F^2Tree: Rapid Failure Recovery for Routing in Production Data Center Networks

Guo Chen, Youjian Zhao, Hailiang Xu, Dan Pei, *Senior Member, IEEE*, and Dan Li, *Member, IEEE*

***Abstract*— Failures are not uncommon in production data center networks (DCNs) nowadays. It takes long time for the DCN routing to recover from a failure and find new forwarding paths, significantly impacting realtime and interactive applications at the upper layer. In this paper, we present a fault-tolerant DCN solution, called F^2Tree, which is readily deployed in existing DNCs. F^2Tree can significantly improve the failure recovery time only through a small amount of link rewiring and switch configuration changes. Through testbed and emulation experiments, we show that F^2Tree can greatly reduce the routing recovery time after failure (by 78%) and improve the performance of upper layer applications when routing failure happens (96% less deadline-missing requests).**

***Index Terms*— Data center networks, failure recovery, routing.**

I. Introduction

DATA Center Network (DCN), which is the key infrastructure of almost all the Internet services we rely on today, scales larger and larger to meet the increasingly stringent demands of users and service providers. However, as the number of network equipments (*e.g.*, switches, links) grows, network failures[1] can happen frequently [1], [2]. Furthermore, recent studies [3], [4] have shown that failure recovery takes long time in the current production DCNs [5] running distributed routing protocols such as OSPF [6] and BGP [7] in multi-rooted tree topologies. The long failure recovery time significantly hurts interactive real-time services such as search, web retail and stock trading. For example, according to [8], to guarantee user experience, most interactive real-time services need to meet stringent service deadline considering both computational and network latencies. This time constraint *between* the moment when a query is originated *and* the moment when the results have returned and been displayed can be as short as 300ms. Furthermore, the time constraint for intra-DC tasks for these services can be even lower than 100ms [8]. This further puts a more stringent requirement on failure recovery time.

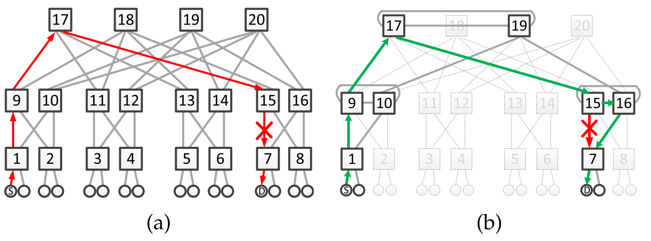


Fig. 1. 4-port, 3-layer fat tree and F^2Tree: One downward link fails. (a) Fat tree. (b) F^2Tree.

We illustrate the slow routing failure recovery problem using an actual testbed experiment. Using virtual machines interconnected in VMware ESXi 5 [9], we have built a 4-port, 3-layer fat tree topology as shown in Fig 1(a). Switches[2] 1∼8 ($S1$∼$S8$) are top-of-rack (ToR), 9∼16 are aggregation, and 17∼20 are core switches running OSPF in Quagga routing software [11], respectively. At time 0ms, node S at the bottom left starts to send a constant-rate UDP flow to node D below $S7$, along the path (S-$S1$-$S9$-$S17$-$S15$-$S7$-D). Then at time 380ms, the link between $S15$ and $S7$ is manually shut down. It takes $S15$'s failure detection mechanism about 60ms to detect the interface failure. Then the OSPF LSA messages take very little time to get propagated from $S15$ to the rest of the network, including $S1$. $S1$, however, waits for OSPF *shortest path calculation timer* (whose default initial value is 200ms, but could be much longer in large operational network [12]) to expire. Then $S1$ calculates the routing table using current global link states. It then knows that the current path has failed and chooses a new path ($S1$-$S10$-$S19$-$S16$-$S7$), and takes another 10ms to update its forwarding table. In total, $S1$ takes more than 272ms before it converges to a working path. Before the convergence, the UDP packets are still forwarded to the failed link of $S15$-$S7$, resulting a 272ms of connectivity loss from S to D, as shown clearly in

Manuscript received November 11, 2015; revised July 9, 2016 and December 11, 2016; accepted January 26, 2017; approved by IEEE/ACM Transactions on Networking Editor Y. Ganjali. This work was supported in part by the National Key Basic Research Program of China (973 Program) under Grant 2014CB347800 and Grant 2013CB329105, in part by the National Key Research and Development Program of China under Grant 2016YFB1000200, in part by the State Key Program of National Natural Science of China under Grant 61233007, Grant 61472210, and Grant 61472214, and in part by the National High Technology Research and Development Program of China (863 Program) under Grant 2013AA013302. The preliminary version [33] of this paper was published in the Proceedings of 35th IEEE International Conference on Distributed Computing Systems (ICDCS), July 2015.

G. Chen, Y. Zhao, D. Pei, and D. Li are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: chen-g11@mails.tsinghua.edu.cn; zhaoyoujian@tsinghua.edu.cn; peidan@tsinghua.edu.cn; tolidan@tsinghua.edu.cn).

H. Xu is with the Department of Computer Science and Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China. (e-mail: harlanxu16@gmail.com).

Digital Object Identifier 10.1109/TNET.2017.2672678

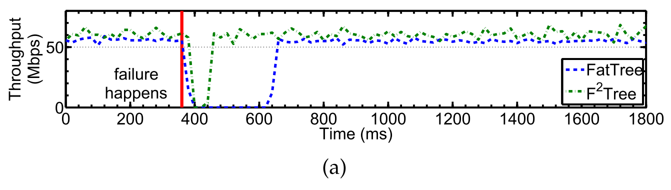
[1]In this paper, *network failure* is defined to be the failure of network equipments related to data forwarding, such as links and switch or router modules. We model all network failures as link failures for simplification. For example, a whole switch failure is modeled as the failures of all its links.

[2]In the rest of the paper, switches in production DCN refer to layer 3 switches [3], [10] that run routing protocols.

TABLE I
THE COMPARISON OF SCALABILITY AND DEPLOYMENT FOR 3-LAYER DCNS BUILT WITH HOMOGENEOUS SWITCHES OF N PORTS USING DIFFERENT SOLUTIONS. (ASSUMING EACH DOWNWARD PORT OF TOR SWITCHES CONNECTED WITH ONE NODE.)

	Nodes supported	Modify routing protocol	Modify data plane
Fat tree [15]	$\frac{N^3}{4}$	n/a	n/a
VL2 [10]	$\frac{N^2}{2}$	n/a	n/a
$\mathbf{F^2}$Tree	$\mathbf{\frac{N^3}{4} - N^2 + N}$	**no**	**no**
Aspen tree [3] $\langle f, 0\rangle$*	$\frac{N^3}{4(f+1)}$	yes	no
F10 [16]	$\frac{N^3}{4}$	yes	yes
DDC [4]	n/a	yes	yes

*f is the fault tolerance value between aggregation and core switches ($f \geq 1$).



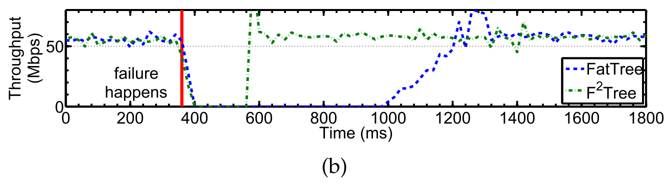


Fig. 2. Failure of one downward link between ToR and aggregation switch in the testbed: Influence to UDP and TCP flows' throughput. (a) Influence to UDP throughput (duration of connectivity loss). (b) Influence to TCP throughput.

Fig. 2(a). While it is normal for distributed routing protocols to have a second or even minute level convergence time in the Internet [12], [13], such a long duration of connectivity loss is apparently unacceptable to a lot of realtime or critical applications in DCN [8]. In a production DCN, the topology is much larger than our small testbed and the failure recovery is more complicated and may be much longer, resulting in path inflation and temporary loops. Furthermore, the holding timer for routing protocol calculation will grow to be very large [14] in a large and unstable network, which leads to a substantial duration of network disruption.

Our key observation from the above simple testbed experiment is that the long duration of connectivity loss is due to two reasons. First, in multi-rooted tree topology such as Fig. 1(a), a downward link (e.g. from $S15$ to $S7$) lacks immediate backup paths. As such, the switch that detects the failure ($S15$) cannot find an immediate working rerouting path. Second, distributed routing protocols such as OSPF take time to learn and react to the failure and find a new working path.

We define the *failure recovery* for routing as the process to recover the connectivity in routing to those affected hosts (which are still physically connected) after network failures happen. In fact, above fundamental reasons for the slow failure recovery in this example are no different from the slow recovery problems in inter-domain routing (BGP) and intra-domain routing (OSPF). Therefore, similar to solutions in BGP and OSPF in the Internet, existing solutions to the DCN slow failure recovery problem are along the following two directions: 1) modifying routing protocol and changing topology [3], and 2) modifying routing protocols and forwarding planes without having to change topologies [4], [16]. More details can be seen in the last two rows in Table I. However, because these previous proposals all rely on non-trivial changes to routing and/or forwarding protocols, it is very challenging to deploy these approaches in an *existing production* DCN.

In this paper, we approach this problem from a different angle. For an existing production DCN with multi-rooted tree topology such as fat tree [15] and distributed routing protocol such as OSPF,[3] we aim to accelerate its failure recovery through only *a small amount of link rewiring and switch configuration changes*, without changing any routing and forwarding protocols or software. Our idea is very intuitive: increasing the downward link redundancy in fat tree topology via rewiring links, and configuring the switch such that it can directly reroute via the newly added backup links when the switch detects a link failure. Because no protocol or software changes are needed, this approach is readily deployed in existing production DCNs, which prior proposals [3], [4], [16] fail to achieve.

