Rewiring 2 Links is Enough: Accelerating Failure Recovery in Production Data Center Networks

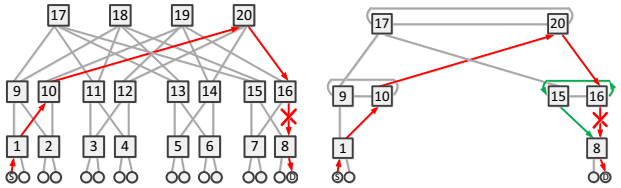
Guo Chen, Youjian Zhao, Dan Pei* and Dan Li
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
Email: {chen-g11@mails., zhaoyoujian@, peidan@, tolidan@}tsinghua.edu.cn

***Abstract*—Failures are not uncommon in production data center networks (DCNs) nowadays, and it takes long time for the network to recover from a failure and find new forwarding paths, significantly impacting realtime and interactive applications at the upper layer. The slow failure recovery is due to two primary reasons. First, there lacks immediate backup paths for downward links in DCN with multi-rooted tree topology. Second, distributed routing protocols in DCN take time to converge after failures. In this paper, we present a fault-tolerant DCN solution, called F^{2}Tree, that can significantly improve the failure recovery time in current DCNs, only through a small amount of link rewiring and switch configuration changes. Because F^{2}Tree does not change any existing software or hardware, it is readily deployed in production DCNs, where other existing proposals fail to achieve. Through testbed and emulation experiments, we show that F^{2}Tree can greatly reduce the time of failure recovery by 78%. Our experimental results also show that, for partition-aggregate applications (popular in DCN) under various failure conditions, F^{2}Tree reduces the ratio of deadline-missing requests by more than 96% compared to current DCNs.**

I. Introduction

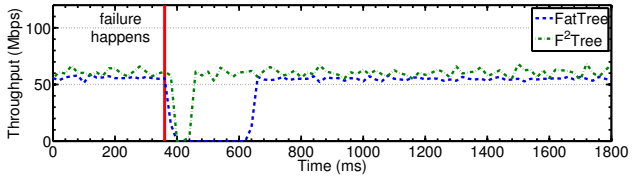
Data Center Network (DCN), which is the key infrastructure of almost all the Internet services we rely on today, scales larger and larger to meet the increasingly stringent demands of users and service providers. However, as the number of network equipments (*e.g.*, switches, links) grows, network failures[1] can happen frequently [1, 2]. Furthermore, recent studies [3, 4] have shown that failure recovery takes long 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.

We illustrate the slow 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 the bottom right node D, along the path (S-$S1$-$S10$-$S20$-$S16$-$S8$-D). Then at time 380ms, the link between $S16$ and $S8$ is manually shut down. It takes $S16$'s failure detection mechanism about 60ms to detect the interface failure. Then the OSPF LSA messages take very little time to get propagated from $S16$ 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

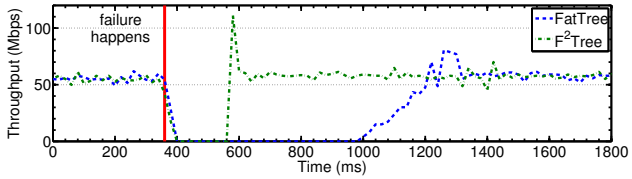


(a) Fat tree (b) F^{2}Tree

Fig. 1. 4-port, 3-layer fat tree and F^{2}Tree: One downward link fails.



(a) Influence to UDP throughput (duration of connectivity loss)



(b) Influence to TCP throughput

Fig. 2. Failure of one downward link between ToR and aggregation switch in the testbed: Influence to UDP and TCP flows' throughput.

[*] Dan Pei is the corresponding author.

[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.

1063-6927/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCS.2015.64

global link states. It then knows that the current path has failed and chooses a new path ($S1$-$S9$-$S17$-$S15$-$S8$), 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 $S16$-$S8$, resulting a 272ms of connectivity loss from S to D, as shown clearly in 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 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 $S16$ to $S8$) lacks immediate backup paths. As such, the switch that detects the failure ($S16$) 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.

