F10: A Fault-Tolerant Engineered Network

Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington
{*vincent, dhalperi, arvind, tom*}*@cs.washington.edu*

Abstract

The data center network is increasingly a cost, reliability and performance bottleneck for cloud computing. Although multi-tree topologies can provide scalable bandwidth and traditional routing algorithms can provide eventual fault tolerance, we argue that recovery speed can be dramatically improved through the co-design of the network topology, routing algorithm and failure detector. We create an engineered network and routing protocol that directly address the failure characteristics observed in data centers. At the core of our proposal is a novel network topology that has many of the same desirable properties as FatTrees, but with much better fault recovery properties. We then create a series of failover protocols that benefit from this topology and are designed to cascade and complement each other. The resulting system, F10, can almost instantaneously reestablish connectivity and load balance, even in the presence of multiple failures. Our results show that following network link and switch failures, F10 has less than 1/7th the packet loss of current schemes. A trace-driven evaluation of MapReduce performance shows that F10's lower packet loss yields a median application-level 30% speedup.

1 Introduction

Data center networks are an increasingly important component to the cost, reliability and performance of cloud services. This has led to recent efforts by the network research community to explore new topologies [10, 11, 12], new routing protocols [10] and new network management layers [3, 4, 19], with a goal of improving network cost-effectiveness, fault tolerance and scalability.

A state of the art approach is taken by Al-Fares et al. [3] and its followup project PortLand [19]. In these systems, the data center network is constructed in a multi-rooted tree structure called a FatTree (inspired by fat-trees [17]) of inexpensive, commodity switches. These proposals provide scalability, both in terms of port count and the overall bisection bandwidth of the network. They also deliver better performance at low costs, primarily due to their use of commodity switches.

The use of a large number of commodity switches, however, opens up questions regarding what happens when links and switches fail. A FatTree has redundant paths between any pair of hosts. If end host operating system changes are possible between these end hosts, the network can be set up to provide multiple paths. The end host manages packet loss and congestion across the paths using MPTCP [21]. In many cases, the data center operator is not in control of the OS, requiring a network-level solution to fault tolerance. A consequence of our work is to show that entirely network-level failure recovery can be practical and nearly instantaneous in a data center setting.

Addressing this need for network-layer recovery, FatTree architectures have proposed using a centralized manager that collects topology and failure information from the switches. It then periodically generates and disseminates back to the switches and end-hosts alternate sets of routes to avoid failures. Centralized route management is both simple and flexible—a reasonable design choice provided that failures do not occur very often.

Recent measurements of network-layer failures in data centers, however, have shown that failures are frequent and disruptive [9]. Network-layer failures can reduce the volume of traffic delivered by more than 40%, even when the underlying network is designed for failure resilience. As data centers grow, the probability of network failures and the consequent disruptions on the system as a whole will likely increase, further exacerbating the problem.

Our goal is to co-design a topology and set of protocols that admit near-instantaneous, fine-grained, localized, network-level recovery and rebalancing for common-case network failures. Because the network is already a significant part of the cost of the data center, we limit ourselves to not introducing any additional hardware relative to PortLand's FatTree. Other work has shown that local repair is possible at the cost of significant added hardware relative to a standard FatTree [8, 11, 12], so our work can be seen as either improving the speed of repair in FatTree networks or in reducing the hardware cost of fast repair in more general networks. A limitation of our work is that we assume that we can change both the network topology and the protocols used between network switches.

Our system is called F10 (the Fault-Tolerant Engineered Network), a network topology and a set of protocols that can recover rapidly from almost all data center network failures. We design a novel variant of a FatTree topology 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 at the cost of additional latency and increased congestion. Some failures are not short-term, so local rerouting eventually triggers a slightly slower pushback mechanism that redirects traffic flows before they reach the faulty components. To address longer-term failures, 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.

We have implemented a Click-based prototype of F10 and its failure detector and have performed a simulation-based evaluation, based on measurements of real-world data center traffic from [5] and measurements of data center network failures from [9]. Our results show that our system dramatically improves packet loss relative to PortLand with no added hardware cost. Our localized reroutes do incur some path inflation and network state, but these effects are small because of our novel topology.

