud Computing*, 2012.

[17] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP Extensions for High Performance. RFC 7323, RFC Editor, September 2014.

[18] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Proc. of ACM SIGCOMM*, 2013.

[19] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122, RFC Editor, October 1989.

[20] R. Braden and J. Postel. Requirements for Internet gateways. RFC 1009, RFC Editor, June 1987.

[21] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *Proc. of USENIX NSDI*, 2005.

[22] CCITT. X.200-Reference Model of Open Systems Interconnection for CCITT Applications. *Fascicle VTIII.4*, 1988.

[23] V. G. Cerf. DARPA Activities in Packet Network Interconnection. In *Interlinking of Computer Networks*. Springer, 1979.

[24] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413, RFC Editor, December 2014.

[25] M. Christensen, K. Kimball, and F. Solensky. Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. RFC 4541 (Informational), RFC Editor, 2006.

[26] D. Clark. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review*, 18(4), 1988.

[27] D. Clark, K. R. Sollins, J. Wroclawski, D. Katabi, J. Kulik, X. Yang, R. Braden, T. Faber, A. Falk, V. K. Pingali, M. Handley, and N. Chiappa. New Arch: Future Generation Internet Architecture. Technical report, ISI, 2003.

[28] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: An Argument for Network Pluralism. *ACM SIGCOMM Computer Communication Review*, 33(4), Aug. 2003.

[29] Datasheet, Texas Instruments. DP83867IR/CR Robust, High Immunity 10/100/1000 Ethernet Physical Layer Transceiver. `http://www.ti.com/lit/ds/symlink/dp83867ir.pdf`, 2015.

[30] B. Decraene, S. Litkowski, H. Gredler, A. Lindem, P. Francois, and C. Bowers. Shortest Path First (SPF) Back-Off Delay Algorithm for Link-State IGPs. RFC 8405, RFC Editor, June 2018.

[31] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proc. of ACM SIGCOMM*, 1989.

[32] G. A. Di Caro and M. Dorigo. Two Ant Colony Algorithms for Best-Effort Routing in

Datagram Networks. In *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, 1998.

[33] M. Duke. QUIC-LB: Using Load Balancers to Generate QUIC Connection IDs. Internet-Draft draft-duke-quic-load-balancers, IETF Secretariat, February 2018.

[34] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*, 2016.

[35] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker. Less Pain, Most of the Gain: Incrementally Deployable ICN. In *Proc. of ACM SIGCOMM*, 2013.

[36] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. Segment Routing Architecture. RFC 8402, RFC Editor, July 2018.

[37] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving Sub-second IGP Convergence in Large IP Networks. *ACM SIGCOMM Computer Communication Review*, 35(3), July 2005.

[38] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud Scale Load Balancing with Hardware and Software. *ACM SIGCOMM Computer Communication Review*, 44(4), 2015.

[39] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Intelligent Design Enables Architectural Evolution. In *Proc. of ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[40] P. B. Godfrey, I. Ganichev, S. Shenker, and I. Stoica. Pathlet Routing. In *Proc. of ACM SIGCOMM*, 2009.

[41] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.

[42] D. Gupta, A. Segal, A. Panda, G. Segev, M. Schapira, J. Feigenbaum, J. Rexford, and S. Shenker. A New Approach to Interdomain Routing Based on Secure Multi-Party Computation. In *Proc. of ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[43] S. Hallyn, S. Graber, D. Engen, C. Brauner, and W. Bumiller. Linux Containers. `https://linuxcontainers.org/`.

[44] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient Support for Evolvable Internetworking. In *Proc. of USENIX NSDI*, 2012.

[45] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, University of California at Berkeley, 2015.

[46] M. Handley. Why the Internet Only Just Works. *BT Technology Journal*, 2006.

[47] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring Control Plane Latency in SDN-Enabled Switches. In *Proc. of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2015.