In fact, above fundamental reasons for the slow failure recovery in this example are no different from the slow convergence 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, 15]. 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 [16] 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, 15] fail to achieve.

[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 §V.

Fig. 1(b) shows part of the topology according to our approach, by only *rewiring two links* for each aggregation and core switch. For example, for $S16$ ($S15$), we first remove two links: one of $S16$ ($S15$)'s upwards link $S16$-$S19$ ($S15$-$S18$) and one of its downward link $S16$-$S7$ ($S15$-$S7$). Then we use the newly available ports to add two links between $S16$ to $S15$ to form a ring. As a result, when the link between $S16$ and $S8$ fails, the number of immediate backup links (details in §II) that can be used downward by $S16$ to reach $S8$ 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 $S16$, so that $S16$ can quickly switch to one of these backup links (*e.g.*, $S16$-$S15$) after the failure of link $S16$-$S8$ is detected, without waiting for OSPF to converge. Therefore, the packets destined to D continue to be forwarded to $S16$, which successfully forwards the packets along the path $S16$-$S15$-$S8$, greatly shortening the failure recovery time. Of course, this path redundancy is achieved at the price of less supported nodes, and the seemingly decrease of upward link redundancy, which we discuss in detail later (§II-D).

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

- We propose F^2Tree, a readily deployable approach to accelerate failure recovery time in existing production DCNs without any protocol or software changes. F^2Tree keeps the merits of fat tree such as no oversubscription and rich path diversity, only trading a little bisection bandwidth for path redundancy (see §II-D).
- F^2Tree significantly increases the path redundancy for downward links in current multi-rooted fat tree DCN, only through rewiring two links for each aggregation and core switch. By adding several routing configurations for aggregation and core switches, F^2Tree offers immediate backup paths against downward link failures. As a result, such a switch can quickly locally reroute around the failure of its downward link.
- Through testbed and emulation experiments, we have shown that F^2Tree can greatly reduce the time of failure recovery by 78% compared to current fat tree. As a result, F^2Tree reduces the ratio of deadline-missing requests of partition-aggregate applications by more than 96%, under different failure conditions, compared to original fat tree.

We believe that the principle behind F^2Tree, *increasing path redundancy and rerouting locally*, is one promising direction and could help with designing fault-tolerant DCNs.

II. F^2Tree Design

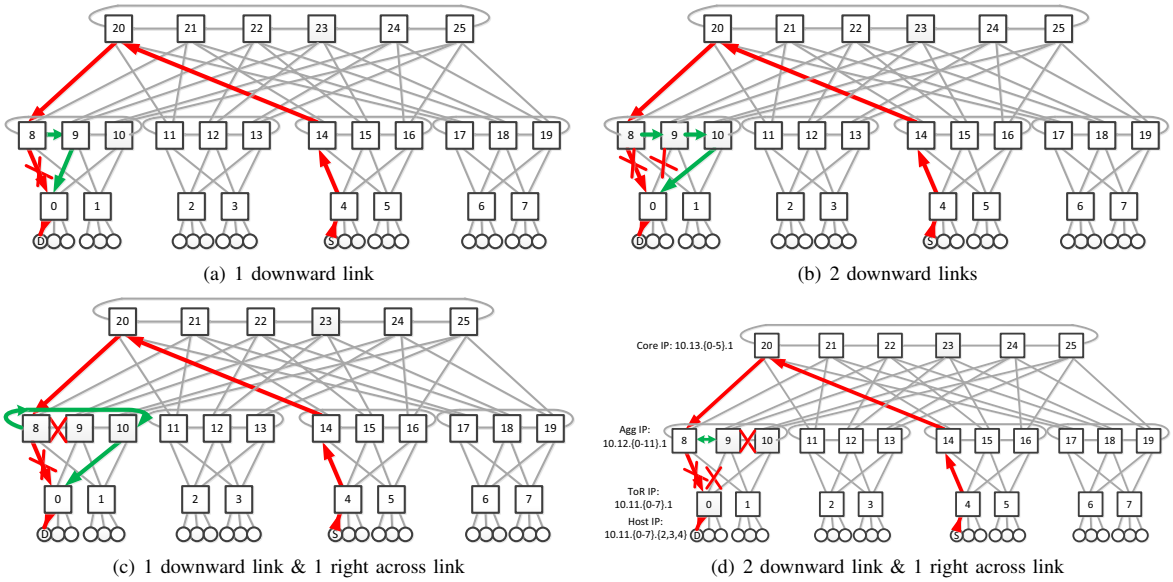
In this section, we first introduce the intuition behind the design of F^2Tree. Then, we present the F^2Tree solution in detail. Next, we discuss that how F^2Tree handles different failure conditions. Finally, we discuss the tradeoffs we made in F^2Tree to accelerate failure recovery.

