©Copyright 2016

Vincent Liu

Improving Fault Tolerance and Performance of Data Center Networks

Vincent Liu

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Thomas E. Anderson, Chair

Arvind Krishnamurthy, Chair

Raadhakrishnan Poovendran

Shyamnath V. Gollakota

Program Authorized to Offer Degree:
UW Computer Science & Engineering

University of Washington

Abstract

Improving Fault Tolerance and Performance of Data Center Networks

Vincent Liu

Co-Chairs of the Supervisory Committee:
Professor Thomas E. Anderson
Professor Arvind Krishnamurthy
UW Computer Science & Engineering

Data center networks are a key component to the explosive growth of cloud computing—enabling the utilization of tens to hundreds of thousands of co-located servers for large-scale computing and services. As applications and data sets continue to grow rapidly, the challenge for data center networks is to keep pace—by providing enough bandwidth while also lowering costs, increasing flexibility, and maintaining reliability.

My thesis is that a key part of the answer is the network's wiring topology: topology has foundational cross-layer effects, and a small amount of intentional asymmetry in the topology can help data center networks meet that challenge.

I present two complementary innovations that demonstrate this.

The first, F10, is a co-design of the network topology and failover protocols to provide efficient, near-instantaneous, fine-grained, and localized recovery and rebalancing for common-case network failures. My results show that following network link and switch failures, F10 has 1/7th the packet loss of current schemes.

The second innovation, Subways, proposes and evaluates a new method to add network capacity by connecting multiple network links per server in an overlapping topology. Using a simulation-based methodology, my work shows that Subways offers substantial performance benefits for popular application workloads: up to a 3.1× speedup in MapReduce and a 2.5×

throughput improvement in memcache for a fixed average request latency, relative to an equivalent-bandwidth network that differs only in its wiring.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . v

Chapter 1: Introduction . . . 1
- 1.1 Summary of results . . . 4
- 1.2 Organization . . . 6

Chapter 2: Background . . . 7
- 2.1 Rack-based Architectures . . . 7
- 2.2 Oversubscribed Folded Clos Networks . . . 8
- 2.3 Equal Cost Multipath (ECMP) . . . 10
- 2.4 TCP-based Transport Layers . . . 11
- 2.5 Tail-sensitive Applications . . . 11
- 2.6 Summary . . . 12

Chapter 3: F10: Fault-tolerant Engineered Networks . . . 13
- 3.1 Failures in Data Centers . . . 13
- 3.2 Failure Recovery in Clos Networks . . . 15
- 3.3 Design Overview . . . 17
- 3.4 The AB Clos Network . . . 18
- 3.5 Handling Failures . . . 21
- 3.6 Load Balancing . . . 32
- 3.7 Prototype and Evaluation . . . 35
- 3.8 Related Work . . . 41
- 3.9 Discussion . . . 42
- 3.10 Final Remarks . . . 46

Chapter 4: Subways . . . 47

4.1 Design Overview . . . 49
4.2 Wiring Types . . . 50
4.3 Adaptive Load Balancing . . . 56
4.4 Detour Routing . . . 60
4.5 Physical Considerations . . . 64
4.6 Evaluation . . . 72
4.7 Related Work . . . 83
4.8 Final Remarks . . . 85

Chapter 5: Conclusions and Future Work . . . 87

Bibliography . . . 91

LIST OF FIGURES

Figure Number — Page

1.1 Two folded Clos networks that use 4-port switches to provide the abstraction of a much larger switch. 2

2.1 A data center network implemented as a folded Clos network with three layers. 9

3.1 Path alternatives in (a) a standard Clos network and (b) an AB Clos network. 16

3.2 A labeled AB Clos network in which the subtrees with dotted blue lines are of type A and the subtrees with solid red lines are of type B. 20

3.3 Illustration of the base cases of local rerouting with a failure at v. 22

3.4 Illustration of pushback when the link from u to v fails (marked by the red 'X'). 24

3.5 TCP congestion window trace with and without failure. 35

3.6 Comparison of throughput of the testbed and the simulator through ten simulated switch failures and the same topology/offered load. 36

3.7 Behavior of F10 under a simulated failure at 10 ms with an all-to-all UDP workload. 38

3.8 Cumulative Distribution Function (CDF) of the congestion losses of both PortLand and F10. 39

3.9 Comparison of MapReduct job performance on PortLand or F10. 40

4.1 A folded Clos data center architecture with Type 0 Subways, $p = 2$. 51

4.2 Example of the server-ToR connection in Type 1 with a single loop of 9 3-port servers. 54

4.3 A two cluster topology for Type 1 (a) and Type 2 (b, c) with $p = 3$ and $l = 3$ and 9, respectively. 55

4.4 A rack-level diagram showing the detour paths of a single hot rack of servers R. 60

4.5 An example side-to-side physical topology for Type 1 and up. 66

4.6 A physical topology for a Paired Row design. 68

4.7 An inter-chassis layout. 70

4.8 The effect of Subways on an all-to-all MapReduce shuffle workload and a range of job sizes. 76

4.9 Speedup of a MapReduce shuffle for different upgrade paths. 77
4.10 Speedup of a MapReduce shuffle for different loop sizes compared to Type 0. 79
4.11 Speedup of a MapReduce shuffle for different per-server port counts compared to Type 0. 80
4.12 Throughput of memcached for various Subways variants for a range of oversubscription ratios. 82
4.13 Detour throughput versus server CPU utilization. 84

LIST OF TABLES

Table Number — Page

3.1 A key to the notation used in this chapter. 18

4.1 The variables that define a Subways architecture. 50

4.2 Approximate maximum wire length for a side-to-side wiring pattern with 2 logical racks per physical rack. 67

4.3 Approximate maximum wire lengths for a paired row design with 2 logical racks per physical rack. 69

4.4 Approximate maximum wire length for a inter-chassis design and a range of port counts. 71

4.5 Comparison of the tail flow completion time of the simulator and the testbed with the same four-rack configurations and single-rack-sink workload. 74

ACKNOWLEDGMENTS

Looking back at the past six years, I could not have predicted my path, but I would not change a single moment. I am endlessly grateful to my advisers, Tom Anderson and Arvind Krishnamurthy, for guiding me along that path and letting me explore any and all problems that interested me. I am especially grateful for all the support both of them have provided to me over the years. From my first conference as a fresh, utterly-lost PhD student to the preparation of this dissertation, I always knew they had my development and my best interests in mind. As I begin to mentor my own students, I sincerely hope that I can begin to emulate your intelligence, kindness and wisdom.

I have also had the privilege to work with many other brilliant professors. Shyam Gollakota and Dan Ports in particular have taught me a lot over the last few years. I have benefited immensely from seeing the way they approach problems and do research. The same is true of the time I spent working with Josh Smith and David Wetherall. It has been a pleasure working with each of them. I would also like to thank Radha Poovendran for agreeing to be on my committee and always being so positive whenever I speak with him. All of these professors have helped to guide me during my time at the University of Washington.

In addition to the professors at UW, there are a great many graduate students to whom I am grateful. The networks lab has been my home, and I still consider our prank on the systems lab one of my crowning achievements. When I arrived, there was a batch of senior students that showed me a glimpse of what I should strive toward. Among them, I got to work with Daniel Halperin, who through his work on F10 showed me how to communicate clearly and effectively. Seungyeop Han, who entered at exactly the same time

as me also taught me a great deal during that initial period. He introduced me to the world of academia, and to this day, he is an amazing source of wisdom. Will Scott and Raymond Cheng entered a year later. They are among the most fascinating people I know. Our technical conversations always teach me something; our non-technical conversations always make me think. Many others have sat in that lab over the years, and though I have only gotten to work with a subset of them—Simon Peter, Qifan Pu, Qiao Zhang and Danyang Zhuo in addition to the people I have already mentioned—I have been blessed to be around such great conversations and smart people. Members of the systems lab (including Jialin Li and Naveen Sharma) and members of the sensor systems lab (Vamsi Talla and Aaron Parks) have been similarly influential in my graduate school career.

Outside of UW, I have thoroughly enjoyed my internships at Google and Facebook, where I have had the opportunity to work with a number of amazing people including Amin Vahdat, Leon Poutievski, Petr Lapukhov and James Zeng.

Finally, I would like to thank my father Hung-wen, my mother Yung-nan, and my two sisters Emmeline and Joceline. They have supported me and taught me both how to work hard and the value of life outside of work. I am also immensely grateful to my loving girlfriend Kate Yang, who has supported me through the last half of this process.

DEDICATION

to my family, friends, mentors and collaborators

Chapter 1

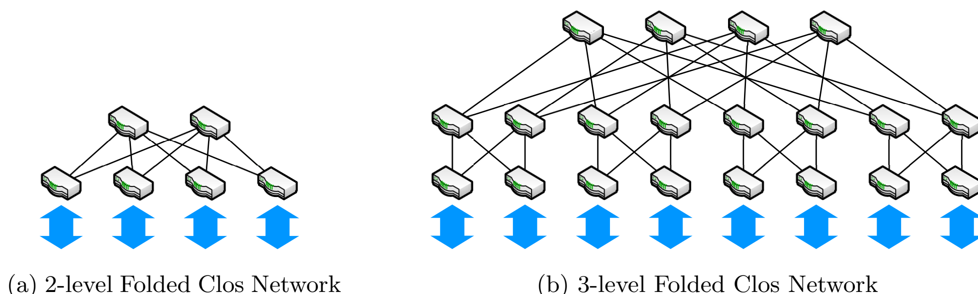
INTRODUCTION

Data centers have existed in one form or another for decades, but they are evolving more quickly than ever before. At a basic level, data centers are facilities devoted to the housing, cooling and maintenance of large amounts of computing infrastructure. For the ENIAC in the 1940's, this meant tens of thousands of vacuum tubes, resistors and capacitors spread across an area equal to half of a basketball court. In today's data centers, this means tens to hundreds of thousands of servers spread across a space two orders of magnitude larger.

Cloud computing, the practice of outsourcing computation and storage to remote servers, owes its recent, rapid rise to the capabilities of data centers; their use has far-reaching and fundamental effects. By aggregating resources, operators amortize capital and operational expenses. By co-locating all of the computers, they enable them to connect to one another at higher bandwidth, lower latency and significantly lower cost. The result is a cheaper, easier to maintain and more efficient computing platform—one that allows applications to scale out to an unprecedented degree that would not be possible otherwise.

The capacity of individual computation and storage elements have not scaled at the same rate as application demand. As cloud applications continue to scale out, their computation and storage must be spread over more and more racks of servers connected by an an increasingly powerful network. Thus, a critical component in the growth of the cloud has been the network that connects the tens to hundreds of thousands of servers within each data center. By some accounts, the total amount of traffic that traverses data center networks exceeds global Internet traffic by an order of magnitude [2, 1].

To handle these traffic demands, data center networks have coalesced around a common structure: the Clos network. Clos networks [21] emulate a single, large switch with a multi-



(a) 2-level Folded Clos Network (b) 3-level Folded Clos Network

Figure 1.1: Two folded Clos networks that use 4-port switches to provide the abstraction of a much larger switch. Folded Clos networks are multi-rooted, multi-level trees where the leaves of the tree serve as both ingresses and egresses to the network. The difference between the two examples is in the number of levels in the tree; more levels results in a bigger network (more ingress and egress ports) using the same port-limited switches. The diagrams show fully-provisioned Clos networks, where each intermediate switch has an equal number of ports going up toward the root and down towards the ingresses/egresses. This is as opposed to oversubscribed and overprovisioned networks. Ingress and egress ports are denoted by the blue arrows.

layer network of relatively small, relatively cheap switches. Figure 1.1 shows two folded Clos networks, where 'folded' indicates that the ingresses to the network are also the egresses. Both examples are composed of switches with four ports each (except the leaves of the trees, which use half their capacity for ingress/egress and half for the Clos interconnect). In Figure 1.1a, when connections enter one of the leaf switches, they are bounced through one of the root switches and back out through the desired egress. In this way, it can connect any ingress to any egress much like a single large switch would. Adding more layers, as in Figure 1.1b, allows the network to scale out using the same four-port switches.

Modern data centers use this exact approach to connect the multitude of servers in each data center, although in practice, switches have higher degree than shown here. Clos networks have been successful at increasing network bandwidth by several orders of magnitude over the past decade [58]. Unfortunately, as data center networks continue to grow, problems remain in their construction:

- *Reliability:* Data center network hardware failures are both frequent and disruptive.

Despite the fact that individual switches and links are relatively reliable with uptimes of $\sim 99.9\%$, a large data center network will have thousands of switches and hundreds of thousands of links. The chance that some switch or link has failed is therefore very high. For example, in a full-deployment of one of Facebook's recent network architectures [14], 99.9% per-switch uptime leads to a 0.6% *fully-up* uptime. If we consider ports/links in addition to switches (there are at least an order of magnitude more links than switches in a data center), failures are even more likely. Recent measurements have shown that network-layer failures are disruptive [26] and can reduce the volume of traffic delivered by more than 40%, even when the underlying network is designed for failure resilience.

- *Cost:* Like reliability, keeping hardware costs manageable is also an ongoing challenge for data center networks. Clos networks scale, but at superlinear cost. Faster and more capable servers, increasing application network-to-compute ratios [31], and kernel bypass techniques [15, 48]—all of these contribute to the challenge of scaling to meet network demand in a cost-effective manner. As a result, the network is a large and growing portion of the total cost of the data center [31]. Moreover, because many data center applications are highly sensitive to tail latencies, networks must be configured with extra capacity and low average link utilization, further increasing costs.

- *Performance:* Inversely correlated to cost is performance. One of the main factors here is how many of a switch's ports to allocate for uplinks to its parents and downlinks to its children. Networks that allocate more ports to downlinks can scale to more nodes or make do with a shallower tree. Networks that allocate more ports to uplinks have more aggregate bandwidth. Our goal is therefore dual in nature: to lower cost for a fixed amount of performance or to raise performance for a fixed network cost.

- *Manageability:* Finally, upgrading a Clos network can be a challenge. Most data

> centers provision only a single link from each server into the network. Not only is this link a bottleneck for both performance and fault tolerance, it is also immutable—upgrades generally involve forklifting in an entirely new data center network and set of servers. Preferable would be an incremental approach to adding capacity while the network continues to carry traffic [14, 58]. Unfortunately, existing methods of adding capacity to servers do not support this type of upgrade.

How, then, do we continue to grow data center networks while improving their reliability, efficiency and manageability? Operators are understandably fearful of changing established practices and abandoning battle-tested techniques. One of the goals of this work is therefore to solve each of the above problems while retaining as many of the the benefits of current industry standards as possible, and without significantly increasing the hardware cost of the network compared to existing solutions. Where possible, we present several similar designs each with a different complexity/benefit tradeoff.

This dissertation proposes that a key part of the answer is the wiring topology of the network. Specifically, *(i) the topology of the network has foundational cross-layer effects, and (ii) intentional asymmetry can help data center networks meet the challenges of scale, reliability, cost, performance and manageability.* Injecting a small amount of controlled asymmetry into the topology, coupled with innovations at other layers can result in large, far-reaching benefits while retaining the benefits of Clos networks. With careful co-design, these benefits go beyond bandwidth and efficiency and result in properties like near-instantaneous failure recovery and incremental upgrades of server capacity.

1.1 Summary of results

This thesis describes the design, implementation and evaluation of a data center architecture that includes two complementary innovations. To be useful, both ideas require a data center architect to understand how they would fit into the operational constraints of a specific data center design. For simplicity, we describe them in the context of a folded Clos data center

network running standard protocols.

1.1.1 F10: A Fault-Tolerant Engineered Network

The first innovation, **F10** is a co-design of the network topology and failover protocols that is the first to provide near-instantaneous, localized recovery for data center network failures. The core of F10 is a novel topology that uses controlled asymmetry to make it easier to do localized repair and rebalancing after failures. We then redesign the routing protocols to take advantage of the modified topology. To satisfy the need for extremely fast failover, we use a local recovery mechanism that reacts almost instantaneously. This comes at the cost of additional latency and increased congestion. In our system, local rerouting after a persistent failure eventually triggers a slightly slower pushback mechanism that redirects traffic flows before they reach the faulty components. If the failure continues to persist, a centralized scheduler rearranges traffic on a much slower time scale in order to create as close to a optimally rerouted configuration as possible. We also introduce a failure detector that benefits from (and contributes to) the speed of our failover protocols while providing fine-grained information not available to traditional failure detection methods.

Other work has shown that local repair is possible at the cost of significant added hardware relative to a standard folded Clos network [23, 28, 29], so this work can be seen as either improving the speed of repair in Clos networks or in reducing the hardware cost of fast repair. A limitation of this work is that we assume that we can change both the network topology and the protocols used between network switches. Complementary work in Software-defined Networking (SDN) has started to open up data center network switches to be re-programmable [18].

F10 can almost instantaneously reestablish connectivity, even in the presence of multiple failures. Further, it can quickly reestablish optimal routes. Our results show that following network link and switch failures, F10 has 1/7th the packet loss of prior approaches. A trace-driven evaluation of MapReduce performance shows that this lower packet loss yields

a median 30% application-level speedup.

1.1.2 Subways: A Case for Redundant, Inexpensive Data Center Edge Links

The second innovation, **Subways** explores how best to add network capacity to servers. Subways is a co-design of the network wiring, routing, and load balancing algorithms when there are multiple links per server. Instead of always wiring servers to a single switch, we cross-wire some server links to adjacent switches in an overlapping pattern.

A counterintuitive result is that, for typical workloads, it is possible to achieve better than a proportional performance improvement when upgrading a data center network in this way. In other words, a doubling of network capacity can result in much better than a 2× performance improvement on the same hardware. We leverage the additional asymmetry and the path diversity of the Subways wiring topology to reduce backbone utilization and improve load balancing. Using a simulation-based methodology, we show that Subways offers substantial performance benefits for popular application workloads: up to a 3.1× speedup in MapReduce and a 2.5× throughput improvement in memcache for a fixed average request latency, relative to an equivalent-bandwidth network with the same number of links and switches, but that differs only in its wiring.

1.2 Organization

The remainder of this thesis proposal is organized as follows. Chapter 2 provides a brief introduction to various best practices in modern data center network design. Chapters 3 and 4 present the two innovations introduced above: F10 and Subways. Finally, Chapter 5 describes my conclusions and future work.

Chapter 2
BACKGROUND

Today's data centers are truly massive facilities that house up to hundreds of thousands of networked servers and a great deal of other infrastructure. Connecting these servers is the data center network, which spans thousands of network switches and hundreds of thousands of links.

Beyond the presence of networked servers, data centers can vary greatly from one deployment to another—sometimes even within the same company. Even so, there are several features that are common among most state-of-the-art data centers. These best practices are necessitated by scalability, cost, and practical limitations.

2.1 Rack-based Architectures

The first is that servers and the network switches that connect them are stacked in physical racks. A typical rack is about half a meter wide and 2 meters tall, which amounts to space for 42 Rack Units (RUs) of equipment. Servers are traditionally 1 RU, but other form factors exist; some deployments have servers that occupy multiple RUs, some have servers that take a fraction of an RU. A typical physical server rack will therefore have approximately 20-120 servers with one or more Top-of-Rack switches (ToRs) that serve as the rack's gateways to the rest of the data center network.

The canonical rack configuration has all of the servers in a rack connecting to a single ToR switch, but other configurations are possible. For instance, some data centers pack multiple *logical* racks within a single *physical* rack. Each logical rack may connect as before, to its own ToR, with the difference being that multiple logical racks and ToRs inhabit the same physical enclosure. In the remainder of this thesis, the term *rack* refers to a logical rack

unless otherwise specified.

Still other data centers may wire links in parallel from each server to the ToR switch. Link Aggregation Groups (LAGs) [3], for example, pair multiple parallel links between the same two devices such that the link group behaves like a single link with the combined bandwidth of its members. LAGs do so by distributing either flows or packets across the available interfaces using consistent hashing [32] or a similar load balancing algorithm. Likewise, operators use Multi-Chassis LAGs (MC-LAGs) [4] to pair switches together rather than just links. With an MC-LAG, servers can connect to two or more switches that appear as a single switch—one that is resilient to both a link failure and a switch failure.

Rack-based architectures simplify installation and management. They allow operators to expand capacity by rolling in a pre-populated rack. They also allow the use of short, inexpensive cables to connect each server to its ToR(s). Even for today's data centers, where server-to-ToR links of 10 Gbps to 50 Gbps are common, these links can be implemented using cheap copper cables rather than more expensive optical cabling. Optical cabling is then used to implement the Clos network that connects racks together as described next.

2.2 Oversubscribed Folded Clos Networks

Above the rack level, ToR switches are connected though a large multi-rooted tree of switches that spans the entire data center. The ToRs serve as the leaves in these trees.

More specifically, data center networks are generally some variant of a folded Clos topology [21]. Most notable of these are the FatTree [10] and its subsequent extensions, Microsoft's VL2 proposal [27], and Facebook's Altoona design [14]. Although inspired by the concept of a fat-tree [40] in which links increase in capacity exponentially as you travel up toward the root, these proposals use multi-rooted, multi-stage tree structures identical to a folded Clos network [38]. The primary distinction between some of these designs and a traditional Clos network is that they allow traffic to take a shortest path rather than requiring that all traffic go through a root switch.

