Fast, Scalable and Robust Centralized Routing for Data Center Networks

Fusheng Lin, Hongyu Wang, Guo Chen, *Member, IEEE*, Guihua Zhou, Tingting Xu, *Student Member, IEEE*, Dehui Wei, Li Chen, *Member, IEEE*, Yuanwei Lu, Andrew Qu, Hua Shao, and Hongbo Jiang, *Senior Member, IEEE*

***Abstract*— This paper presents a fast and robust centralized data center network (DCN) routing solution, called Primus. For fast routing calculation, Primus uses centralized controllers to collect/disseminate the network's link-states (LS), and offload the actual routing calculation onto each switch. Observing that the routing changes can be classified into a few fixed patterns in DCNs which have regular topologies, we simplify each switch's routing calculation into a table-lookup manner, i.e., comparing LS changes with pre-installed base topology and updating routing paths according to predefined rules. As such, the routing calculation time at each switch only needs 10s of us even in a large network topology containing 10K+ switches. For efficient controller fault-tolerance, Primus purposely uses reporter switch to ensure the LS updates successfully delivered to all affected switches. As such, Primus can use multiple stateless controllers and little redundant traffic to tolerate failures, which incurs little overhead under normal case, and keeps 10s of ms fast routing reaction time even under complex data-/control-plane failures. We design, implement and evaluate Primus with extensive experiments on Linux-machine controllers and white-box switches. Primus provides $\sim$1200x and $\sim$100x shorter convergence time than current distributed protocol BGP and the state-of-the-art centralized routing solution, respectively. Furthermore, Primus maintains good routing controllability/manageability thanks to its centralized architecture, which enables us to build several advanced routing features in our testbed, including routing failure visualization and weighted-cost-multi-path routing.**

***Index Terms*— Data center networks, centralized routing, network protocols.**

Manuscript received 16 April 2022; revised 24 October 2022 and 20 February 2023; accepted 26 February 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor C. Wu. Date of publication 4 April 2023; date of current version 19 December 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62222204 and Grant 62172148, in part by the National Key Research and Development Program of China under Grant 2020YFB2104000, in part by the Natural Science Foundation of Hunan Province for Excellent Young Scholars, in part by the Training Program for Excellent Young Innovators of Changsha, and in part by Tencent. This article was presented in part at the IEEE INFOCOM 2021 [DOI: 10.1109/INFOCOM42981.2021.9488689]. *(Fusheng Lin and Hongyu Wang are co-first authors.) (Corresponding author: Guo Chen.)*

Fusheng Lin was with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410012, China. He is now with Tencent, Shenzhen 518054, China (e-mail: linfusheng@hnu.edu.cn).

Hongyu Wang, Guo Chen, and Hongbo Jiang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410012, China (e-mail: why2021@hnu.edu.cn; guochen@hnu.edu.cn; hongbojiang2004@gmail.com).

Guihua Zhou and Andrew Qu are with Tencent, Shenzhen 518054, China (e-mail: guluguluhuli@hnu.edu.cn; andrewxqu@tencent.com).

Tingting Xu is with the Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu 210093, China (e-mail: xutingting@smail.nju.edu.cn).

Dehui Wei is with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China (e-mail: dehuiwei@bupt.edu.cn).

Li Chen is with the Zhongguancun Laboratory, Beijing 100081, China (e-mail: Crischenli@gmail.com).

Yuanwei Lu and Hua Shao were with Tencent, Shenzhen 518054, China. They are now with Pinduoduo, Shanghai 200000, China (e-mail: yuanweilu612@live.com; huashao770@gmail.com).

Digital Object Identifier 10.1109/TNET.2023.3259541

I. INTRODUCTION

A. Current Distributed Routing

BGP [1] is the current *de facto* data center network (DCN) routing protocol [2], [3]. However, such distributed routing protocol has two well-known open issues [4], [5], which now become increasingly problematic as the DCN scales larger. First, the *routing convergence procedure is slow*. As large number of switches independently react to network changes without a centralized coordination, it may incur excessively unnecessary routing communication and calculation, which can cause long routing connectivity loss although physical network remains connected [6], [7], [8]. Second, it is *hard to control and manage* the whole network's routing with thousands of switches making decisions independently, *e.g.*, BGP configurations in large-scale DCNs can be daunting [9].

B. The Rise of Centralized Routing and Remaining Problems

At least from [10] and [11], the community has started to think of using centralized way to address above intrinsic problems in the distributed routing protocols. Ethane [12] may be the first successful application of centralized control on a medium-scale campus network. However, before Google published Firepath (*i.e.*, its DCN routing system) in 2015 [9], people were still unsure about whether the centralized way can handle the whole network's routing for large DCNs, which contain more than thousands of routing nodes (*i.e.*, L3 switches) and require stringently high networking performance.