Fig. (1b) shows part of the topology according to our approach, by only *rewiring two links* for each aggregation and core switch. For example, for $S15$ ($S16$), we first remove two links: one of $S15$ ($S16$)'s upwards link $S15$-$S18$ ($S16$-$S20$) and one of its downward link $S15$-$S8$ ($S16$-$S8$). Then we use the newly available ports to add two links between $S15$ to $S16$ to form a ring. As a result, when the link between $S15$ and $S7$ fails, the number of immediate backup links (details in §II) that can be used downward by $S15$ to reach $S7$ increased from 0 in fat tree to 2 in our approach. Then we configure two static routes via the two links in the ring at $S15$, so that $S15$ can quickly switch to one of these backup links (*e.g.*, $S15$-$S16$) after the failure of link $S15$-$S7$ is detected, without waiting for OSPF to converge. Therefore, the packets destined to D continue to be forwarded to $S15$, which successfully forwards the packets along the path $S15$-$S16$-$S7$, greatly shortening the failure recovery time. Of course, this path redundancy

[3]We focus on fat tree topology and OSPF for ease of presentation in the rest of our paper, but the slow failure recovery and our proposed solution is also applicable to other multi-rooted topologies such as Leaf-Spine [17] and VL2 [10] and distributed routing protocols such as BGP. More discussions on this can be found in §VI.

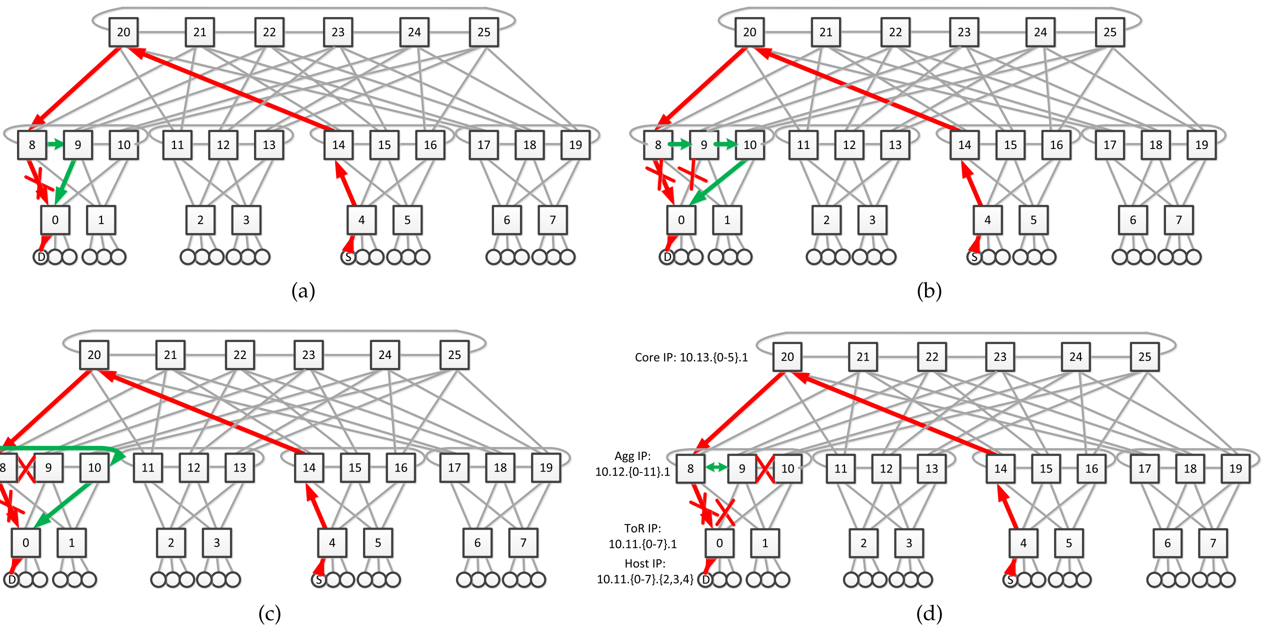


Fig. 3. 6-port, 3-layer F^{2}Tree: Different failure conditions. Fig 3(d) shows an example of address assigning. Hosts connected to each ToR are assigned with addresses in the same subnet of each ToR. (a) 1 downward link. (b) 2 downward links. (c) 1 downward link & 1 right across link. (d) 2 downward link & 1 right across link.

is achieved at the price of less supported nodes, and the seemingly decrease of upward link redundancy, which we discuss in detail later (§III).

We call our above approach F^{2}Tree (standing for Fault-tolerant Fat Tree). Our contributions in this paper can be summarized as follows:

- Through detailed testbed experiments and analysis, we identify the causes of the slow routing recovery problem in current DCN.
- We propose F^{2}Tree, a readily deployable approach to accelerate failure recovery time in existing production DCNs, which only requires rewiring two links and adding a few backup routes in each switch.
- Through testbed and emulation experiments, we have shown that F^{2}Tree can greatly reduce the time of failure recovery by 78% compared to current fat tree. As a result, F^{2}Tree reduces the ratio of deadline-missing requests of partition-aggregate applications by more than 96%, under different failure conditions, compared to original fat tree.

II. F^{2}Tree Design

A. Intuition of F^{2}Tree

ECMP Background: In current fat tree DCNs running distributed routing protocols as Fig. 1(a) shows, ECMP [18] is often used as the load balance scheme. With ECMP, a switch stores the shortest paths with equal costs in its routing table. Each five-tuple flow is forwarded along a particular (based on hashing results) path out of the set of the shortest paths. If the switch detects that the next hop of a path fails, it will just eliminate this failed path from the set of the shortest paths, and the forwarding can continue without any control plane calculation if the set of the remaining shortest paths with same path length is not empty.

For the ease of presentation, we first define a term of **immediate backup link** for a certain link L. *Once L fails, the switch S directly connected to L can continue to use this* ***immediate backup link*** *to forward packets that are originally forwarded through L to their destinations, only with local information.* Thus, in original fat tree using switch with N ports, there are $N/2$-1 immediate backup links for each upward link from ECMP. However, a switch in original fat tree has no immediate backup link for its downward links. Therefore, the switch that detects its downward link failure cannot find an alternative working route without triggering control plane calculation. The goal of F^{2}Tree is to add immediate backup links for the downward link in fat tree, in order to accelerate recovery from downward failures.

B. Link Rewiring & Switch Configurations

We now introduce how F^{2}Tree achieves above goal through topology rewiring and configuration changes.

1) Link Rewiring: In original fat tree as shown in Fig. 1(a), there is no link between switches in the same pod.[4] Once a downward link fails, packets in the detecting switch cannot be immediately forwarded to neighbors in the same pod, although the neighbors have working paths to the destination (e.g. $S15$). F^{2}Tree attempts to add immediate backup links for downward links, utilizing these neighbor switches in the same pod who still can successfully reach the destination.

Specifically, F^{2}Tree reserves a downward and an upward port of each aggregation and core switch to provide fault-tolerance as Fig. 3 shows. As we can see in Fig. 3, the topology of F^{2}Tree is almost the same as fat tree except for a slight modification within each pod. Each aggregation or core switch

[4]A pod is defined as a set of switches that directly connected to the same subtree [3]. E.g., $S9$ and $S10$ in Fig. 1(a) are within a pod, connecting to the same subtree of $S1$ and $S2$.

TABLE II

PART OF THE ROUTING TABLE OF $S8$ IN FIG 3(d). THE LAST TWO LINES SHOW AN EXAMPLE OF CONFIGURATIONS TO USE IMMEDIATE BACKUP LINKS FOR DOWNWARD AND UPWARD LINKS IN F^{2}TREE

No.	Destination	Next Hop
1	D (10.11.0.0/24)	$S0$ (10.11.0.1)
2	S (10.11.4.0/24)	$S20$ (10.13.0.1)
		$S21$ (10.13.1.1)
3	Prefix of all hosts (10.11.0.0/16)	$S9$ (10.12.1.1)
4	Shorter prefix covering all hosts (10.10.0.0/15)	$S10$ (10.12.2.1)

in F^{2}Tree has two ports connected to their neighbors in the same pod (the leftmost switch is considered to a neighbor to the rightmost one). We call the neighbors in the same pod of F^{2}Tree as *across neighbors*, and the links/ports between across neighbors as *across links/ports*. Thus, the switches in each pod form a ring through the across links.

Assuming each switch has N ports in F^{2}Tree, these immediate backup links only cost 2 of the N ports of corresponding switches. The rest N-2 ports of each aggregation or core switch are half connected to switches in the layers above and half below, exactly as in fat tree. As such, F^{2}Tree increases the immediate backup links for each upward and downward link, from $N/2$-1 and 0 in original fat tree, to $N/2$ (including $N/2$-2 ECMP links and 2 across links) and 2 respectively, only at the cost of a negligible bisection bandwidth (discussed in §III).

2) Configuring Backup Routes: We now introduce how F^{2}Tree achieves fast reroute[5] for downward link failures by simple switch configurations.