2 Motivation

Our goal is to design a data center network architecture that can gracefully and quickly recover and rebalance after failures, without any additional hardware relative to a standard FatTree. To motivate our approach, we outline 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.

2.1 Failures in Data Centers

A recent study by Gill et al. provides insight into the characteristics of data center network failures [9]. 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.

The authors assume a model where hardware is either up or down and transitions between those two states, but certain parts of their data—along with anecdotal evidence of *gray failures* from industry—conforms to a stochastic model of failures in which hardware loses a certain percentage of packets. There is thus an additional concern:

- **Existing failure detection mechanisms are too coarse-grained.** Links are marked as down after losing a certain number of heartbeats and marked as up after a brief handshake. Within a short time frame, it is difficult to distinguish between a complete failure, where no packets are getting through, and a situation where the link is congested, and had gotten unlucky with the heartbeats. Conversely, a flaky link that just happened to allow a handshake would appear to be reliable.

2.2 Next-Generation Data Center Networks

Today's data center networks are multi-level, multi-rooted trees of switches. The leaves of the tree are Top-of-Rack (ToR) switches that connect down to many machines in a rack, and up to the network core which aggregates and transfers traffic between racks. A modern data center might have racks that contain 40 servers connected with 1 Gbps access links, and one or two 10 Gbps uplinks that connect the ToR switch to the core, which contains a small number of significantly more expensive switches with an even faster interconnect. The primary challenges with these networks are that they do not *scale*—port counts and internal backplane bandwidth of core switches are limited and expensive—and that they are *dramatically oversubscribed*, with reported factors of 1:240 [10].

Recent proposals for the next generation of data center networks [3, 10, 11] overcome these limitations. In this paper, we focus on a class of these networks based on the FatTree [3] proposal and its subsequent extensions. Inspired by the concept of a fat-tree [17], these FatTrees use a multi-rooted, multi-stage tree structure identical to a folded Clos network [15].[1]

[1] Since there are a few key distinctions between their instantiations, we clarify them here. We use fat-tree to denote the classical concept where links increase in capacity as you travel up toward the root. We use FatTree to denote the proposal of Al-Fares et al. [3], which uses multiple rooted trees to approximate a fat-tree. A similar caveat applies to the research literature's use of the terminology for Clos networks, which route messages along equal-length paths between distinct input

The benefit of these networks is that they are made of cheaper, commodity switches and provide much more path diversity within the network. PortLand takes advantage of this path diversity by using ECMP, which randomly places flows across physical paths. While ECMP lets us take advantage of the increased bandwidth provided by multiple paths, placing a flow on a single physical path means that failures will disrupt entire flows. An alternative is to upgrade the OS and let the end host use a protocol like MPTCP; however, it is not always the case that network operators have the ability to change end host OSes. In this paper, *we explore whether we can make network failures lightweight from the perspective of the end host* so that data center operators can run any end host system and not what is needed for the network.

To ease exposition, we will focus on a *non-oversubscribed* FatTree, in which half of the ports are used as downlinks to connect nodes within the same subtree, and half used as uplinks to access other parts of the tree. However, our system handles both *oversubscribed* (which allocate more ports to downlinks and can scale to more nodes or use few layers) and *overprovisioned* (which allocate more ports to uplinks for reliability and bisection bandwidth) variants, discussed further in Section 8. The root nodes, which do not have uplink edges, use all ports for downlinks. Figure 1a depicts a 3-level FatTree built from 4-port switches.

Our goal is near-instantaneous recovery from failures and load spikes with no added hardware. The original design of Clos networks was more concerned with non-blocking behavior than fault tolerance. Similarly, the papers introducing FatTrees and related proposals [3, 4, 19] discuss basic failover mechanisms, but are principally focused on achieving good bisection bandwidth with commodity switches [3], scalability, resilience to (but not rapid recovery from) faults [19], and centralized load-balancing [4]. These proposals are inherently limited in their ability to recover quickly and thoroughly from faults.

Limited local rerouting: While modern data centers have a variety of failover mechanisms, few are truly local. Data centers that use a link-state protocol such as OSPF require updates sent across the entire network before convergence. PortLand uses a centralized topology manager. VL2 [10] 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 1a.[2] 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 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.