Firepath [9] is (possible) the first and only published work that has successfully designed/implemented/operated a full routing protocol and system using a centralized control for DCNs (other works are not a full routing protocol and [13] still uses Firepath's algorithm for intra-DCN routing. See §II for details). Centralized architecture does help Firepath greatly accelerate routing convergence, by eliminating broadcasting communication between switches and reducing the routing inconsistency caused by an independent calculation on each switch. Moreover, it significantly simplifies the routing control and management. Nonetheless, there remain two major challenges not well addressed in Firepath, which limit the performance of its centralized routing when DCN scales larger. Specifically:

- **How to calculate the routing fast enough?** Obviously, using a centralized controller to directly calculate the routing of the entire network using shortest-path first (SPF) algorithms will be too slow. Therefore, the centralized controller in Firepath is only used to collect and store the whole network's link-state database (LSDB), and disseminates link-state (LS) changes to the switches. Each switch then distributedly calculates its routing paths. On the downside, we emphasize that it is still very time-consuming to calculate shortest paths on each switch, since performing SPF algorithms on the whole large DCN topology is required. In a topology with *n* nodes, *m* edges and *k* equal-cost shortest paths, a typical k-SPF algorithm has a very high time complexity of $O(kn(m+n\log n))$ [14].[1] Our experiments show that for a DCN topology with 10K switches (Fig. 1), it takes more than 3 seconds for a switch to calculate the shortest paths upon one LS change. This may be the reason why a single link failure causes 4s of routing connectivity loss to a rack of servers in Firepath (Table IV in [9]).
- **How to gracefully handle control-plane failure?** In case of controller failures, Firepath runs multiple backup controllers, each maintaining an LSDB of the whole network. To avoid routing inconsistency, all controllers always keep their LSDBs synced. However, this delays the routing reaction. For example, if using consensus protocols (*e.g.*, [17]) to keep LSDBs consistent among multiple backups, when the controller processes an LS, it will incur extra overhead such as logging and replicating states between multiple backups. Moreover, as DCN scales larger, failures would also be the norm in the control-plane network, since there are a large number of control-plane switches/links. Therefore LS updates reported to the controller may be lost due to control-plane network failures. Then, the reporting switch either has to wait for some retransmission timeout (*e.g.*, upon temporary failures) or wait for a controller reelection procedure (*e.g.*, when the controller's access link permanently down), both incurring significant delay.

C. Our Contributions

To tame above challenges, we propose Primus, a fast and robust centralized intra-DCN[2] routing protocol and system. Primus takes philosophies totally different from Firepath for routing calculation and control-plane failure handling:

- **Primus simplifies the routing calculation into a table-lookup manner**, which is fast and scalable. In Primus, we follow the architecture of [9], using a centralized master[3] to monitor all the link-states and each switch calculates routes by itself. However, each switch calculate the routes in a more efficient way than the classical calculation. Observing that DCNs have regular topologies and the routing changes can be classified into a few fixed patterns, we let each switch simply compare the current link-states with the *preinstalled base topology*, and disable or enable the routing entries in its *preinstalled base routing table* according to *predefined rules*. According to the routing change patterns, we develop a smart indexing technique which provides *O(1)* routing-path table lookup time for an LS change.[4] The whole routing updating time at a switch for an LS change only takes *10s of* μs even in a large network topology containing 10K+ switches. Moreover, we devise novel data structures so the memory footprint at each switch is only *<10MB* for such large network. (§III-B)
- **Primus uses multiple stateless controllers and little redundant traffic to tolerate control-plane failures**, which incurs low overhead in the normal, meanwhile achieving fast routing reaction even under complex control-plane failures. Particularly, we adopt multiple hot-standby backup masters as in [9] to tolerate master failure. However, the reporter switch is logically responsible for the success of delivering LS changes to the whole network (but still physically through the master). As such, *masters can be stateless* without remembering the whole network's link-states. Therefore, handling master failure is easy and low-cost because any backup master can process an LS change and there is no need to wait for synchronization between multiple masters.[5] Moreover, to keep fast reaction upon failures, Primus *adds some redundancies whenever passing LS messages*. Those redundant LS messages are often on different failure domains (*e.g.*, processed by different backup masters and control-plane network devices), and the routing can be correctly performed if at least one LS message copy has been successfully processed. As such, Primus can keep fast routing convergence (10s of ms for a network with 10K+ switches), even under control-plane network failure and master failure. (§III-C)

We have an open-sourced implementation of Primus (available at [18]). Our Primus master implementation runs on Linux machines, and Primus switch implementation can run both on Ruijie white-box switches [19] and Linux-based software switches. Our testbed evaluation shows that for a large network containing 10K+ switches, upon an LS change, Primus can finish the routing updates of the whole network within 33ms, which is ~98.8x faster than Firepath. Based on fast and robust routing, applications using Primus have three orders of magnitude better 99_{th} and 99.5_{th} percentile performance compared to those using BGP, and ~100x better

[1] We note that there exist some optimizations to the k-SPF algorithm (*e.g.*, [15], [16]). However, their calculation time still grows fast as the topology scales larger, which is undesired for scalability.

[2] We still assume the use of BGP for external routing. Details in §III-E.

[3] "Master" and "controller" are used interchangeably in this paper.

[4] Note that the table update time is not *O(1)* since it depends on the number of routing-path entries which are affected by the LS change.