Specifically, to achieve fast reroute, in each aggregation and core switch, we add *one static route to the prefix containing all hosts in the DCN network (called DCN prefix) with the next hop being its rightward across neighbor, and one static route to the prefix just covering the DCN prefix (called covering prefix) with the next hop being its leftward across neighbor.*

To be more specific, we show how to configure $S8$ in Fig. 3(d) as an example. The DCN prefix is 10.11.0.0/16, and the covering prefix is 10.10.0.0/15. $S8$'s rightward across neighbor is $S9$, and its leftward across neighbor is $S10$. The last two rows in Table II show the two newly added static routes in $S8$'s routing table. These two static routes serve as backup routes for the routes (through both downward and upward links) to all the destinations in 10.11.0.0/16. Upon detecting the link $S8$-$S0$ fails, $S8$ realizes that D is not reachable via 10.11.0.1. When a new packet destined to D arrives, $S8$ looks up its routing table, and finds it can still reach D with the 3th route (to 10.11.0.0/16), and will directly forward the packet via the next hop $S9$, only incurring the normal FIB lookup time. If $S9$ is also detected as unreachable, the 4th route (to 10.10.0.0/15) which has a shorter prefix will be chosen and $S8$ will forward packets through $S10$.

Note that these static routes added by F^{2}Tree are only used when $S8$ cannot find any other routes to a specific destination, because they have shorter prefix than the prefix originally in OSPF routes. We also deliberately disable OSPF routing protocols in all the across interfaces in the ring, thus the newly added across links are only used for backup fast rerouting which will not be advertised and calculated in the routing protocol.

One more thing need to mention is that we deliberately configure the two backup routes with different prefix length. Because if the two backup routes have the same length, a temporary loop may occur during fast rerouting while the downward links of two adjacent switches in the same pod both fail. For example in Fig. 3(b), while $S8$ forwarding packets to $S9$ after detecting the failure of its downward link, $S9$ may forward those packets back to $S8$ by picking up one of its two immediate backup links, because its downward link fails as well. To avoid a potential forwarding loop under this kind of conditions, we assign a longer prefix for the backup route through the right across link than the one through the left across link. With that, during fast rerouting, packets will be forwarded rightward if the right across link works.

C. Deployment

Next, we discuss how to deploy F^{2}Tree in existing production DCNs.

There are two key deployment challenges to update a running large-scale production DCN into F^{2}Tree: 1) how to update the whole DCN in an automated manner with as few manual processes as possible, 2) how to make the update hitless to the existing network traffic (*i.e.*, keep non-stop forwarding during the update). To address these challenges, we have designed the following 5-step deployment scheme of F^{2}Tree.

1) *Planning the topology changes.* First, before making any changes, we need to plan which switch ports & how they are going to be rewired, and which nodes & switches are going to be pruned from the network. The topology change can be conducted in a very simple and regular manner, by reserving the same two ports in each switch for constructing the ring structure in F^{2}Tree. Fig. 1 shows an example of one possible way to rewire the topology. In each aggregation switch, we reserve the rightmost downward port for the right across port and the rightmost upward port for the left across port. Similarly, in each core switch, we reserve the third downward port for the right across port and the second downward port for the left across port. Then, we update the topology database which records the whole DCN topology information (including the switch locations, wiring between the ports, *etc.*).
2) *Migrating services.* Next, we need to migrate the services running in the nodes that are going to be pruned in the new F^{2}Tree.
3) *Disabling the routing protocol on the across ports.* Then, we disable the routing protocols in the across ports which are used for redundancy in F^{2}Tree. This is the key step that ensures the non-stop forwarding during the update, which will move all the traffic away from those

[5]We define *fast reroute* as the process of routing packets around failures with only local failure detection information and without control plane communication and calculation.

across ports before the real topology changes are made. Note that this step can be conducted in an automated manner using a centralized configuration script, which finds the ID of across ports in each switch by querying the topology database and then change configurations to disable routing protocols on these ports.

4) *Configuring backup routes.* Next, we add two backup routes in each aggregation and core switch, thus to utilize the across ports (once the physical rewiring is done) as backup paths. This step can also be automated similarly as the step before.
5) *Conducting physical topology changes.* Finally, after all the configurations are done, we can make the physical topology changes meanwhile avoid affecting the running traffic. Existing DCN topology verification tools (*e.g.*, [19]) can be used to ensure the correctness of topology changes.

D. Handling Failures

We now discuss how F^{2}Tree utilizes the two newly added immediate backup links to deal with different kinds of failure conditions. Upon upward link failure, F^{2}Tree potentially performs better than original fat tree, because it offers one more immediate backup link for each upward link. However, because upward link failures are already handled reasonably well using the ECMP scheme in current production DCNs, we omit the analysis here due to the space limitation. In the rest of the paper, we focus on the the failures of downward and across links. The way to handle upward failures can be easily derived from the way to handle downward failures.

We begin our discussion assuming there is a flow between two end hosts belonging to different pods of aggregation switches, such as S and D in Fig. 3. Without failures, this kind of flow will traverse through the path from bottom to top and top to bottom, as the red line in Fig. 3 shows. Then, we analyze the performance of F^{2}Tree, keeping the precondition that the downward link of a certain switch x (Sx) fails, which is in the flow's downward forwarding path. F^{2}Tree handles failures in the same way regardless of whether Sx is an aggregation or a core switch. Also, the combination of failures above different layers will not affect the working scheme of F^{2}Tree (shown by experiments in §V), because that the fast rerouting scheme can work locally at a switch for each packet that arrives at this switch. Therefore, we only present analysis assuming Sx to be an aggregation switch, and only consider the failures that happen in the same layer. Conditions where physical path does not exist are beyond the discussion. The failure conditions can be summarized as the following four types:

1) *The right across link of Sx and the downward link of the switch right to Sx still work.* Fig. 3(a) shows an example under this condition, assuming $Sx = S8$. During fast rerouting, $S8$ will forward the packets to $S9$ once the link failure is detected. Then $S9$ will forward these packets to the destination D.
2) *Downward links of all the switches in the same pod right to Sx and left to Sy fail, and Sy has a working downward link to the destination (at least 1 switch between Sy and Sx). Meanwhile, across links right to Sx and left to Sy are working.* Fig. 3(b) shows an example of this condition, with $Sx = S8$ and $Sy = S10$. During fast rerouting in this situation, $S8$ will forward the packets to $S9$, and $S9$ will relay these packets to $S10$ because its downward link fails as well. Finally, packets will be forwarded to the destination through $S10$.
3) *The right across link of Sx fails, while Sx's left across link and the downward link of the switch left to Sx still works.* This condition can be illustrated by the example in Fig. 3(c), where $Sx = S8$. During fast rerouting under this condition, $S8$ will not forward packets to $S9$ because it detects failure of both $S0$ and $S9$'s port. So $S8$ will choose the second backup route, forwarding packets to $S10$ using its left across link.
4) *The right across link of a certain switch (Sy) in the same pod fails, and the downward links of those switches right to Sx and left to Sy (include Sy) all fail (If Sy = Sx, the downward link of the switch left to Sx should also fail).* This is a much tougher situation, as shown in the example in Fig. 3(d) ($Sx = S8$, $Sy = S9$). Under this situation, fast rerouting of F^{2}Tree will fail. Specifically, packets will be bounced between $S8$ and $S9$, before $S8$ knows the failure of $S9$'s right across link and downward link, and calculates a new route. In this situation, the time for failure recovery will degrade to that in fat tree.

III. Cost for Routing Redundancy

The philosophy of F^{2}Tree is to trade some aggregate throughput for the increase of path redundancy. In this section, we analyze the scalability of F^{2}Tree and other prior failure recovery solutions. We show that unlike other solutions such as [3], F^{2}Tree only sacrifices a little aggregate throughput, which can be negligible for a large fat tree topology. Table I summarizes the analysis results.

We conduct the analysis by assuming to build a 3-layer non-oversubscribed DCN using homogeneous switches each with N ports, and then compare the number of nodes (hosts) supported in different solutions' topologies. This reflects the aggregate throughput of a non-oversubscribed network. We assume that each fabric link (connected between switch ports) and access link (connected between ToR switch ports and nodes) has the same capacity. Note that although we only analyze 3-layer DCN here, our analysis method can easily be extended to DCN with more layers. We take the two well-known multi-rooted tree topologies in production DCN, fat tree [15] and VL2 [10], as the baselines to compare the scalability and cost with other failure recovery solutions.

For a fat tree DCN [15] with 3-layer switches, each core-layer switch has all its N ports connected to the pods below. Thus in total there are N pods in the aggregation layer. Each aggregation and ToR switch has half of its ports connected upward and the other half connected downward. Therefore, there are $N/2$ ToR switches and $(N/2)^2$ nodes in each pod. Thus, the total number of nodes in a 3-layer fat tree DCN is

$$\frac{N^3}{4}$$

Fig. 1(a) shows an example of fat tree with 4-port switches, supporting 16 nodes in total.

VL2 [10][6] is also a variant of multi-rooted tree topology (shown later in Fig. 9(b)). It has a much denser interconnection than fat tree, which improves the fault tolerance. We only analyze the scalability and cost of VL2 here, and defer the discussion of its performance in failure recovery and how to apply F^{2}Tree for VL2 to §VI. In VL2, each core switch has all its N ports connected to N aggregation switches below in total. Each aggregation switch has half of its ports connected to $N/2$ ToR switches. In a non-oversubscribed VL2, each ToR switch has $N/2$ ports connected to $N/2$ aggregation switches, and the other $N/2$ connected to hosts. Thus in VL2, there are $N/2$ core switches, N aggregation switches and N ToR switches. Thus the total number of nodes in a 3-layer VL2 DCN is

$$\frac{N^2}{2}$$

The left part of Fig. 19(b) shows an example of VL2 with 6-port switches, supporting 18 nodes in total.