Irregular tree structure because of long-term faults: While data center operators aim to rapidly repair or replace failed equipment, as a practical matter, failures can persist for long periods of time. This can leave the system in a suboptimal state with poor load balancing. 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 from a failed link to all remaining links at a local level, but does not evenly shift load to the remaining paths.

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:

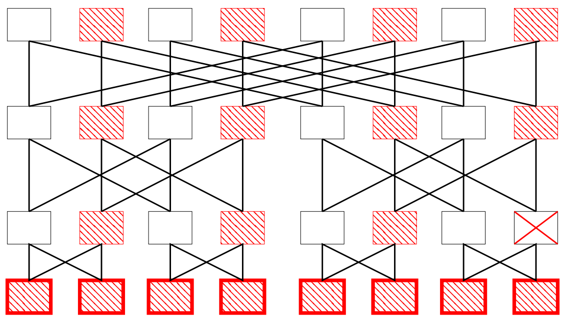
AB FatTree: We introduce a novel topology, the AB FatTree, that is essentially a reworked FatTree with better fault tolerance properties. By skewing the symmetry of a traditional FatTree, the AB FatTree allows for efficient local rerouting. The benefits come at almost no cost. The new topology requires no extra hardware, does not lose bisection bandwidth, and has similar properties to standard FatTrees (e.g., unique paths from a root to leaf, non-blocking behavior, etc.).

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

and output terminals; folded Clos networks, which make no distinctions between terminals; and FatTrees, which allow short-circuiting of paths between nodes in a folded Clos network subtree.

[2]For simplicity, we omit from several of our figures the doubled subtrees generated by folding the root uplinks into downlinks.


(a)

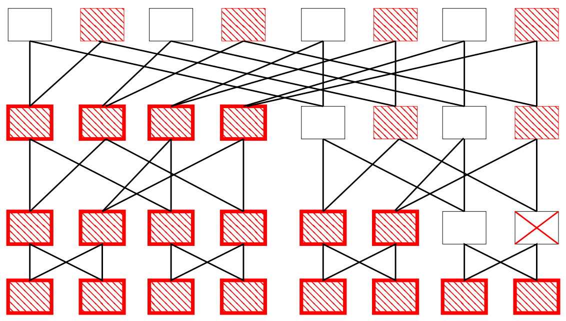

(b)

Figure 1: Path alternatives in (a) a standard FatTree and (b) an AB FatTree. The X indicates a failure, and the hashed rectangles represent switches that are affected by it when trying to send to its children. Bold borders indicate affected switches that have a path around the failure. In the AB FatTree, more switches are affected, but more have alternatives, and they are closer to the failure.

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 1: A key to the notation used in this paper.

pushback mechanism that removes the additional latency, reducing 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.

4 The AB FatTree

As we saw in Section 2.2, the standard FatTree design by Al-Fares et al. [3] has a structural weakness that makes it difficult to locally reroute around network failures. We introduce a novel topology, the AB FatTree, that skews the symmetry of a traditional FatTree to address this issue.

The key weakness in the standard FatTree 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 FatTree 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 FatTree 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 1 contains a summary of the notation we use in this paper.

Figure 2 shows the labeled structure of an AB FatTree 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.[3] 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 FatTree 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 FatTree contains only type A subtrees, whereas in an AB FatTree only half the subtrees are of type A. The remainder are of **type B**,

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

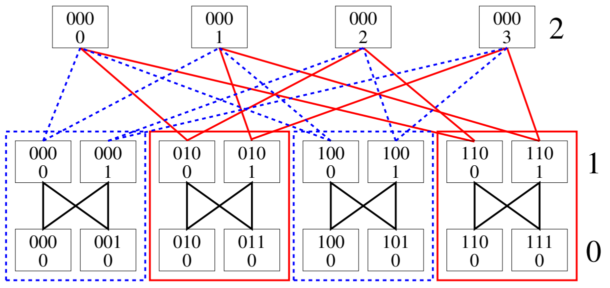


Figure 2: A labeled AB FatTree in which the subtrees with dotted blue lines are of type A and the subtrees with solid red lines are of type B. The numbers to the right of the tree are the *level*, the top number in each switch is the *location*, and the bottom number is the *index*.

wherein children connect to parents with a *stride of* p^i: root j is connected to parents j, $j+p^i$, $j+2p^i$, 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 $L-1$)...(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 [19]—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 FatTree. Revisiting Figure 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.