[5] For centralized control/management demands, masters can later slowly synchronize the latest complete LS changes with each other after the routing has been updated. See §III-C and §IV for details.

compared to Firepath, respectively. We believe that Primus has set a new performance milestone for building centralized routing for large scale DCNs. Furthermore, based on Primus's high routing controllability/manageability, we build several advanced routing features in our testbed, including routing failure visualization and weighted-cost-multi-path (WCMP) routing.

A shorter conference version [20] of this paper appeared in INFOCOM 2021. The previous version did not demonstrate Primus' characteristics beyond being more efficient in data-plane routing recovery. Compared with the previous version, in this paper, we show the controllability and manageability of Primus through two scenarios: network failure location and dynamic WCMP. Also, Primus implementations on some classical topologies other than fat tree are discussed. Moreover, the effectiveness of the redundancy mechanism proposed by Primus is modeled and evaluated. In addition, we introduce how to update the base topology for Primus and evaluate its efficiency, and discuss several design choices in more details.

II. Related Work

Other centralized routing control: Besides Firepath, there are also previous works (*e.g.*, [2], [13], [21], [22], [23], [24], [25], [26], [27], [28]) using centralized control to address part of the routing problems in DCN. However, they are *not complete routing protocols*. For example, [23], [24], and [25] use centralized controllers to help scheduling network flows on certain paths, thus to minimize flow completion time or balance the network utilization. However, they still rely on underlying routing protocols to maintain and calculate the routing paths (*e.g.*, [24] is implemented on top of BGP). [21] takes the same idea of using centralized controllers to collect/disseminate link-states. However, it aims to build a layer 2 routing based on MAC address, which does not match the layer 3 IP routing architecture in modern data center physical networks. Moreover, [21] did not address the problems discussed in §I-B. [2], [22] utilize centralized control to translate between physical and virtual addresses for network virtualization. [26], [27], [28] build underlying system (*e.g.*, switches softwares and distributed systems) to provide centralized abstraction for data center networks. However, they do not build routing protocols and algorithms on top of the system. Orion [13] presents Google's latest distributed software-defined networking (SDN) platform, which can be used as the control plane for data center routing. However, Orion's intra-DCN routing control plane still takes Firepath as the specific routing algorithms.

Improved distributed routing protocol: Many works, *e.g.*, [29], [30], and [31], try to improve BGP using various techniques. Although big improvements have been achieved, the intrinsic drawbacks in distributed routing still remain to be open questions (*e.g.*, convergence still requires seconds to minutes) [4], [30]. Reference [32] presents a BGP-based routing design for data center. Although providing better reliability and flexibility, it still requires a second-level convergence time. RIFT [33] utilizes the pre-knowledge of the DCN fat tree topology to simplify the routing and limit the broadcasting area. However, staying as a distributed protocol,

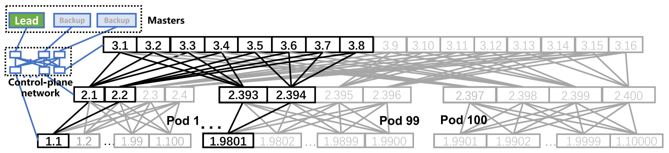


Fig. 1. Primus over an example DCN.

RIFT still has intrinsically poor routing controllability (making decisions distributedly) and slow convergence time (link-state broadcast, calculation and waiting timers). Since RIFT is still an RFC draft (working in progress) lacking implementation details, we are not able to compare with it in our testbed.

Data-plane connectivity recovery: There are many works (*e.g.*, [6], [7], [8], [34], [35], [36], [37], [38], [39], [40]) aiming to provide fast data-plane connectivity recovery before routing convergence. Fast rerouting (FRR) techniques (*e.g.*, [34], [35], [36]) focus on the Internet scenarios. However, in DCN with dense fat tree topology, for downward routing paths, it does not satisfy the loop-free requirements of these FRR techniques and still require control-plane convergence (single next-hop),[6] so typically they are not applied in DCNs [3]. Several works [6], [7], [8], [38] focus on fast data-plane recovery in DCN. However, they either require significant changes to physical topology [6], [38], or may incur temporary routing loops or use non-shortest bounce-back paths [7], [8]. Moreover, these works are complementary to Primus. Primus can leverage those data-plane techniques to further accelerate routing recovery before control-plane routing convergence.

Centralized routing in WAN: Previous works build centralized routing system for traffic engineering in inter-DCN wide-area networks (WANs) (*e.g.*, [41], [42], [43]), which is different from the intra-DCN environment.

III. Primus Design

A. Architecture

In Primus, we follow the architecture of [9], using a centralized master to collect/disseminate all the LSes. Each switch simply compares the current link-states with the *preinstalled base topology*, and disables or enables the routing entries in its *preinstalled base routing table* according to *predefined rules*. This table-lookup manner greatly simplifies the routing calculation. Moreover, it maintains the centralized routing manageability/controllability through monitoring global LSes.

Master communicates with each switch through an out-of-band control-plane network as shown in Fig. 1. Each switch monitors its local data-plane links (using standard failure detection scheme such as [44]) and reports to the master upon a local LS change. After receiving an LS, the master delivers updates to all the switches whose routing may be affected. Possibly affected switches are fixed in a certain DCN topology, so the master uses predefined rules to quickly find them. Notes that upward link affects all switches in the subtree below it, *e.g.*, in Fig. 1 link 2.1→3.1 possibly affects routing in switch 2.1 and 1.1-1.100. Similarly, downward link affects all upper layer switches connected to it (with one or two hops) and the subtree below those switches, *e.g.*, in Fig. 1

[6]For upward routing paths, there are multiple equal-cost next-hops so it simply uses ECMP fast data-plane rerouting instead of those FRR techniques.