Whatever the routing methodology, the benefit of using cheaper, commodity switches is

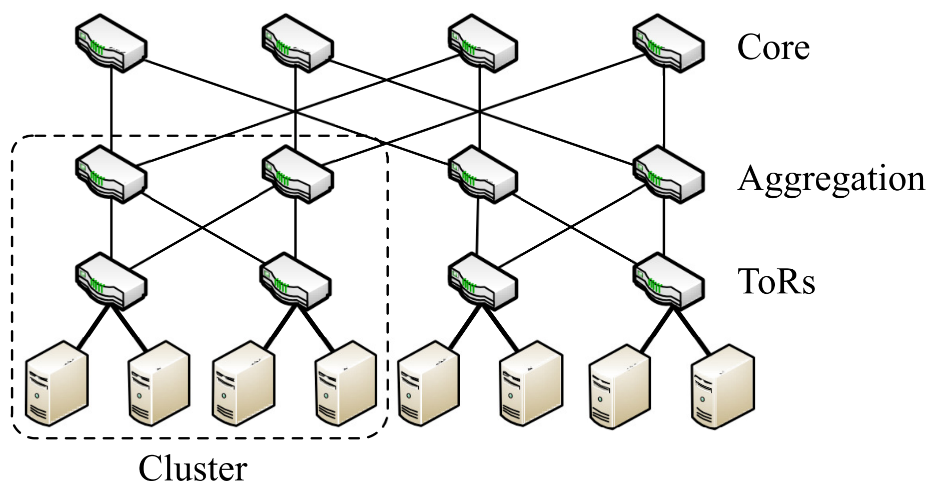


Figure 2.1: A data center network implemented as a folded Clos network with three layers. The ToRs connect a rack of servers together, the aggregation switches connect several ToRs together into a cluster, and the core switches connect the entire data center together. For simplicity, we omit from several of our figures the doubled subtrees generated by folding the root uplinks into downlinks. In practice, individual switches will be much higher degree than shown. The configuration of any particular data center network depends on port count, switching capacity, and demand constraints.

twofold. First, it decreases the cost of the network substantially compared to legacy designs that use specialized routers at the upper layers of the tree. Second, these designs include much more path diversity within the network.

A canonical folded Clos data center network, illustrated in Figure 2.1, has three layers: (1) ToR switches, which connect a single physical or logical rack of several dozen servers, (2) aggregation switches, which connect the ToRs of tens of racks into a single cluster, and (3) core switches, which connect the clusters of the entire data center together. Each layer has many physical switches operating in parallel.

For switches with a given port count, the number of layers determines the maximum number of servers the network can connect. Connecting more servers requires adding more layers to the network—something that many data centers already do.

Although copper links connect racks to ToR switches, the rest of the data center network uses more expensive fiber optics between switches: from ToR switches to the aggregation

layer and then at higher layers. This is because electrical signaling is not feasible for cables of more than a few meters. Since optical connections are expensive, the tree is often thinned immediately above the ToR level. That is, the aggregate bandwidth from servers into a ToR is often more than that of the ToR's optical uplinks. Higher levels of the tree are often even more oversubscribed, again for cost reasons [27, 58]. Note that for simplicity, Figure 2.1 depicts a *non-oversubscribed* network in which half of the ports are used as downlinks to connect nodes within the same subtree, and half are used as uplinks to access other parts of the tree.

2.3 Equal Cost Multipath (ECMP)

Clos network topologies allow data center operators to build large networks out of relatively inexpensive switches of limited degree. Each individual switch has only a tiny fraction of the network's switching capacity, but the network as whole provides high bandwidth and redundancy to servers by using many paths through many switches.

The most common technique for taking advantage of this path diversity is ECMP [6]. At a high level, ECMP allows switches to distribute flows over multiple possible next-hops with equal probability. In the example in Figure 2.1, each ToR has two possible next hops as does each aggregation switch. If all of those switches use ECMP, a given server's flows will be spread evenly across its two aggregation switches and four core switches. Here, the symmetry of the network ensures that between any two hosts, traffic will use all four available paths with equal probability. In practice, there are often hundreds of equal-cost paths between any pair of servers.

An important point to note is that, with ECMP, switches typically pin entire flows to single physical path to prevent performance-degrading TCP reordering effects (explained below). They accomplish this by taking a consistent hash over a subset of each packet's header. For instance, a common technique is to calculate a CRC or XOR over each packet's 5-tuple (i.e., source/destination IP addresses, source/destination ports, and protocol number). Given a table of possible next hops, the index of the entry to use is result of the

CRC/XOR modulo the number of entries in the table. Consistent hashing allows each switch to make independent ECMP decisions while also guaranteeing that all packets from a given connection flow along the same path.

ECMP handles failures by relying on the control plane protocol (e.g., OSPF [44] or BGP [54]) to remove failed next-hops and update the routing tables to reflect unreachable destinations. ECMP will continue to spread load randomly over available next hops—an approach that can result in load imbalance as we describe in Chapter 3.

2.4 TCP-based Transport Layers

Although a variety of transport protocols are in use, most rely heavily on TCP [51]. TCP is a reliable streaming protocol that retransmits packets lost due to congestion or hardware failures. A fundamental part of TCP dating to its origins as a protocol for wide area communication is to reduce its rate if packets arrive out of order; a skipped sequence number is taken to indicate a drop. It is because of this property that ECMP hashing keeps packets from a single flow together on the same path.

An alternative is to rethink the end host protocol to take advantage of path diversity with a protocol like Multipath TCP [52]. Failures then result in more traffic on the remaining paths. However, such protocols can sometimes exacerbate performance issues in data centers [12]. In addition, many data centers do not have the ability to control the protocol in the host operating system.

2.5 Tail-sensitive Applications

Of the pieces of a data center, applications are the most diverse. Not only is every data center host to a wide array of applications, different data centers can have very different workload characteristics. However, a common pattern among applications is a dependence on tail latency.

For example, imagine assembling a complex social-network web page like `facebook.com`. A single web request needs to pull data and computation from a variety of sources (e.g.,

advertisements, recent friend activity, personalized news, etc.). As mentioned in Chapter 1, this data and computation is increasingly spread over more and more servers; a recent Facebook study showed that 44% of all memcache requests contact 20 or more memcached servers [46] and that fan-outs of several hundred are not rare.

The same is true for analytics frameworks like MapReduce [22], which spreads computation over many machines by splitting computation into two primary stages: the map stage and the reduce stage. Data is sent across the network from mappers to reducers between stages.

In both of the above applications, tail latency plays a large role in the completion time of requests. If there are stragglers, applications need to either wait for the straggler, repeat the request or continue with incomplete results. This application dependence on tail latency can have an impact on network design. For example, a non-oversubscribed Clos network with no switch or link failures using ECMP and TCP is likely to provide reasonably fair service to all application flows. As we increase the oversubscription of the network, tail latencies go up accordingly even though the network may have enough capacity to handle the *average utilization*.

2.6 Summary

Data center networks are diverse, but the above features outline the key architectural elements and restrictions of their design. Though these features have made today's massive data centers possible, as discussed in Chapter 1, issues remain in the reliability, cost, performance and manageability of these networks. The remainder of this thesis details how cross-layer design and a small amount of intentional asymmetry can address these issues.

Chapter 3

F10: FAULT-TOLERANT ENGINEERED NETWORKS

The first challenge we examine relates to failure recovery. The use of a large number of commodity switches opens up questions regarding what happens when links and switches fail. As we discussed in the previous chapter, we restrict ourselves to assuming only network-layer changes and not end host protocol changes.

With network-layer failure recovery, one possibility is a centralized manager that collects topology and failure information from the switches. This then generates the current set of routes that exclude failed parts of the topology and disseminates these routes back to the switches. Centralized route management is both simple and flexible—a reasonable design choice provided that failures do not occur very often. However, it is also relatively slow.

Our goal in F10 is to design a data center network architecture that can quickly recover and rebalance after failures, without any additional hardware relative to a standard Clos network. To motivate this approach, we begin by outlining the results of previous measurements of data center network failures and then discuss the implications of these results on the design of fault-tolerant data center networks.

3.1 Failures in Data Centers

A recent study by Gill et al. provides insight into the characteristics of data center network failures [26]. The authors found that a large majority of devices are failure-free over the course of a year; commodity switches are mostly reliable. Their data also shows, however, that there are a large number of short-term failures, that link failures are common and that the network responsiveness to failures is limited. We emphasize a few results from their study:

- **Many failures are short-term.** Devices and links exhibit a large number of short-term failures. In fact, the authors observed that the most failure-prone devices have a median time-to-failure of 8.6 minutes.

- **Multiple failures are common.** Devices often fail in groups. 41% of link failure events affect multiple devices—often, just a few (2–4) links, but in 10% of cases, they do affect more than 4 devices. There are also often multiple independent ongoing failures.

- **Some failures have long downtimes.** Though most failures are short-term, failure durations exhibit a long tail. Gill et al. attribute this to issues such as firmware bugs and device unreliability. Hardware that fails often stays down and contributes heavily to network-level unavailability.

- **Network faults impact network efficiency.** The data centers studied by Gill et al. have 1:1 redundancy dedicated to failure recovery, yet the network delivered only about 90% of the traffic in the median failure case. Performance is worse in the tail, with only 60% of traffic delivered during 20% of failures. This suggests better methods are needed for exploiting existing redundancy.

- **Failures impact application performance.** More generally, because packets from a single flow are always kept together on the same path, failures severely and inordinately impact a subset of network flows. While the number of impacted flows may be small relative to the total number of flows in the network, for applications that are tail-latency sensitive, even a few stragglers can significantly affect the end-to-end performance of the system.

The authors assume a model where hardware is either up or down and transitions between those two states. However, their data suggests that there are also partial, or *gray*, failures in which hardware continues to work partially, but loses a certain percentage of packets. There is thus an additional concern:

- **Detection of gray failures.** Existing protocols like Bidirectional Forwarding Detection (BFD) mark links as down after losing a certain number of heartbeats and as up after a brief handshake. Within a short time frame, it is difficult to distinguish between (*i*) a complete failure, where no packets are getting through, (*ii*) a situation where the switch is overloaded and unlucky with the heartbeats, and (*iii*) partial failures, which might allow a handshake and so appear to be working.

3.2 Failure Recovery in Clos Networks

The papers introducing FatTrees and related proposals [10, 11, 45] discuss basic failover mechanisms, but they are principally focused on achieving good aggregate bandwidth with commodity switches [10]. These proposals are resilient to faults, but are inherently limited in their ability to recover quickly and gracefully from faults. There are at least three reasons for a lack of these properties in Clos networks.

Limited local rerouting: While modern data centers have a variety of failover mechanisms, few are truly local. For instance, data centers that use a link-state protocol (e.g., OSPF [44]) send updates across the entire network before convergence. PortLand [45] uses a centralized topology manager. VL2 [27] suggested detouring around a fault on the upward path, but it does not reroute around failures on the downward path because (as we explain below) there is only one path from any given root to a leaf switch.

Failure information must propagate to many and distant nodes: This deficiency goes beyond the lack of a suitable protocol. Consider Figure 3.1a, which for ease of exposition, depicts a *non-oversubscribed* 4-layer folded Clos network. No parent or grandparent of the failed node has any downlink path to the affected subtree. This property follows from the fat-tree-style construction that there is only ever one downlink path from the root of a subtree to any of its children. Among the nodes whose routes could reach a failed node, only those located *lower in the tree than the failure* have a route that avoids the failure. In other words, *no protocol that informs only nodes in the top portion of the tree will restore*

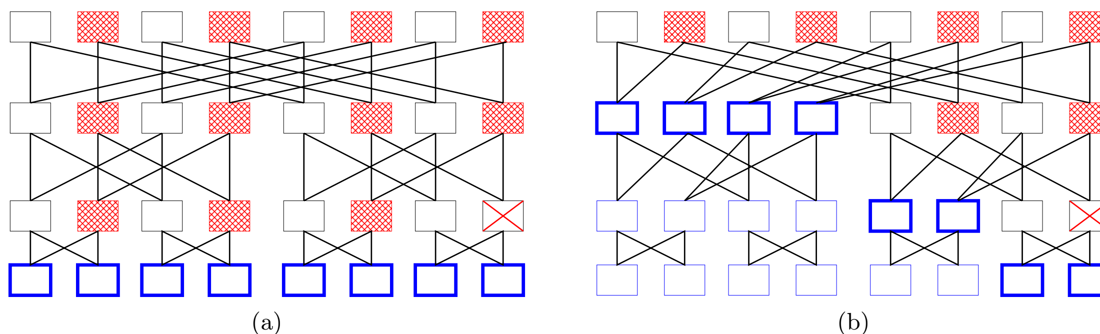


Figure 3.1: Path alternatives in (a) a standard Clos network and (b) an AB Clos network. The X indicates a failure, and the red hashed rectangles represent switches that lose connectivity to the children of the failure as a result of the failure. Blue bordered rectangles represent switches that are affected by, but have alternative paths around the failure. Bold blue borders indicate the frontier of this zone—the closest switches that need to be informed of the failure in order to route around it. In the AB Clos network, more switches are affected, but more have alternatives, and they are closer to the failure. These figures depict fully-provisioned networks, but similar results apply to oversubscribed and overprovisioned networks.

connectivity. In the case of a failure on the downward portion of a path, any detour or pushback/broadcast protocol will be forced to travel from the parent of the failure all the way back to every node in the entire tree lower than the failure.

Some data center architectures include redundant paths in each subtree in order to correct for the topological need to involve distant nodes in failure recovery. While this provides a degree of fault-tolerance, it also requires extra hardware or decreases the maximum number of hosts the network can handle for a fixed amount of hardware. The decrease is exponential in the number of layers of the tree. We limit ourself to failure recovery mechanisms that use no additional hardware relative to a traditional Clos network.

Irregular tree structure because of long-term faults: While data center operators aim to rapidly repair or replace failed equipment, because repairs happen on a human time scale, failures can persist for a considerable period of time. During this time, the local, random path choices of ECMP can result in poor load balancing. For example, in

Figure 2.1 a failure of an aggregation switch renders two of four core switches unusable for any traffic sent from or destined to the switch's cluster. The remaining two core switches will persistently have higher load. Multiple failures make this problem even worse.

In our view, it is crucial that data center networks gracefully handle missing links and loss of symmetry. A negative example of this is the simple application of ECMP, which spreads load across all next-hops randomly. When a link fails, ECMP will spread the load from the failed link to all remaining next-hops at a local level. Unfortunately, local rebalancing does not imply global rebalancing [11, 12].

3.3 Design Overview

Taking the above concerns into account, we create an engineered network and routing protocol that can rapidly restore network connectivity and performance. Our system, F10, relies on the following ideas:

AB Clos network: We introduce a novel topology, the AB Clos network, that is essentially a reworked Clos network with better fault tolerance properties. By skewing the symmetry of a traditional Clos network, the AB Clos network allows for efficient local rerouting. The benefits come at almost no cost. The new topology requires no extra hardware, does not lose bisection bandwidth, and has similar properties to standard Clos networks: unique paths from a root to leaf, non-blocking behavior, redundancy, and compatibility with Valiant Load Balancing.

Local rerouting: To satisfy the need for fast failover, we use a local recovery mechanism that is able to reroute the very next packet after failure detection. Because we fix the topology of the network, we can design a purely local mechanism that is initiated and torn down at the affected switch and does not cause any convergence issues or broader disruptions.

Pushback notification: The reroute uses extra hops then the global optimum. Our system adds a slightly slower pushback mechanism that removes the additional latency, reducing

Notation	Definition or Value
k	# of ports per switch, e.g., 24
$L+1$	# of levels in the network, e.g., 3
p	$k/2$: # of up/downlinks per switch
N	$2p^{L+1}$: # of end hosts in the data center
b	$\lceil \log_2(p) \rceil$: # of bits per level in a node location
$\mathit{prefix}(a, i)$	$a \gg (ib)$: relevant prefix of location a at level i
$\mathit{same_prefix}(a, a', i)$	$(\mathit{prefix}(a, i) \equiv \mathit{prefix}(a', i))$: whether a and a' share a prefix at level i

Table 3.1: A key to the notation used in this chapter.

the impact on congestion of local recovery.

Global re-optimization: On a much slower time scale, a centralized scheduler rearranges traffic to optimally balance load, despite failures.

Failure Detector: The lightweight and local nature of our failover protocols means that we can be more aggressive in marking links and switches as down, improving network performance. Our failure detector also provides and uses finer-grained information about the exact loss characteristics of the connection.

To accomplish the above, we assume a few things about the hardware. On the most basic level, we assume that we can modify the control plane of switches to execute our protocols locally and that switches can do local neighbor failure detection. We also assume the presence of a fault-tolerant centralized controller, as in PortLand. For flow scheduling, we assume switches support consistent flow-based assignment for each source-destination pair. Our system can also benefit from the ability of switches to randomly place flows based on configured weights calculated by the central controller; however, this weighted placement is not essential for correct operation.

3.4 The AB Clos Network

As we saw in Section 3.2, the standard Clos network has a structural weakness that makes it difficult to locally reroute around network failures. I introduce a novel topology, the AB Clos network, that skews the symmetry of a traditional Clos network to address this issue.

The key weakness in the standard Clos network is that all subtrees at level i are wired to the parents at level $i+1$ in an identical fashion. A parent attempting to detour around a failed child must use roundabout paths (with inflation of at least four hops) because all paths from its $p-1$ other children to the target subtree use the same failed node. The AB Clos network solves this problem by defining two types of subtrees (called *type A* and *type B*) that are wired to their parents in *two different ways*. With this simple change, a parent with a failed child in a type A subtree can detour to that subtree in two hops through the parents of a child in a type B subtree (and vice versa), because those parents *do not* rely on the failed node.

We now make the design concrete. Let k be the number of ports on each switch element, and L be the number of levels; as in the standard Clos network we use $p = k/2$ ports each for uplink and downlink at each switch, and can connect a total of $N = 2p^L$ end hosts in a rearrangingly non-blocking manner to the network. Table 3.1 contains a summary of the notation we use in this chapter.

Figure 3.2 shows the labeled structure of an AB Clos network for $k = 4$ and $L = 3$, explained in the next few paragraphs.

Connectivity. For levels numbered 0 through L, each level $i < L$ contains $2p^L$ switches arranged in $2p^{L-i}$ groups of p^i switches.[1] Each group at level i represents a multi-rooted subtree of p^{i+1} end hosts with p^i root switches. The distinction between the standard version and an AB Clos network is in the method of connecting these root switches to their parents.

Let j denote the index of a root node numbered 0 through $p^i - 1$ in level i. In a **type A** subtree, root j will be connected to the p *consecutive parents* numbered jp through $(j+1)p-1$. A standard Clos network contains only type A subtrees whereas in an AB Clos network, only half the subtrees are of type A. The remainder are of **type B**, wherein children connect to parents with a *stride of* p^i: root j is connected to parents j, $j+p^i$,

[1] The top level ($i = L$) has one group of p^L switches, using all ports for downlinks.

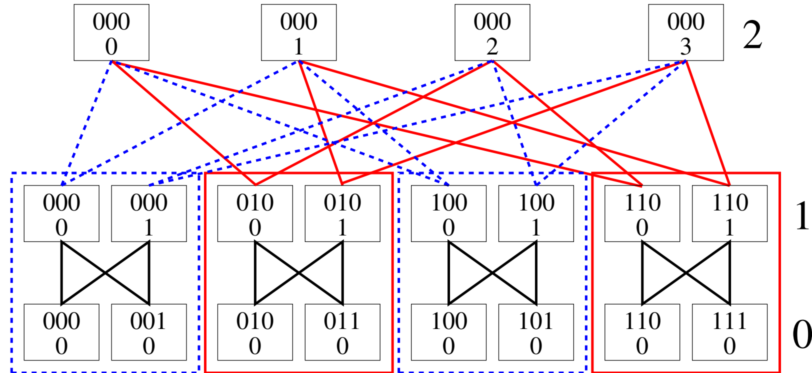


Figure 3.2: A labeled AB Clos network in which the subtrees with dotted blue lines are of type A and the subtrees with solid red lines are of type B. The numbers to the right of the tree are the *level*, the top number in each switch is the *location*, and the bottom number is the *index*.

$j + 2p^i$, etc.

Addressing and Routing. Three values uniquely identify any switch in the system:

- *level* i – The level of the subtree of which it is a root.
- *index* j – The roots of a specific subtree are consecutively numbered as described above.
- *location* – The location of a node is an $Lb + 1$-bit number constructed such that all nodes in the same level i subtree share a prefix of $(L-i)b+1$ bits that encodes the path from the root group to the subtree, where $b = \lceil \log_2 p \rceil$. The location has the format: $(b+1$ bits for level $L).(b$ bits for level $L-1)\ldots(b$ bits for level $i+1)$, concatenated with ib zero bits for levels i through 0.

In the absence of failures, routing occurs much like in PortLand [45]—each packet is routed upwards until it is able to travel back down, following longest-prefix matching. By construction, each subtree owns a single location address and the roots of a subtree can access one child in each of its subsubtrees. When a packet's destination lies within the subtree rooted in the current node, it will be routed downwards, otherwise it is forwarded

upward.

Versus a standard Clos network. Revisiting Figure 3.1, we see that this rewiring allows nodes in subtrees of a different type to route around failures, in addition to nodes on a lower level that already had alternate paths. While the number of switches with affected paths increases, the total number of failed paths stays the same, and therefore the effects of the failure are distributed across more switches. As a consequence, more nodes have alternate paths, and there are alternatives closer to the failure.

3.5 Handling Failures

Our failover protocol consists of three stages that operate on increasing timescales. (1) When a switch detects a failure in one of its links, it immediately begins using *local rerouting* to reroute the very next packet. (2) Since local rerouting inflates paths as well as increases local congestion, the switch initiates a *pushback protocol* that causes upstream switches to redirect traffic to resume using shortest paths. (3) Finally, to deal with long-term failures that create a structural imbalance in the network, a *centralized rerouting* protocol determines an efficient global rearrangement of flows. In addition, the key to fast failure recovery is rapid and accurate failure detection, which is discussed at the end of this section.

3.5.1 Local Rerouting

Our first step after a failure is to quickly establish a new working route using only local information. We explain this using Figure 3.3, which shows a 3-level AB Clos network with $k = 6$. We label nodes u, v, and w, where v has failed.