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 CONNECTING WITH ONE NODE.)

	Switches consumed	Nodes supported	Modify routing protocol	Modify data plane
Fat tree	$\frac{5}{4}N^2$	$\frac{N^3}{4}$	n/a	n/a
VL2 [10]	$\frac{5}{2}N$	$\frac{N^2}{2}$	n/a	n/a
F^{2}Tree	$\frac{5}{4}N^2 - \frac{7}{2}N + 2$	$\frac{N^3}{4} - N^2 + N$	no	no
Aspen tree [3] $\langle f, 0\rangle$*	$\frac{5}{4(f+1)}N^2$	$\frac{N^3}{4(f+1)}$	yes	no
F10 [15]	$\frac{5}{4}N^2$	$\frac{N^3}{4}$	yes	yes
DDC [4]	n/a	n/a	yes	yes

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



(a) 1 downward link (b) 2 downward links

(c) 1 downward link & 1 right across link (d) 2 downward link & 1 right across link

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. Intuition of F^2Tree

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[4] 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.

[4]Each link in DCN is assumed to have the same cost for simplification.

B. Topology & Working Schemes

Topology: In original fat tree as shown in Fig. 1(a), there is no link between switches in the same pod[5]. 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. To maximize bisection bandwidth, the original fat tree uses all ports of a switch to interconnect with switches above and below. Each switch within a same pod in fat tree has no

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

links connected with each other. In contrast, 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 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 between across neighbors as *across links*. 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 later).

Fast rerouting: We define *fast rerouting* as the process of routing packets around failures with only local failure detection information and without control plane communication and calculation. F^{2}Tree accelerates the failure recovery by using the newly added across links as immediate backup links and by installing static routes directly into forwarding table. Therefore switches that detect the failures can get rid of the time waiting for communication and recalculation in control plane and FIB updating, greatly reducing failure recovery time.

Next, we introduce our fast rerouting scheme more specifically under typical DCN address assignment. According to our interview with operators of one top global cloud provider, usually switches in DCN bundle all ports into one layer 3 interface using one IP address. Hosts in each rack are assigned addresses within the same subnet of each ToR switch they connect to. Each ToR will redistribute (propagate) the subnet address containing hosts below into OSPF. Fig 3(d) shows an example of such DCN address assignment. For example, $S8$ has an IP address of 10.12.0.1, $S0$ has an IP address of 10.11.0.1, and the subnet containing all the hosts below $S0$ is 10.11.0/24.

To acheive 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), and one static route to the prefix just covering the DCN prefix. Take Fig. 3 as an example, this DCN prefix is 10.11.0.0/16, and the covering prefix is 10.10.0.0/15. These two backup routes are not redistributed to OSPF, thus they are used only locally at each switch. Take $S8$ as an example, one static route's next hop is the rightward across neighbor $S9$, and the other's next hop is the leftward across neighbor $S10$. 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.