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.

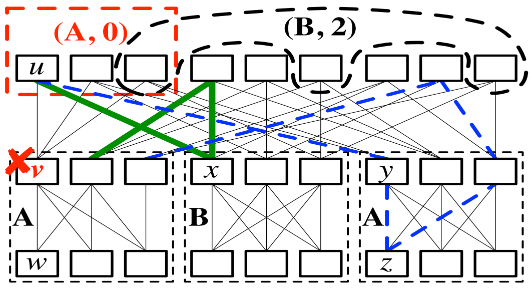


Figure 3: Illustration of the base cases of local rerouting with a failure at v. In the upward direction, w avoids v by routing to any other parent. Downward, u must find detours that avoid the failure group (A,0). The bold green path shows Scheme 1 rerouting through a type B child x, and the dotted blue path shows Scheme 2 rerouting through a child y of same type A.

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, which shows a 3-level AB FatTree 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 node is a leaf node. Most data center services are designed to tolerate rack-level failures. Alternatively, eavh 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 FatTree 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 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. This does not exist in a standard FatTree.

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 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 FatTree, so our work can also be seen as showing how to do local rerouting in a standard FatTree. 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 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.*

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 FatTree enables these notifications to occur closer to the source of the failure than in a regular FatTree. Reducing the number of notifications speeds recovery and minimizes network state.

Consider Figure 4, which shows a 4-level AB FatTree 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 FatTrees and standard FatTrees is that AB FatTrees can install state higher in the tree, at fewer nodes. As a result, pushback message travel less far in AB FatTrees 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

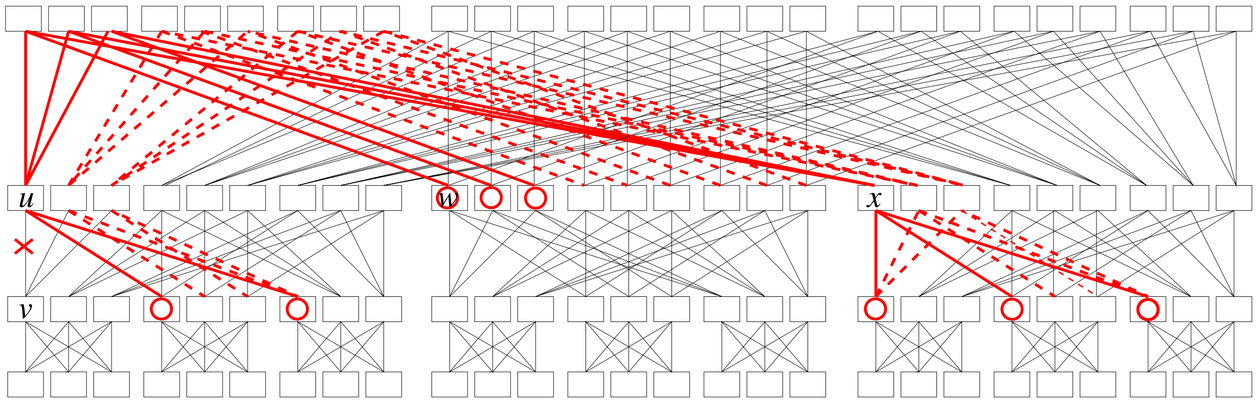


Figure 4: Illustration of pushback when the link from u to v fails (marked by the red 'X'). Solid red lines are the paths along which the notification travels, and the switches with red circles are the set of nodes that need to be notified of the failure. In the case of the entire switch v failing, the dashed red lines show the paths along which associated notifications travel and state would be installed at all the endpoints they touch.