Note that local rerouting for **upward links** in any multi-rooted tree is simple. A child (w) can route around a failed parent (v), by simply redirecting affected flows to any working parent. This restores connectivity without increasing the number of hops or requiring control traffic. In the unlikely event that all parents have failed, the child drops the packet; an alternative route will soon be configured by the pushback schemes discussed later unless the

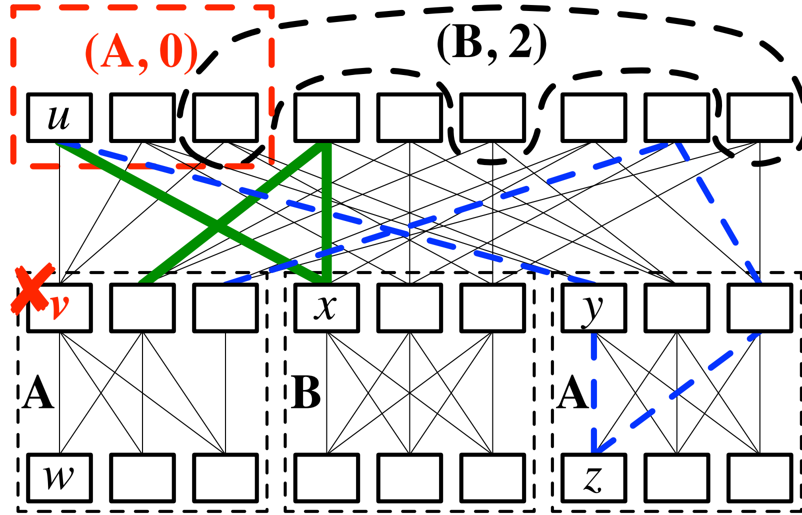


Figure 3.3: Illustration of the base cases of local rerouting with a failure at v. In the upward direction, w avoids v by routing to any other parent. Downward, u must find detours that avoid the failure group (A,0). The bold green path shows Scheme 1 rerouting through a type B child x, and the dotted blue path shows Scheme 2 rerouting through a child y of same type A.

node is a leaf node. Most data center services are designed to tolerate rack-level failures. Alternatively, each leaf node can be wired into multiple ToR switches.

The rest of this section discusses rerouting of traffic for failed **downward links**. This case is significantly more complex, because when a child (v) fails, its parents (e.g., u) lose the only working path to that subtree (identified by *prefix*(v)) that follows standard routing policy. Instead, we propose two local detouring schemes. The first mechanism results in shorter detours, but $p/2$ failures located at specific locations can cause it to fail. The second mechanism succeeds in more cases, but will have longer paths.

Scheme 1: three-hop rerouting. In most cases, we can route around a single failed child in an AB Clos network with two additional hops (three hops in total, but one replacing the link that would have been traversed anyway), without any pre-computation or coordination.

Suppose, without loss of generality, that the failed child (v) is located in a type A subtree. By construction, the parent (u) has connections to $p/2-1$ children in type A subtrees, and

$p/2$ children in type B subtrees. Each of these children has $p-1$ other parents (u's siblings), which all have a link into the affected subtree. By detouring through one of its siblings, u can establish a path.

Not any sibling will work. With only local information, u must assume that the entire switch v has failed, rather than just the link $\langle u, v \rangle$. If so, none of the other parents of v have a route to the affected subtree. We call this set of v's parents a *failure group* and identify it by a tuple (t, j) consisting of v's *subtree type* t and its *index* j, since each parent is connected to the jth node in all type t subtrees. In this example, we would denote the failure group of v as $(\mathrm{A}, 0)$. Figure 3.3 shows $(\mathrm{A}, 0)$ and $(\mathrm{B}, 2)$ failure groups.

All of u's children in type A subtrees only have parents in the $(\mathrm{A}, 0)$ failure group, and thus cannot reach the target prefix. Thus, in Scheme 1, u will simply pick a random child, say x, in a type B subtree. By construction, x has parents in all type A failure groups, and thus *any parent of x except u does not route through v.* One of the alternate paths from u to v's subtree is shown by the bold, green line in Figure 3.3. This does not exist in a standard Clos network.

Multiple failures can be handled in most cases. When failures are located on different levels of the tree, Scheme 1 will always find a path. Multiple failures on the same level can sometimes block Scheme 1. For the first hop, u has $p/2$ links into type B subtrees; if none of these links work ($p/2 + 1$ targeted failures) then u must use Scheme 2. At the second hop, if x has no other working parents (p targeted failures and a $p/2$ random choice) then the scheme fails and packets will be dropped for the brief period until the pushback mechanism (described in Section 3.5.2) removes u from all such paths. At the third hop, if the link from u' into the affected subtree has also failed (2 targeted failures and $(p/2)(p-1)$ random choice), u' will invoke local rerouting recursively.

Scheme 2 – five-hop rerouting. We saw that in some cases of at least $p/2 + 1$ failures, Scheme 1 will fail because u will have no working links to type B subtrees. This situation trivially arises in the case of any single failure in a standard Clos network, so our work can

also be seen as showing how to do local rerouting in a standard Clos network. Scheme 2 uses u's type A children, but it must go two levels down to find a working route to v, for a total of four additional hops in the detour path. One such path is illustrated in Figure 3.3 using the bold, dashed blue line. In Scheme 2, u picks any type A child $y \neq v$ in a different type A subtree, y picks any of its children, and that child proceeds to use normal routing to v's prefix after ensuring it routes through a parent (y's sibling) not in a currently-known failure group. This results in a five-hop path from u to the target prefix. Scheme 2 can fail in the presence of sufficiently many (at least p) targeted failures and unlucky random choices. These unlikely cases will be resolved by our pushback schemes, described next. *With fewer than p failures, local rerouting will always succeed.*

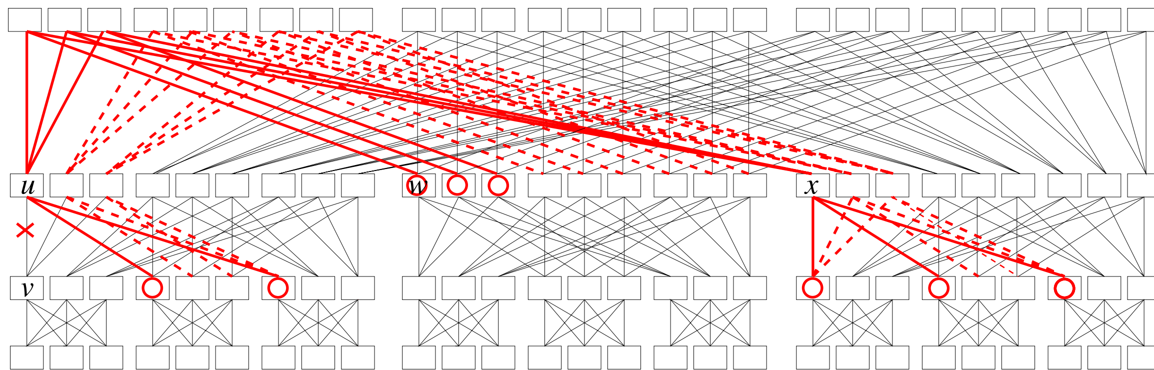


Figure 3.4: Illustration of pushback when the link from u to v fails (marked by the red 'X'). Solid red lines are the paths along which the notification travels, and the switches with red circles are the set of nodes that need to be notified of the failure. In the case of the entire switch v failing, the dashed red lines show the paths along which associated notifications travel and state would be installed at all the endpoints they touch.

3.5.2 Pushback Flow Redirection

The purpose of local rerouting is to find a quick way to reestablish routing immediately after detecting a failure. The detour paths it sets up are necessarily inflated, and the schemes we use can sometimes fail although a working path exists. We introduce pushback routing to reestablish direct routes and handle cases where local rerouting fails, but where connectivity

is still possible. Pushback flow redirection solves both of these issues by broadcasting failure notifications back to the closest switch that has an alternate route that does not include the failure. The AB Clos network enables these notifications to occur closer to the source of the failure than in a regular Clos network. Reducing the number of notifications speeds recovery and minimizes network state.

Consider Figure 3.4, which shows a 4-level AB Clos network built with 6-port switches. This figure illustrates the extent of pushback propagation in the network when the link $\langle u, v \rangle$ has failed. A total of 14 pushback messages are sent (indicated by the bold red lines), and state has to be installed at the 8 switches marked with red circles. Note that in our pushback scheme—conversely from the local rerouting schemes—all messages indicate link failures, not node failures. If the entire node v had failed, u's two siblings would also send pushback messages along the red dashed lines, for a total of 32 additional messages and an additional 12 switches installing state. For pushback, the main difference between AB Clos networks and standard Clos networks is that AB Clos networks can install state higher in the tree, at fewer nodes. As a result, pushback message travel less far in AB Clos networks and the network will converge to optimal working routes more quickly.

Pushback algorithms. There are three important scenarios in which a switch u will push failure notifications to its neighbors:

1. u cannot route to some prefix in its subtree, either because of the failure of an immediate child v or upon receiving a notification from v of a failure further downstream. Then u will broadcast that it can no longer route to the affected prefix to all of its neighbors excluding v.

2. When all uplinks from u have failed, u can only route traffic destined to its own prefix, and will inform its children so that they route upward traffic to other parents.

3. When all non-failed uplinks from u are partially affected by failures, there may exist some external prefixes for which u is unable to route traffic. u informs its children of

these partial failures so that they can use upward detouring for those prefixes.

To handle these scenarios, we define two types of pushback messages. **PBOnly** messages indicate that the sender cannot route to the specific prefix indicated in the message. **PBExcept** messages mean that the sender cannot reach any prefix except its own subtree (or the subtree indicated in the message). Together, **PBOnly** and **PBExcept** can represent any set of routable prefixes. Note that **PBOnly** messages are used in scenario 1 described above, **PBExcept** messages match scenario 2, and a combination of both is used in scenario 3.

Downward failures. The most common pushback case is scenario 1, in which any downlink fails. For this case, the detecting node would send a **PBOnly** message to all neighbors so that they no longer send it traffic to route down that link.

When a node n receives a **PBOnly** message telling it that the edge $\langle u, v\rangle$ has failed, how does it know whether it can route around the failure—in which case it installs pushback state locally and does not forward the message on—or whether it needs to forward the notification on to its neighbors? The intuition behind this is that if a node n can connect to a root node that node u cannot (in the absence of failures), then n has paths using this root that can reach v's subtree without going through the failed edge. Thus when the edge $\langle u, v\rangle$ fails, n has an alternative path to v's prefix if and only if it is connected to such a root.

There is one trivial way that n is guaranteed to be wired to a root that u is not: when n is located at a lower level than u, then at most one of its parents routes through u, and an alternative path exists. In a more complex case that occurs in AB Clos networks (and not in standard Clos networks), pushback state can be sometimes be stored higher in the tree. However, determining when this is the case is not as simple as comparing node levels.

To implement a method by which n at a level above u can know that it has an alternative root, we use *subtree type stack* that represents the types of the trees on the path from a given switch to the roots of AB Clos network. When a switch that receives a pushback

notification has the same type stack as the originator (or partial type stack, if the recipient is higher in the tree), then it has no alternative route and must forward the message on to its neighbors. In Figure 3.4, u and w are both have stacks {A}, while x has a type stack $\{B\}$. Since u and w have the same type stack, when v fails neither u nor w can route around it, while x can as long as it uses a parent it does not share with u and w. *Formally, a node in a subtree has a path around a failure precisely if (i) it is at a lower level than the failure, or (ii) its subtree type stack is different than the top of the type stack of the failure.*

The **PBOnly** algorithm to handle downward failures works as follows:

1. When a parent, u, detects a failure on a downward link $\langle u, v \rangle$, u floods a message $\langle \textbf{PBOnly}, v.location, v.level, S \rangle$, to all neighbors (except v) indicating that it does not have a path to *only* the given location. The type stack $S = \{v.type\}$.

2. Recursively, when w receives a **PBOnly** message from x:

   (a) If x is a child of w:

      i. Push the type of x onto S

      ii. Flood the notification to all neighbors except x

   (b) If x is a parent of w:

      i. If w's level is equal to $v.level$ or w's type is not equal to the top of S, put an entry in the forwarding table that redirects traffic to the location prefix of v to another parent x'

      ii. Else, pop S and flood the notification to all children of w

Upward links failures. As mentioned above, the most common of our three failure scenarios is scenario 1, in which a downlink fails and so the parent can no longer reach the destination. When upward links fail, or **PBOnly** messages come from above, a child can usually route around the failure using a detour to any of its $p-1$ parents that has a working route. However, switches can run out of upward routes, as described in scenarios 2 and 3.

In particular, if there is a switch failure, some switches on the same level can run out of upward routes toward a given location (Figure 3.4 contains three such switches). This can also happen if all upward links on a given switch fail or if different pushbacks compose to block a particular location.

If all upward alternatives toward a given location prefix have failed for some switch u, then u is considered failed for packets traveling upwards through it. u broadcasts to its children a message $\langle \textbf{PBExcept}, u.location, u.level \rangle$ that indicates that it no longer has any routes *except* to its children (switches with a *location* such that $same_prefix(location, u.location, u.level)$).

This case is handled by recursive use of the upward flow redirection scheme. Whenever a switch installs a new pushback block or detects a new failure, it checks to see if there is any blocked location prefix that is shared between all links. If so, the block must be propagated down the tree. Special cases include the fact that a *PBOnly* block will never block as many locations as a *PBExcept*, *PBOnly*s block more locations when they are for higher levels, and *PBExcept*s block more locations when they are for lower levels.

1. Given a pushback block b, let $foundCounterExample = \text{false}$

2. For each upward link, l:

   (a) If l is down or b is installed on l, continue

   (b) Let $foundPrefix = \text{false}$

   (c) Else, for each block b' installed on l:

      i. If both b and b' are of type $PBOnly$,

         A. If $b'.level < b.level$, continue

         B. If $same_prefix(b'.location, b.location,$
            $b'.level + 1)$, then $foundPrefix = \text{true}$

      ii. If b' is of type $PBExcept$,

         A. If $b'.level > b.level$, continue

B. If $same_prefix(b'.location, b.location,$
$b.level + 1)$, then $foundPrefix = \text{true}$

(d) If $foundPrefix = \text{false}$,
then $foundCounterExample = \text{true}$

3. If $foundCounterExample = \text{false}$, push b further down to all children

3.5.3 Epoch-based Rerouting

After pushback terminates, all traffic will be routed along shortest paths (provided a route exists), but load may be unbalanced. Traffic that would have traversed failed links are shunted onto the remaining links. The third step is then to repair load balancing by reassigning flows. This is a global process that is somewhat more involved than the previous two schemes, so while failures are immediately reported to a centralized controller, the rebalancing of load occurs periodically at discrete epochs.

We describe a centralized load balancing server in Section 3.6.3; the same mechanism is used to rebalance flows after failures. The mechanism for reporting traffic characteristics and scheduling will be discussed subsequently. Failures are communicated to the centralized controller and taken into account in scheduling. Only shortest paths are considered by the controller—local detours are intended to be a temporary patch. Since all paths have the same length, the controller assigns flows to minimize the maximum traffic across any link. If there is no direct path available, the flow will continue to take a locally rerouted path if possible. Additionally, if a packet from a scheduled flow encounters a failed link or node before the centralized controller is informed or reflects the change, it is treated as non-scheduled from that point onwards. If it remains stable, it will be rescheduled in the next epoch.

When a node recovers, the switch or link must prove that it is stable by remaining up for an extended period of time before the centralized scheduler will assign it traffic. This minimizes lost packets due to repeated failures of flaky devices. By putting recovery of

hardware on a somewhat slower time scale, we aggregate frequent and correlated failures into a single event and only incur the compulsory losses once. When the controller does decide to reinstall the device, all neighbors are informed, and they are responsible for tearing down local reroutes and pushback blocks. Only when the neighboring switches acknowledge reinstallation is complete does the central controller use the new device for scheduled flows.

3.5.4 Failure Detection

To gain the full benefit of near-instantaneous rerouting, we need to be able to rapidly and accurately detect failures. If hardware has fail-stop behavior, the high-level anatomy of a failure event will start with the actual failure, followed by eventual detection, and then a recovery of connectivity by the protocol. In this case, MTTR is bounded by the time to detection, plus the time to compute and install any changes into the routing table, and so it is useful to have a failure detector that can quickly and accurately detect failures without needing to wait for multiple losses of relatively infrequent heartbeats. Stochastic failures can also benefit from a quicker, more accurate failure detector that does not rely on periodic sampling of packet loss.

Most current detection methods do not provide either of these properties and wait for multiple, relatively slow heartbeat intervals before declaring failure. In IP routers, OSPF and IS-IS implement 330 millisecond heartbeats with 1 second dead intervals. Similarly, layer 2 Ethernet switches will report failures only after a waiting period on the order of multiple milliseconds. (This is called debouncing the interface.)

These methods depend on heartbeats because the networks they traditionally handle are not necessarily physically connected and/or operate on shared media. In these settings, congestion can cause false positives. Worse, some routing algorithms are prone to positive feedback loops during rapid changes [62].

We argue that these protections are not necessary in our system. F10 has very fast neighbor-to-neighbor failure detection because switches are directly connected and routing

loops are impossible by construction. Our failure detection mechanism requires that switches continually send packets, even when idle. These packets test the interface, data link, and to an extent, the forwarding engine.

F10's failure detector takes advantage of the fact that packets should be continually arriving, and allows the network administrator to define two sets of values (one for bit transitions to detect power loss and one for valid packets to detect corruption):

- t, the time period over which to aggregate
- c, the required number of bit transitions/valid packets per t for a working link to not be declared as down
- d, the number of bit transitions/valid packets per t before a failed link is brought back up

This allows customization of the threshold for stochastic losses, as well as the amount of time necessary before the link can be declared as down. When a node detects a neighbor failure, it also begins to send dummy "failure detected" packets to ensure the detection is symmetric while maintaining a way to detect when it comes back up. c and d should be set such that the link will rarely flap. To avoid fluctuation at a scale slightly longer than t, we use exponential backoff.

From a protocol design standpoint, our system eliminates the usual concerns with fast failure detection. Firstly, our failover protocols only deal with one link at a time, meaning that a spurious failure will not affect any other link, cascade failures or create feedback loops. The only possible concern is that the increased load from rerouted paths will cause congestion. However, local rerouting is intended to be short-term. Further, global load balancing is done based on the measured end-to-end traffic matrix, ignoring the temporary detour routes.

Secondly, local rerouting is initiated and can be removed at the affected node. Instead of having an extended period during which the network propagates status updates until

the system converges, our rerouting protocol completes in the time it takes for a switch to update its routing table. The choice of whether to send along the link in question or to deflect to a new path is made at the detecting switch, thus limiting the issue of convergence of local rerouting to a single switch and guaranteeing that the protocol converges before the next failure.

Note that blasting traffic and expecting it to continually arrive assumes certain properties of the link layer. The type of Ethernet used in data centers are mostly full-duplex between switches and therefore are not affected by collisions. In fact, Cisco gigabit Ethernet switches and Ethernet standards starting from 10GbE do not even support half-duplex or CSMA/CD.

3.6 Load Balancing

Balancing load across the network is important in data center networks. Failures compound problems of load balance since they reduce the overall bandwidth of the network and destroy the regular structure of the network.

Just like failures, traffic in data centers also follows a long-tailed distribution [16]. The majority of flows are small and short-lived, but their longer-lived counterparts can cause long-term congestion and inefficiencies if not handled correctly. To handle this type of workload, we take the same 'cascading' approach to load as we do failures. We again introduce three mechanisms:

- A flow-placement mechanism that allows each switch to locally place flows based on expected load.
- A version of our pushback mechanism that is able to gracefully handle momentary spikes in traffic.
- The same epoch-based centralized scheduler that is also used for failure recovery.

At a high level, the centralized scheduler preallocates a portion of each link for long-term, stable flows. The remainder is used for new and unstable flows—these are randomly

scheduled in the remaining capacity, but with pushback to deal with short term congestion.

3.6.1 Weighted Random Load Balancing

To make immediate, local load balancing decisions, we use random placement of short-term traffic across all of the available shortest paths. Because TCP dynamics make packet reordering is undesirable. Instead, we schedule traffic on a per-flow basis and rely on the central controller to handle any long-term congestion.

Unscheduled traffic is that which is too short-lived to benefit from our centralized scheduling algorithm. Each flow is directed along upward edges randomly, and in the case that the centralized scheduler makes paths unequal in terms of scheduled load, we use weighted ECMP that is based on the residual capacity left after scheduling.

When new links are installed, we set their residual capacity to zero. New flows do not use the link so that the centralized controller is able to ensure consistent weighting. If all links have zero remaining capacity, a new flow is placed across some non-failed link with equal probability.

3.6.2 Push Back Load Balancing

When traffic suddenly spikes, there is a period before the centralized controller can react. Measurement studies have shown that this congestion usually manifests itself in isolated hotspots across the network [36].

As mentioned previously, switches have information about their expected remaining capacity. When its expected usage is significantly exceeded, the switch notifies other nodes about the change with the same mechanism as described in Section 3.5.2, except that notified switches modify the ECMP weights instead of blocking all traffic.

The switch keeps traffic statistics for the last 500ms, and calculates the average difference between instantaneous and scheduled load. The difference between a link's randomly-placed load and the average randomly-placed load is *LBdelta*, and if any link has an *LBdelta* above

20%, a congestion pushback message will be sent back upstream for the link, rerouting a portion of upstream traffic around the spontaneous congestion. These push back blocks are removed after each scheduling epoch.

3.6.3 Centralized scheduling

Longer-term, predictable flows can and should be scheduled centrally to ensure good placement to avoid persistent congestion. For these longer flows, we use a similar approach to MicroTE [17], which advocates centralized scheduling of ToR to ToR pairs that remain predictable for a sufficient timespan. The authors found from measurement data that data center traffic is predictable at the granularity of a couple seconds. They propose a system in which a server in each rack saves traffic statistics for two seconds, and, every second, sends to a centralized controller a list of "predictable" flows that have instantaneous values within some delta of their average value over the last two seconds (they used a factor of .2).

In F10, these flows are scheduled with a greedy algorithm that sorts the flows from largest to smallest and places them in order on the paths with the least cost, where the cost of a path is defined as $\sum \frac{1}{R(e)}$ over the edges e in the path P, where $R(e)$ is the remaining capacity of edge e. The controller informs ToRs about scheduled flows, and residual capacities are sent to each switch to use for weighted ECMP. If a scheduled flow runs into a failure, it becomes unscheduled at the point of failure, and gets placed using weighted ECMP.

In general, optimal rearrangement is an NP-complete problem for single-source unsplittable flows. We choose the greedy algorithm for scalability reasons, but the exact choice of algorithm is orthogonal to our work. Multipath flows are more flexible from a load balancing perspective, but require end host changes to the TCP stack.