Note that the backup routes for all the addresses have shorter length of prefixes than the route introduced by the routing protocol, thus the backup ones have also been written in the forwarding information base (FIB) before failure happens. This design totally avoids the FIB update time, which would normally take a substantial time for large networks [19].

These static routes 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. For example in Table II, normally, $S8$ only has the first two routes calculated by the routing protocol for host D and S, which pass through its downward and upward links respectively. The two added backup routes (3 and 4) will not be stored in the routing entry for D or S by original OSPF and ECMP. In F^{2}Tree, 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 forwarding table, and finds it can still reach D via 10.11.0.0/16, and will directly forward the packet via next $S9$, only incurring the normal FIB lookup time. And if $S9$ is also detected as unreachable, the 4th route which has a shorter prefix will be chosen and $S8$ will forward packets through $S10$.

TABLE II
PART OF THE ROUTING TABLE OF $S8$ IN FIG 3. THE LAST TWO LINES SHOW AN EXAMPLE OF CONFIGURATIONS TO USE IMMEDIATE BACKUP LINKS FOR DOWNWARD AND UPWARD LINKS IN F^{2}TREE. THE DCN ADDRESSES ARE ASSIGNED AS FIG 3(D) SHOWS.

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)

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, as shown 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. Handling Failures

After introducing the topology and working schemes of F^{2}Tree, we now comprehensively 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 §IV), because that the fast rerouting scheme can work locally at a switch for each packet that arrives at this switch. Therefore, due to the space limitation, 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.

Actually, the above four conditions have summarized all the failure conditions above the same layer related to the downward forwarding path (except the failures of both two across links of $S8$, which F^{2}Tree obviously degrades to fat tree). Failure of a whole switch is also included, by modeling as the failure of all of its links. For instance in Fig. 3, the condition that $S9$ fails belongs to the 3rd condition mentioned before. From the above discussion, F^{2}Tree is shown to be able to greatly reduce the time for failure recovery with fast rerouting, under all the failure conditions with no more than 2 concurrent link failures. If there are 3 links concurrently failed in a certain manner in the same pod (e.g. the 4th condition), F^{2}Tree will fail to fast reroute. However, we believe that this extreme situation could rarely happen in real network (our emulations later also confirm this). Moreover, if we reserve more ports (*e.g.* 4) for across links and configure them as immediate backup links, following the philosophy of F^{2}Tree, it is able to deal with this extreme condition as well.

D. Trading (negligible) Bisection Bandwidth for Downward Redundancy

F^{2}Tree's topology is only slightly different from fat tree: only two links of each aggregation and core switch are rewired to form a ring in each pod. Also, the backup routes are not used in forwarding unless failures happen. Thus, F^{2}Tree keeps all the merits of fat tree such as no oversubscription and rich path diversity.

The design of F^{2}Tree is to trade some bisection bandwidth for the increase of path redundancy. However, the reduction of bisection bandwidth can be negligible for a large fat tree topology. To understand the cost more clearly, in Table I, we compare the number of nodes supported (which reflects the aggregate throughput of a non-oversubscribed network) in different multi-rooted tree topologies, assuming using homogeneous switches with N ports. 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.

III. TESTBED EXPERIMENT

Prototype implementation: As introduced in Section I, we rewire the 4-port, 3-layer fat tree into an F^{2}Tree prototype, as Fig. 1(b) shows. We have configured backup routes in Quagga [11] for each aggregation ($S9$, $S10$, $S15$, $S16$) and core ($S17$, $S20$) switch, to use the two across links as immediate backup links for each downward and upward link. This prototype is a small but full implementation of F^{2}Tree solution.

Experiment setup: We build a UDP and a TCP flow respectively, from the leftmost node S to the rightmost node

D in F^2Tree prototype. 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 [20] (about 60ms), thus approximating the real DCN. The same experiments are also done in the fat tree prototype shown in Fig. 1(a). Both UDP and TCP flows send a segment of 1448 bytes data to the destination every 100μs. All other settings are set as the default ones in Linux and Quagga.