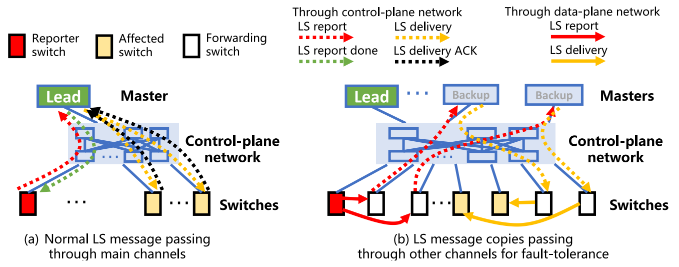


Fig. 2. Primus's fault-tolerant link-state updating scheme.

link 2.1→1.1 possibly affects switch 2.1, 3.1-3.4, the leftmost Agg switch in each pod, and all ToR switches except 1.1.

Each switch is pre-configured with the static address of the master, and setups a long-lived bidirectional reliable transport connection (*e.g.*, TCP) to the master through control-plane network for passing LS messages (called *main channel*). As Fig. 2(a) shows, upon detecting a local LS change, a switch will report it to the master through the main channel (dashed red arrow-line). Each LS change has a unique ascending ID per link. Whenever receiving a new LS, the master will deliver the LS to all the possibly affected switches[7] through their main channels (dashed yellow arrow-line). A switch will reply an acknowledgment to the master over main channel after successfully receiving the LS and updating its routing (dashed black arrow-line). The master will reply a response to the reporting switch over the main channel (dashed green arrow-line), when it successfully delivers the LS to all the affected switches and receives their acknowledgments. If not receiving the master's response, the reporting switch will keep retransmitting the LS (after certain timeout) until succeed or the master changes (in case of master reelection). Note that the timeout can be relatively long since we have fast fault-tolerant schemes (§III-C). Another possible option is to let the leader immediately send an ACK to the reporter switch without merging all the ACKs into one. However, this may incur higher overhead on several aspects: 1) The reporter switch needs to maintain ACK states and selectively retransmit LSes to each affected switch (possibly maintain an RTO timer for each affected switch) if any of the ACK does not return. However, in our design, the sending of LS to all affected switches is treated as a single event on the reporter, and the master only needs to count the number of returned ACKs, which is easy to handle and light-weight. 2) This may incur higher traffic volume for passing LSes, including many ACKs back to the reporter and LS retransmission from the reporter to the master.

Next, we introduce in details how Primus's design achieves *1) quick routing calculation (§III-B), 2) efficient fault-tolerance (§III-C), and 3) maintains high controllability/manageability (§III-D)*, respectively.

B. Routing Calculation

For ease of presentation, in the rest of paper, we assume that the DCN uses the most popular three-layer fat tree topology [2], [9], [21], [45]. The topology contains three layers of switches, *i.e.*, Core, Aggregation (Agg) and Top-of-Rack (ToR) switches, respectively. k ToR switches and s Agg switches form a *pod* with each ToR connected to each Agg. We denote the number of pods as p. Each Agg switch in a pod is connected to n different Core switches, and each Core switch is connected to every pod. The total number of Core, Agg, and ToR switches is $s \times n$, $s \times p$ and $k \times p$, respectively. Figure 1 shows an example topology with $k = 100$, $s = 4$, $p = 100$ and $n = 4$, which is used for production DCNs in Tencent. Note that Primus also works for fat tree with more layers, and can be easily adapted to other topologies (see §IV).

Each switch maintains two table data structures for routing calculation.[8]

Path table: It includes all the equal-cost shortest paths to every destination in the topology and lists all the links along that path.[9] We do not merge paths (*e.g.*, with the same next-hop) in the path table for ease of routing calculation (detailed reasons in §IV). The path table also records the number of failed links (denoted as FL) in each path. Once receiving a link-state update from the master, the switch first finds the path entries that contain that link (discuss how to find them later), then increases (for link failure) or decreases (for link recovery) FL of those path entries by one. The switch can detect if a path can be used for routing in $O(1)$ time by checking whether its FL equals zero.

For the topology shown in Fig. 1, there are ∼160K paths in the path table, which will only take about 7.7M bytes in each switch (experiments latter in §VI-A). Note that this table is an internal data structure used by the Primus routing calculation

TABLE I

SNAPSHOT OF LINK TABLE OF SWITCH 1.1 IN FIG. 1

No.	Next	Dest	FL
...	... → ... → ...	...	...
39201	→2.1→3.1→2.393	1.9801	1
39202	→2.1→3.2→2.393	1.9801	1
...	... → ... → ...	...	...
39204	→2.1→3.4→2.393	1.9801	1
39205	→2.1→3.1→2.393	1.9802	1
...	... → ... → ...	...	...
39208	→2.1→3.4→2.393	1.9802	1
...	... → ... → ...	...	...
39601	→2.1→3.1→2.397	1.9901	1
...	... → ... → ...	...	...
39604	→2.1→3.4→2.397	1.9901	1
39605	→2.1→3.1→2.397	1.9902	1
...	... → ... → ...	...	...
79201	→2.2→3.1→2.393	1.9801	0
...	... → ... → ...	...	...