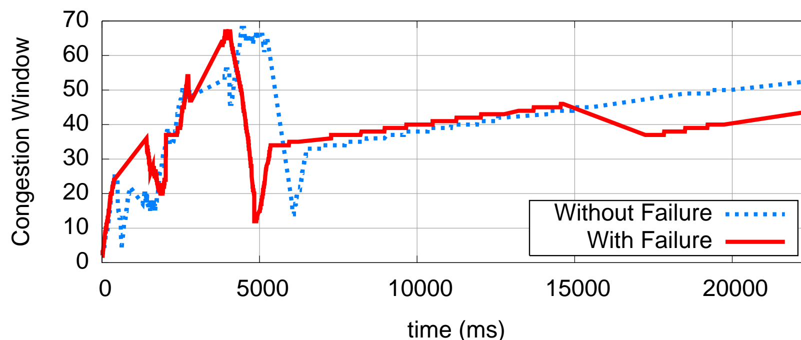


Figure 3.5: TCP congestion window trace with and without failure. In the case of the failure, a link went down at 15sec and F10 recovered before a timeout occurred.

3.7 Prototype and Evaluation

3.7.1 Prototype

We built a Click-based implementation of F10 and tested it on a small deployment in Emulab [61]. The prototype runs either in user-mode or as a kernel module. The implementation is a proof of concept and correctly performs all of the routing and rerouting functionality of F10. It is able to accept traffic from unmodified servers and route them to their correct destinations.

Failure Characteristics. We instrumented a Linux kernel to gather detailed TCP information, including accurate information about congestion window size; we used this instrumented kernel to test the effect of a failure on a TCP stream. Tests were performed in Emulab, but since bandwidth limitations in both the links and the Click implementation are lower than in a real data center, we lowered the packet size so that the transmission time and the number of packets in flight are comparable to a real deployment. We used this testbed to compare the evolution of a congestion window with and without failure during a 25 second interval in Figure 3.5. F10 is able to recover from the failure before a timeout occurs, making the performance penalty minimal.

Failure Detector. We have also implemented an approximation of F10's failure detector

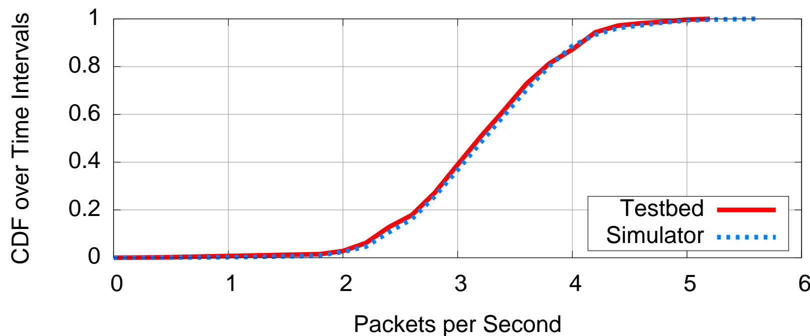


Figure 3.6: Comparison of throughput of the testbed and the simulator through ten simulated switch failures and the same topology/offered load.

using Click in polling mode. The detector would ideally be built in hardware, but preliminary results indicate that we can approximate the ideal detector with a Click-based implementation. Unfortunately, with Click, it is not possible to track bit transitions on the wire, and there is some amount of jitter between successive schedulings of the network device poller. Even so, our Pentium III testbed machine was able to accurately detect failures after as little as 30μs—much less than a single RTT in a data center. With this property, we were able to fail based on the rate of valid packets.

At each output port, we placed a strict priority scheduler that pulls from the output queue if possible, or else generates a test packet. The dummy packets are intercepted and dropped by the downstream failure detector before being passed to the rest of the system. The detector asserts a failure and notifies the rest of the system when the arrival rate of either good or nonce packets drops below the specified threshold.

3.7.2 Evaluation Environment

Simulator. We created an event-driven simulator to test the efficacy of F10 with medium- to large-scale data centers—resources limited the feasibility of such experiments in our testbed setting. The simulation includes the entire routing and load balancing protocol along with the fast failure detection algorithm. When there is no traffic, each switch generates

nonce messages to its neighbors. The link is marked as failed if three consecutive packets are not received correctly.

Our multicore, packet-level, event-driven simulator comprises 4181 lines of Java. It implements both low-level device behaviors and protocols. The Layer 2 Ethernet switches use standard drop-tail queues and have unbounded routing state; our evaluation shows that even with many failures in the network, only a modest amount of state needs to be installed. The simulator models 100 ns latency across each link to cover switch and interface processing as well as network propagation latencies.

Our experiments are performed assuming 24-port 10GbE switches in a configuration that has 1,728 end hosts, resulting in a standard or AB Clos network with three layers. Except in Section 3.7.6, we use UDP traffic in our experiments so that we can more precisely measure the impact of the failure on load. This enables us to understand how well the evaluated mechanisms improve *network capacity*. TCP will generally back off quickly, resulting in lower bandwidth and fewer losses than shown here.

We have compared the measurements generated by both the testbed and simulator, for an identical topology and offered load. Figure 3.6 is a CDF of throughput for a single source-destination pair that experienced a sequence of ten failures, which each went through all of the stages of failover in F10. We found that, in all cases tested, the simulator and testbed results matched each other closely.

Workload model. We derive our workload from measurements of Microsoft data centers given by Benson et al. [16]. We generate log-normal distributions for (1) packet interarrival times, (2) flow ON-periods, and (3) flow OFF-periods, parameterized to match the experimental data from the paper. In certain experiments (labeled explicitly below), we scale the packet interarrival times to adjust the load on the network.

Failure model. Failures are based on the study by Gill et al. [26] that investigated failures in modern data centers. We generated log-normal distributions for (1) the time between failures and (2) the time to repair for both switches and individual links based on their

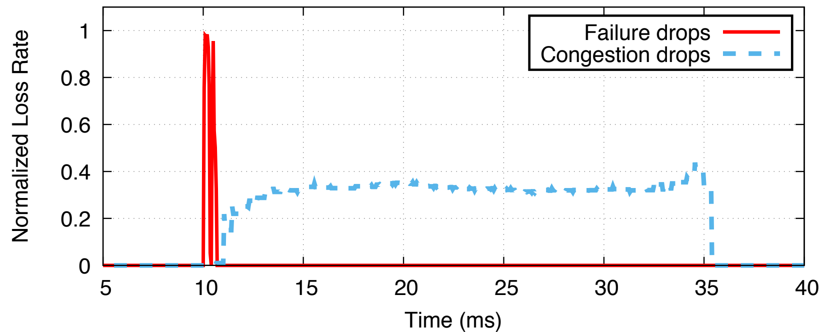


Figure 3.7: Behavior of F10 under a simulated failure at 10 ms with an all-to-all UDP workload. The graph shows the loss rate over a 40 sec window due to lack of connectivity and congestion in the case a single failure.

experimental data.

Note that we do not consider leaf (ToR) switch failures, as these are not handled by traditional Clos networks or by F10. We provide a solution to this issue in the next chapter.

3.7.3 Recovering from a Single Failure

Figure 3.7 shows a breakdown of the losses over time after a single switch failure in F10 running a uniform all-pairs workload at 50% (UDP) load. The y-axis in this graph shows the loss rate normalized to the expected number of packets traversing each switch.

When the failure occurs at 10 ms, there is a burst of packet drops due to the failure. At around 11 ms, the neighbors of the failed switch detect the failure, and local rerouting installs new working routes and eliminates failure drops. Local rerouting reduces the capacity of the network, triggering congestion. When the pushback scheme is initiated later, it quickly and effectively optimizes paths, spreading the extra load and eliminating the congestion loss.

3.7.4 Comparison with PortLand

F10 recovered from the single failure evaluated in the prior section within 1 ms of the failure; this is roughly two orders of magnitude faster than PortLand [45], a state of the art design for fault tolerance in data center networks, which reports minimum failure response times

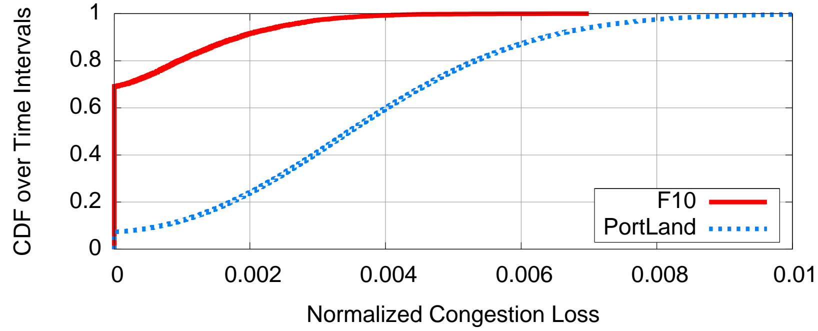


Figure 3.8: Cumulative Distribution Function (CDF) of the congestion losses of both PortLand and F10. The traffic and failure conditions were derived from recent measurement studies [26, 16].

of 65 ms. In addition, F10 was able to recover load balancing in 35 ms, while PortLand does not handle congestion losses at all. In this section, we compare F10 against PortLand using the realistic, synthetic traffic and failure models described in Section 3.7.2.

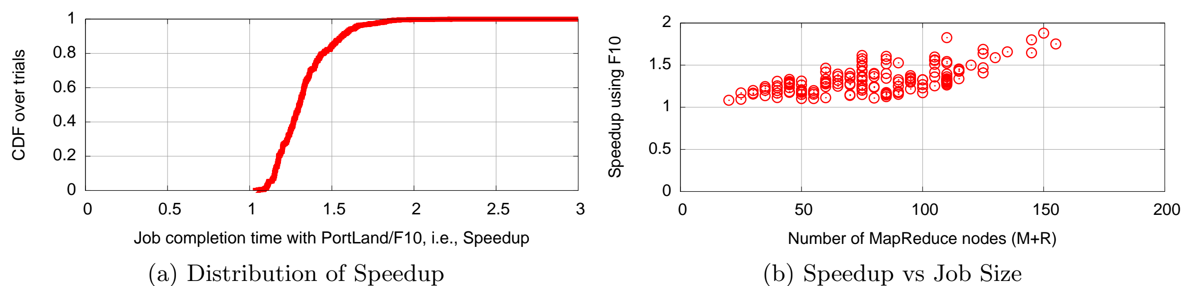
Figure 3.8 shows the congestion rate in each system. We generated workload and failure events from a random seed and fed the same trace into PortLand, which uses a standard Clos network, and F10 with an AB Clos network and all our techniques. We aggregated loss statistics over a 500 μs time interval, and report the distribution of congestion loss over these intervals. The figure aggregates data points for multiple runs that start from different initial conditions.

Overall, F10 has much less congestion than PortLand. F10 sees negligible loss for ¾ of time periods, whereas PortLand nearly always has congestion. In total, Portland has 7.6× the congestion loss of F10 for UDP traffic.

3.7.5 Local Rerouting and AB Clos networks

Note that both standard and AB Clos networks can perform local rerouting, but the former is unable to exploit the shorter detours of F10. Here, we evaluate the impact of the novel structure of AB Clos networks during local reroutes.

We measured the path inflation of local reroutes using varying numbers of switch failures



(a) Distribution of Speedup

(b) Speedup vs Job Size

Figure 3.9: Comparison of MapReduct job performance on PortLand or F10. Background traffic and failure conditions were derived from recent measurement studies [26, 16].

(up to 15 concurrent failures, implying up to 360 failed links) in standard vs AB Clos networks. We found that *local reroutes in AB Clos networks experience roughly half the path inflation than in standard Clos networks*, owing to F10's ability to use Scheme 1 rerouting in addition to Scheme 2. Even for many concurrent failures, the vast majority—more than 99.9%—of reroutes use the minimum number of hops (2 for AB Clos networks, and 4 for standard Clos networks). We also looked at random link failures as opposed to switch failures, and obtained similar results in terms of how the path dilation in F10 compares with that of standard Clos networks.

3.7.6 Speeding up MapReduce

We conclude our evaluation by simulating the behavior of a MapReduce [22] job (with TCP flows) in our data center. We used a MapReduce trace generated from a 3600-node production data center [20], and considered the performance of just the shuffle phase, where flows are initiated from mappers to reducers, with mappers and reducers assigned randomly to servers. We focus our study on only those MapReduce computations that involved fewer than 200 mappers and reducers in total.

Figure 3.9 compares the performance of the shuffle operation under the two architectures—F10 and PortLand—and the failure model used thus far. Since the shuffle operation completes only after all the constituent flows are complete, it suffers from the well-known strag-

glers problem. If any of the flows traverse a failed or rerouted link, it suffers from suboptimal performance. We measure the speedup of an individual job as the completion time under PortLand divided by that of the job under F10.

Figure 3.9a shows the distribution of the speedup; we find that F10 is faster than PortLand with a median speedup of about 1.3×. Figure 3.9b, shows the distribution of speedup vs job size, and we find that gains are larger when more nodes participate and compete for bandwidth. We conclude that F10 offers significant gains over PortLand, and this will improve in larger future data centers.

3.8 Related Work

The topic of fault tolerance in interconnection networks has a long history [7, 23, 39]. Most previous work on this topic, most notably [8], has added hardware in the form of stages, switches and links to existing topologies to make them more fault tolerant while keeping latency and non-blocking characteristics constant. Modern data centers may also improve fault tolerance by connecting subtrees redundantly such that there is more than a single path between every root and every destination [27]. Instead of requiring additional redundancy, we instead allow for a temporary increase in latency for paths affected by faults in exchange for no increase in hardware cost.

Researchers have also recently proposed several alternative interconnects. Our work directly builds on FatTrees [10] as they are commonly used in data center networks. We leverage many of the earlier mechanisms in our work. We replace the interconnect with our novel AB Clos network and co-design local rerouting, pushback, and load balancing mechanisms to exploit the topology. Hedera [11] implements centralized load balancing on top of PortLand. Hedera only schedules new flows, whereas we choose to globally rearrange flows periodically.

DCell [29] and BCube [28] introduce structured networks that are not multi-rooted trees.The key difference is that these topologies trade more hardware for their increased robustness. DCell performs local rerouting after a failure but is not loop free (unlike ours).

Loop freedom is important to enable fast failure detectors at the link layer without compromising reliability.

Jellyfish [59] takes a different approach to datacenter design—unstructured, random-wiring. It trades regularity and rearangeable, non-blocking guarantees for better average-case performance with less hardware. Our mechanisms might apply to their topology, though it would require precomputation of all detour paths, and it is unclear how much path dilation would be needed on average.

Our failure recovery schemes leverage existing techniques. Our local rerouting scheme uses tags and failure lists analogous to MPLS and Failure-Carrying Packets [37], respectively. MPLS supports a similar style of immediate local detours (Fast Reroute) while waiting for the failure to propagate upstream (Facility Backup) [47]. MPLS failover requires manual preconfiguration and stored state, whereas our system has easy-to-compute backup paths and stores state only when there is a failure.

DDC [42] is an attempt to do local failure recovery in unstructured networks. They make no assumptions about network topology, and so they cannot benefit from preset local reroutes. In order to handle unstructured networks, their approach reroutes for each destination separately and does not result in paths that are as efficient as the ones produced by our local rerouting scheme.

3.9 Discussion

The preceding chapter describes the basic design of F10 and how it recovers from common-case switch and link failures quickly and gracefully. Our exposition was based on a fully-provisioned Clos network; however, the concepts presented here allow for a broader family of designs. Similarly, our architecture is flexible enough to support existing data center operations. In this section, we briefly discuss a few of the ways in which F10 handles a broader class of base topologies and operations.

3.9.1 Oversubscribed and Overprovisioned Networks

So far, we have assumed that the number of uplinks at any switch is equal to the number of downlinks. In an oversubscribed or overprovisioned network, these numbers can potentially differ between levels. Fortunately, these networks require little to no change in our algorithms.

The placement of flows by the global rebalancer is straightforwardly extended to this case. Pushback similarly does not rely on the number of links; notifications are broadcast to all uplinks and downlinks, and termination only depends on level and type stack. For basic routing, local rerouting and recursive pushback, a few generalizations of functions must be made, and for this we require configuration of the number of downlinks for switches at each level, D_{level}. All references to p should be replaced by D_{level} and protocols should be changed to take the nonuniformity into account (e.g., $\mathit{prefix}(a, i) = a \gg (\Sigma_{l=1}^{i}(\lceil \log(D_l) \rceil)))$.

3.9.2 Beyond AB Clos Networks

Our architecture introduces an extra type of subtree that connects to a different set of roots and thus provides additional path diversity closer to a given rerouting node. A natural question to ask is whether we can get even more diversity with more types.

In the limit, we can create a p-type Clos network in which all subtrees are connected to a slightly different set of roots. This is accomplished by rotating the set of roots to which a subtree connects—subroot j of the first subtree connects to the jp through the $(j+1)p-1$ roots, subroot j of the second subtree connects to roots $jp+1$ through $(j+1)p$, and in the same manner, each additional subtree incrementally shifts by one. This guarantees that every sibling of a given node n has at least one alternative path.

At first glance, this seems to improve the potential for efficient reroutes. However, more choices at the first hop of local rerouting comes at the cost of fewer at the second. While an AB Clos network provides $p-1$ alternatives for the second hop of Scheme 1 given a single failure, a p type Clos network will have an average of $p/2-1$, with some nodes having more

alternatives than others. Increasing the number of types does not, in general, increase the chance of finding a two-hop detour.

For pushback, more alternatives means that the notifications can stop earlier (in the case of a single failure in a p-type Clos network, pushback can terminate after the message traverses any downward link). However, traffic destined for the failed path is split over a smaller number of alternate paths, disproportionately increasing the load on those paths. In sum, the tradeoffs are complex, and we leave a fuller comparison for future work.

3.9.3 Central Controller Fault Tolerance

We previously assumed the central controller to be fault tolerant. The implementation for this is not complicated: it can be replicated with a primary/backup approach, with the benefit that data like traffic matrices (which are thrown away after each epoch) and liveness of switches is soft-state. Even if the controller or its links do fail, the role the central controller plays is not essential to correct operation.

If the controller fails to receive the traffic matrix from any leaf switch, schedule flows or install weights at switches, the switch will use the last set of information it received, if available, and randomly place the rest of the flows—load balancing may suffer, but connectivity is not affected.

If the controller fails to receive a failure notification, flows through that failure will hit a pushback block and become a randomly placed flow from that point onward. If the controller misses an installation notification or fails to install a new device or link, we again fall back on the philosophy that a node that fails and recovers is still a faulty node. It must wait until the controller comes back up, but this is fine since installation already occurs on a slower time scale and, if all installed alternatives go down, uninstalled but active alternatives are used.

3.9.4 Symmetry

Another potential concern is that the structure of an AB Clos network is no longer symmetric. Mechanisms like ECMP rely on symmetric shortest paths, present in standard Clos networks. However, AB Clos networks also have symmetric shortest paths, and the distribution of load is similar so ECMP is just as effective in our architecture as it is in standard Clos networks.

3.9.5 Topology Verification

Good network administration is an essential prerequisite of high-bandwidth, scalable and fault-tolerant network operation. Thus we assume administrators plan out the network to adhere to our topology and, from the structure, can set the *level*, *location* and *index* of each switch. However, configuration errors do occur.

Each switch checks in with the centralized controller to aid in load balancing and fault tolerance at a global level. We take advantage of this single point of control to assist in correctly implementing our architecture—it can verify that the switches are addressed properly, connected properly and address values correctly correspond to the actual wiring. To protect against multiple switches taking the same three values accidentally, a UUID is added to the three existing address variables, which can simply be the lowest MAC address of any of the switch's ports.

In addition to checking that no two nodes share the same three address variables, we run the following check whenever the controller detects new switches or links:

1. Let c be the current node and N be the set of neighbors of the current node
2. For each node, n in N
   (a) If $n.level = c.level + 1$
      i. Assert $\textit{same_prefix}(n.location, c.location, n.level)$

ii. If $\textit{prefix}(c.location, c.level) \pmod 2 = 0$, then assert $n.index \geq c.index * p$ and $n.index < (c.index + 1) * p$

iii. Else, assert $n.index - c.index \pmod{p^{\text{c.level}}} = 0$

(b) Else assert $n.level = c.level - 1$

In this way, we can verify that the address values correctly correspond to the actual wiring.

3.10 Final Remarks

Scalable, cost-efficient and failure resilient data center networks are increasingly important for cloud-based services. In this chapter, we described F10, a novel multi-tree topology and routing algorithm to achieve near-instantaneous restoration of connectivity and load balance after a switch or link failure. Our approach operates entirely in the network with no end host modifications, and experiments show that routes can generally be reestablished with detours of two additional hops and no global coordination, even during multiple failures. We couple this fast rerouting with complementary mechanisms to quickly reestablish direct routes and global load balancing. Our evaluation shows significant reduction in packet loss and improved application-level performance.

Chapter 4
SUBWAYS

The previous chapter considered how controlled asymmetry and cross-layer design can improve the reliability of the backbone of data center networks. Controlled asymmetry provides path alternatives closer to failures, which can then be used by cross-layer design.

This chapter considers the topology at the edge of the network, between the ToR switches and servers. Specifically, Subways connects servers to both their own ToR switch and the ToR switches of neighboring rack in an overlapping pattern. We find that doing so enables incremental server capacity upgrades, decreases network congestion, mitigates the effects of oversubscription, and increases the fault tolerance within a physical rack, all at reasonable cost.

It is beyond the scope of this work to fully characterize installation and operational costs of a Subways architecture (these costs depend on proprietary volume discounts for optical and electrical cables, switches, network interfaces, etc.). However, the largest added cost for Subways is likely to be the added cable length from the server to the adjacent ToR switches. We develop a set of wiring algorithms that show that physical wire lengths can be kept short enough to be implemented with copper and also made relatively easy to install.

We next discuss several motivating factors to consider in the design of the server-ToR interconnect.

- *Technology and Application Trends*

As mentioned in Chapter 1, cloud applications continue to scale beyond the limits of a single machine. Compounding this effect is a simultaneous tendency for modern data centers to be oversubscribed, i.e., to have less capacity in the core of the network compared to the

capacity at the edge.

There is a resulting need to keep communication local where possible. For instance, if we could pack all nodes from a MapReduce job into a single rack, shuffle traffic would never need to traverse the oversubscribed core network.