[48] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile Edge Computing – A

Key Technology Towards 5G. White paper 11, ETSI, Sept. 2015.
[49] IEEE Standards Association. 802.1ag-2007 - IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management.
[50] IEEE Standards Association. 802.1aq-2012 - IEEE Standard for Local and Metropolitan Area Networks - Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks - Amendment 20: Shortest Path Bridging.
[51] IEEE Standards Association. 802.1D-2004 - IEEE Standard for Local and Metropolitan Area Networks - Media Access Control (MAC) Bridges.
[52] IEEE Standards Association. 802.1Q-2014 - IEEE Standard for Local and Metropolitan Area Networks - Bridges and Bridged Networks.
[53] IEEE Standards Association. 802.1s-2002 - IEEE Standards for Local and Metropolitan Area Networks - Amendment to 802.1Q Virtual Bridged Local Area Networks: Multiple Spanning Trees.
[54] IEEE Standards Association. 802.2-1989 - IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 2: Logical Link Control.
[55] Internet2. `http://www.internet2.edu`.
[56] V. Jacobson, B. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, RFC Editor, May 1992.
[57] V. Jacobson and R. Braden. TCP Extensions for Long-Delay Paths. RFC 1072, RFC Editor, October 1988.
[58] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *Proc. of ACM CoNEXT*, 2009.
[59] D. Jansen and H. Buttner. Real-Time Ethernet: The EtherCAT Solution. *Computing and Control Engineering*, 15(1):16–21, 2004.
[60] D. B. Johnson. Routing in Ad Hoc Networks of Mobile Hosts. In *Proc. of Workshop on Mobile Computing Systems and Applications (WMCSA)*, 1994.
[61] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *Proc. of USENIX NSDI*, 2015.
[62] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, RFC Editor, 2010.
[63] M. Kempf. Bridge Circuit for Interconnecting Networks, 1986. US Patent 4,597,078.
[64] T. Kohno, A. Broido, and K. C. Claffy. Remote Physical Device Fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.
[65] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *Proc. of ACM SIGCOMM*, 2007.
[66] T. Koponen, S. Shenker, H. Balakrishnan, N. Feamster, I. Ganichev, A. Ghodsi, P. B. Godfrey, N. McKeown, G. Parulkar, B. Raghavan, J. Rexford, S. Arianfar, and D. Kuptsov. Architecting for Innovation. *ACM SIGCOMM Computer Communication Review*, 41(3), July 2011.
[67] M. Kuehlewind and B. Trammell. Applicability of the QUIC Transport Protocol.

Internet-Draft draft-ietf-quic-applicability-01, IETF Secretariat, October 2017.

[68] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving Convergence-Free Routing Using Failure-Carrying Packets. In *Proc. of ACM SIGCOMM*, 2007.

[69] P. Lapukhov. OSPF Fast Convergence. *INE Blog*, Jun 2010. `https://blog.ine.com/2010/06/02/ospf-fast-convergenc`.

[70] T. Lee, C. Pappas, D. Barrera, P. Szalachowski, and A. Perrig. Source Accountability with Domain-Brokered Privacy. In *Proc. of ACM CoNEXT*, 2016.

[71] J. Liu. Routing Along DAGs. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-155*, 2011.

[72] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *Proc. of USENIX NSDI*, 2013.

[73] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. of USENIX NSDI*, 2013.

[74] A. Malis and W. Simpson. PPP over SONET/SDH. RFC 2615, RFC Editor, June 1999.

[75] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Diot. Characterization of Failures in an IP Backbone. In *Proc. of INFOCOM*, 2004.

[76] J. McCauley, N. Agarwal, B. Raghavan, and S. Shenker. Resilient Routing for the Rest of Us. *In submission*, 2020.

[77] J. McCauley, Y. Harchol, A. Panda, B. Raghavan, and S. Shenker. Enabling a Permanent Revolution in Internet Architecture. In *Proc. of ACM SIGCOMM*, 2019.

[78] J. McCauley, A. Mushtaq, N. Agarwal, S. Ratnasamy, and S. Shenker. Packet-State Load Balancing. *Unpublished*, n.d.

[79] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker. Thoughts on Load Distribution and the Role of Programmable Switches. *ACM SIGCOMM Computer Communication Review*, 49(1), Feb. 2019.

[80] J. McCauley, A. Sheng, E. J. Jackson, B. Raghavan, S. Ratnasamy, and S. Shenker. Taking an AXE to L2 Spanning Trees. In *Proc. of ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[81] J. McCauley, M. Zhao, E. J. Jackson, B. Raghavan, S. Ratnasamy, and S. Shenker. The Deforestation of L2. In *Proc. of ACM SIGCOMM*, 2016.

[82] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[83] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, Mar. 2014.

[84] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[85] Mininet. `http://mininet.org/`.

[86] J. Moy. OSPF Version 2. STD 54, RFC Editor, April 1998.

[87] D. Naylor, M. K. Mukerjee, P. Agyapong, R. Grandl, R. Kang, M. Machado, S. Brown, C. Doucette, H.-C. Hsiao, D. Han, T. H.-J. Kim, H. Lim, C. Ovon, D. Zhou, S. B. Lee,

Y.-H. Lin, C. Stuart, D. Barrett, A. Akella, D. Andersen, J. Byers, L. Dabbish, M. Kaminsky, S. Kiesler, J. Peha, A. Perrig, S. Seshan, M. Sirbu, and P. Steenkiste. XIA: Architecting a More Trustworthy and Evolvable Internet. *ACM SIGCOMM Computer Communication Review*, July 2014.

[88] D. Naylor, M. K. Mukerjee, and P. Steenkiste. Balancing Accountability and Privacy in the Network. In *Proc. of ACM SIGCOMM*, 2014.

[89] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An End-Host Stack for Service-Centric Networking. In *Proc. of USENIX NSDI*, 2012.

[90] ns-3. `http://www.nsnam.org/`.