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 FatTrees (and not in standard FatTrees), 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 FatTree. 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 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 messsage $\langle$**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 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 $\langle$**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 $foundCounterExample =$ false
2. For each upward link, $l$:
    (a) If $l$ is down or $b$ is installed on $l$, continue
    (b) Let $foundPrefix =$ 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 $foundPrefix =$ 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 $foundPrefix =$ true
    (d) If $foundPrefix =$ false, then $foundCounterExample =$ true
3. If $foundCounterExample =$ false, push $b$ further down to all children
```

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 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.

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 [24].

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.

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 [5]. 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.

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.

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 [13].

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 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.

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 [6], 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).

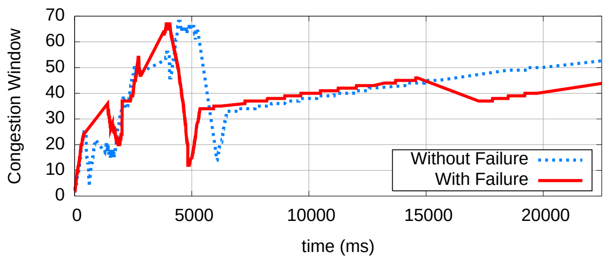


Figure 5: TCP congestion window trace with and without failure. In the case of the failure, a link went down at 15sec and F10 recovered before a timeout occurred.

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 scability 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.

7 Prototype and Evaluation

7.1 Prototype

We built a Click-based implementation of F10 and tested it on a small deployment in Emulab [23]. 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 5. F10 is able to recover from the failure before a timeout occurs and the performance hit is minimal.

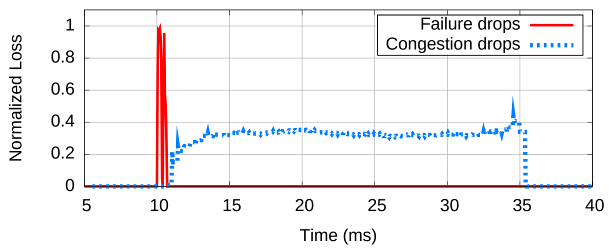


Figure 6: Aggregate losses due to lack of connectivity and congestion in the case a single failure.

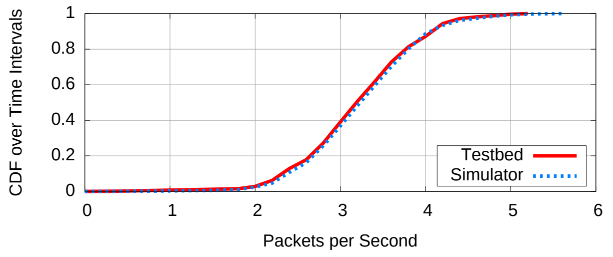


Figure 7: Comparison of throughput of the testbed and the simulator through ten failures and the same topology/offered load.

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

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 FatTree with three layers. Except in Section 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 7 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. [5]. 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. [9] 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 experimental data.

Note that we do not consider leaf (ToR) switch failures, as these are well handled by cloud software. Fault tolerance of rack failures is orthogonal to our work on the robust interconnection between them.

7.3 Recovering from a Single Failure

Figure 6 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 10ms, there is a burst of packet drops due to failure. At around 11ms, the neighbors of the failed switch detect the failure, and local rerouting installs new working routes and eliminates failure drops.

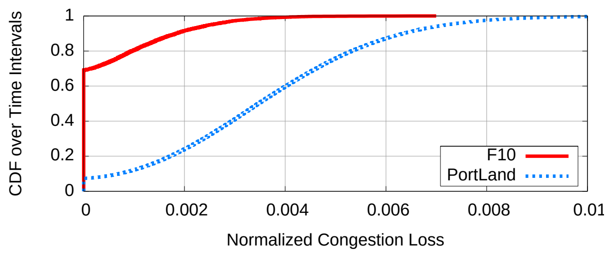


Figure 8: CDF of the congestion losses of both PortLand and F10 under realistic traffic and failure conditions.

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.

7.4 Comparison with PortLand

F10 recovered from the single failure evaluated in the prior section within 1 ms of the failure; this is more than two orders of magnitude faster than possible with PortLand [19], the state of the art research proposal for fault tolerance in data center networks, which reports minimum failure response times 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 7.2.

Figure 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 FatTree, and F10 with an AB FatTree and all our techniques. We aggregated loss statistics over a $500\mu s$ 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 3/4 of time periods, whereas PortLand nearly always has congestion. In total, Portland has 7.6× the congestion loss of F10 for UDP traffic.

7.5 Local Rerouting and AB FatTrees

Note that both standard and AB FatTrees 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 FatTrees during local reroutes.

We measured the path inflation of local reroutes using varying numbers of switch failures (up to 15 concurrent failures, implying up to 360 failed links) in standard vs AB FatTrees. We found that *local reroutes in AB FatTrees experience roughly half the path inflation than in standard FatTrees*, 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 FatTrees, and 4 for standard FatTrees). 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 FatTrees.