Despite this, inter-rack communication is often unavoidable. Sometimes, this distribution is by design, e.g., network storage blocks are often stored across racks and power failure domains for fault tolerance. Individual jobs are often too large to fit within a single rack [56]. In a recent Google trace [53], 63% of jobs required more nodes than a single 40-node rack can handle; because of fragmentation, it is likely that an even larger percentage of jobs would span multiple racks in practice. The increasing use of large-scale analytics implies that inter-rack communication will continue to grow at a very fast pace [31].

- *Server Traffic is Often Correlated*

Data center traffic is very bursty, particularly on links closer to the edge [13, 16, 30]. This is partly a consequence of the desire to preserve traffic locality. To reduce inter-rack communication, multiple servers in the same rack can be assigned to the same job, but by virtue of performing related tasks, when they do send inter-rack traffic, they are more likely to do so simultaneously.

A second reason is that servers/jobs with similar purposes are often placed in the same rack. Facebook, for example, expands its memcache and web frontend infrastructure by rolling out an entire rack of servers that are optimized for that particular service [46]. Although this approach significantly reduces operational complexity, it implies that the traffic patterns of servers within a rack can be correlated; all of the servers in a rack get hot or cold simultaneously.

Correlated server behavior is a problem because of oversubscription. When servers are connected to the same ToR switch, the aggregate capacity of the uplinks from that ToR switch define how much the servers in that rack can send or receive at any time, gating

application performance.

- *ToRs Are a Single Point of Failure*

In addition to demand growth, the edge of the network is also a fault tolerance bottleneck. Data center operators go to great lengths to ensure that their systems are resilient to failures. In fact, it is common to see techniques such as redundant power supplies, a great deal of network path diversity, and redundant network controllers. In most data centers, there is no such redundancy for the ToR switch—it remains as one of the few remaining single points of failure in the data center. This is particularly important as ToRs have relatively high failure rates compared to other network devices [26].

4.1 Design Overview

We introduce Subways, a wiring pattern where servers in a rack are wired to the ToRs of adjacent racks in addition to their own. Clever usage of multiple server-ToR connections can mitigate many fundamental issues in today's data centers:

- *Simpler upgrades:* A typical edge capacity upgrade requires large amounts of up-front investment and/or rewiring of both server-ToR links and links in the backbone. By augmenting connectivity and allowing servers to connect to adjacent ToRs, we enable cheap, potentially incremental upgrades that reduce or eliminate the need for rewiring.

- *Less backbone traffic:* An overlapping connection pattern creates shorter paths for more destinations, keeping traffic off the data center backbone.

- *Smoother hot spots:* Traffic is very bursty, particularly at the ToRs. By connecting servers to multiple, differently-loaded ToRs and clusters rather than just one, we gain better load balance.

- *Fault tolerance:* Redundant server-ToR links remove one of the remaining single-points-of-failure in data centers.

Var.	Definition
N	# of end hosts in the data center
p	# of ports per server
q	# of downward facing ports per ToR switch
r	# of servers per logical rack ($\frac{q}{p}$)
c	# of clusters over which a loop is mapped (Type 2)
l	# of racks in a single Subways loop (Types {1,2})

Table 4.1: The variables that define a Subways architecture. Some only apply to a subset of the wiring methodologies in Section 4.2.

The Subways design varies on two dimensions, which we discuss in the following sections. First is topology: how are the server links distributed among ToRs, and how are those ToRs distributed into clusters? These choices have a large impact on reducing and smoothing traffic at each layer. The second dimension is load balancing: Subways can work with uniform load balancing mechanisms like ECMP, but it can also benefit from adaptive mechanisms to spread traffic bursts over neighboring ToR switches and from detour routing, to spread bursts over even more switches.

As every data center instantiation may have different numbers of links per server or servers per rack, we parameterize our discussion using the notation sketched in Table 4.1. Because Subways is an edge architecture, it is compatible with *any* aggregation and core topology, although for concreteness we concentrate on folded Clos topologies.

4.2 Wiring Types

We begin by describing a baseline topology, Type 0 Subways, corresponding to the current, industry-standard approach to using a p-port server where all ports are connected to the same switch. We then introduce two new wiring types that link servers to adjacent ToRs. To keep the discussion concrete, we assume for now that all topologies use a simple ECMP-like load balancing algorithm; we relax that assumption in the next section. We defer a discussion of the implications of Subways for physical cable lengths until Section 4.5.

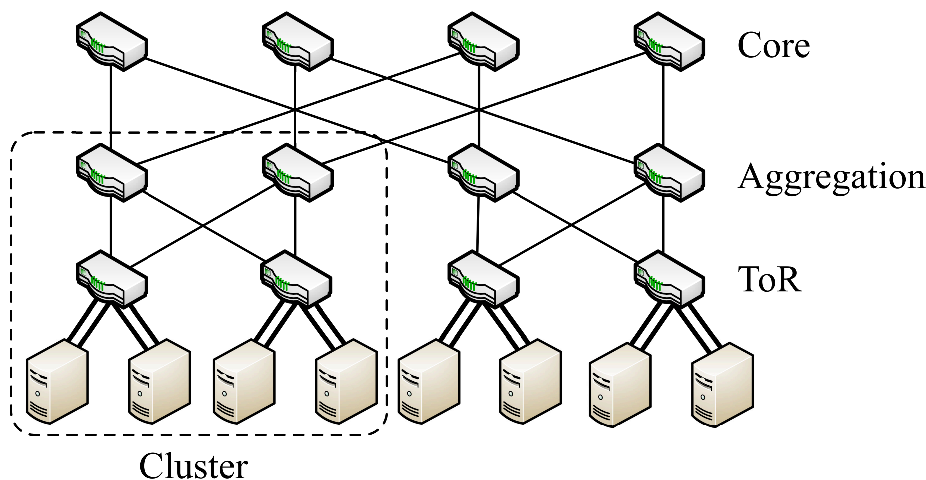


Figure 4.1: A folded Clos data center architecture with Type 0 Subways, $p = 2$. Three layers of switches connect servers together. Each server has two ports that are both connected to the same ToR switch.

4.2.1 Starting Point: Type 0 Subways

The simplest wiring model is to trunk p connections from each server to the same ToR switch, as shown in Figure 4.1. A Link Aggregation Group (LAG) can be used to treat the multiple physical links as a single logical connection using a protocol like LACP. This simplifies routing but requires the entire trunk to terminate at a single, physical switch.

As a result, although the network core has ample redundant paths for load balancing and fault tolerance, all paths to and from a rack feed through a single ToR switch. Likewise, the server itself has redundant links, but if the switch fails or is overloaded, all p connections suffer.

When using Type 0 Subways as an upgrade path, operators are required to rewire many existing connections. For instance, let us assume she wants to double capacity while keeping the number of servers (N_s) and oversubscription ratio constant. Any such upgrade requires N_s new server-ToR connections and double the number of switches in the network. In addition, Type 0 requires her to rewire $\frac{N_s}{2}$ of the existing server-ToR links so that all links from a single server go to the same switch, and then to rewire/expand the existing

interconnect to ensure that new ToRs have connectivity to the old ones.

Perhaps most importantly, most trunking protocols are limited to links of the same speed. This prevents heterogeneous upgrades, e.g., where a 25 Gbps link is added to an existing 10 Gbps server connection, something allowed with the wiring patterns we discuss next.

4.2.2 Type 1: Shared ToRs Within a Cluster

One way to improve fault tolerance is to wire the p ports of each server to p different ToR switches at the top of the physical rack. With a multihoming approach like MC-LAG, these multiple physical switches can be aggregated into a single, virtual ToR. To make routing and failover completely transparent, the physical switches are typically wired identically into the network fabric.

Instead, we propose to wire each group of servers to a distinct, overlapping set of nearby ToR switches. In doing so, we gain an extra degree of freedom and, as we describe later, considerable performance improvement relative to both Type 0 and multihoming. For example, with $p = 2$, we wire each server to its ToR plus the one for the closest (logical) rack to the left. In this way, neighboring logical racks share at least one ToR. With Type 1, each server is connected to ToR switches that connect into the same aggregation-level cluster; we relax this assumption for Type 2.

This simple act of sharing ToRs has far-reaching benefits. Using $p = 2$ as an example, two adjacent logical racks are connected to three ToRs, instead of just two. Each server has a high-capacity path to 50% more servers than before, improving performance and reducing the traffic reaching the oversubscribed data center network. When those servers do need to send data through the backbone, they have, in aggregate, 50% more peak throughput than with Type 0.

Topology. Conceptually, each logical rack still has an associated ToR. However, instead of connecting servers to their own ToR p times, servers connect to their own ToR and their

$p-1$ closest ToRs exactly once. This overlapping chain of racks and ToRs is wrapped around to eventually form a *loop* of l racks/ToRs, where each server is connected to its own ToR, the $\lfloor \frac{p-1}{2} \rfloor$ ToRs clockwise, and $\lceil \frac{p-1}{2} \rceil$ ToRs counter-clockwise from it. A loop is thus a connected component in the server-ToR topology. In the degenerate case where the number of server links equals the loop size, $p = l$, we have multihoming. We show that performance improves with increased loop size, but there are practical limitations due to physical wiring constraints that we discuss in Section 4.5. For simplicity, we assume a single loop length l for all loops in the data center. Figure 4.2 shows a single loop of 3-port servers and a length of 9. Figure 4.3a shows two loops where the loops are the same size as the clusters.

Upgrades. One of the interesting things about Subways-style server-ToR connections is that adding capacity to servers does not require rewiring their existing connections. Figure 4.2 shows an example of this, where an upgrade from 2 ports to 3 requires adding new ToRs, but leaves the existing ToR connections untouched. Note, however, that to make room for the added ToRs, an operator may still need to rewire the upper layers of the data center network to ensure that all of a server's ToRs are contained within the same cluster.

Routing. In vanilla Type 1, we keep routing similar to today's data centers, implemented entirely with existing protocols. In particular, because all of a server's ToRs are contained within a single cluster, routing in most of the network does not change—traffic to a particular server still flows to a single cluster, typically through longest-prefix matching.

Servers and aggregation switches have an additional responsibility to assign connections randomly to the available ToRs. This can be done using ECMP just as it is done today—routing table entries can map to multiple possible output ports, to which connections are randomly assigned on a per-connection basis.

Only nodes that are directly connected to ToRs (i.e., servers and aggregation switches) need to install extra ECMP rules. Servers only need one such rule with p options: one for each ToR. Aggregation switches need t rules with p options each, where t is the number of racks in its cluster; they do not need any rules for traffic destined for other clusters beyond

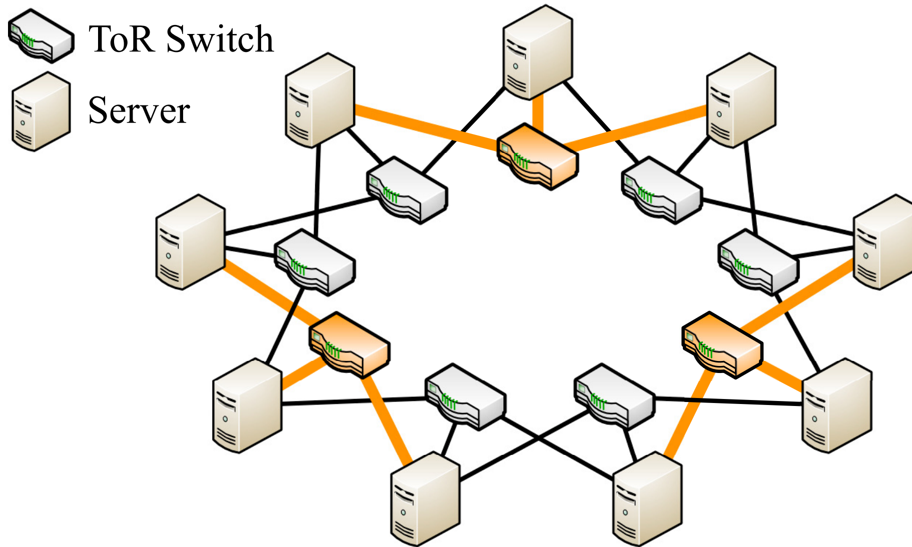


Figure 4.2: Example of the server-ToR connection in Type 1 with a single loop of 9 3-port servers. Bold links/switches are added to upgrade from a 2-port to a 3-port configuration without rewiring.

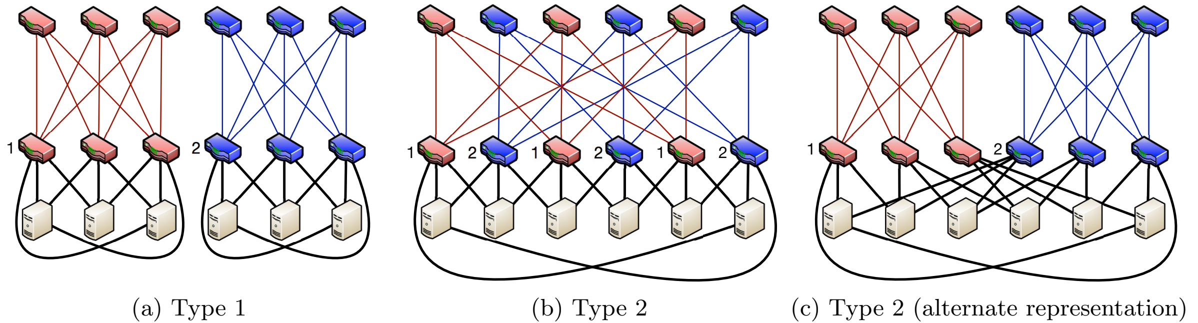
what is needed today.

4.2.3 Type 2: ToRs in Different Clusters

Subways provides one more degree of freedom: adjacent ToR switches in the same loop can be wired into different clusters. In a traditional multi-rooted tree, there is no benefit to declustering. In our case, however, it can spread load more evenly across clusters, and it increases the number of servers that can be reached without going through the core layer. These shortcuts allow a greater degree of oversubscription, for a given level of performance, or equivalently, less congestion for those jobs whose traffic must traverse the core.

Topology. Every server in a Type 2 Subways is connected to multiple clusters as evenly as possible. The number of clusters that are connected to a single loop is configurable and depends on factors such as how much mixing is needed, loop length, and physical constraints.

Let us assume that we have c clusters whose ToRs should be connected to a single Subways loop. In order to find a mapping between the ToRs in Figure 4.2 and the clusters'



(a) Type 1 (b) Type 2 (c) Type 2 (alternate representation)

Figure 4.3: A two cluster topology for Type 1 (a) and Type 2 (b, c) with $p = 3$ and $l = 3$ and 9, respectively. We omit the core layers and color/number ToRs according to their cluster.

ToRs, pick an arbitrary ToR and assign it to the first cluster. Move to the next ToR in clockwise order and assign it to the second cluster. Continue assigning each ToR to a cluster in this manner, looping back to cluster 1 after assigning a ToR to cluster c. Note that the position within a cluster does not matter as all ToRs within a cluster are logically equivalent. It also does not matter that a server is not necessarily connected to all c clusters, as long as assignment is as even as possible.

Figure 4.3b and Figure 4.3c show two equivalent examples where $c = 2$. Note that servers still only connect to nearby ToRs (Figure 4.3b), and at the same time, the cluster topology is still amenable to cable bundling (Figure 4.3c)—we only change the interface between these two topologies.

Upgrades. Because we no longer require all of a server's ToRs to be in the same cluster, an upgrade in Type 2 does not actually require any rewiring of existing links. In the case of augmentation with a parallel backbone, we do not even need to touch the existing core layer. In the absence of rewiring, it is possible to perform a capacity upgrade without any disruption in service. Further, the upgrades can be heterogeneous.

As an example, in an entirely 10 GbE data center, an operator could choose a single cluster of servers and augment them with a parallel 25 GbE FatTree, cross-wired so that

each logical rack of servers connects to a different, overlapping pair of 10 GbE and 25 GbE ToR switches. Instead of wiring every server in a physical rack to the same two 10 and 25 GbE switches, we wire half of the servers to one 25 GbE switch (at the top of the physical rack) and half to the 25 GbE switch at the top of the neighboring rack.

Addressing/Routing. Regardless of whether servers are connected to parallel interconnects or different clusters in the same interconnect, the networks are configured as they are today. Each interconnect has its own address space, and each of its clusters has a subnet within that space. Routing tables can be set up for longest-prefix matching with no increase in size. The primary change is that each server has p addresses.

Servers spread flows over possible paths using a simple, static version of ECMP or WCMP [64]. For example, servers with parallel 10 and 25 GbE connections would place flows on those networks with probability $2/7$ and $5/7$, respectively.

4.3 Adaptive Load Balancing

Wiring servers to ToRs in an overlapping pattern is by itself enough to provide significant performance benefits, but it also opens up the possibility of more advanced load balancing. In this section, we introduce two load balancing mechanisms; they can be combined with either Type 1 or Type 2 topologies. For simplicity, we assume $l > p$ in this section.

As a motivating example, consider the case where $p = 2$ and two adjacent racks are simultaneously hot—a situation that could be more likely as jobs are placed to take advantage of short paths in Subways. With ECMP, the shared ToR between the racks would have twice the load of the two non-shared ToRs. This hurts tail latency: flows placed on the shared ToR take twice as long as the other flows. Using adaptive load balancing, we can equalize utilization across all three ToRs. Note that with Type 0, ToR switches can become overloaded, but load balancing does not help: all paths to the same set of servers lead through the same ToR, and in the absence of a switch or link failure, all paths are identical.

We introduce two possible solutions: a multipath transport-layer protocol or controller-based scheduling. We describe both approaches, but our evaluation assumes the latter.

4.3.1 Multipath Transport-layer Protocols

Transport-layer protocols like MPTCP [63] allow each end host to utilize multiple paths through the network to maximize resource usage. Like TCP, MPTCP detects congestion and varies the amount of traffic sent along a given path. However, the specific protocol is orthogonal to our design, as long as it is able to utilize multiple paths effectively.

In the context of Subways, a multipath transport-layer protocol would allow the system to adaptively balance load over all available paths. This is done in a distributed fashion at each end host and thus can adapt to changes in load relatively quickly and at a fine granularity. In the example described above where adjacent racks are hot, a multipath protocol would let each server split all connections over both ToRs. Instead of a situation where some connections have half the throughput of others, every connection would receive an approximately equal portion of the available bandwidth—a 50% decrease in tail latency.

Exposing multiple paths to the servers. To support MPTCP, we need to expose the multiple paths to each server. For each server-server pair, there are up to p^2 combinations of ToRs through which a path can pass—p ToRs at the source and p at the destination. Note that we assume path diversity in the core interconnect is handled separately.

The source ToR is chosen by directing traffic out of the associated physical port. The destination ToR is chosen by giving each server p addresses, just as in Section 4.2.3. This is necessary regardless of whether the topology is Type 1 or 2; in Type 1, however, each address will come from the same subnet so that routing in the middle of the network remains the same.

4.3.2 Weighted-ECMP

An alternative approach that does not require changes to the end host TCP implementation is to use a locally adaptive variant of the previous work on Weighted-ECMP [11, 17, 43, 64]. In these proposals, senders periodically obtain current utilization information from a centralized controller and use that information to change the probability that flows are placed on a given next hop.

The problem is much simpler here. Because the ToR tends to be the bottleneck, we only need to look at ToR utilization (and not entire paths) when making traffic engineering decisions. Because of this, load balancing calculations are local; ToR utilization information can be maintained by a sharded set of controllers, rather than a single centralized server. This makes statistic collection and load balancing calculations faster and more efficient. Loops in particular represent a natural sharding boundary.

Initial flow placement. Like traditional FatTree routing, the source will prefer shorter paths if available. If there are one or more paths that pass through a shared ToR (and avoid using the data center backbone), the source will choose one of them with equal probability. Otherwise, it must route the flow through one of its ToRs and one of the destination's ToRs. Each is chosen independently and the decision is based on current congestion information. The source ToR is always chosen by the source server, while the destination ToR is chosen by the destination-side aggregation switch or source server in Type 1 and Type 2 topologies, respectively.

Setting weights for source ToRs. The probability that a flow is placed on any particular ToR is based on the remaining capacity of that ToR. For the choice of source ToR, we care about the remaining capacity in the outbound direction.

More formally, each server obtains from its loop's controller the values $[U_1, U_2, .., U_p]$. These represent, for each of the server's ToRs, an exponentially-weighted moving average of the fraction of the ToR's uplinks that are utilized. From utilization, we can calculate the remaining capacity on each ToR: $V_x = B_x(1 - U_x)$, where B_x is the total uplink capacity of

ToR x. The probability of placing a new flow on ToR i is then simply $\frac{V_i}{\sum_{j=1}^{p} V_j}$.

Setting weights for destination ToRs. The main difference between choosing a destination ToR and a source ToR is that we care about *inbound* traffic, rather than outbound. In particular, for each destination, we have $[D_1, D_2, .., D_p]$, the per-ToR downlink utilization of the Aggregation-ToR links.

The resulting weights for a server's ith ToR is $\frac{E_i}{\sum_{j=1}^{p} E_j}$, where E_x is the remaining downlink capacity, $B_x(1 - D_x)$. In Type 2, these weights are provided to each server for each communication partner. In Type 1, they are provided to each aggregation switch for their own cluster. Alternatively, these could be implemented using pushback mechanisms like pause frames and ECN.

Subsequent load balancing. We periodically reschedule existing flows to adapt to changing traffic conditions in the presence of long-lived flows. Each end host and/or aggregation switch will rebind all existing connections every T_{rebind}.

The first step in this process is to calculate a target utilization for each ToR, which for the outgoing direction, is $U_{target} = \frac{\sum_{i=1}^{p} B_i U_i}{\sum_{i=1}^{p} B_i}$, i.e., the total amount of traffic divided by the total bandwidth. With these targets in mind, a server will shift its current utilization to achieve a more even split.

Specifically, let $[u_1, u_2, .., u_p]$ be the current fraction of traffic the server is sending over each ToR. Based on the target ToR utilization and the server's current split, it will calculate a target split for its own traffic. Toward ToR i, its target is $t_i = cu_i \frac{U_{target}}{U_i} + (1 - c)u_i$, where c is a scaling factor that determines the aggressiveness our algorithm. In our evaluation, we use $c = 0.5$.