TABLE II

SNAPSHOT OF LINK TABLE OF SWITCH 1.1 IN FIG. 1

From	To	State	Type: First Entry
1.1	2.1	Fail	1: 1
1.1	2.2	OK	1: 40001
...	...	OK	...
2.1	3.1	OK	2: 401
...	...	OK	...
3.1	2.397	OK	3: 39601
...	...	OK	...
2.397	1.9901	OK	4: 39601
...	...	OK	...

[7]There must be affected switches in Fig. 1, if not, the master will not deliver the LS.

[8]We only discuss routing to the servers for ease of presentation. Routing to switches is similar and can be easily drawn from the following design.

[9]Access links from servers to ToRs are not listed in the paths, since those links are only used in L2 switching but not in L3 routing.

algorithm, and the actual routing/forwarding table in the switch data plane can be much smaller because the paths with the same first next-hop and the same destination can be merged into one route. Each switch can distributedly merge its routes and check whether the merged route is working by detecting if there is at least one path working (*i.e.*, FL $= 0$) within that route. It is also simple to make this merging and checking process very fast, *e.g.*, using bitwise AND on the FLs of all paths. Table I shows a snapshot of switch 1.1's path table, assuming the link from switch 1.1 to 2.1 fails (the affected entries' FL increased by 1).

Link table: It records the current state of all the links in the base topology. Apparently, once a link-state changes, it would cost too much for a brute-force search in the path table to find which entries will be affected. As such, we pre-calculate all the affected paths for each link, and maintain a data structure in each link table entry to smartly index all the path entries in the path table affected by this link, in $O(1)$ time complexity.

An intuitive data structure for maintaining such index would be a bitmap, with each bit indicating whether a path contains this link. However, such simple bitmap would consume too much memory since there are $k \times s \times n$ entires in total in the path table. For example, in Fig. 1, such bitmap needs 160Kb memory to index all the path entires and each of the 40K links needs one such bitmap, which costs ~6.6Gb in total. Such memory consumption would be prohibitive as the scale grows larger.

Luckily, in DCN topology there are only several fixed patterns of how a link-state change will affect the routing paths. Therefore, it is not necessary to use a bitmap that can represent any combination of all the routing paths. Specifically, from a switch's point of view, links in the fat tree topology can be classified as four types:

- *Type 1*. For upward link from ToR to Agg, it will affect $n \times k \times p$ path entries in total, *i.e.*, the n paths stemmed from this Agg to all the $k \times p$ ToRs.[10]
- *Type 2*. For upward link from Agg to Core, it will affect $k \times (p-1)$ path entries in total, *i.e.*, the single path through this link to all the $k \times (p-1)$ destinations in all other pods.
- *Type 3*. For downward link from Core to Agg, it will affect k path entries in total, *i.e.*, the single path through this link to all the k destinations in this pod.
- *Type 4*. For downward link from Agg to ToR, it will affect n path entries in total, *i.e.*, the n paths through this Agg to the single destination of the ToR.

As such, we can use a compact data structure that only needs to indicate the type and the first path affected by this link, and we can quickly index all the affected path entries according to the above patterns. Such data structure only requires 2 bits (for type) plus $log(k \times s \times n)$ bits for the index of the first affected entry. For the example topology shown in Fig. 1, the link table in our implementation only consumes about 1.3MB memory in total (experiments in §VI-A).

[10]For ease of organizing tables, we assume that the routing between switches must go up to the Core and then go down, even in the same pod. However, the actual routing in data-plane within a pod use shorter paths that only traverse the Agg or ToR.

Table II shows a snapshot of switch 1.1's link table in Fig. 1. The last column is the index of the affected path entries. According to the index, the switch can find the first affected path table entry based on First and find all the rest of the affected path table entries based on Type. For example, if link 1.1→2.1 failed, switch 1.1 would find Type: First to be 1:1 in the link table. This means the first affected path table entry is No.1 (First $= 1$) and the next 40000-1 entries ($4 \times 100 \times 100$ as Type $= 1$) are also affected. Then switch 1.1 will increase all these entries' FL by one. Similarly, if link 2.1→3.1 failed, it would check Type:First and find that the first affected path table entry is No.401 (First $= 401$). Also, the next 9900-1 entries (100×99 as Type $= 2$) are also affected. Note that there is a fixed gap of four to the next affected path table entry since there are four upward links from each Agg to Core. All these entries' FL will then be increased by one.

C. Handling Control-Plane Failure

The reliable LS report scheme described in §III-A ensures the network has an eventual *consistent view* to the latest LS change, without relying on states maintained in the master or other parts of the network. As such, we can easily use control-plane redundancy (backup masters and redundant LS messages) to tolerate failures. Specifically, Primus takes the following fault-tolerant schemes.