TABLE III
RESULTS UPON FAILURE OF ONE DOWNWARD LINK BETWEEN TOR AND AGGREGATION SWITCH IN THE TESTBED.

	Duration of connectivity loss (us)	Packets lost	Duration of throughput collapse (us)
Fat tree	272847	1302	700000
F^2Tree	60619	310	220000

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^2Tree 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 about 60ms in F^2Tree, 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. We record the time of the last UDP packet arrived at the receiver before this duration, and the time of the first UDP packet just after this duration. The time difference of the arrival of these two packets reflects the *duration of connectivity loss*, with a time granularity of 100μs (the packet sending interval). The detailed comparison of the length of this duration and the number of lost packets is shown in Table III. With no need of control plane calculation and FIB update, the 60ms of connectivity loss in F^2Tree shown in the table comes from the time of failure detection, which approximates the time scale of BFD. 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. Due to a significantly shorter time of connectivity loss in F^2Tree, the number of packets lost has been reduced by 75%, compared to fat tree.

Next, we discuss how failures impact the TCP flows in F^2Tree and fat tree, respectively. From Fig. 2(b) we can see, the TCP flow in F^2Tree 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 (with time bin of 20ms), and list it in Table III as the *duration of throughput collapse*. While fat tree's duration of throughput collapse is 700ms, F^2Tree's is only 220ms. The big gap between them results from the TCP retransmission timeout (RTO), which is 200ms in initial as default. As stated before, after failure happens, there are periods time of about 60ms and 270ms in F^2Tree 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^2Tree, the retransmitted packets successfully get to the destination. However, the retransmitted packets in fat tree fail to 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. This difference between the duration of connectivity loss has greatly affected the performance of TCP flows. Setting a shorter initial RTO down to hundreds of μs could successfully reduce the duration of TCP throughput collapse both in fat tree and F^2Tree. However, it will not narrow the gap between these two methods to be less than the difference between the duration of connectivity loss shown in Table III. How to set proper initial RTO is beyond the scope of our work.

From these results in the testbed experiments, F^2Tree is shown to have a much shorter failure recovery time than fat tree. And the advantage would be larger as the network scales, since it would consume much more time for updating FIB and calculating OSPF shortest path [12]. Ideally, shortening the OSPF calculation timer may accelerate the connectivity recovery in fat tree. But a shorter timer may cause severe routing oscillation and a large amount of calculation in real DCN, because of the unstable state of the network. Thus, the initial holding timer for OSPF calculation may be set even longer to reduce the routing table calculations in the operational network [12]. Moreover, it will even grow to be as long as several seconds while the network is unstable (see §IV-B). However, these delay can be gotten rid of in F^2Tree, using fast rerouting through immediate backup links.

IV. EMULATION AND PERFORMANCE EVALUATION

In this section, we evaluate the performance of F^2Tree in the emulation environment with a larger scale. First, we study that how F^2Tree performs under different failure conditions discussed in §II-C. Then, we evaluate F^2Tree's improvement to upper layer applications using the workload derived from production DCNs. Prior fault-tolerant DCN solutions such as Aspen Tree [3] and F10 [15] all fail to be readily deployed in existing production DCNs, thus are beyond the scope of comparison with our solution.

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 [21] through the Direct Code Execution (DCE) [22] 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^2Tree topology within NS3, using switches and nodes

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

Label	Failures	Belong to which failure condition in §II-C
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

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, to make a $\sim 250\mu s$ round-trip-time (RTT) including transmission and processing delay, which approximates the RTT in real data centers [23]. A 60ms failure 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^2Tree and fat tree are left as default. We have emulated different scales of DCNs on a powerful server (2.4GHz CPU, 12 cores), and found that 8-port, 3-layer DCN is the largest scale that could be emulated in an acceptable time. Thus, we present the result of 8-port, 3-layer F^2Tree and fat tree DCN in this section as prior study [4]. However, we believe that F^2Tree will outperform fat tree even more as the DCN scales larger, because the control plane will converge slower in a larger network [12], which will not affect the time for failure recovery under most failure conditions in F^2Tree.