The probability that a flow is rebound to the ith ToR is then $P[i] = \frac{t_i}{\sum_{j=1}^{p} t_j}$. Note that the target traffic split might not be possible due to the bandwidth of the server-ToR link, b_i. Ignoring short paths for simplicity, this happens when $P[i](\sum_{j=1}^{p} u_j b_j) > b_i$. In this case, we set $P[i] = \frac{b_i}{\sum_{i=1}^{p} u_i b_j}$ and spread the excess load evenly over the remaining links. Also note that reordering can be prevented by waiting for flowlet inactivity gaps [35].

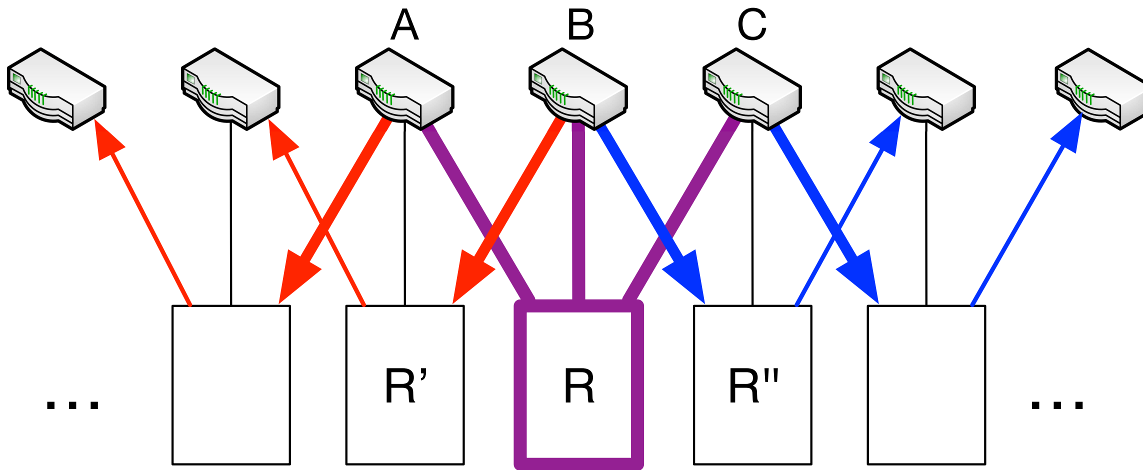


Figure 4.4: A rack-level diagram showing the detour paths of a single hot rack of servers R. Boxes represent racks. Colors differentiate the direction of the detour (red for left, blue for right).

4.4 Detour Routing

An interesting side effect of the Subways wiring pattern is that each loop is a connected graph even without the use of the inter-ToR network. Because of this, whenever a server's ToRs are overloaded, it is possible for servers to detour traffic through adjacent racks (using only Subways links) in order to shunt excess load to remote ToRs.

As an extreme example, consider the topology in Figure 4.4. Assume that rack R has 20 servers and all of them wish to fully utilize their three 10 GbE links. Despite R's 600 Gbps of demand, its ToRs A, B, and C have only 40 Gbps of total uplink capacity apiece—a 15:1 oversubscription ratio. Even with adaptive load balancing, the ToRs can satisfy at most 1/5th of R's offered load. If R instead bounced traffic through adjacent racks, it could enlist its neighbors to handle all 600 Gbps. Here, some portion of the offered load would be sent through A, B, and C while the remainder is detoured away from the loaded racks, i.e., through R' and R''. This happens recursively, with each successive ToR egressing a portion of the remaining traffic until all of it is handled.

In the common situation where just a single rack is hot, the only limit to the amount of traffic we can detour is the rack's NIC capacity. In other words, if all server-ToR links have the same capacity, detouring can, in principle, achieve full burst bandwidth to/from the loaded rack *regardless of the network's oversubscription ratio.*

Design overview. To ensure that detours are both efficient and do not interfere with adjacent servers, we rely on a per-loop scheduler in the same mold as the one in Section 4.3. These can be one in the same. When a group of servers is bottlenecked by oversubscription in the network, the scheduler can allocate a set of detour paths for them. These detours can extend to as many remote ToRs as necessary, limited in 3 ways:

- *Loop size:* detours can only use remote ToRs in the same loop.
- *Other hotspots:* for simplicity, multiple distinct hotspots are not allowed to detour through the same remote ToR.
- *Path dilation:* operators can limit the associated increase in latency and additional server utilization by imposing a cap on path length.

The scheduler will provide each server with an equal share of the available capacity, but will limit the usable uplink capacity of each remote ToR based on their current utilization. In our experiments, we set the usable capacity to be 60% of the remaining bandwidth (before any detours) so as to provide sufficient headroom for momentary spikes in traffic.

Note that for simplicity, we focus on outbound detours in this section; the process for inbound detours is similar. Also note that we start by assuming isolated hotspots and discuss groups of adjacent, hot racks at the end of this section.

When to detour. The scheduler initiates a detour whenever there is persistent congestion at the ToR level. Persistent congestion is measured by gathering pre-detour ToR utilization statistics every T_{detect} ms. Whenever a ToR is using $> 60\%$ of its total uplink capacity (incoming or outgoing) over an entire period, it is considered "hot" for a minimum of $T_{duration}$ ms. If all of a source rack's ToRs are hot in the outgoing direction, the rack is

considered "hot" and warrants detour paths. Likewise, when all of a destination rack's ToR uplinks are hot in the incoming direction, the scheduler will allocate detour paths and notify the sources of the traffic. Notifications continue to arrive every T_{detect} ms until the congestion subsides.

Computing detour paths. As an example, consider Figure 4.4. In this case, we have a hot rack of servers and wish to shunt traffic through neighboring racks. To shunt traffic left, neighboring racks will take traffic from their rightmost $h = \lfloor \frac{p}{2} \rfloor$ ToRs and forward it along their leftmost h ToRs. Similarly, neighboring ToRs will take traffic from their rightmost h connected racks and forward it to their leftmost h racks. Rightward detours proceed in a similar fashion. Each intermediate ToR egresses a portion of the detoured traffic until all of it exits the Subways server-ToR network.

More concretely, from the ToR utilizations, a per-loop scheduler should be able to deduce the set of overloaded racks and the uplink utilization of every ToR in the loop. A hot rack of servers will manifest itself as p contiguous, heavily-utilized ToRs. From this information, it can determine the set of rack-level detour paths and the usable capacity on each:

1. Consider each group of hot ToRs.

2. Take the 2 ToRs on either side of the set under consideration. These ToRs will be used as egress points for some number of detour paths.

3. If one of the ToRs is already being used for a detour, do not consider it or any further ToRs in that direction.

4. Otherwise, for each of these edge ToRs, calculate the set of detour paths that should use the ToR's spare uplink capacity. We can do this by iteratively backtracing the paths and pruning based on a maximum path length. Specifically, the left ToR should be fed by its rightmost $\lfloor \frac{p}{2} \rfloor$ connected racks, which should in turn be fed by their rightmost $\lfloor \frac{p}{2} \rfloor$ connected ToRs. The detours are allowed to use a total of 60% of the measured residual capacity. The paths split this usable capacity equally.

5. Repeat 2-4 for every group of hot ToRs.
6. Repeat 2-5 until we either run out of capacity on the original sources' server-ToR links or no additional ToRs are under consideration.

Servers source route along these paths using recursive encapsulation with headers that specify the next hop in reverse order. Forwarding can be implemented using software switches that are increasingly common (e.g., Open vSwitch [49]) and with NICs that are beginning to include similar functionality for forwarding among VMs [33]. While these solutions are primarily aimed at providing a communication bridge among various virtual machines executing on a server, they also have the capability to switch among server NIC ports. In Section 4.6.8, we test the software overhead of forwarding and show that it is small.

Note that, even without pruning based on path length, the path dilation of detouring is bounded. Specifically, if the total amount of detourable traffic is t and each ToR has an 60% capacity of s, $\frac{t}{s}$ ToRs are required. This results in just a few hops for typical oversubscription ratios.

Detouring for groups of racks. As in Section 4.3, shortcut-aware job scheduling policies can lead to cases where multiple adjacent racks are simultaneously loaded. The scheduler can detect this case when it notices that more than p contiguous ToR switches are heavily loaded. When this occurs, a couple of modifications must be made to the above algorithm.

The first is that not all servers can detour in all directions. For some, their detours would pass through other loaded racks, which would create contention. As a result of this restriction, only the $\lfloor \frac{p}{2} \rfloor$ leftmost racks in a contiguous group can detour traffic to the left, while the $\lfloor \frac{p}{2} \rfloor$ rightmost racks can detour traffic to the right.

The second modification concerns fairness. Because only the edges of the group can utilize detours, we can achieve a slightly more even division of capacity by forcing the edges—those racks that can utilize detours—to not send any traffic through directly connected ToRs that other members of the group could use. Doing so can improve tail job completion time

for the entire group.

4.5 Physical Considerations

As with any topology, physical design considerations can affect the practicality of an architecture.

In this section, we explain how to physically realize a Subways architecture with minimal increase in cost and complexity. We begin by describing more about the physical layout of modern data centers beyond what was discussed in Chapter 2.

4.5.1 The Layout of Modern Data Centers

The servers of a data center are housed in physical racks that are approximately 19 inches wide and six feet tall. These racks stand side-by-side in long rows that are separated by alternating hot and cold aisles. To cool the servers, air is taken in from the cold aisles and expelled into the hot aisles, where vents take hot air away to be cooled. Server ports and other components that must be readily-accessible by operators face the cold aisles.

Even when a single physical rack holds multiple logical racks, the logical racks are placed in contiguous slots in the rack so that wires can be more easily bundled together. Logical racks do not span multiple physical racks.

Similarly, clusters often reside in a single row so that all the ToR-Aggregation wires can be more easily bundled together. This provides easier cable management at the cost of increased length. These links are optical links because, for the speeds and lengths required, attenuation makes sending a signal over a typical copper wire physically impossible [24]. On the other hand, links within a rack are short enough to be implemented with copper wires, which are significantly cheaper than their optical counterparts because the device electronics can be implemented entirely in high-volume CMOS.

4.5.2 Subways-style Wiring

Like classical data centers, each server in our system connects to the ToR in its rack. Unlike modern data centers, each server also connects to the ToRs closest to it. Until now, we have abstractly described Subways links as connections between servers and nearby ToRs. On the data center floor, however, there are multiple ToRs that are physically close to a target server:

- ToRs in racks that are in the same row and to either side of the target server's rack.
- ToRs in racks across a cold/hot aisle.
- ToRs in the same physical rack as the target server, but part of a different logical rack.

We describe three physical topologies that utilize the above types of links. For two (Section 4.5.2.1 and Section 4.5.2.2), we also discuss concretely how a data center operator might plan for and implement an upgrade.

4.5.2.1 Side-to-Side

In this topology, servers are connected to ToRs in adjacent racks within the same row. Figure 4.5 shows a side-to-side connection pattern for a small loop with $p = 2$. Looking at a row of racks from a cold aisle, we map a loop onto the row by starting at a logical rack. If there is another logical rack above the current one (in the same *physical* rack), that rack is considered to be the next counter-clockwise neighbor in the loop. Otherwise, the next counter-clockwise neighbor is at the bottom of the physical rack that is to the immediate left of the current one. The loop is completed by considering the left-most, top-most rack to be the neighbor of the right-most, bottom-most rack. The physical implementation of this "back edge" must essentially span the entire loop. A visual depiction of the resulting loop of Figure 4.5 is shown in form of a purple overlay.

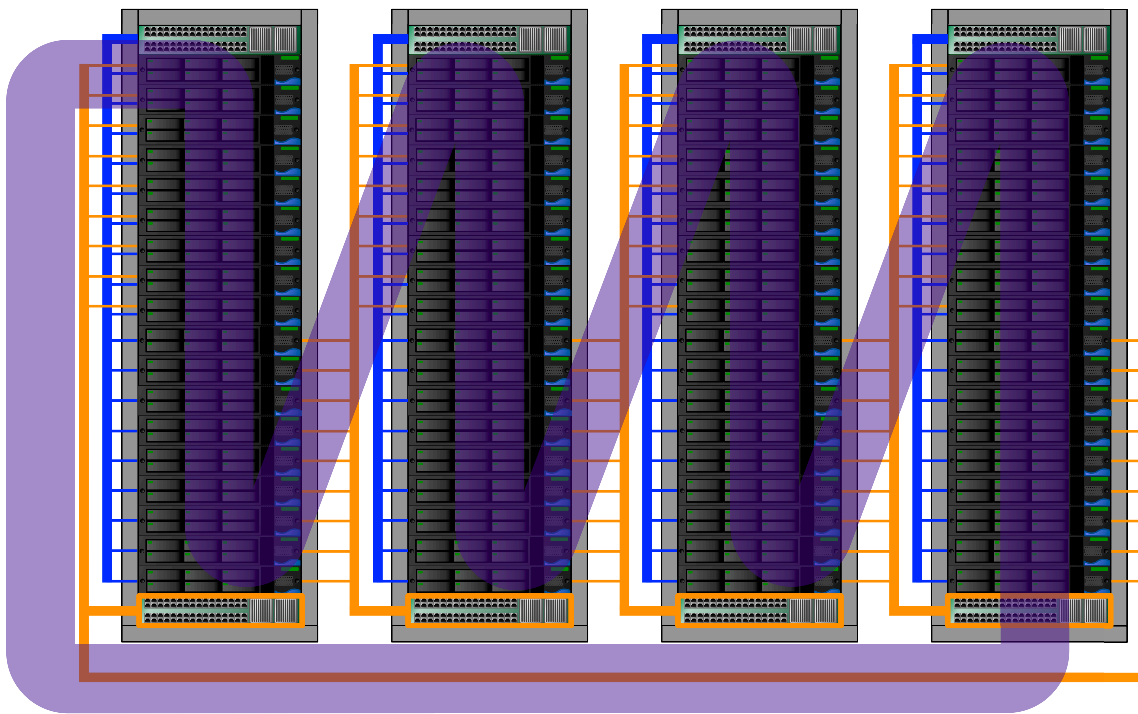


Figure 4.5: An example side-to-side physical topology for Type 1 and up. The image shows eight logical racks split between four physical racks. Blue links are part of the initial 1-port configuration. Orange links and outlined switches denote hardware that is added as part of the upgrade to 2 ports.

Upgrades. In addition to showing an example physical topology, Figure 4.5 also depicts an example upgrade path. Blue links are the initial connections, while orange links and switches represent hardware that is added during the upgrade.

More concretely, this design starts simply: operators connect all servers in a rack to a ToR switch in the same rack. ToRs are placed at the top of the rack just as they are today, with an empty 2U slot at the other end. When upgrading, the operator will install a new ToR at the bottom of each rack that is used to connect either (1) adjacent racks within a row of racks or (2) racks on the edges of the loop. The back edge in this example goes under the raised flooring to minimize wire length. This example is optimized for expansion to $p = 2$, but similar techniques can be used to handle higher port counts.

Loop Length	**Max Wire Length**
8	4 m
16	6 m
48	14 m
128	34 m

Table 4.2: Approximate maximum wire length for a side-to-side wiring pattern with 2 logical racks per physical rack.

Wire length. The longest wires in this design are the back edges. Assuming that each physical rack contains exactly b logical racks, the back edges must cross the width of $\frac{l}{b}$ racks. The maximum length is therefore $h + \frac{l}{b}w$, where h and w are the height (2 m) and width (.5 m) of a rack, respectively. Table 4.2 shows the the resulting maximum wire length for different values of l and $b = 2$. For small loop lengths, it is possible to implement these wires in copper. For larger loop lengths, it is still possible to implement most wires in copper; however, the back edges may need to be optical links.

Cabling Complexity. An advantage of this design is that the initial configuration requires little to no work beyond what is done today: racks can still be preconfigured and there are no cross-rack wires. The upgrade step requires some additional cable installation beyond what is traditional; however, this is such a structured, symmetric topology that we anticipate this

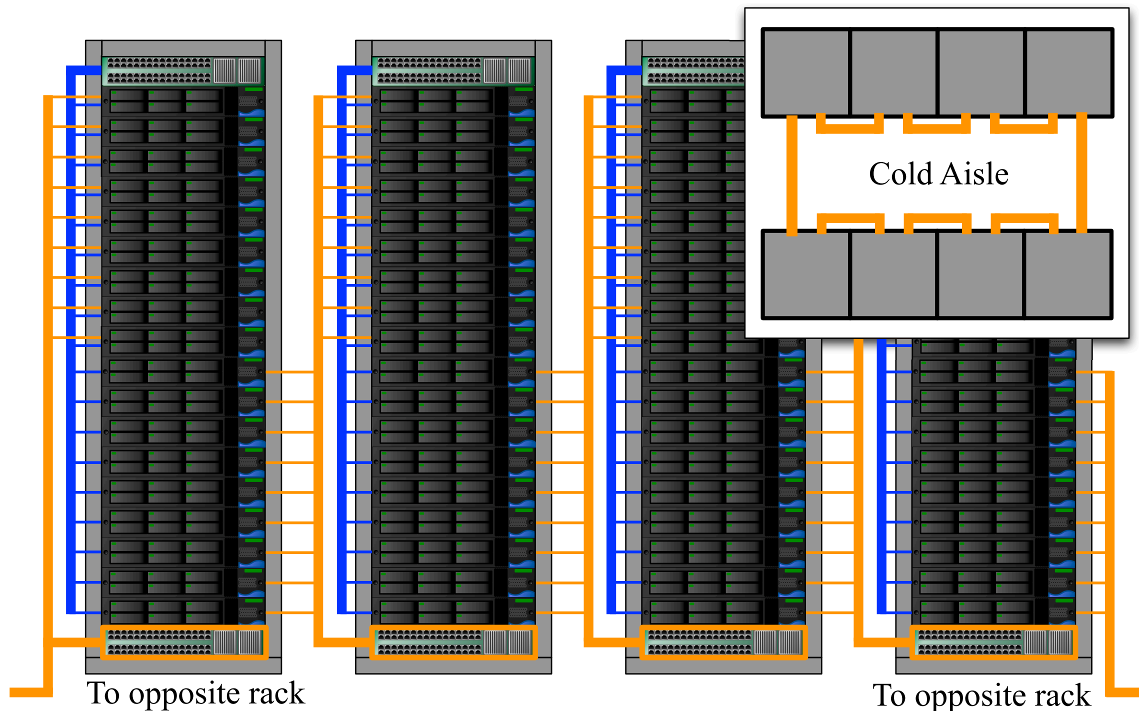


Figure 4.6: A physical topology for a Paired Row design. Like the previous diagram, blue links are part of the initial configuration, and orange links and switches are part of an upgrade. The main image is the frontal view of a row of racks, while the inset is an aerial view of the entire Subways loop.

complexity to be manageable. Further, it can be simplified by bundling the wires between racks into a single cable and connector in a manner similar to the pin headers in modern desktop computers.

4.5.2.2 Paired Rows

The Paired Row design takes advantage of the fact that, in a real data center, servers have multiple ToRs that are physically "adjacent"—not just those in the next rack in the row. In particular, we place racks from a single loop in two rows that are separated by a cold aisle. Within a row, racks are connected side-to-side as in the above design. The difference is that back edges now connect to ToRs across the aisle resulting in much shorter wire lengths.

Upgrades. The upgrade path for this design is very similar to that of the side-to-side design. Figure 4.6 shows the corresponding example. Again, the design starts with a rack layout that is very similar to a typical modern layout. When upgrading, the operator will install a new ToR in each rack that is used to connect either (1) adjacent racks within a row of racks or (2) racks that are separated by a cold aisle. In this way, an operator creates a Subways loop using two parallel rows of racks.

Wire length. In this design, rather than including a back edge that needs to cross the entire loop, we minimize wire length by forming a loop with racks across a cold aisle. Because the network interfaces of both sides of the cold aisle face each other, wires only need to span the width of an aisle (1.2 m [5]).

Assuming wires only cross the aisle at the edges of each loop (through a cross-aisle cable tray or underneath the raised floor), the maximum length of a wire is then approximately $a + w\lfloor \frac{p-1}{b} \rfloor + \frac{h}{b}\left[(p-1) \pmod b\right]$, where a is the width of a cold aisle. The maximum length is independent of the loop length and only depends on the number of racks that a ToR needs to reach $(p-1)$. Some resulting lengths are given in Table 4.3.

Server Port Count	**Max Wire Length**
2	2.2 m
8	3.7 m
16	5.7 m

Table 4.3: Approximate maximum wire lengths for a paired row design with 2 logical racks per physical rack. These values are independent of loop length.

For a 2-port case in particular, the longest intra-row wire is about 2 m, the height of a physical rack. If we also assume we can place wires on both overhead trays and beneath a raised floor, then the length of the cross-aisle wires will be ~ 2.2 m. This is not much more than is required for a single rack and, for currently available link speeds, can be implemented using copper cables.