Slight redundancy for speed: We use multiple backup masters and one lead master. As Fig. 2 shows, the lead master is active to collect/disseminate all LS messages from main channels, and multiple backup masters work as hot standbys and process redundant LS messages. Each switch is pre-configured with the static address of each backup master. Each switch also maintains a *backup main channel* (reliable) between each backup master. Different from the main channel, those backup main channels are only lazily monitored through slow hello, being prepared for the possible master reelection, but never used to pass LS messages until a backup becomes the lead. For a switch, whenever detecting a local LS change in the data-plane, besides reporting the LS through its main channel, it also sends multiple copies of the LS to backup masters through other reachable switches using normal data-plane network (solid red arrow-line in Fig. 2(b)). Reporter sends multiple copies of the LS through other reachable switches in data-plane, so it can tolerate the failure of the reporter's single control-plane access link. These switches will immediately forward those copies to the backup masters through their own control-plane links (dashed red arrow-line in Fig. 2(b)). Similarly, backup masters will send such copies to some other switches (which will forward) to deliver LS updates to a target switch (yellow arrow-line in Fig. 2(b)).

Above LS copies (called *redundancies*) are transferred using low-overhead unreliable transport (*e.g.*, UDP) through randomly picked (and different) forwarding switches and backup masters, and are never ACKed or retransmitted. This creates multiple *other channels* between the master and switches. Target switches will process the first arrived LS message among all the copies (including the origin) and neglect others (based on the message ID). Note that a switch always sends an ACK to the lead master through the main channel when it has processed this LS for the first time (no matter from main channel or other channels), thus the lead master can quickly notify the reporting switch when the whole network has finished the routing updates. Based on above redundancy schemes, the routing reaction in Primus still keeps fast even under complex control-plane or master failures, as long as at least one main/other channel is working.

Slow detection and synchronization with low-overhead: Switches detect the main channel status through periodical hellos. With the aforementioned fault-tolerant scheme, this hello can be very slow (*e.g.*, minute-level TCP keep-alive) without delaying routing reaction upon control-plane failures. Once the main channel is detected as dead, we will setup indirect channels to work as the new main channel to pass control-plane messages. For a switch, if it detects the failure of its main channel, it notifies several other switches (possibly) reachable in data-plane (according to its local routing table) and picks the first responding one to establish an indirect reliable data-plane communication channel through it to the master, working as the new main channel. Similarly, indirect main channel will be setup when the master detects the failure of certain target switch's main channel.

We run a consensus protocol (*e.g.*, Raft [17]) among all masters (including the lead and the backups). Note that the consensus protocol does not affect normal LS processing, but only runs in the background to detect master failure, and reelect lead master. Once a new lead is elected after the original one fails, it will notify all the switches. For centralized manageability and controllability, to maintain global network states in case of master failure, the consensus protocol also slowly exchanges latest global link-states between the lead and backup masters in the background (just best effort but not mandatory for routing correctness, discussed in §IV). There exists a minor problem that the reporter cannot receive the ACK from the leader that indicates the success of delivering all the LSes to all the affected switches. However, the actual routing recovery has already completed and it may only incur some retransmission of the LSes from the reporter after relatively long timeouts.

Note that the consensus protocol is decoupled with normal routing reaction upon LS changes, so it can run at a very low frequency, incurring low overhead. We mentioned earlier that *masters can be stateless*, which does not contradict with the description here. The main tasks for consensus protocol are running in the background to detect master failure and reelect the lead master, but not for maintaining states. Upon the lead master failure, other backup masters or data-plane switches can help delivering LS changes. For data-plane routing recovery, masters can be fully stateless about global LS states, and the reporter switch will retransmit the LS if it is not received by all the affected switches. Since each switch maintains the path and link table locally, all the things that switches need to do is to change its local tables according to certain rules, such as routing equally or in a weighted way using those working paths. Once a new lead is elected after the original one fails, it will notify all the switches and collect/deliver LSes with no need of history states. If there is a need for advanced centralized manageability and controllability, such as for visualizing failure localization, masters may need to be stateful for control-plane usage. Under such case, consensus protocol can slowly exchange latest global link-states among masters in the background. However, such control plane management functionalities can be asynchronous, which does not delay the fast data-plane recovery.

Effectiveness and bandwidth usage of LS redundancy: We quantitatively analyze the effectiveness of the redundancy mechanism in theory. Our analysis is based on the following basic assumptions:

- *The reporter and the affected switches are alive.*
- *Switches/Links in the control-plane have same failure possibilities as the data-plane counterparts.*

By assuming the failure probability of each link, we model the probability of successful transmission of LSes between the main channel and backup channels, respectively, and finally derive the effectiveness model of the LS redundancy. Solution to the above model under different scenarios can be found in the results in §IX-A (Fig. 11). Our results show that if using 3-UDP backup channels, the average probability of success is 99.6675%.

We then evaluate bandwidth utilization by evaluating the number of packets generated in redundant links when transferring LSes. Solution to this model can be found in the results in §IX-B (Table. IV). The results show that even with 3UDP redundant channels, the average additional traffic in the network is only 101.03KB/s.