A. Handling Different Failure Conditions

Experiment setup: As shown in the testbed environment before, F^2Tree can gracefully handle single failure between ToR and aggregation switch. Now, we conduct experiments under different failure conditions, to evaluate F^2Tree's performance comprehensively.

In these experiments, we also 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 IV) 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-C. C1 and C2 belong to the first failure condition we discussed before, with Sx being aggregation and core switch respectively. C3 is a combination of C1 and C2, thus it also belongs to the first condition. C4 and C5 are special cases of the second failure condition, and C6 belongs to the third condition. Finally, C7 is a tough and rarely happened situation that belongs to the fourth condition in §II-C. For C1 to C5, we compare the performance of F^2Tree and fat tree. For C6 and C7, we only evaluate F^2Tree, because they are specific failure conditions in F^2Tree. All the link failures in our emulation are bidirectional, and we plan to consider more complicated failures mixed with unidirectional and bidirectional links in our future work.

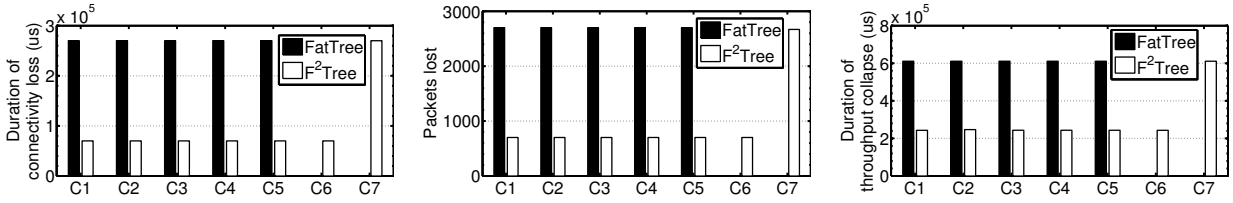
Results: Fig. 4 shows the duration of connectivity loss, the number of packets lost of a UDP flow, and duration of throughput collapse of a TCP flow both in fat tree and F^2Tree. For failure condition C1, F^2Tree reduces the duration of connectivity loss by about 78%, from 270ms to 60ms, compared to fat tree. Also, the number of lost packets is reduced by 75% in F^2Tree compared to fat tree. As for the TCP flow, there are 220ms and 610ms of throughput collapse in F^2Tree 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^2Tree 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^2Tree performs almost the same as in C1, which verifies our analysis before. The results are shown in Fig. 4. 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^2Tree 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^2Tree 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 up-layer performance, which is verified by the results in Fig. 4.

C6 belongs to the third condition discussed before in §II-C.



(a) Duration of connectivity loss (b) Packets lost (c) Duration of throughput collapse

Fig. 4. Results upon different failure conditions in an 8-port 3-layer DCN.

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-C. F^{2}Tree degrades to fat tree under this condition (see Fig. 4 & 5), which confirms the analysis before.

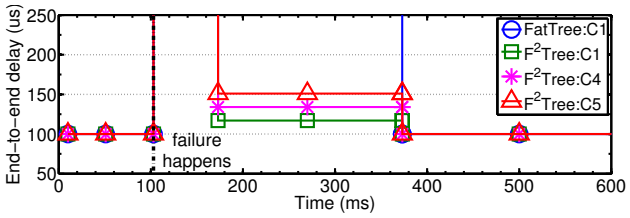
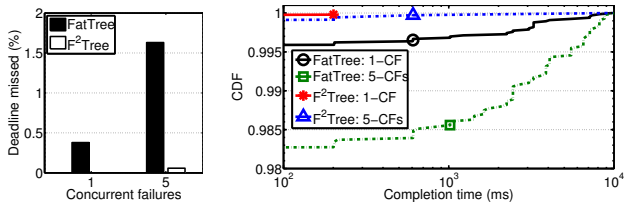


Fig. 5. Comparison of end-to-end delay during the process of failure recovery.