Now we analyze the number of nodes supported in F^{2}Tree. As introduced before in §II, F^{2}Tree has exactly the same wiring manner with fat tree, except that each aggregation/core switch in F^{2}Tree reserves two ports connected to their neighbors in the same pod. Thus in F^{2}Tree, the core-layer switches are connected to $N-2$ pods of aggregation switches below in total, and each pod contains $N/2-1$ ToR switches. The same as fat tree, each ToR switch has half of its ports connected downward to the nodes, thus in each pod there are $(N/2-1)(N/2)$ nodes. Therefore, the total number of nodes in a 3-layer F^{2}Tree DCN is

$$\frac{N^3}{4} - N^2 + N$$

Fig. 1(b) and Fig. 3(a) show two examples of F^{2}Tree with 4-port and 6-port switches respectively, supporting 4 nodes and 24 nodes in total.

We also analyze the number of nodes supported in other failure recovery solutions. We only discuss the scalability of these solutions in this section and leave other details in §VII. Aspen tree [3] changes the topology of fat tree to accelerate failure recovery, by making the switches in the upper layer connect to each pod below with more than one link. Originally, each upper-layer switch only has one link connected to each pod below. Aspen tree adds the number of such links from 1 to $f+1$, and calls f as the fault tolerance value. We analyze the Aspen tree with only $f > 0$ between aggregation and core switches, which is recommended by [3] as a practical trade-off between fault-tolerance, scalability and cost. In Aspen tree, each core-layer switch is connected to $N/(f+1)$ pods of aggregation switches. Each pod has the same $(N/2)^2$ nodes/hosts in fat tree. Thus, the total number of nodes in a 3-layer Aspen tree DCN (fault-tolerance between the core and aggregation layer) is

$$\frac{N^3}{4(f+1)}$$

F10 [16] changes the wiring manner of fat tree, but it keeps the same aggregate throughput as fat tree. DDC [4] makes no changes to the topology.

We summarize the results above in Table I to show the cost more clearly. F^{2}Tree can support $\frac{N^3}{4} - N^2 + N$ nodes, which is only $N^2 - N$ less than $\frac{N^3}{4}$ nodes in standard fat tree. Only smaller with a low-order terms, we can see that F^{2}Tree is able to support approximately the same number of nodes as fat tree as the network scales larger. For instance, if 128-port switches are used, F^{2}Tree only supports about 2% nodes less than original fat tree. However, other fault-tolerant topologies such as Aspen tree improves the fault-tolerance at the cost of much more aggregate throughput. Aspen tree supports only $\frac{1}{f+1}$ of nodes of original fat tree, where f (always ≥ 1) is the fault tolerance value between aggregation and core switches. For example, if adding one fault tolerance degree (*i.e.* $f=1$) in Aspen tree, it trades off 50% aggregate throughput compared to original fat tree.

IV. Testbed Implementation & Experiment

A. Implementation

Basic Fat Tree: As briefly introduced before (§I), we have built a basic 4-port, 3-layer fat tree prototype (Fig. 1(a)), based on virtual machines interconnected in VMware ESXi 5 [9]. Specifically, all the switches and end-hosts are virtual machines (VMs), running Ubuntu 14.04.1 LTS. Each switch VM has 4 1Gbps ports, and each end-host has 1 1Gbps port. Switches run Qugga OSPF routing software [11] (v0.99.22) with ECMP enabled.

Live Migration Scheme to **F^2***Tree:* We have implemented the deployment scheme (introduced in §II-C), to update the basic fat tree prototype (Fig. 1(a)) into F^{2}Tree (Fig. 1(b)). Specifically, we first plan the topology changes as Fig. 1(b) shows. We plan to use the rightmost downward/upward port in each aggregation switch and the second/third port in each core switch of the original fat tree for the across ports in F^{2}Tree. As such, we update the property of those ports as across ports in the topology database. We develop a bash script which first disables the OSPF routing protocol on all the across ports in each aggregation/core switch (by looking up the topology database), and then adds two backup routes in each aggregation/core switch. Then after all the configurations are done, we manually rewire the physical topology from fat tree (Fig. 1(a)) to F^{2}Tree (Fig. 1(b)). Finally, those switches and nodes pruned from the network are shutdown.

B. Experiment

Next, we use testbed experiments to evaluate the non-stop forwarding during migration to F^{2}Tree (§IV-B.1), and the fast routing recovery to failure after successful migration to F^{2}Tree (§IV-B.2).

[6]Originally, VL2 is designed using high speed fabric links (*e.g.*10Gbps switch links) and relatively low speed access links (*e.g.*1Gbps node/host links). However, to make the comparison of the scalability and cost under the same fair condition, we also assume each fabric link and access link with the same capacity as in fat tree [15].

1) Non-Stop Forwarding During Migration: Experiment Setup: We start a UDP flow and a TCP flow respectively from node S to D during the migration to F^{2}Tree, and examine whether those traffic are affected. Both UDP and TCP flows send unlimited data to the destination with maximum rate. All other settings are set as the default ones in Linux and Quagga.

Results: Results show that *no packets* are dropped in the UDP flows and the throughput of TCP flows keeps very steady during the network updates. We omit the result figures here for the interest of space. The results are confirmed by running the experiments for multiple times, and using both our customized flow generator and iperf [20] to generate traffic.

It is the third step of the deployment scheme (§II-C), *i.e.*, *disabling the routing protocol on the across ports*, which mainly helps the network keep non-stop forwarding during updates. Specifically, there are two phases after the routing is disabled on (to-be) across ports:

- Originally, flows from S to D traverse the path $S9$-$S18$-$S15$ before the migration. After the routing is just disabled on the ports between $S9$-$S18$ and $S18$-$S15$ (3^{rd} step in §II-C), packets continue to be forwarded through the old path $S9$-$S18$-$S15$ before routing converges to the recent changes. Because we only disable the routing on the ports and these ports are still working physically, those packets going through $S9$-$S18$-$S15$ are not dropped and successfully forwarded to $S7$ and then to D.
- Next when the routing converges, $S9$ will delete the routing entry passing through $S18$ from its routing table. Then the flows will be switched to the other working entry $S9$-$S17$-$S15$. Note that switching traffic between different routing entries only costs a normal routing lookup time, thus no packets are dropped during the switching. As such, the flows in $S9$-$S18$-$S15$ are moved to $S9$-$S17$-$S15$, with non-stop forwarding.

2) Fast Routing Recovery in F^2*Tree: Experiment Setup:* Next, we generate both a constant-rate UDP and TCP flow from S to D sending a segment of 1448 bytes data every 100μs. During the data forwarding, we tear down a downward link between ToR and aggregation switch along the forwarding path, to evaluate the network performance against failure. Links are torn down by shutting down certain interface of the switches. The time for interface failure detection is similar to the fast failure detection techniques such as BFD [21] (about 60ms), thus approximating the real DCN. The same experiments are done in both the original fat tree and F^{2}Tree.

Results: Fig. 2 shows the instantaneous receiving throughput of both UDP and TCP flows, with a time bin of 20ms. The red vertical line indicates the time when failure happens. As we can see, both UDP and TCP flows in F^{2}Tree recover from failure much faster than original fat tree. In Fig. 2(a), we can see that the UDP receiving throughput falls to zero for only about 60ms in F^{2}Tree, while it lasts more than 270ms in fat tree. This duration of throughput fall comes from the *loss of connectivity*, during which packets of the flow fail to be forwarded to the destination. With no need of control plane calculation and FIB update, the 60ms of connectivity loss in F^{2}Tree only comes from the time of failure detection. As for fat tree, there is a duration of connectivity about 272ms. This duration mainly consists of a 60ms period of failure detection, a 200ms period of OSPF default initial shortest path calculation timer, and a 10ms period of FIB update. Also, the LSA propagation and the CPU processing delay contribute a small part.

Next, we discuss how failures impact the TCP flows in F^{2}Tree and fat tree, respectively. From Fig. 2(b) we can see, the TCP flow in F^{2}Tree has a significant shorter time for throughput recovery than that in fat tree. We measure the time when TCP throughput is lower than 1/2 of the average throughput before failure happens as the *duration of throughput collapse*. While fat tree's duration of throughput collapse is 700ms, F^{2}Tree's is only 220ms. The big gap of the TCP recovery time between the two solutions is due to the TCP retransmission timeout (RTO). Specifically, after failure happens, there are about 60ms and 270ms in F^{2}Tree and fat tree respectively, when the destination is out of connectivity. During this time, packets of the TCP flow are all lost and incur a retransmission after initial timeout of 200ms. In F^{2}Tree, the retransmitted packets successfully get to the destination, while in fat tree, the retransmitted packets cannot reach the destination because the connectivity has not been recovered yet. Thus, it leads to another retransmission after a doubled RTO, which increases another 400ms of throughput collapse. Setting a shorter initial RTO down to hundreds of μs may successfully reduce the duration of TCP throughput collapse both in fat tree and F^{2}Tree. However, the TCP throughput collapse will still last for at least the duration of connectivity loss.