Cabling Complexity. The main difference between this design and the side-to-side design

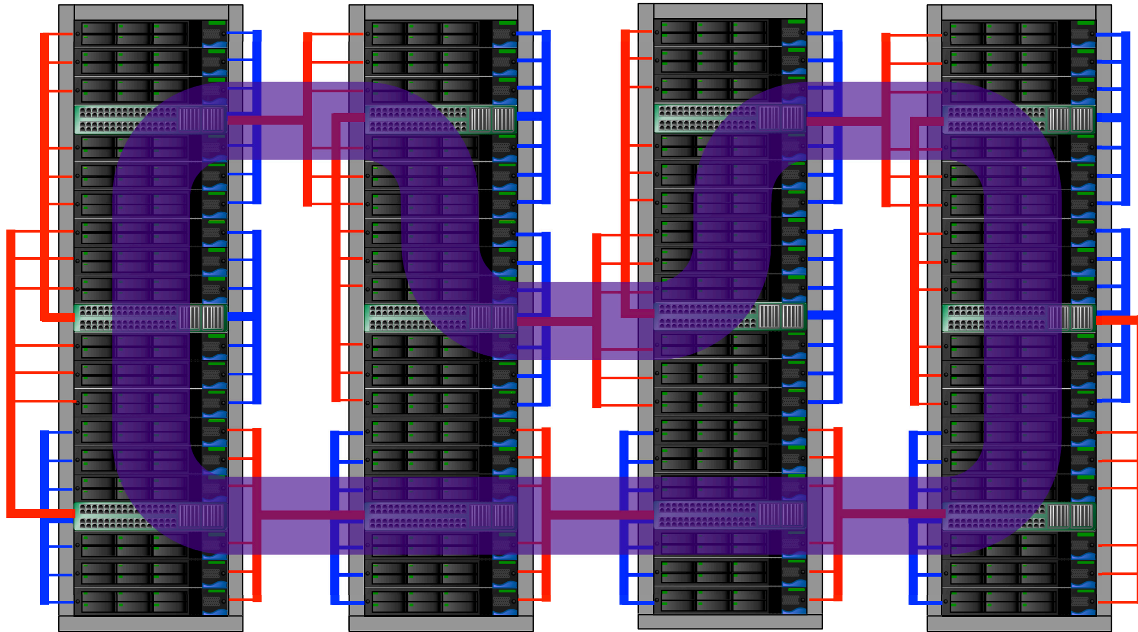


Figure 4.7: An inter-chassis layout. A loop of 12 logical racks is contained in 4 physical racks. Blue lines show wiring between a logical rack and its own ToR. Red lines show links to other ToRs. The purple overlaid loop visually depicts how we map a loop onto the physical layout using a meandered wiring pattern.

is that cables can now cross aisles. This change may add some additional installation effort, but the difference is small as the number of connections and wire bundles are the same as the previous design.

4.5.2.3 Inter-chassis Loops

Inter-chassis loops provide yet another option for decreasing wire length. This design operates within a single row and does not include long back edges. For instance, when physical racks contain exactly two logical racks, we can wire the top and bottom logical racks in separate, side-by-side configurations. We can then place vertical links at the two edges to complete the loop without requiring any long back edges.

When there are more than two logical racks in each physical rack, we can meander the connections so as to cover an arbitrarily large loop size with minimal wire length. Figure 4.7

shows an example with three logical racks per physical rack. The bottom row of chassis are all connected in a line, while the remaining chassis are connected in a zig-zag pattern: up one physical rack and down the next, bridging physical racks through either the top or second-from-the-the-bottom chassis in an alternating pattern.

Wire length. Subways wires in this design never connect components that are more than a few physical racks away. More formally, the maximum length of a wire is usually bounded by $w(p-1) + \frac{h}{2b}$.

The first term corresponds to the maximum wire lengths of the chain at the bottom of Figure 4.7, which will generally be the longest links in this design. The exception is when b and is small, in which case the physical height of a logical rack is larger than its width. For example, in Figure 4.7, the longest wires are those of the top-most ToR on the right-most rack, which need to span half of the physical rack ($\sim 1\,\text{m}$).

The second term measures the vertical distance that the bottom chain must span (i.e., half the height of a logical rack), and it assumes that ToRs are placed in the middle of a logical rack. It also assumes that wires cannot be placed diagonally and instead must travel along the edge of the rack or parallel to the floor.

Port Count	**Max Wire Length**
2	0.75 m
4	1.75 m
8	3.75 m

Table 4.4: Approximate maximum wire length for a inter-chassis design and a range of port counts. We show values for a design with 4 logical racks per physical rack.

The resulting values are given in Table 4.4. Like the Paired Row design, maximum wire length is independent of loop length and instead depends on port count and the number of logical racks in each physical rack. For currently available link speeds and all port counts shown, the lengths are feasible for copper links.

Cabling Complexity This design achieves the shortest cables of any of the three designs

presented and contains all wires within the same row. The tradeoff is that the initial configuration requires slightly more preplanning. Specifically, it must involve wires that cross physical racks making preinstallation of complete racks impossible. Instead, a subset of the server-ToR links can be preinstalled, but a few must be deployed on the data center floor. Even so, the cable bundling techniques outlined in the previous designs can still be used here.

4.6 Evaluation

To evaluate Subways, we implemented the following:

- A packet-level simulator that we used to test medium to large deployments of Subways.
- A small Cloudlab [55] testbed to validate our simulator.
- A server detour implementation to test the feasibility of software-based detouring through servers.

In this section, we denote Subways variants with adaptive load balancing as Type 1LB and 2LB, while detour variants are denoted as Type 1D and 2D. Our evaluation attempts to illustrate the key aspects of Subways, but our results are limited to the specific workloads and parameters that we tested.

4.6.1 Simulator Implementation

We used a modified version of a packet-level simulator that we previously used in the evaluation of other systems [43, 50]. It implements both low-level switch behavior and all of our Subways protocols. The Layer-3 switches use drop-tail queues and flow-level ECMP. The queues are per-port with size based on the bandwidth-delay product of the network. Switching latency along with network propagation delays total to 60 µs per hop. Our workload varies by experiment. For experiments that use TCP, we implemented TCP New Reno in the end hosts using the MPTCP codebase [63] as a reference.

We simulate a standard 3-layer FatTree topology in which ToR switches have 36 10 GbE ports and all other switches have 12 10 GbE ports. Each cluster consists of 12 racks and up to 6 aggregation switches. In some of the following experiments, we evaluate the sensitivity of Subways to different configurations by beginning with a default configuration and varying one parameter at a time.

Our default configuration has two ports per server and 15 servers per rack. For Type 0, we wire both ports to the same ToR switch; for Type 1 and Type 2, we wire servers to overlapping ToRs with a loop size of 12—the size of an entire cluster. The ToR layer has a 5:1 oversubscription ratio while the aggregation layer has a ratio of 4:1. When varying the oversubscription ratio at a given layer, we do so by removing links and aggregation/core switches. For instance, we create our 5:1 ToR-layer oversubscription ratio by only including 6 aggregation switches per cluster and adjusting the number of core switches accordingly.

We assume that ToRs have per-port packet counters that are aggregated by local controllers. Every $T_{rebind} = 10\,\text{ms}$, the controllers collect all the packet counters and disseminate them to any subscribed end hosts. For detouring, we use $T_{detect} = 10\,\text{ms}$, $T_{duration} = 100\,\text{ms}$, and limit the number of hops to at most two intermediate servers.

4.6.2 Validation Using Our Testbed

We validate our simulator using a small Cloudlab [55] testbed. The testbed emulates 16 dual-ported servers, 4 ToRs, and 2 aggregation switches connected through a 2-layer FatTree topology with an oversubscription ratio of 4:1. Because Cloudlab's network topology and queuing characteristics differ from our system, we emulated the ToR and aggregation switches using servers. For this purpose, we have added support for flow-based ECMP routing to the Linux kernel version 3.13. ECMP was implemented using the same consistent hash function that memcache uses to map keys to servers [34]. To prevent the emulated switches from becoming overloaded, we limited link rates to 500 Mbps. For comparison purposes, we replicated this setup in our simulator.

	Type 0	Type 1
Testbed	179.1 s	47.5 s
Simulator	171.3 s	47.6 s

Table 4.5: Comparison of the tail flow completion time of the simulator and the testbed with the same four-rack configurations and single-rack-sink workload. Type 1 is faster because senders and receivers share ToRs for this workload.

For validation, we emulated a scenario where three racks of servers all send traffic to the servers in a single rack. Since a rack has 4 servers, there are a total of 12 senders and 4 receivers. Each sender sends 4 simultaneous 100 MB flows to each receiver. In the testbed, this was accomplished using `iperf`. In the simulator, we started 192 simultaneous TCP flows. In both cases, we record tail flow completion time.

Table 4.5 shows the time from the first flow start to the last flow completion for simulation and emulation, for Subways Type 0 and Type 1. With Type 0, all flows traverse the aggregation layer and are bottlenecked at the downlink into the ToR switch for the receivers. With Type 1, most senders have shortcuts through a shared ToR to the receiver rack. The flows from the remaining senders traverse the aggregation layer, but with less contention. For both topologies, our simulator and testbed results matched each other closely. Some of the remaining difference can be attributed to kernel scheduling latency and jitter resulting from our need to emulate switches in software.

4.6.3 Speeding up a MapReduce Shuffle

Our first evaluation considers the effect of Subways on the performance of an all-to-all MapReduce-like shuffle.

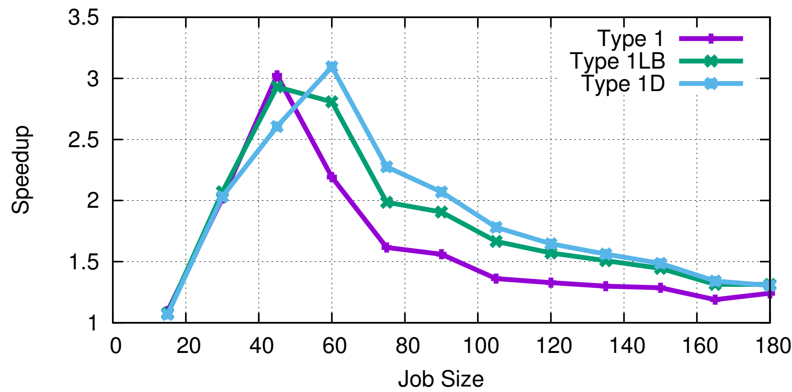
Experiment. Using the default configuration described above, we tested a range of shuffle job sizes. We measured the speedup in completion time of different Subways types compared to Type 0 with the same hardware. For each job, all of the servers in several contiguous racks act as both mappers and reducers. We minimize the number of racks used and group

them together as much as possible in order to promote locality. Mappers initiate flows to reducers with a shuffle size derived from [19], which, in this experiment, is 15 MB between each pair. We mark the job as completed when the last TCP flow finishes and compare the completion time with that of Type 0. Note that we ignore the effects of cross-traffic.

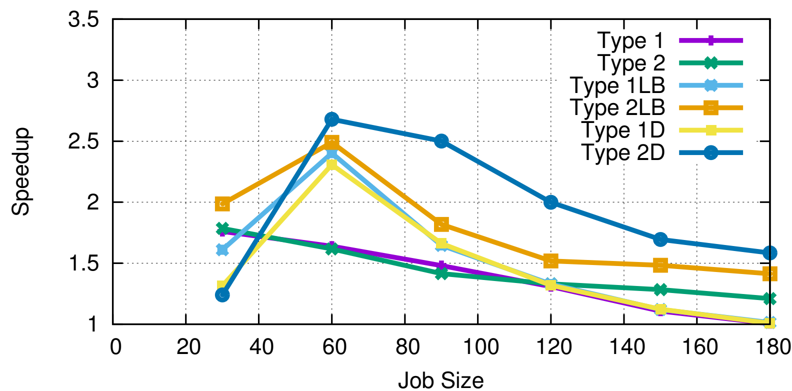
We evaluate two situations. The first involves job sizes of up to 180 nodes (12 racks) within a single cluster. For these, we only test Type 1 and its load balancing variants as Type 2 provides very little additional benefit for a purely intra-cluster and intra-loop workload. The second situation also tests several different job sizes, but splits those servers equally between two different Subways loops. For this case, we test both Type 1 and Type 2 variants.

Results. Figure 4.8 shows results for both intracluster and cross cluster configurations. In most cases, Subways significantly outperforms a Type 0 architecture using the same amount of hardware. This is most pronounced for mid-sized jobs (up to a 3.1× speedup for 60 nodes in an intracluster job) with the speedup tapering off for larger jobs. The primary reasons for these performance benefits are (i) more short paths that avoid the network backbone, (ii) the ability to spread uplink/downlink traffic to the backbone across more ToRs, and (iii) for Type 2, increased cross cluster bandwidth.

In the intracluster case, these benefits peak at a job size of 45/60, i.e., 3 or 4 racks of servers. In these cases, most of the traffic can be transmitted directly through shared ToRs; this is in contrast to Type 0 and traditional rack-based architectures where short paths only exist within a single rack. Load balancing has a greater relative effect—with $p = 2$, 3 racks can spread their uplink/downlink traffic across 4 ToRs versus 3 ToRs with Type 0. Adaptive load balancing and detours only serve to enhance and extend this benefit. As the job size grows to encompass an entire loop, the load balancing effect disappears as there are no free racks for either load balancing or detours. This represents a worst case for our load balancing algorithms. However, even so, short paths continue to provide a modest benefit (about a 1.3× speedup).



(a) Intracluster



(b) Cross cluster

Figure 4.8: The effect of Subways on an all-to-all MapReduce shuffle workload and a range of job sizes. For both intracluster and cross cluster configurations, we show the speedup of different Subways wiring and load balancing variants compared to Type 0.

Like the intracluster configuration, our cross cluster experiments show significant benefits for small to medium job sizes, peaking at a speedup of 2.7× for Type 2D compared to Type 0. For large, cross-cluster jobs, the bottleneck becomes the core network For this reason, Type 2 with its greater amount of cross-cluster bandwidth is particularly effective for this workload. Adaptive load balancing and detours provide further benefits for Type 2 for the same reason.

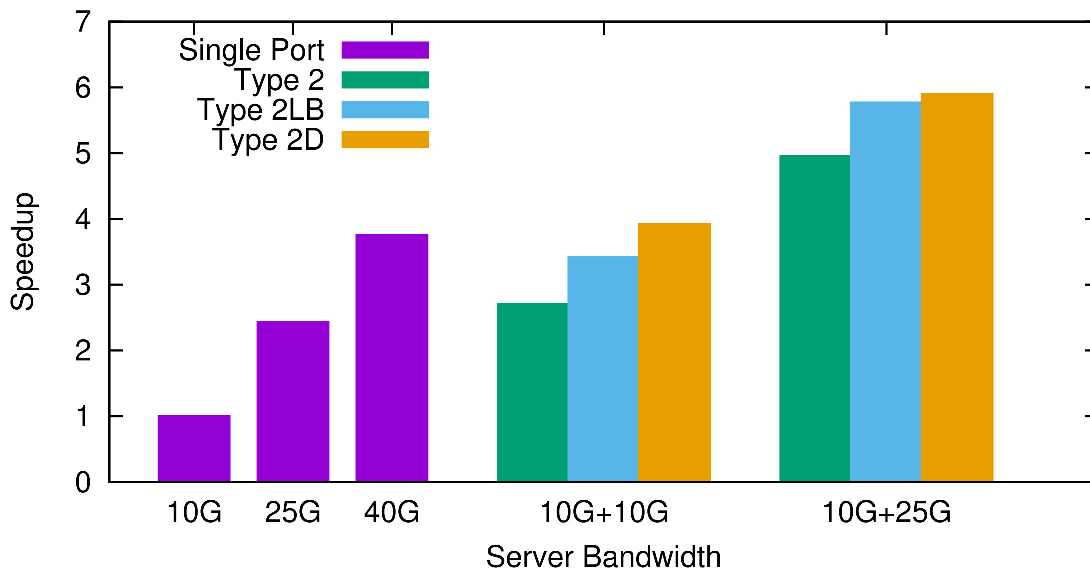


Figure 4.9: Speedup of a MapReduce shuffle for different upgrade paths. The baseline is a configuration with a single 10 GbE port on every server.

4.6.4 Comparison of Upgrade Paths

Next, we examine the effect of the above performance improvements on different upgrade paths.

Experiment. We reconsider the MapReduce shuffle pattern of the previous section, but focus on a job size of 90 nodes all contained within a single cluster.

Our experiment measures speedup in job completion time versus a baseline where every server has a single 10 GbE link and the backbone is also made of 10 GbE links. From this baseline, we evaluate four potential upgrade paths: a full network replacement with 25 GbE links, one with 40 GbE links, a Subways-style 10 GbE augmentation, and a Subways-style 25 GbE augmentation. Note that for both the single port and Subways configurations, we maintain a 15 server rack and add/remove aggregation and core switches to keep the oversubscription ratio the same across upgrade paths. Because this is a heterogeneous configuration, we use Type 2 wiring.

Results. Figure 4.9 compares the upgrade paths. As expected, performance of the 25 GbE and 40 GbE network replacements provide $\sim 2.5\times$ and $\sim 4\times$ speedup respectively. In contrast, with a Type 2D design, a 10 GbE augmentation provides about a $4\times$ speedup. Despite the fact that the servers only have 20 GbE of total bandwidth, the performance is on par with a 40 GbE network because of decreased inter-ToR traffic and better load balancing. Augmenting with a 25 GbE link shows additional performance benefits.

4.6.5 The Effect of Loop Size

In this experiment, we evaluate the effect of loop size on performance. In particular, we look to answer two questions: (1) Is Subways better than a multihoming solution ($l = p$)? and (2) How sensitive is Subways to loop size?

Experiment. Again we consider a shuffle pattern with 90 contiguous nodes. We evaluate Subways Type 1, with and without load balancing, on loop sizes ranging from 2, essentially a multihomed configuration, to 12, spanning an entire cluster. The remainder of the topology adheres to our default configuration.

Results. Our results are shown in Figure 4.10. From the graph, we can see a few interesting effects. First, all Subways Type 1 variants benefit significantly from $l > p$. Multihoming provides a $1.3\times$ speedup on this workload because it has short paths between more pairs of servers. However, a loop size of 3 provides a $1.64\times$ speedup for Type 1 and a $1.85\times$ speedup for Type 1LB and Type 1D, because of even more short paths and opportunities to spread backbone traffic across more ToRs.

Second, ECMP is sensitive to interactions between the job- and loop-size. When $l = 3$, the 6 racks of MapReduce servers fit perfectly in 2 loops and their ToRs are therefore naturally balanced when all flows are operating at full capacity. At $l = 4$, we must split the job into two groups of racks that do not fit perfectly in the loop; with ECMP, this leads to load imbalance at some ToRs as described at the beginning of Section 4.3. Adaptive load balancing fixes this issue. Finally, as expected, detours improve with loop size once the loop

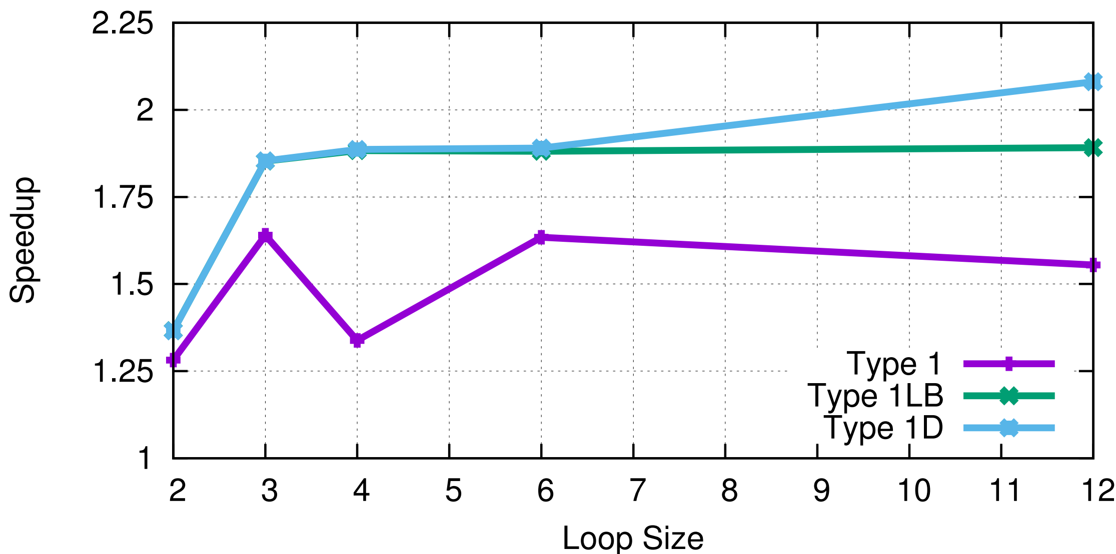


Figure 4.10: Speedup of a MapReduce shuffle for different loop sizes compared to Type 0.

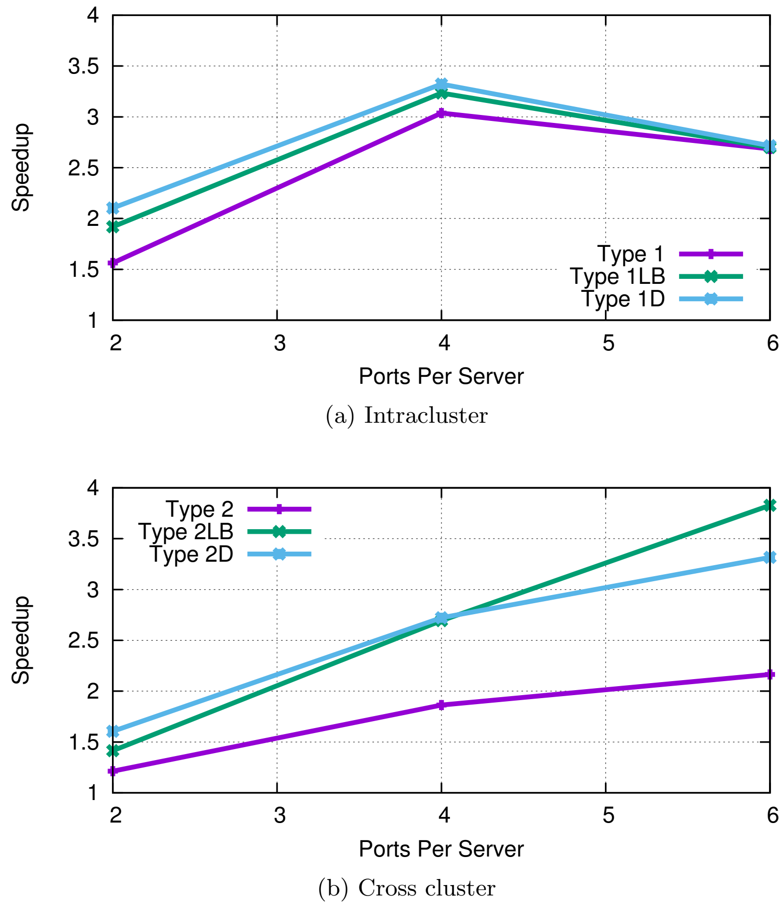
is large enough to encompass both the busy and free ToRs.

4.6.6 The Effect of Port Count

Our default configuration has two ports per server because it is the simplest Subways upgrade step for many data centers. In this section, we extend our discussion to study the effect of further increases in the port count.