B. Impact to Partition-aggregate Workload

Experiment setup: Next, we evaluate F^{2}Tree and fat tree in a much more complicated condition with random failures, with an input traffic workload derived from real operational data centers [24]. The failed links are randomly picked among all the 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, 15]. 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 also inject the background traffic coexisting with the partition-aggregate requests in our emulation. The flow sizes and inter-arrival intervals of the background traffic obey the log-normal distribution derived from real operational DCNs [25]. 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, 23], 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 [23]), 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.



(a) Missed deadlines (%) (b) CDF of request completion time

Fig. 6. Impact to partition-aggregate workload, while experiencing different number of concurrent failures (CF).

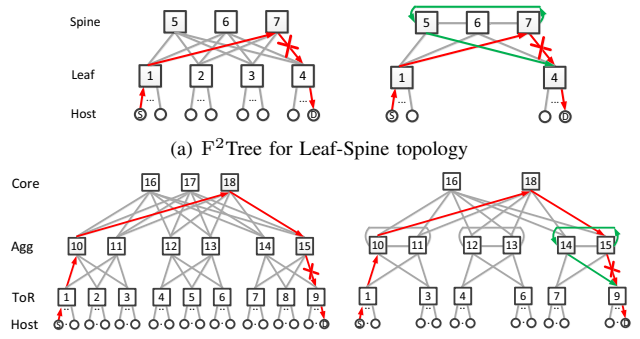
To be clearer, Fig. 6(b) shows the CDF of requests' completion time longer than 100ms, both in fat tree and F^{2}Tree. As we can see, there are more than 0.4% requests taking longer than 100ms to complete in fat tree, under the condition with only 1 concurrent failure. Specifically, there are about 0.05% 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 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-C 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, even in such extreme conditions, there are only about 0.03% requests taking that long time to be completed.

V. Discussion

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



(a) F^{2}Tree for Leaf-Spine topology

(b) F^{2}Tree for VL2 topology

Fig. 7. F^{2}Tree for other multi-rooted tree topologies

Other Multi-rooted Tree Topologies: Not only for standard fat tree, F^{2}Tree's scheme (rewiring links and adding backup routes) is also applicable to other multi-rooted tree topologies such as VL2 [10] and two-layer Leaf-Spine [17], helping to reduce failure recovery time. Fig 7 illustrates how to rewire these topologies according to F^{2}Tree's scheme to add path redundancy and reroute locally with proper switch configurations. The details of rewiring and configuring switches are similar to that in standard fat tree, which are omitted in the interest of space. 1) Although there are only 2 tiers in Leaf-Spine topology, it still lacks immediate backup link for downward links. As the example in Fig 7(a) shows, if link $S7$-$S4$ fails, original Leaf-Spine DCN needs to propagate failure message and wait control plane calculation until $S1$ finds a new path (e.g. $S1$-$S5$-$S4$) to route around the failure. In contrast, F^{2}Tree for Leaf-Spine can locally reroute quickly after $S7$ detects the failure, which costs far less time. 2) VL2 has a denser interconnection than fat tree, which improves its fault tolerance. 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. By applying F^{2}Tree scheme to VL2, we can improve its failure recovery time as shown in the example in Fig 7(b).

Other Distributed Routing Schemes: Besides OSPF, other distributed routing schemes used in DCN such as BGP [7] also suffer from slow failure recovery problem [13]. Essentially, both of them recover from failure slowly due to the same reason, which is the need of control plane communication and calculation under failures, in the lack of local rerouting paths. Thus, F^{2}Tree is also applicable to the DCN running distributed routing schemes other than OSPF.