D. Routing Controllability/Manageability

Besides the very fast data-plane routing recovery to network changes benefited from the table-lookup method and stateless masters, in this section, we show that Primus still can meet the common features of centralized DCNs, i.e., controllability and manageability. Specifically, benefited from the table data structures listing all the routing paths, it is natural and easy to control/manage each switch's routing by manipulating its path entries. We use two example scenarios to show Primus' controllability and manageability. Note that these are only example scenarios. Primus' controllability and manageability are by no means limited to these scenarios. Moreover, these are common features which are also applicable to other centralized routing schemes for DCN.

1) Network Failure Localization: Primus master collects global link-states in a centralized manner, making it natural and simple to locate failures in switches or links through the master. As experiments in §VIII show, we can easily visualize the location of network failure only based on existing information maintained in Primus master, without any extra monitoring systems. We note that information in the master can only help with connectivity problems, and other failures (*e.g.*, packet random loss) still require extra tools to debug. However, quickly figuring out connectivity problems alone can already significantly simplify the daily DCN management [46].

2) Dynamic WCMP: Link or switch failures usually happen in large DCNs which break the network symmetry. For example, in Fig. 1, if the links from switch 2.1 to

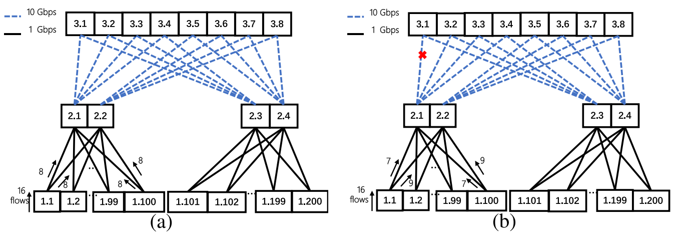


Fig. 3. *(a)* While there is no link failure in the network, the traffic is evenly distributed among network links. *(b)* While the link failure breaks the network symmetry, dynamic WCMP recalculates the weight and redistribute the network traffic.

switch 3.1 and switch 3.2 fail, ECMP weights to the four next-hops (switch 2.1 to 2.4) in switch 1.1 is 1:1:1:1. The bandwidth for paths containing switch 2.1 will be half compare to other paths. The ideal weights in switch 1.1 should be 1:2:2:2.

Weighted-cost multi-path routing (WCMP) [47] is an effective method for balancing traffic among paths in a network that has become asymmetric. Primus can easily support above dynamic WCMP since it lists all the base paths in the path table. Specifically, if link or switch failures occur and break the network's symmetry, the master will deliver the LS messages to the affected switches. Each switch can then dynamically calculate the weight of each next-hop route according to the current weight of paths through this route. The detailed steps and algorithms for this process are described below.

Both master and switches store the link bandwidth information on the link table (as shown in Table II). Switches locally detect the link bandwidth according to physical-/link-layer information. Once the link bandwidth changes (or link down), the information will be embedded into LS messages and reported to the master. After receiving the LS messages, the master delivers the LS message to all the affected switches (calculated from the base topology).

Primus master doesn't calculate WCMP weights themselves, but each switch performs these calculations locally. This approach reduces the computational burden on the master. Similar to [25], the weights can be calculated by the max-flow min-cut algorithms for any source-destination pair. As shown in Fig.3(a), assume that each ToR switch in pod 1 has 16 network flows sending data at the same speed to switches in pod 2 (*i.e.*, 1.1→1.101, 1.2→1.102, …, 1.100→1.200). In this condition, WCMP will distribute the network flows evenly among network links. Taking switch 1.1 for example, there will be 8 flows going through link 1.1→2.1 and the other 8 flows go through 1.1→2.2. Once the link failure breaks the network symmetry, WCMP will update link weights and redistribute the network traffic among these links. As shown in Fig.3(b), the link between switch 2.1 and 3.1 has broken, which causes bandwidth capacity imbalance between switch 2.1's uplink and 2.2's uplink (total uplink bandwidth ratio between Agg switch 2.1 and 2.2 is 3:4). Our dynamic WCMP can perceive the bandwidth capacity imbalance and calculate the WCMP weights, *e.g.*, $\frac{3}{7}$ of network flows from ToR switch 1.1 go through link 1.1→2.1, and the others go through link 1.1→2.2. The link weights can be easily calculated by the switches using the max-flow min-cut algorithm base on the global network topology and the link table. After the weights have been calculated, Primus injects the path weights into the routing table, which maximizes the link utilization.

E. Other Design Details

Controlling routing flaps. Primus adopts per-local-link timer in each switch to monitor each switch's local LS changes. The timer does not apply to the first state change of each link, but throttles updating subsequent continuous changes[11] to the master. As such, Primus reacts fast on normal LS changes while having routing flaps well controlled. In case of buggy switches, master can also throttle disseminating continuously changing LSes by monitoring LSes itself. Note that such timer incurs very small overhead, which can be easily implemented by adding a timestamp for each local link in the link table, indicating the time of its last state change.