V. Emulation Experiment

In this section, we evaluate the performance of F^{2}Tree in the emulation environment with a larger scale. First, we study that how F^{2}Tree performs under different failure conditions discussed in §II-D. Then, we evaluate F^{2}Tree's improvement to upper layer applications using the workload derived from production DCNs. We also evaluate F^{2}Tree under all-to-all traffic scenario and different settings of routing calculation timers.

Emulation Environment: In order to use realistic routing and forwarding implementation, we choose a feasible software-based solution, using Quagga [11] software router running OSPF and Linux network stack as the control and data plane of our emulated network. We introduce these real implementations into NS3 [22] through the Direct Code Execution (DCE) [23] environment. DCE is a framework that provides an environment to execute, within NS3, existing implementations of userspace and kernelspace network protocols or applications. Thus, we can build networks of fat tree and F^{2}Tree topology within NS3, using switches and nodes implemented with Quagga and Linux. We implement real TCP and UDP based applications on Linux, and install them on the nodes in NS3 to generate different traffic.

ECMP is used in our simulation, just like in existing production DCNs. Each link within DCN is set with a bandwidth of 1Gbps, and a propagation delay of 5μs. A 60ms failure

TABLE III
LABELS THAT REPRESENTS DIFFERENT FAILURE CONDITIONS IN AN 8-PORT 3-LAYER DCN

Label	Failures	Belong to which failure condition in §2.4
C1	1 link between ToR and aggregation switch	1st
C2	1 link between core and aggregation switch	1st
C3	1 link between ToR and aggregation switch & 1 link between core and aggregation switch	1st
C4	2 adjacent links between ToR and aggregation switches in the same pod	2nd
C5	All links between ToR and aggregation switches in the same pod except the one of the left across neighbor	2nd
C6	1 link between ToR and aggregation switch & 1 right across link	3rd
C7	2 links between ToR and aggregation switches & 1 right across link	4th

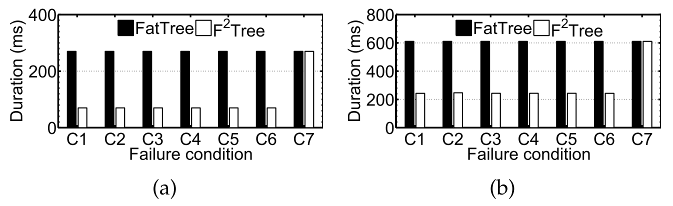


Fig. 4. Results upon different failure conditions in an 8-port 3-layer DCN. (a) Duration of connectivity loss. (b) Duration of TCP throughput collapse.

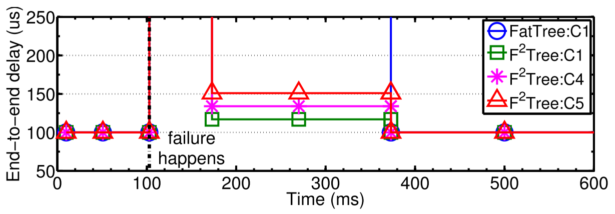


Fig. 5. Comparison of end-to-end delay during the failure recovery.

detection delay and 10ms FIB update delay are added to the emulation, according to the results measured in our testbed. All the rest of configurations in F^{2}Tree and fat tree are left as default.

A. Handling Different Failure Conditions

Experiment Setup: We set up a UDP flow and a TCP flow from the leftmost end host to the rightmost one. During the data transmission, we inject 7 different types of failure conditions (shown in Table III) containing links *either* along the path, *or* not on the path but may impact the packet forwarding. These conditions have covered all the failure conditions discussed in §II-D. C1 and C2 belong to the 1st failure condition in §II-D, with Sx being aggregation and core switch respectively. C3 is a combination of C1 and C2, which also belongs to the 1st condition. C4 and C5 are special cases of the 2nd failure condition, and C6 belongs to the 3nd condition. Finally, C7 belongs to the 4th condition in §II-D. We compare the performance of F^{2}Tree and fat tree for C1 to C7. Note that there is no across links in fat tree, thus for C6 and C7, we only tear down the downward link between ToR and aggregation switch in fat tree. All the link failures in our emulation are bidirectional.

Results: Fig. 4 shows the duration of connectivity loss, and the duration of TCP throughput collapse both in fat tree and F^{2}Tree. For failure condition C1, F^{2}Tree reduces the duration of connectivity loss by about 78%, from 270ms to 60ms, compared to fat tree. As for the TCP flow, there are 220ms and 610ms of throughput collapse in F^{2}Tree and fat tree, respectively. All these results are similar to those in the testbed experiments analyzed before.

Fig. 5 demonstrates the variation of end-to-end delay during the process of failure recovery under several representative failure conditions. Except for the 270ms duration of connectivity loss between time 100ms and 370ms, the end-to-end packet delay in fat tree under C1 remains to be $100\mu s$, which consists of propagation, transmission and processing delay. During the fast rerouting in F^{2}Tree between time 170ms and 270ms in Fig. 5, packets successfully reach the destination through backup paths with one extra hop. Thus, the end-to-end is a slightly higher, which is $117\mu s$ during this period. After the control plane converges at 270ms, the end-to-end delay falls down to $100\mu s$, the same as that in fat tree.

Besides the link above the ToR layer, we also consider the condition that the link in the higher layer fails (C2) and the situation that links from both layers fail together (C3). In C2 and C3, fat tree performs almost the same as C1. Compared to C1, fat tree takes a little bit shorter time for LSA to propagate to the ToR switch that is connected with the source end host, which is negligible to the whole failure recovery time. For C2 and C3, F^{2}Tree performs almost the same as in C1, which verifies our analysis before. The end-to-end delay performance is the same as the one in C1, which is omitted in Fig. 5 for brief.

C4 and C5 are tougher conditions for F^{2}Tree belonging to the second condition discussed before, with Sy right to Sx and Sy left to Sx respectively. This could lead to a longer path while using backup routes. As discussed before, under C4 and C5, F^{2}Tree has paths of more than one extra hop during the fast rerouting period. This leads to a slightly longer end-to-end delay during fast rerouting as shown in Fig. 5. However, these slightly longer paths during fast rerouting negligibly affect the upper layer performance, which is verified by the results in Fig. 4(b).

C6 belongs to the third condition discussed before in §II-D. Under this condition, F^{2}Tree performs the same as in C1, except that packets are forwarded through the left across link during fast rerouting. Furthermore, we evaluate F^{2}Tree under the extreme condition C7, which belongs to the fourth condition in §II-D. F^{2}Tree degrades to fat tree under this condition, which confirms the analysis before.

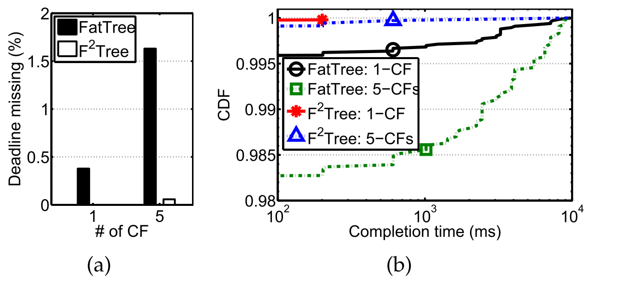


Fig. 6. Impact to partition-aggregate workload, while experiencing different number of concurrent failures (CF). (a) Deadline missing ratio (%). (b) CDF of request completion time.

B. Impact to Partition-Aggregate Workload

Experiment Setup: Next, we evaluate F^{2}Tree and fat tree under more complicated conditions. We generate failures on randomly picked links. The time between failures and the length of lasting time both obey log-normal distribution, which derives from the measurement results of operational DCNs [1]. We inject a partition-aggregate workload to our DCN emulation environment, following the convention of prior works [4], [16]. In this workload, we randomly pick some end hosts, each of which sends a small TCP single request to each of 8 other end hosts, and waits for a 2KB response from each machine. This traffic pattern often exists in front-end data centers. We measure the completion time of these requests, which means all the 8 responses are received by the sender, under 1 and 5 concurrent failure conditions respectively. We have generated more than 3000 such requests, and 1500 background flows during 600s experiment time. About 40 and 100 link failures are respectively generated in the 1 and 5 concurrent failure conditions, during the experiment time.

Results: Following the convention of the literatures [4], [24], we use deadline missing ratio as our main evaluation metric in this section. Fig. 6(a) shows the ratio of requests that miss the completion deadline (assuming to be 250ms according to [24]), under two failure conditions in fat tree and F^{2}Tree, respectively. In fat tree, there are about 0.4% and 1.6% requests, having completion time more than 250ms under 1 and 5 concurrent failure conditions. However in F^{2}Tree, no request is completed for a time longer than 250ms under 1 concurrent failure, and only about 0.06% requests are completed after the deadline under 5 concurrent failures. Compared to fat tree, F^{2}Tree reduces the ratio of deadline missing requests by 100% and about 96.25% under these failure condition, respectively.