Centralized Routing DCNs: As for multi-rooted fat tree DCNs with centralized routing schemes such as [26], F^{2}Tree can also help to improve the failure recovery. Originally, when a failure happens in these centralized routing DCNs, 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 time, causing a substantial duration of connectivity loss upon failures. Again, F^{2}Tree's scheme can be applied to centralized routing DCNs, by rewiring two links in each pod of the aggregation and 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, we believe that our proposed scheme in F^{2}Tree can also significantly reduce the time for failure recovery in centralized routing DCNs, especially in a large scale network. A more thorough study on F^{2}Tree's scheme in this environment is part of our future work.

VI. Related Work

DCN fast failure recovery: Recently, there are several related works [3, 4, 15] (see Table I) on improving network fault tolerance in the context of modern DCNs. Different from F^{2}Tree, they accelerate failure recovery by adding new networking protocols, or even new data plane forwarding hardware. Thus, they are not readily deployed in existing production DCNs.

Aspen Tree [3] requires a change to the fat tree fabric and a new routing protocol. It shares the same intuition as F^{2}Tree 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 (see Table I). 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 [15] 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) [27]. 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. In contrast, the backup route configuration in F^2Tree is very simple and is similar at each switch (see the last two lines in Table II). Thus F^2Tree is much more practical to be deployed in existing production DCN than MPLS FRR.

VII. 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 significantly increases 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 which we plan to further study in our future work.

Acknowledgment

We thank Jilei Yang and Xiaohui Nie for their help in the experiments. Also, we thank Kai Chen and the anonymous reviewers for the suggestion to improve this paper, and Kun Tan for offering us useful DCN background knowledge. Additionally, we thank Kaixin Sui and Juexing Liao for their proofreading on this paper.

This work has been supported by the National High Technology Research and Development Program of China (863 Program, No. 2013AA013302), the National Key Basic Research Program of China (973 program, No. 2013CB329105) and the State Key Program of National Natural Science of China (No. 61233007).

References

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

[2] R. Potharaju and N. Jain, "When the network crumbles: An empirical study of cloud network failures and their impact on services," in *SOCC*, 2013.

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

[4] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *NSDI*, 2013.

[5] "Cisco data center infrastructure, design guide 2.5," 2010.

[6] J. Moy, *OSPF version 2*. Internet Engineering Task Force, July 1997. RFC 2178.

[7] Y. Rekhter and T. Li, *A border gateway protocol 4 (BGP-4)*, March 1995. RFC 1771.

[8] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *SIGCOMM*, 2012.

[9] "vsphere esx and esxi info center." http://www.vmware.com/products/esxi-and-esx/.

[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] "Quagga routing suite." http://www.nongnu.org/quagga/.

[12] M. Goyal, M. Soperi, E. Baccelli, G. Choudhury, A. Shaikh, H. Hosseini, and K. Trivedi, "Improving convergence speed and scalability in ospf: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 14, no. 2, pp. 443–463, 2012.

[13] A. Fabrikant, U. Syed, and J. Rexford, "There's something about mrai: timing diversity can exponentially worsen bgp convergence," in *INFOCOM*, 2011.

[14] "Ospf shortest path first throttling." http://www.cisco.com/c/en/us/td/docs/ios/12_2s/feature/guide/fs_spftrl.html/.

[15] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A fault-tolerant engineered network," in *NSDI*, 2013.

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

[17] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM*, 2014.

[18] C. Hopps, *Analysis of an Equal-Cost Multi-Path Algorithm*, November 2000. RFC 2992.

[19] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure, "Achieving sub-second igp convergence in large ip networks," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 3, pp. 35–44, 2005.

[20] D. Katz and D. Ward, *Bidirectional forwarding detection*. Internet Engineering Task Force, June 2010. RFC 5880.

[21] "ns-3." http://www.nsnam.org/.

[22] "Direct code execution." http://www.nsnam.org/overview/projects/direct-code-execution/.

[23] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM*, 2011.

[24] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM*, 2010.

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

[26] 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.

[27] P. Pan, G. Swallow, A. Atlas, *et al.*, *Fast reroute extensions to RSVP-TE for LSP tunnels*, May 2005. RFC 4090.