7.6 Speeding up MapReduce

We conclude our evaluation by simulating the behavior of a MapReduce job (with TCP flows) in our data center. We used a MapReduce trace generated from a 3600-node production data center [7], 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 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 stragglers 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.

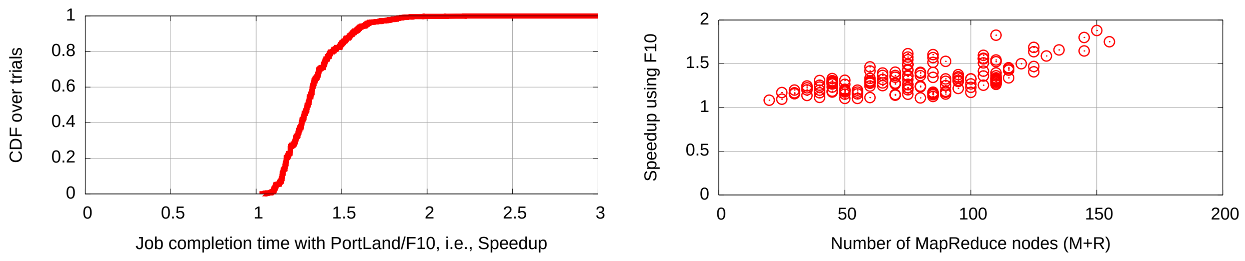
Figure 9a shows the distribution of the speedup; we find that F10 is faster than PortLand with a median speedup of about 1.3×. Figure 9b, shows the distribution of speedup vs job size, and we find that gains are larger when more nodes participate and compete for bandwidth. We conclude that F10 offers significant gains over PortLand, and this will improve in larger future data centers.

8 Discussion

Symmetry: Another concern is that the structure of an AB FatTree is no longer symmetric. Mechanisms like ECMP rely on symmetric shortest paths, present in standard FatTrees. However, AB FatTrees 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 FatTrees.

Beyond AB FatTrees: 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 FatTree 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



(a) Distribution of Speedup (b) Speedup vs Job Size

Figure 9: An end-to-end evaluation using PortLand or F10 for MapReduce jobs.

$(j+1)p$, 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 FatTree provides $p-1$ alternatives for the second hop of Scheme 1 given a single failure, a p type FatTree 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 FatTree, 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.

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., $prefix(a,i) = a \gg (\Sigma_{l=1}^{i}(\lceil\log(D_l)\rceil)))$.

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.

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 $same\_prefix(n.location, c.location, n.level)$

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

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

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

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

Automatically Addressing Nodes: Once the topology of the F10 data center network is verified, we are guaranteed that children and parents are wired together in precisely the way specified in Section 4. In this case, switches can automatically learn their identifiers (level, location, and index) using similar mechanisms as those employed by PortLand.

Although we expect network administrators to carefully plan out their networks, it is possible to automatically address each node. Due to space constraints and the fact that this is not the main thrust of our work, we will not delve into all the details. The main modification between the administrator-defined and automatically-configured address is that the index is not actually required. What we essentially need is just a way to determine failure groups, and so, when ease of sanity checking is not a factor, we can actually replace the *index* with two integers that represent failure groups directly: *groupA* and *groupB*. Nodes that are in the same failure group share the same value for either *groupA* and *groupB*, but this value can be any integer. The level can be immediately determined by the controller by propagating upwards. Initially, all switches connected to at least one end host have $level = 0$. The propagation then occurs in rounds, with all switches connected to level 0 neighbors taking $level = 1$, and so on. From the above we learn parent-child relationships, which we use to actually construct the subtrees (which can be represented by the set of roots of the subtree). We first take a node that is not yet in a set of root, adding it to a new set, S and repeatedly adding parents of children of every node in S until the algorithm converges. After this process, S will be the roots of a subtree. Assigning types and failure groups is a bit more complicated, but essentially occurs in two phases: creating a base type assignment and then filling in other assignments to create a full assignment that is consistent with the base. The intuition is that the types are actually interchangeable and a tree entirely made of type A subtrees is equivalent to a tree entirely made of type B subtrees—the key difference is the way that they are wired in relation to each other. In fact, given a diagram of a tree with labeled type A and type B subtrees like the one in Figure 2, we can swap labels and rearrange the nodes in the diagram to create the same diagram. Type A subtrees will still connect to consecutive groups of nodes and type B subtrees will still connect in a strided manner. For any given tree, we can arbitrarily choose one subtree to be of type A and then assign types for other trees based on that base grouping. Specifically, for each tree, we look at