To be clearer, Fig. 6(b) shows the CDF of requests' completion time longer than 100ms, both in fat tree and F^{2}Tree. As we can see, there are more than 0.4% requests taking longer than 100ms to complete in fat tree, under the condition with only 1 concurrent failure. Specifically, there are about 0.05% requests delayed for about 600ms due to the duration of TCP throughput collapse as analyzed before. Moreover, among all the requests in fat tree, there are even more than 0.3% requests completed after 1s, which is apparently unacceptable for upper layer applications. Digging into the trace, we find that, due to the frequent failures, large amount of LSAs are generated. This

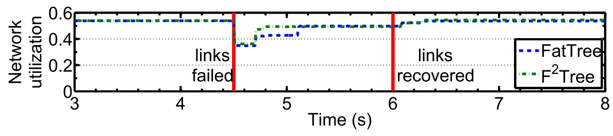


Fig. 7. Overall network utilization under all-to-all traffic.

leads to a dramatic growth of OSPF calculation timer up to about 9s, caused by the exponential backoff scheme [14] to adjust the hold time in OSPF. Thus, some requests are even delayed for such a long time by these large timers in fat tree. In contrast, due to the path redundancy and local rerouting in F^{2}Tree, packets can be forwarded through immediate backup links without waiting for control plane communication and calculation under only 1 concurrent failure, as analyzed before. As a result, there are only about 0.04% requests completed for about 200ms in F^{2}Tree, which are delayed by the failure detection time as stated before, under 1 concurrent failure.

As for a tougher case in which 5 failures occur concurrently, the ratio of requests with completion time of more than 200ms increases both in fat tree and F^{2}Tree. However, F^{2}Tree still performs much better than fat tree, by reducing the ratio of those requests by 93.5%. Because of the large number of concurrent failures, the 4th failure condition discussed in §II-D has occurred in our experiment, which degrades the F^{2}Tree's performance down to fat tree, and leads to a 9s completion time for some requests waiting for large OSPF calculation timers. However, there are only about 0.03% requests taking that long time to be completed.

C. All-to-All Traffic

Next we evaluate the overall network utilization under all-to-all traffic, before failure happens, during failures, and after recovered from failures, respectively.

Experiment Setup: Starting from time 0s, we generate an all-to-all permutation traffic [25] with each end-host sending an unlimited TCP flow to another end-host (in different Agg pod) with maximum rate. At time 4.5s, we manually tear down 1/8 links between the Core layer switches and the Agg layer switches. After 1.5s at time 6s, we recover all these failed links.

Results: Fig. 7 shows the overall network utilization. Each point in the figure are the average of 20 runs. Before failure happens, F^{2}Tree and fat tree have almost the same network utilization of ∼0.54. Due to hash collision in ECMP, the overall utilization cannot reach 1, which has also been shown in many previous studies such as [17], [26].

When failure happens at time 4.5s, flows traversing these failed paths have multiple consecutive packets dropped, which leads to a 200ms timeout to them. It makes the overall network utilization in both topologies drop to ∼0.35 from time 4.5s to 4.7s. Then at time 4.7s, all flows going through those failed links in the upward direction (denoted as upward flows) have recovered their throughput, thanks to the quick routing recovery for upward link failures (§II-A). In F^{2}Tree, for flows going in the downward direction through those failed links (denoted as downward flows), their transmission is also recovered thanks to the fast rerouting scheme. This leads to the

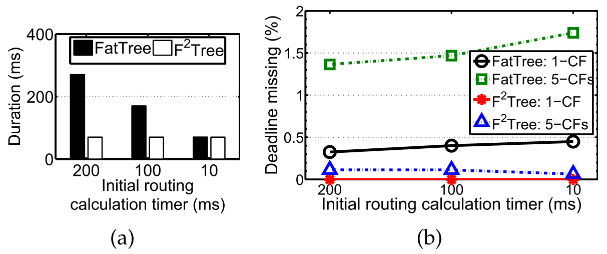


Fig. 8. Different routing calculation timer settings: Duration of connectivity loss caused by single link failure, and the impact to partition-aggregate workload when experiencing different number of concurrent failures (CF). (a) Duration of connectivity loss. (b) Deadline missing ratio (%).

overall network utilization going back to ∼0.49 at time 4.7s. However, in fat tree, these downward flows' transmission still cannot be recovered since the downward routing recovery is not finished yet. As such, at time 4.7s, the overall utilization only goes up to ∼0.43. These downward flows encounter another 400ms timeout. Then at time 5.1s, these downward flows' transmission get recovered, and the overall utilization goes back to ∼0.49.

At time 6s, all failed links are physically recovered. Then at time 6.07s, upward flows can utilize those recovered links, after Agg switches detect the link recovery and update their forwarding tables. At time 6.27s, the whole routing plane converges to the link recovery, and the overall network utilization is recovered in both topologies.

D. Different Routing Calculation Timer Settings

Now we evaluate the impact of different routing calculation timer settings on the routing recovery performance of both fat tree and F^{2}Tree.

Experiment Setup: Besides the default 200ms used in experiments before, here we set smaller initial OSPF calculation timers from a moderate 100ms to a more aggressive 10ms. We rerun the experiments in §V-A (condition 1) and §V-B to see how different timers affect the duration of connectivity loss caused by single link failure, and the deadline-missing ratio in partition-aggregate workload when experiencing random and complex failures.

Results: Fig. 8(a) shows that, for single link failure, a smaller timer can help fat tree to achieve better routing recovery time. It is because that, with a smaller timer, the control plane will sooner calculate the new route to avoid the failed link. The results show that using an aggressive 10ms timer can make fat tree (which has no fast rerouting schemes) have a very similar routing recovery time as F^{2}Tree (which fast-reroutes with no need to wait control plane calculation).

However, a smaller timer may also make the routing unstable and more difficult to converge under complex and relatively frequent failure scenarios, thus even lengthens the failure recovery time. Same as before (§V-B), we generate random failure conditions derived from real measurements [1] and a practical partition-aggregate workload, to see whether a smaller timer can handle such more realistic scenarios. Results in Fig. 8(b) show that the deadline-missing requests in fat tree grows as the initial routing calculation timer decreases.

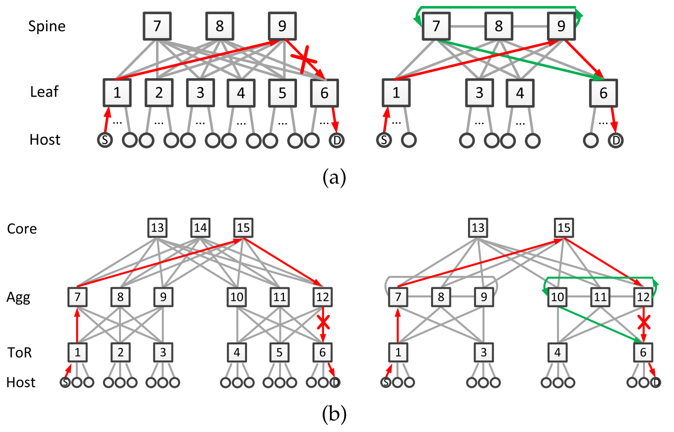


Fig. 9. F^{2}Tree for other multi-rooted tree topologies. (a) F^{2}Tree for Leaf-Spine topology. (b) F^{2}Tree for VL2 topology.

It indicates that simply decreasing the timer can not adapt to complex failure conditions in production DCNs. On the contrary, F^{2}Tree keeps behaving well under various timer settings, benefited from the fast rerouting scheme before the control plane communication and calculation.

VI. F^{2}Tree Extension

In this section, we introduce how F^{2}Tree can be adapted in other existing DCN environments.

A. Other Multi-Rooted Tree Topologies

Besides fat tree, several other multi-rooted tree topologies are also used in some existing production DCNs, such as VL2 [10] topology and two-layer Leaf-Spine [17] topology. In addition to standard fat tree, F^{2}Tree's scheme (rewiring links and adding backup routes) is also applicable to other multi-rooted tree topologies, helping to reduce failure recovery time.

1) F^{2}Tree for Other Multi-Rooted Tree Topologies: We use Fig. 9 as an example to briefly illustrate how to rewire these topologies according to F^{2}Tree's scheme, thus to add path redundancy and reroute locally with proper switch configurations.

F^2*Tree for Leaf-Spine:* Actually, Leaf-Spine [17] is a special instance of fat tree topology which contains only two layers, commonly used for small-scale or middle-scale DCNs. The left part of Fig 9(a) shows an example of a Leaf-Spine topology. Like fat tree, Leaf-Spine also lacks immediate backup link for downward links. As the example in Fig 9(a) shows, if link $S9$-$S6$ fails, original Leaf-Spine DCN needs to propagate failure message and wait control plane calculation until $S1$ finds a new path (e.g. $S1$-$S8$-$S6$) to route around the failure. As the right part of Fig 9(a) shows, it is easy to apply F^{2}Tree to the Spine layer in Leaf-Spine, using the same way as that in fat tree. Therefore, F^{2}Tree for Leaf-Spine can locally reroute quickly after $S9$ detects the failure, which costs far less time.

F^2*Tree for VL2:* VL2 [10] is another kind of well-known multi-rooted DCN topology, which has a denser interconnection than fat tree and improves the ability of fault tolerance.

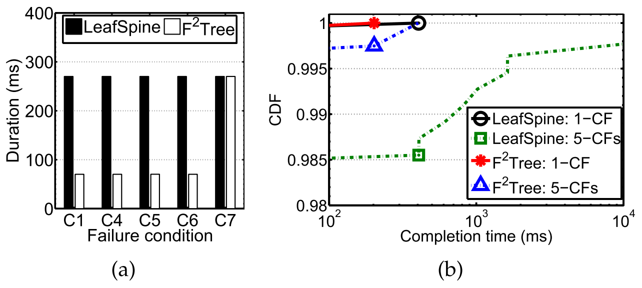


Fig. 10. F^{2}Tree for Leaf-Spine DCN: Duration of connectivity loss upon different failure conditions *and* the CDF of request completion time when experiencing different number of concurrent failures (CF). (a) Duration of connectivity loss. (b) CDF of request completion time.