[91] V. Olteanu and C. Raiciu. Datacenter Scale Load Balancing for Multipath Transport. In *Proc. of ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2016.

[92] Open Networking Foundation. OpenFlow Switch Specification Version 1.5.1. `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`, 2014.

[93] P. Pan, G. Swallow, and A. Atlas. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. RFC 4090, RFC Editor, May 2005.

[94] A. Panda, J. M. McCauley, A. Tootoonchian, J. Sherry, T. Koponen, S. Ratnasamy, and S. Shenker. Open Network Interfaces for Carrier Networks. *ACM SIGCOMM Computer Communication Review*, 46(1), Jan. 2016.

[95] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud Scale Load Balancing. *ACM SIGCOMM Computer Communication Review*, 43(4), 2013.

[96] R. Perlman. An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN. In *Proc. of ACM SIGCOMM*, 1985.

[97] R. Perlman, D. Eastlake, D. Dutt, S. Gai, and A. Ghanwani. Routing Bridges (RBridges): Base Protocol Specification. RFC 6325 (Proposed Standard), RFC Editor, 2011.

[98] J. Postel. Internet Protocol. RFC 791, RFC Editor, September 1981.

[99] J. Postel. NCP/TCP Transition Plan. RFC 801, RFC Editor, November 1981.

[100] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proc. of ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[101] B. Raghavan, T. Koponen, A. Ghodsi, V. Brajkovic, and S. Shenker. Making the Internet More Evolvable. Technical report, International Computer Science Institute, September 2012.

[102] N. Rath. libfuse: Filesystem in UserSpace. `https://github.com/libfuse/libfuse`.

[103] E. Rojas, G. Ibañez, J. M. Gimenez-Guzman, J. A. Carral, A. Garcia-Martinez, I. Martinez-Yelmo, and J. M. Arco. All-Path Bridging: Path Exploration Protocols for Data Center and Campus Networks. *Computer Networks*, 79, 2015.

[104] G. N. Rouskas, I. Baldine, K. Calvert, R. Dutta, J. Griffioen, A. Nagurney, and T. Wolf.

ChoiceNet: Network Innovation through Choice. In *Proc. of the International Conference on Optical Networking Design and Modeling (ONDM)*. IEEE, April 2013.

[105] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proc. of ACM SIGCOMM*. ACM, 2015.

[106] A. L. Russell. *Open Standards and the Digital Age*. Cambridge University Press, 2014.

[107] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)*, 2(4), 1984.

[108] R. R. Sambasivan, D. Tran-Lam, A. Akella, and P. Steenkiste. Bootstrapping Evolvability for Inter-Domain Routing with D-BGP. In *Proc. of ACM SIGCOMM*, 2017.

[109] R. Schroeppel. Hasty Pudding Cipher Specification (Wayback Machine archive). `https://web.archive.org/web/20110717205702/http://richard.schroeppel.name:8015/hpc/hpc-spec`, 1998.

[110] I. Seskar, K. Nagaraja, S. Nelson, and D. Raychaudhuri. MobilityFirst Future Internet Architecture Project. In *Proc. of the Asian Internet Engineering Conference (AINTEC)*, 2011.

[111] P. Shuff. Building a Billion User Load Balancer - USENIX LISA, Dec 2016. `https://www.youtube.com/watch?v=bxhYNfFeVF4`.

[112] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. of ACM SIGCOMM*, 2015.

[113] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *ACM SIGCOMM Computer Communication Review*, 26(2), Apr. 1996.

[114] B. Trammell, M. Welzl, T. Enghardt, G. Fairhurst, M. Kühlewind, C. Perkins, P. S. Tiesel, and C. A. Wood. An Abstract Application Layer Interface to Transport Services. Internet-Draft draft-ietf-taps-interface-03, Internet Engineering Task Force, Mar. 2019. Work in Progress.

[115] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. RFC 1075 (Experimental), RFC Editor, 1988.

[116] Y. Wang, I. Matta, F. Esposito, and J. Day. Introducing ProtoRINA: A Prototype for Programming Recursive-Networking Policies. *ACM SIGCOMM Computer Communication Review*, 44(3), July 2014.

[117] J. Williams. Introducing the GitHub Load Balancer, Sep 2016. `https://githubengineering.com/introducing-glb/`.

[118] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldin, and A. Nagurney. ChoiceNet: Toward an Economy Plane for the Internet. *ACM SIGCOMM Computer Communication Review*, 44(3), July 2014.

[119] X. Yang, D. Clark, and A. W. Berger. NIRA: A New Inter-Domain Routing Architecture. *IEEE/ACM Transactions on Networking*, 15(4), Aug. 2007.

[120] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-Limiting Network Architecture. *IEEE/ACM Transactions on Networking*, 16(6), Dec. 2008.

[121] P. Zave and J. Rexford. The Compositional Architecture of the Internet. *Commun. ACM*,

62(3), Feb. 2019.

[122] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.