the children of the set of roots and choose a node c with the the greatest number of uplinks to be in a subtree of type A. All of c's parents would then have $groupA = c.UUID$. Everything else is filled in with two simple rules: if at least two of a node's parents share a value for either *groupA* or *groupB*, all parents should share that value, and if at least two of a node's parents differ on a value for either *groupA* or *groupB*, it is of the opposite type and should fill in its parent's fields accordingly. Locations are assigned in a straightforward manner by starting at the topmost level (whose location is all zeroes), and propagating the information downward. For each subtree, s of a tree, t, the controller chooses a unique value, i, less than $\lceil \log p \rceil$ (again, even for type A subtrees and odd for type B). We then set $s.location = prefix(t.location, t.level)$. Nodes are not considered for installation until all of these address variables have been defined. There may be instances of switches that are barred from operation because they are partially connected (or their neighbors are faulty) in such a way that some address variable is ambiguous, but those cases are not of much import since they are extreme corner cases that have the network severely crippled.

9 Related Work

The topic of fault tolerance in interconnection networks has a long history [1, 8, 16]. Most previous work on this topic, most notably [2], 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. We instead allow for a temporary increase in latency for paths affected by faults in exchange for no increase in hardware cost.

In the context of today's data centers, researchers have recently proposed several alternative interconnects. Our work directly builds on FatTrees [3] as they are used in PortLand [19], although our ideas generalize to other multi-rooted trees like VL2 [10] and beyond. We leverage many of the earlier mechanisms in our work. We replace the interconnect with our novel AB FatTree network and co-design local rerouting, pushback, and load balancing mechanisms to exploit the topology.

DCell [12] and BCube [11] 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 [22] 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 [14], respectively. MPLS supports a similar style of immediate local detours (Fast Reroute) while waiting for the failure to propagate upstream (Facility Backup) [20]. 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 [18] has the same intuition that failover should be done at the network layer. 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.

Hedera [4] implements centralized load balancing on top of PortLand. Hedera only schedules new flows and does so in real-time, whereas we choose to globally rearrange flows periodically.

10 Conclusion

Scalable, cost-efficient and failure resilient data center networks are increasingly important for cloud-based services. In this paper, we describe F10, a novel fat-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 30% improvement for MapReduce tasks.

References

[1] G. B. Adams, III, D. P. Agrawal, and H. J. Seigel. A survey and comparision of fault-tolerant multistage interconnection networks. *Computer*, 20:14–27, June 1987.

[2] I. Adams, G.B. and H. Siegel. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Trans. Comput.*, C-31(5):443–454, May 1982.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[5] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained traffic engineering for data centers. In *CoNEXT*, 2011.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[8] C. C. Fan and J. Bruck. Tolerating multiple faults in multistage interconnection networks with minimal extra stages. *IEEE Trans. Comput.*, 49:998–1004, September 2000.

[9] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, 2011.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[12] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[13] S. Kandula, J. Padhye, and P. Bahl. Flyways to de-congest data center networks. In *HotNets*, 2009.

[14] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *SIGCOMM*, 2007.

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

[16] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Trans. Comput.*, 41:578–587, 1992.

[17] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34:892–901, October 1985.

[18] J. Liu, B. Yang, S. Shenker, and M. Shapira. Data-driven network connectivity. In *HotNets*, 2011.

[19] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a scalable fault-tolerant Layer 2 data center network fabric. In *SIGCOMM*, 2009.

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

[21] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM*, 2011.

[22] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: networking data centers randomly. In *NSDI*, 2012.

[23] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, 2002.

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