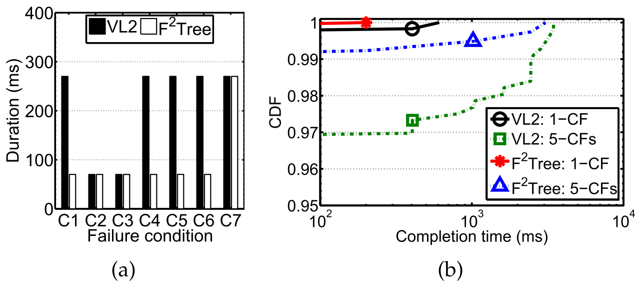


Fig. 11. F^{2}Tree for VL2 DCN: Duration of connectivity loss upon different failure conditions *and* the CDF of request completion time when experiencing different number of concurrent failures (CF). (a) Duration of connectivity loss. (b) CDF of request completion time.

In VL2, the core switches and aggregation switches are interconnected in a full-mesh. Thus for downward link between core and aggregation layer, there are immediate backup links that can be used for local rerouting. However, downward links between aggregation and ToR switches still lack redundancy, thus an aggregation switch has to wait for control plane communication and calculation if any of these links fails. The left part of Fig. 9(b) shows an example of a VL2 topology. If link $S12$-$S6$ fails, VL2 DCN has to wait control plane communication and calculation, before the connection from host S to D is recovered. Therefore, to further improve VL2's fault tolerance, we can apply F^{2}Tree scheme to the aggregation layer in VL2, as the right part of Fig. 9(b) shows. As such, $S12$ can locally reroute using the across links once $S12 - S6$ fails, before the control plane reacts.

2) Evaluation Results: We evaluate the performance of F^{2}Tree for the Leaf-Spine and VL2 DCNs, using the same emulation environment as before (§V). Similarly, we use micro benchmarks (§V-A) to examine Leaf-Spine, VL2 and F^{2}Tree's performance under different failure conditions, and use macro benchmarks (§V-B) to show how F^{2}Tree can benefit the performance of upper layer application based on the acceleration of failure recovery. We build the topology using homogeneous 8-port switches as before. All the experiment settings are the same as in §V.

Fig. 10 and 11 show the results of using F^{2}Tree for Leaf-Spine and VL2 DCN respectively. For Leaf-Spine, a two-layer DCN topology, F^{2}Tree can protect the downward link from Spine switches to ToR switches. Thus, as shown in Fig. 10(a), F^{2}Tree can shorten the duration of connectivity loss from about 270ms to about 70ms, under failure condition C1, C4, C5 and C6, except for the extreme case C7. Note that C2 and C3 are failure conditions related to links between core and aggregation switches in three-layer topology, thus are not applicable to Leaf-Spine. Fig. 10(b) shows the CDF of request completion time of Leaf-Spine DCN and F^{2}Tree for Leaf-Spine, using the same application and evaluation settings as §V-B. In Leaf-Spine, there are about 0.17% and 2.36% requests that missed the deadline under one and five concurrent failure conditions, respectively. Benefiting from acceleration of failure recovery, F^{2}Tree can reduce the deadline missing ratio down to 0% and ~0.25% under these two failure conditions, respectively.

As for VL2, F^{2}Tree offers protection to the downward links from aggregation switches to ToR switches. Fig. 11(a) shows the the duration of connectivity loss in the micro benchmarks under different failure conditions. In original VL2, there is no immediate backup links for downward links between aggregation and ToR layer. Thus for failure conditions C1, C4, C5 and C6, which are related to the failure of these links, F^{2}Tree can shorten the recovery time from about 270ms to about 70ms. For link failures between core and aggregation layer, VL2 has already offered a full-mesh connection with many immediate backup links, which makes it perform well under conditions C2 and C3. F^{2}Tree makes no modification between core and aggregation layer, and performs the same as VL2 under these two conditions. Note that in condition C3, where both the link between the aggregation and the ToR switch *and* the link between the core and the aggregation switch fail, VL2 can quickly rerouted around the failure. It is because that once the the link between the core and the aggregation switch fails, the core switch will locally choose another aggregation switch below to forward packets, which also helps it rerouted around the failed link below between the aggregation and ToR switch. As shown in Fig. 11(b), the acceleration of failure recovery brings a 100% and ~76% reduction of deadline missing requests in F^{2}Tree for VL2, under conditions of 1 concurrent failure and 5 concurrent failures, respectively.

B. Centralized Routing DCNs

There are also several existing data centers [27], [28] using centralized routing DCNs. For example, recently, Google has published the routing architecture of its data centers [28], which uses logically centralized state and control. In this section, we show that F^{2}Tree can also accelerate failure recovery for these centralized routing DCNs.

1) F^{2}Tree for Centralized Routing DCNs: Originally, when a failure happens in the centralized routing DCNs, if there is no local backup routes, the detecting switch will pass the failure message up to the controller. Then the controller calculates new routing paths using global link message, and delivers the new routing tables to affected switches. Besides the time for centralized calculation, in the worst case, original centralized routing DCNs require one message from the switch detecting failure to the controller, and one message from the controller to *each* affected switches. As the DCN scales larger, the communication and processing will take quite a long

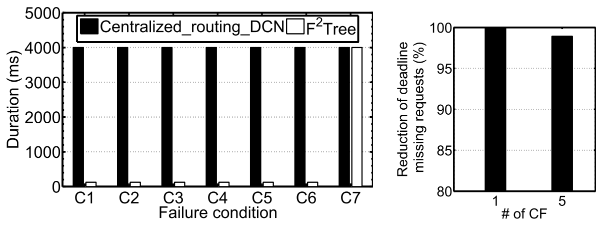


Fig. 12. F^2Tree for centralized routing DCN: Duration of connectivity loss upon different failure conditions, *and* the reduction of deadline missing requests using F^2Tree compared to original centralized routing DCN, when experiencing different number of concurrent failures (CF). (a) Duration of connectivity loss. (b) Reduction of deadline missing requests.

time, causing a substantial duration of connectivity loss upon failures. For example, Google [28] reports that for failures of downward links without local backup routes, it takes about four seconds for routing connectivity recovery, although the physical connection exists.

Again, F^2Tree's scheme can be applied to centralized routing DCNs, by rewiring two links in each pod of the aggregation or core layer. By adding backup paths in the corresponding switches' routing table, switches could locally reroute around failures, before uploading and waiting for the new routes calculated by the controller. Therefore, F^2Tree can also significantly reduce the time for failure recovery in centralized routing DCNs, especially in a large scale network.

2) Evaluation Results: We also evaluate the performance of F^2Tree for centralized DCNs, using the same micro benchmarks (§V-A) and macro benchmarks (§V-B) as before. Note that in this experiment, we use the global routing module in NS3 [22] to simulate the centralized routing. We take a centralized routing DCN with fat tree topology as the baseline, and use F^2Tree on it. We set the local recovery time of failures where local switch has backup routes and paths (*e.g.* one of the upward links fails) to be 125ms, a value as reported in [28]. For the recovery time of failures which needs communications and calculation of the controller, we set it to be 4s [28]. All the rest experiment settings are the same as in §V. The topology is built using homogeneous 8-port switches as before.

Fig. 12(a) and 12(b) respectively show the results of recovery time under different failure conditions, and how the upper layer applications benefit from the acceleration of failure recovery using F^2Tree for the centralized routing DCN. F^2Tree shortens the recovery time from 4s to 125ms under almost all the failure conditions, except the extreme case C7 which we discussed before. Furthermore, for the up-layer applications, Fig. 12(b) shows that F^2Tree can reduce the deadline missing requests by 100% and $\sim$ 99% compared with the original centralized routing DCN, under conditions of 1 and 5 concurrent failure(s), respectively.

VII. Related Work

DCN Fast Failure Recovery: Aspen Tree [3] requires a change to the fat tree fabric and a new routing protocol. It shares the same intuition as F^2Tree to reduce the failure recovery time by increasing the path redundancy in current fat tree DCNs, but in a different way. Switches in the upper layer connect to each pod below with more than one link in Aspen Tree. Through a new failure reaction and notification protocol combining with the modified topology, Aspen Tree shortens the routing convergence time compared to fat tree. However, except introducing a new protocol, the modification to original topology is at the expense of more than half of the network bisection as fat tree (§III). Moreover, Aspen Tree only has immediate backup links for downward links in the fault-tolerant layer, which may still incur a substantial time for recovery from downward failures at other layers.

DDC [4] requires both a new routing protocol and data plane forwarding hardware. Before control plane converges after failures, DDC will reverse a packet's forwarding direction once it encounters failure. Packets will be bounced in the network, according to an ingenious algorithm, and finally get to its destination. However, packet bouncing in DDC could greatly inflate the paths and may cause congestion due to the lack of global control. Furthermore, the characteristic of fat tree topology may cause the packet bouncing to its sender switch to find an alternate path under certain failures, which incurs a longer delay and potential congestion.

Unlike the works mentioned before, F10 [16] presents a whole new fault-tolerant DCN solution. F10 combines new topology, failover and load balancing protocols, and failure detector to provide a completely novel solution for fault-tolerant centralized routing DCNs, thus not applicable to existing production DCNs.