Experiment. In this experiment we again use a shuffle workload. We begin with the default configuration and vary the number of ports per server. More ports increases the bandwidth per server. To keep the oversubscription ratio constant, each configuration includes a different number of aggregation switches. For $p = 2$, there are 6 aggregation switches per rack, for $p = 4$, 12, and for $p = 6$, 18. Within a given port count and job size, all Subways types operate under the same constraints.

Like Section 4.6.3, we test two job configurations. The first involves 90 nodes across 6 racks within a single cluster to evaluate Type 1 and its variants. The second involves 180



(a) Intracluster

(b) Cross cluster

Figure 4.11: Speedup of a MapReduce shuffle for different per-server port counts compared to Type 0.

nodes across 12 racks split equally between two different Subways loops. For this case, we only test Type 2 variants.

Results. Figure 4.11a depicts the results for the intracluster configuration. By creating more short paths to more nodes, increasing the number of ports per server also increases the speedup of Subways. There are diminishing returns for very high port counts. With a job size of 6 racks, 6 ports is overkill—servers will prefer the short paths while rarely using their ToR uplinks. For the same reason, the relative benefits due to adaptive load balancing and detours decrease with 6 ports since most of the traffic bypasses the data center backbone.

The cross cluster results in Figure 4.11b provide a complementary view of the effect of port counts on Subways performance. Because the job is divided amongst two loops, short paths will never dominate as they do in the intracluster case. Instead, the bottleneck is in the cross cluster network, where more ports leads to a greater degree of interconnection and speedup compared to Type 0. Here, load balancing becomes even more effective with more ports.

4.6.7 Faster Memcache with Less Hardware

We also look at the effect of Subways on the throughput of a Facebook-like memcache deployment. In particular, we test a range of oversubscription ratios to evaluate the degree to which we can achieve the same performance as a Type 0, but with less hardware.

Experiment. We model this experiment on Facebook's memcache architecture [46]. Each rack within a single cluster consists of either memcache or web servers, but not both. Out of a cluster with 12 racks, 2 are memcache racks while the rest hold web servers. The web servers perform lookups for random keys spread across the memcache servers. Requests are done with UDP packets of 50 bytes, while responses are 1500 bytes. Because the nodes reside in a single cluster, we only evaluate Type 1 and its variants.

As in [46], the metric we use is the maximum sustainable memcache request rate. More specifically, all web servers in a cluster send requests at a constant rate to all memcache servers in the same cluster. We then record the average latency for responses after the system enters a steady state. To find the maximum sustainable throughput, we increase each web server's request rate until the average latency over all requests goes above 1 ms.

Results. Figure 4.12 shows our results. We draw two conclusions from the graph. The first is that, for a given oversubscription ratio, Type 1 and each successive load balancing variant provides higher memcache throughput while maintaining a fixed latency. This benefit remains fairly stable across oversubscription ratios, with Type 1 hovering at around a 1.2× speedup compared to Type 0 and Type 1LB at around 1.6×. The relative benefit of

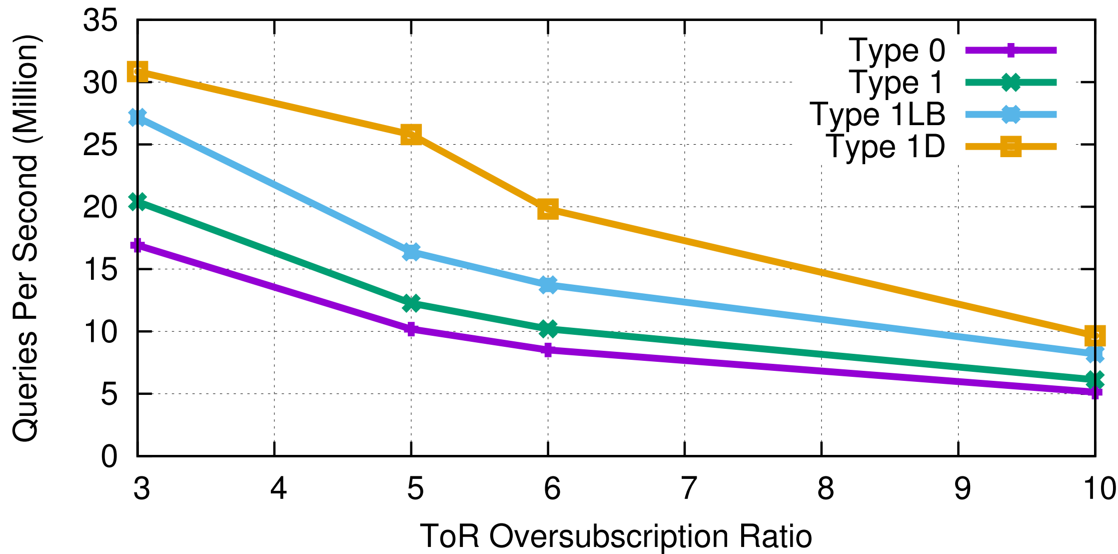


Figure 4.12: Throughput of memcached for various Subways variants for a range of oversubscription ratios. For a given ratio, each variant including Type 0 has the same amount of hardware. Throughput is the maximum number of queries per second while preserving an average response latency of 1 ms.

Type 1D fluctuates between 1.8 and 2.5×, rising as the ToR becomes the bottleneck and falling due to our limit on detour path length. We note that, despite the fact that this experiment does not have any intra-rack locality, it does exhibit intra-*cluster* locality that allows Subways to decrease traffic in the data center backbone. Because responses are much larger than requests, the bottleneck is the outgoing capacity of each memcache rack. Load balancing and detours improve performance by spreading the load more evenly across the available ToRs.

The second conclusion is that, for a target latency and number of queries per second (qps), we can achieve equivalent performance with less hardware. For instance, to achieve a target qps of 10 million and a 1 ms average latency, Type 0 requires a ToR oversubscription ratio of at most 5. On the other hand, Type 1D can provide the same performance with a ToR oversubscription ratio of 10—a factor of two less backbone capacity.

4.6.8 Detour Forwarding Overhead

How much load does software detouring place on servers? We benchmark a prototype of our detour protocol and measure CPU utilization. Our prototype is implemented on the Arrakis high-performance server OS [48]. We note that any solution providing low-latency access to the network would have been sufficient (e.g., a Linux kernel module).

We conduct our experiment on a six machine cluster consisting of 6-core Intel Xeon E5-2430 systems at 2.2 GHz. Each system has an Intel X520 dual-port 10 Gb Ethernet adapter. Both ports of each machine are connected to a single Dell PowerConnect 8024F 10 Gb Ethernet switch. One machine is the detour server under scrutiny. The other machines generate detour traffic to one port of the server. The server decapsulates each detour packet and forwards it along its other port to the next hop.

Our prototype uses a simple pipeline of two server cores: the first core receives packets from the input NIC port, checks whether they are detour packets and, if so, puts a pointer to each one in a shared memory queue. For each queue entry, the second core decapsulates the corresponding packet and sends it to the other NIC port. When done, it uses another shared memory queue to inform the first core that the packet buffer can be reused. We do not copy the packet payload.

Figure 4.13 shows the average CPU utilization for each pipeline step over a period of 5 s. We vary the detour load between 1 and 10 Gbps, the line rate of a single NIC port. We see that total utilization grows linearly with the detour throughput, with decapsulation contributing most to the load. Although our prototype requires two cores, in principle, a single core would be able to sustain a detour workload of up to 7 Gbps.

4.7 Related Work

There has been a large body of work on the topology and management of the data center backbone, above the ToR switch level [10, 27, 43, 45]. Our work is complementary to these, in that wiring servers to overlapping sets of ToR switches can be combined with various

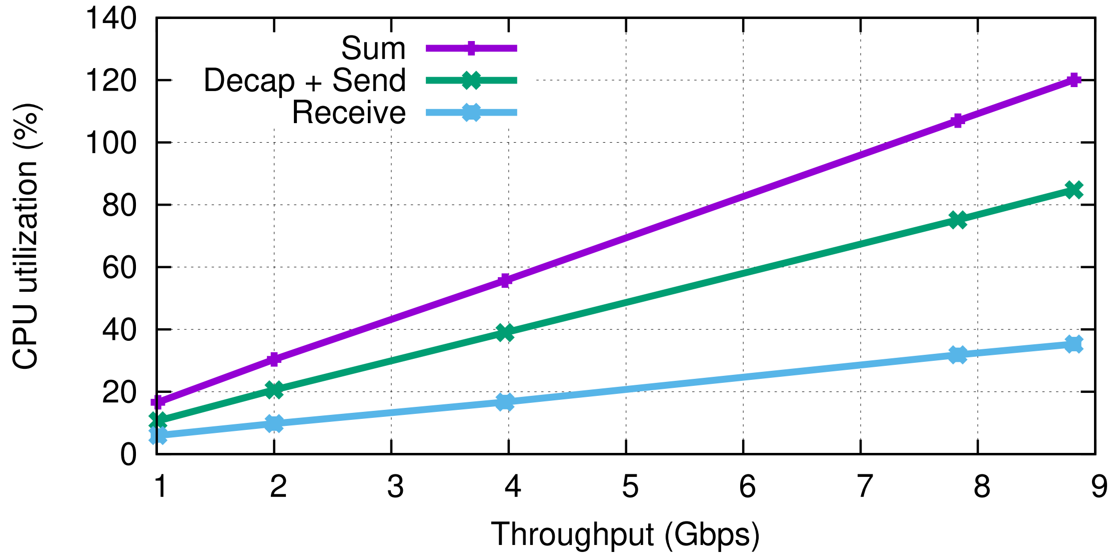


Figure 4.13: Detour throughput versus server CPU utilization. Two cores are involved in detouring. We show the utilization of each individually, and the sum.

data center backbone topologies, including the AB trees of Chapter 3.

BCube, Hypercube, and Torus [28, 38] are backbone topologies based on an assumption of multi-port servers, e.g., by connecting each server port along a different dimension of a hypercube. Although performance in Subways is better with higher values of p, we demonstrate performance benefits for $p = 2$, a port count far smaller than in this earlier work. Further, since we preserve the ToR switch layer, our work is compatible with existing data center operational constraints.

Port trunking (e.g., LACP (802.1ax), MC-LAG, and Trill [60]) is a well-known technique for connecting multiple server links to the same physical (LAG) or virtual (MC-LAG) switch. These architectures are equivalent to Type 0, or in the case of MC-LAG, Type 1 with $p = l$. We show that Subways provides significantly better performance than port trunking, along with further benefits when combined with adaptive load balancing and detour routing.

GRIN [9] is an interesting proposal that looks at wiring adjacent servers together directly, in addition to the ToR switch. Traffic through the adjacent server is detour routed through

that server's ToR link. If the servers are in the same logical rack, this has the effect of increasing the burst bandwidth into each server, at low cost. If the servers are in adjacent racks, GRIN becomes a way to wire each server indirectly into multiple ToR switches, at the cost of dedicating CPU cycles and server link bandwidth to forwarding traffic. Our work is complementary to GRIN, in that our wiring topologies, adaptive load balancing, and detour load shedding could be applied in a GRIN context.

Another approach to reducing hotspots by adding capacity to ToR switches is to use specialized hardware like optical circuit switches and Wi-Fi [25, 30, 41, 65]. Our work can be combined with these approaches. For example, if optical switching is used in combination with a traditional multi-level tree, then Subways can ameliorate any overload that occurs in the oversubscribed switched network. Our work also shows that it is possible to reduce the impact of hotspots with existing hardware.

4.8 Final Remarks

Growing pains are often most acute at the edge of the network. This chapter described Subways, a novel way to wire servers to ToR switches that enables incremental upgrades and reuse of existing hardware. Connecting servers to the ToR switches of neighboring racks decreases the traffic reaching the network backbone. When combined with advanced load balancing techniques, it also spreads the remaining backbone traffic across more ToRs. In addition, Subways improves locality and increases fault tolerance, all while remaining compatible with modern data center network practices, including multi-level switch backbones, wire bundling, rack-based servers, and inexpensive server cables.

Chapter 5

CONCLUSIONS AND FUTURE WORK

This dissertation explores how we can increase the reliability, lower the cost, raise the performance and improve the manageability of data center networks through the wiring topology of the network. With a small amount of controlled asymmetry and a cross-layer architecture to take advantage of that controlled asymmetry, we can design better data center networks.

Specifically, we make the following contributions:

- *A Fault-tolerant Engineered Network:* This dissertation presents F10, a novel fat-tree topology, routing algorithm and failure detector to achieve near-instantaneous restoration of connectivity and load balance after a switch or link failure. Our approach operates entirely in the network with no end host modifications, and experiments show that routes can generally be reestablished with detours of two additional hops and no global coordination, even during multiple failures.

- *Subways:* We also present Subways, a novel way to wire ToR switches to reduce congestion and the cost of the network. By connecting to the ToR switches of neighboring racks, we allow servers to dynamically balance load and decrease peak utilization. Our approach is compatible with any multi-rooted ToR interconnection network and maintains most of the benefits of existing designs (e.g., cheap, short cables from the server; wire bundling; a backward-compatible, rack-based architecture; etc.).

A significant open question for both of these innovations is adoption. Subsequent to the work described in this thesis, data center operators have explored asymmetrical topologies [64, 57] as they provide more resilience to simultaneous failures than a classic folded Clos

network. As localized recovery using AB Clos networks requires network switch support, it is likely that further progress will be enabled by the next generation of reprogrammable switches [18]. For Subways, as we mentioned, a crucial question guiding adoption will be the relationship between total cost of ownership and application performance. Although we have shown that communication-intensive applications can show a large benefit to the Subways approach, fully quantifying operational costs are beyond the scope of this thesis; they depend on technology and business-specific issues.

Although this thesis has made progress toward understanding how to improve the fault tolerance, cost, performance and manageability of data center networks, several additional questions merit future work:

Assisting network operators with failure diagnosis. While F10 tries to quickly and gracefully handle locally-detectable network failures, other types of failures also exist and are becoming increasingly prevalent with the size and complexity of the network; end-to-end failures, performance issues, and network-level misconfigurations are all examples of errors that also need to be handled quickly and gracefully in data centers. An important challenge in this space involves failure localization. Modern data centers have a glut of path diversity—between a given source and destination server, packets can take one of hundreds and sometimes thousands of potential paths and, with ECMP, will do so transparently to the end hosts. When a failure occurs that cannot be detected without end-to-end metrics, it can be almost impossible for hosts to tie the failure back to a particular hardware component.

One approach is to actively probe paths using traceroute. While traceroute can catch some types of failures, the large path diversity means that coverage is difficult to achieve without very high overhead. This is true when trying to cover all paths in the network and is also true when trying to detect gray failures that drop packets at a relatively low rate.

We observe that transparent routing is not a necessity. For instance, when ECMP chooses a next hop for a packet, it does so in an intentionally simple, deterministic fashion. A potential direction is therefore to explore how to enable cooperation between end hosts

and the network for the purpose of failure localization.

Handling misconfiguration. Another challenge in the space of failure handling is the ability to tolerate and recover from misconfiguration. This is a very challenging problem as networks, by their very nature, are connected—a change in one part of the network can necessitate changes elsewhere. For instance, when a data center operator adds/removes an IP prefix from a rack of servers, the network may need to distribute new BGP updates to the rest of the network to ensure that the new prefix is routable. If she mistakenly added a prefix that is too large, a great deal of traffic can be incorrectly diverted to the rack. If the routing updates trigged latent bugs in the BGP implementation, the entire network can go down. Instead, it may be possible to co-design better topologies, routing protocols and control planes such that incorrect and poisonous routing updates cannot propagate from one region of the network to another.

Incremental upgrades/expansion. Subways showed that it is possible to expand per-server bandwidth incrementally with minimal wiring and downtime. Is the same possible when adding to all or part of the data center network backbone? When adding new servers? What about the wide-area network that connects data centers to each other and to edge deployments located in Internet Points-of-Presence (PoPs)? Network operators have existing mechanisms to add capacity and servers to the network, but as this was not an original goal of Clos networks, so-called 'network provisioning' techniques often involve downtime and/or inefficient use of ports. Again, it may be possible to design better topologies and data center architectures that are explicitly optimized for simple and efficient growth.

Smart switches. Finally, we note that today's switches are extremely flexible and powerful. The underlying switch ASIC (application-specific integrated circuit) provides a vast array of detailed statistics and configuration options. In addition, switches include relatively-powerful general-purpose CPUs—often with multiple cores—which are able to run fairly complex control logic.

Traditionally, most of this power and flexibility has been hidden from network operators

by switch vendors. This has changed with the advent of sophisticated data center operators like Facebook and Google that both need additional control/flexibility and have the resources to use advanced switch features. F10 leverages this flexibility for custom, local failure handling at the switch; however there is much more potential for innovation.

For example, network operators could gather fine-grained network statistics, perform arbitrary computation over them, and set applicable configuration values all on the local switch. This is in contrast and complementary to Software-Defined Networking (SDN) which assumes simple switches (for generalizability) and a coarser-grained, centralized control plane. A fast, local control loop opens up an entirely new design space for network load balancing, control plane protocols, and failure detection/recovery techniques.

More broadly, data center networks are an essential technology that has enabled the move towards cloud computing. We have shown in this thesis that topological asymmetry, coupled with cross-layer protocol changes, can significantly improve data center networks along a number of dimensions. As cloud computing becomes more ubiquitous, the stresses on the data center network design are likely to intensify, and there remain a number of important challenges yet to be addressed in this space.

BIBLIOGRAPHY

[1] *Cisco Global Cloud Index: Forecast and Methodology, 20132018.* http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf.

[2] *Cisco Visual Networking Index: Forecast and Methodology, 20132018.* http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf.

[3] Link aggregation (LAG). *IEEE Standard 802.3ad*, 2000.

[4] Shortest path bridging. *IEEE Standard 802.1aq*, 2012.

[5] (2012), TIA standard ANSI/TIA-942-A, data center cabling standard amended. Telecommunications Industry Association, 2012, http://www.tiaonline.org.

[6] Equal cost multiple paths (ECMP). *IEEE Standard 802.1bp*, 2014.

[7] George B. Adams, III, Dharma P. Agrawal, and Howard Jay Seigel. A survey and comparision of fault-tolerant multistage interconnection networks. *IEEE Computer*, 20:14–27, June 1987.

[8] III Adams, G.B. and H.J. Siegel. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Transactions on Computers*, C-31(5):443–454, May 1982.

[9] Alexandru Agache, Razvan Deaconescu, and Costin Raiciu. Increasing datacenter network utilisation with GRIN. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the 2008 ACM Conference on Special Interest Group on Data Communication*, 2008.

[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[12] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, 2014.

[13] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication*, 2010.

[14] Alexey Andreyev. Introducing data center fabric, the next-generation Facebook data center network. `https://code.facebook.com`, November 2014.

[15] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, 2014.

[16] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 2010 ACM Internet Measurement Conference*, 2010.

[17] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine grained traffic engineering for data centers. In *Proceedings of the 7th International Conference on emerging Networking EXperiments and Technologies*, 2011.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[19] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. *Proceedings of the VLDB Endowment*, 2012.

[20] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with Orchestra. In *Proceedings of the 2011 ACM Conference on Special Interest Group on Data Communication*, 2011.

[21] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[23] Chenggong Charles Fan and Jenoshua Bruck. Tolerating multiple faults in multistage interconnection networks with minimal extra stages. *IEEE Transactions on Computers*, 49:998–1004, September 2000.

[24] Nathan Farrington. *Optics in Data Center Network Architecture*. PhD thesis, UC San Diego, December 2012.

[25] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the 2010 ACM Conference on Special Interest Group on Data Communication*, 2010.

[26] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the 2011 ACM Conference on Special Interest Group on Data Communication*, 2011.

[27] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the 2009 ACM Conference on Special Interest Group on Data Communication*, 2009.

[28] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the 2009 ACM Conference on Special Interest Group on Data Communication*, 2009.

[29] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the 2008 ACM Conference on Special Interest Group on Data Communication*, 2008.

[30] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the 2011 ACM Conference on Special Interest Group on Data Communication*, 2011.

[31] James Hamilton. AWS innovation at scale. Presented at re:Invent 2014, Las Vegas, NV, 2014.

[32] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992 (Informational), 2000.

[33] Intel Corporation. *Intel 82599 10 GbE Controller Datasheet*, December 2010. Revision 2.6. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`.

[34] Richard Jones. libketama: Consistent hashing library for memcached clients. http://www.metabrew.com, 2007.

[35] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, March 2007.

[36] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to de-congest data center networks. In *The Eighth ACM Workshop on Hot Topics in Networks*, 2009.

[37] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. In *Proceedings of the 2007 ACM Conference on Special Interest Group on Data Communication*, 2007.

[38] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, Inc., 1992.

[39] F. Thomson Leighton and Bruce M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41:578–587, 1992.

[40] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34:892–901, October 1985.

[41] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[42] Junda Liu, Baohua Yang, Scott Shenker, and Michael Shapira. Data-driven network connectivity. In *The Tenth ACM Workshop on Hot Topics in Networks*, 2011.

[43] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[44] J. Moy. RFC 2328: OSPF Version 2. RFC 2328 (Standard), 1998.

[45] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the 2009 ACM Conference on Special Interest Group on Data Communication*, 2009.

[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[47] P. Pan, G. Swallow, and A. Atlas. Fast reroute extensions to RSVP-TE for LSP tunnels. *Internet RFC 4090*, 2005.

[48] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO, October 2014.

[49] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[50] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[51] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981.

[52] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the 2011 ACM Conference on Special Interest Group on Data Communication*, 2011.

[53] Charles Reiss, Alexey Tumanov, Gregory R. Ranger, Randy H. Katz, and Michael A. Kozuch. Towards understanding heterogeneous clouds at scale. ISTC-CC-TR-12-101, October 2012.

[54] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Standard), 2006.

[55] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), December 2014.

[56] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[57] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better topologies through declarative design. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[58] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[59] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: networking data centers randomly. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[60] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556 (Informational), May 2009.

[61] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[62] Russ White. High availability in routing. `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-1/high_availability_routing.html`.

[63] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[64] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the 9th ACM EuroSys Conference on Computer Systems*, 2014.

[65] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proceedings of the 2012 ACM Conference on Special Interest Group on Data Communication*, 2012.