Existing Fast Rerouting Schemes: There are also other existing fast rerouting (FRR) schemes designed for IP network, such as MPLS Fast Reroute (MPLS FRR) [29]. It is commonly used to protect an individual link by providing a backup path, which can route traffic around failure. There are two major differences between F^2Tree and MPLS FRR. First, MPLS FRR itself does not offer additional path redundancy, it just speeds up the process of switching to the backup path. Second, the backup path in MPLS FRR is manually configured based on additional lower layer information and non-trivial algorithms such as shared risk groups. In multi-rooted tree DCNs that lack redundancy for downward links, it is inherently hard to provide planned backup paths for all complicated failure situations, which needs extremely careful pre-configuration.

Facebook's DCN Topology: First revealed in a 2-page paper [30] and recently revisited in [31], Facebook's DCN also constructs a ring in each pod of aggregation switches and core switches. The use of the ring is similar to that of topology of F^2Tree. However, [30], [31] only briefly mention the ring while presenting Facebook overall DCN topology, and the ring's primary usage in Facebook's DCN is to provide faster and more direct paths for large amount of traffic between core switches. It is unclear whether the ring is designed for failure recovery in Facebook and how/whether it works for failure recovery studied in our paper. In contrast, our paper has a few important differences from the Facebook papers. First, our paper presents an thorough study on the cause of DCN slow failure recovery problem. Second, we propose a comprehensive solution F^2Tree which includes not only the ring structure, but also fast rerouting configurations. In other words, without fast rerouting configuration, the ring structure alone does not help improve failure recovery time. Third, we show how to use F^2Tree for various other multi-rooted tree topologies as well as centralized routing DCNs.

Randomly Wired DCN Topology: Recent works such as Jellyfish [32] use randomly wired topology between switches which can significantly increase the overall capacity and scalability of DCN. Such randomly wired topology could also potentially help to accelerate routing recovery from failures. To compare the routing recovery efficiency between F^2Tree and such methods, we implement Jellyfish[7] in the emulation and use the same macro benchmark as in §V-B to evaluate their performance. Results show that Jellyfish performs slightly better than F^2Tree. Specifically, for 1 concurrent failure condition, both F^2Tree and Jellyfish encounter *no* deadline-missing requests. But for 5 concurrent failure condition, the fraction of deadline-missing requests is $\sim$0.11% in F^2Tree but only $\sim$0.09% in Jellyfish. However, there are several key difficulties to deploy random topology in current production DCNs: 1) Turning existing fat tree into random topology requires significant wiring effort, 2) The physical layout of switches and servers also needs a lot of changes, 3) It is difficult and still remains unknown that how to keep live traffic unaffected during rewiring fat tree topology into randomly wired.

[7]We use ECMP routing for Jellyfish instead of k-shortest-path, which requires significant changes to current DCN routing protocols.

VIII. Conclusion

In this paper, we present a readily deployable fault-tolerant solution called F^2Tree for existing production DCNs. Through only rewiring two links and changing configurations, F^2Tree increases the downward link redundancy and achieves local fast rerouting for downward link failures, greatly accelerating the failure recovery and improving upper layer application's performance. F^2Tree is one important step towards improving the fault-tolerance of existing production DCNs. We believe that the principle behind F^2Tree, *increasing path redundancy and rerouting locally*, is one promising direction for accelerating failure recovery for routing in DCNs.

Acknowledgment

The authors would like to thank J. Yang and X. Nie for their help in the experiments. Also, they greatly thank K. Tan for offering useful DCN background knowledge, and K. Chen for the suggestions to improve this paper. In addition, they thank K. Sui, D. Deng, and J. Liao for their proofreading on this paper.

References

[1] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. SIGCOMM*, Aug. 2011, pp. 350–361.

[2] R. Potharaju and N. Jain, "When the network crumbles: An empirical study of cloud network failures and their impact on services," in *Proc. SOCC*, Oct. 2013, Art. no. 15.

[3] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, "Aspen trees: Balancing data center fault tolerance, scalability and cost," in *Proc. CoNEXT*, Dec. 2013, pp. 85–96.

[4] J. Liu *et al.*, "Ensuring connectivity via data plane mechanisms," in *Proc. NSDI*, 2013, pp. 113–126.

[5] (2010). *Cisco Data Center Infrastructure 2.5 Design Guide*. [Online]. Available: http://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf

[6] J. Moy, *OSPF Version 2*, document RFC 2178, Internet Engineering Task Force, California, USA, Jul. 1997.

[7] Y. Rekhter, T. Li, and S. Hares, *A Border Gateway Protocol 4 (BGP-4)*, document RFC 1771, Internet Engineering Task Force (IETF), California, USA, 2006.

[8] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *Proc. SIGCOMM*, Aug. 2012, pp. 115–126.

[9] *vSphere ESX and ESXi Info Center*, accessed on Jul. 1, 2015. [Online]. Available: http://www.vmware.com/products/esxi-and-esx/

[10] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. SIGCOMM*, Aug. 2009, pp. 51–62.

[11] *Quagga Routing Suite*, accessed on Jul. 1, 2015. [Online]. Available: http://www.nongnu.org/quagga/

[12] M. Goyal *et al.*, "Improving convergence speed and scalability in OSPF: A survey," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 2, pp. 443–463, 2nd Quart., 2012.

[13] A. Fabrikant, U. Syed, and J. Rexford, "There's something about MRAI: Timing diversity can exponentially worsen BGP convergence," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 2975–2983.

[14] *OSPF Shortest Path First Throttling*, accessed on Jul. 1, 2015. [Online]. Available: http://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fs_spftrl.html/

[15] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM*, Aug. 2008, pp. 63–74.

[16] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A fault-tolerant engineered network," in *Proc. NSDI*, 2013, pp. 399–412.

[17] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. SIGCOMM*, Aug. 2014, pp. 503–514.

[18] C. Hopps, *Analysis of an Equal-Cost Multi-Path Algorithm*, document RFC 2992, Internet Engineering Task Force (IETF), California, USA, Nov. 2000.

[19] *Complex Topology and Wiring Validation in Data Centers*, accessed on Jul. 1, 2015. [Online]. Available: https://cumulusnetworks.com/blog/complex-topology-and-wiring-validation-in-data-centers/

[20] *iPerf—The Network Bandwidth Measurement Tool*, accessed on Jul. 1, 2015. [Online]. Available: https://iperf.fr/

[21] D. Katz and D. Ward, *Bidirectional Forwarding Detection (BFD)*, document RFC 5880, Internet Engineering Task Force, California, USA, Jun. 2010.

[22] *ns-3*, accessed on Jul. 1, 2015. [Online]. AVailable: http://www.nsnam.org/

[23] *Direct Code Execution*, accessed on Jul. 1, 2015. [Online]. Available: http://www.nsnam.org/overview/projects/direct-code-execution/

[24] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. SIGCOMM*, Aug. 2011, pp. 50–61.

[25] C. Raiciu *et al.*, "Improving datacenter performance and robustness with multipath TCP," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 266–277.

[26] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. 7th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, Apr. 2010, p. 19.

[27] R. N. Mysore *et al.*, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. SIGCOMM*, Aug. 2009, pp. 39–50.

[28] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 183–197.

[29] P. Pan, G. Swallow, and A. Atlas, *Fast Reroute Extensions to RSVP-TE for LSP Tunnels*, document RFC 4090, Internet Engineering Task Force (IETF), California, USA, 2005.

[30] N. Farrington and A. Andreyev, "Facebook's data center network architecture," in *Proc. IEEE Opt. Interconnects Conf.*, May 2013, pp. 49–50.

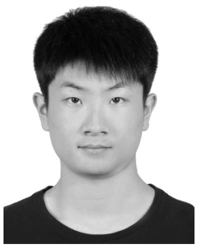
[31] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 123–137.

[32] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, Berkeley, CA, USA, 2012, p. 17.

[33] G. Chen, Y. Zhao, D. Pei, and D. Li, "Rewiring 2 links is enough: Accelerating failure recovery in production data center networks," in *Proc. 35th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2015, pp. 569–578.

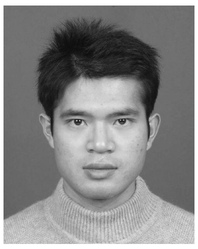
Guo Chen received the B.S. degree from Wuhan University in 2011 and the Ph.D. degree from Tsinghua University in 2016. He is currently an Associate Researcher with Microsoft Research Asia, Beijing, China. His current research interests focus on data center networking.

Youjian Zhao received the B.S. degree from Tsinghua University in 1991, the M.S. degree from the Shenyang Institute of Computing Technology, Chinese Academy of Sciences, in 1995, and the Ph.D. degree in computer science from Northeastern University, China, in 1999. He is currently a Professor with the CS Department, Tsinghua University. His research mainly focuses on high-speed Internet architecture, switching and routing, and high-speed network equipment.



Hailiang Xu is currently pursuing the bachelor's degree with the Department of Computer Science, Beijing University of Posts and Telecommunications. His current research interests are data center networks and wireless networks.

Dan Pei received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 1997 and 2000, respectively, and the Ph.D. degree from the University of California at Los Angeles, Los Angeles, CA, USA, in 2005. He is currently an Associate Professor with Tsinghua University. His current research interests are management and improvement of the performance and security of the networked services, through big data analytics with feedback loop. Right now, he is focusing on improving the mobile Internet performance over Wi-Fi networks and data center networks.



Dan Li received the Ph.D. degree in computer science from Tsinghua University in 2007. He is currently an Associate Professor with the Computer Science Department, Tsinghua University. His research interest includes future Internet architecture and data center networking.