Routing initialization/reboot. We preconfigure the base DCN topology, the expected position in the topology, and all the masters' static addresses to each switch.[12] Based on the configuration, switches can generate their base link and path table. When a switch initializes or reboots from crash, it will build main channel with all masters, reporting its base information, and finding out the current lead master (also masters can discover topology wiring error from the position information reported by switches). Meanwhile, it finds out the highest LS event ID of all its local links from all the masters, and then checks all local links' states and reports all current states using highest LS event ID++, to ensure that the whole network view its latest link-states. Similarly, masters are also pre-configured with the base DCN topology and all other masters' static addresses. When a master initializes or reboots from the crash, it first finds out (or reelect) the lead, and starts listening/processing switches' LS messages.

Multi-link failures and switch failures. Failures of multiple links are processed as multiple independent LS events, using the same methods described before. A switch is detected as dead if all masters cannot reach it (even with redundancy schemes). Then the master takes it as all its data-plane links are down and notifies other switches to update routes accordingly.

Interacting with external routes. Primus still uses BGP to interact with external Internet routers. Specifically, border switches in DCN (*e.g.*, Core switches) both run Primus and BGP routing instances, but only enable BGP on outside ports. Border switches will disseminate BGP routes learned from outside to each internal switch, notifying which address they can reach. As such, when having traffic going outside, a switch can route the traffic first to the border using the intra-DCN routes calculated by Primus, and then to the outside.

Routing in control-plane. Since the control-plane network is relatively small compared to the data-plane and our redundancy scheme helps to handle control-plane failures fast at low-cost, we simply use existing distributed protocols (*e.g.*, OSPF [48]) for routing in the control-plane network.

[11]A subsequent link-state change which happens after the timer length is considered as the first state change again for this link.

[12]If the base topology changes, *e.g.*, due to scale upgrade, we will reconfigure each switch.

Base topology update for Primus. Primus simplifies the routing calculation into a table-lookup manner. When base topology changes (*e.g.*, adding new pods), we have to update the base link and path table in each switch. We use the way similar to the routing initialization discussed before to update base topology in each switch. Specifically, after the topology has been physically changed, the lead master will notify each switch the new way of how to generate the link/path table according to each own expected position in the new topology. Note that the master does not send the actual tables to them for saving network bandwidth, but only the rule to generate the tables. During this period, the whole network's routing is stopped. We have evaluated the time cost of topology update with an experiment results shown in §IX-C, which is about dozens of seconds in a large topology.

Since base topology change is a low-frequency event, for simplicity, we choose the initialization-style topology update because it does not cost much time. We note that there may exist incremental and faster way such as to insert or modify entries in the existing tables. How to further shorten the downtime during base topology update is left for our future work.

IV. Discussions

Master state loss. Since we use stateless master to achieve fast routing reaction upon complex failures, it may lose global LSes (although not likely). It may temporarily affect some centralized control/management functionalities (such as the routing failure localization introduced before in §III-D.1). However, the master will eventually get the correct latest states and restore these functionalities, after running correctly for a while. These control/management functionalities are relatively less time sensitive, so we choose to decouple them with a normal network routing reaction to reduce overhead. Note that some centralized control/management functionalities will not be affected by master state loss. For example, for the WCMP feature we build on Primus, it is rarely affected by master state loss since the master only conducts stateless LS forwarding and each switch independently calculates the latest WCMP. For such advanced routing control which works on the data-plane, since each switch maintains the path and link table locally, all the things that switches need to do is to change its local tables according to certain rules. Normal routing convergence will not be affected by master state loss as described in §III-C. If considering advanced routing managements such as global failure localization, upon the lead master fails, the total recovery of accurate failure localization needs to sync LS states among (backup) masters, which can be several seconds (as shown in Fig. 7). However, such control plane management functionalities can be asynchronous, which does not delay the fast data-plane recovery.

Routing correctness. We preinstall all the shortest paths into switches, and only disable/enable preinstalled paths upon LS changes. As such, Primus will only use those (correct) shortest paths, without worrying about routing loops. Routing may be unreachable when only non-shortest paths exist physically (*e.g.*, bounce back to upper-layer switches), but DCNs already have plenty of shortest paths to tolerate network failures. Since there is only one generator, *i.e.*, reporter switch, for a certain LS (masters and other switches are only for forwarding), routing is eventually consistent in the whole network.

Why not use merged routes? Once a link's state changes, a switch cannot know whether a route entry is still working or not if only using a merged route instead of monitoring the status of all the links along that route. For example, in Fig. 1, if upward link 2.1→3.1 fails, switch 1.1 has no idea about whether its next-hop 2.1 is still valid or not, since 2.1's other three upward links (to 3.2/3.3/3.4) may have already failed. As such, it may require master calculation based on history states, hurting performance and bringing consistency issue.

Why not centralized table-lookup? As introduced in §III-B, each switch needs a link table and a path table for updating routes. These two tables are different for each switch which are calculated based on location. As such, putting all the tables in the master will consume too much memory. Taking the topology in Fig. 1 as an example, the total memory cost of all the tables in all the switches is more than 30GB. Moreover, if routing calculation is done on the master, it raises performance and consistency issues in case of control-plane failure.

Traffic incast/outcast to/from the master? There are two conditions possibly generating incast [49] traffic to the master, but neither of them will cause performance issues: 1) Multiple switches simultaneously report local LS changes to the master. However, the number of concurrent LS changes is typically very small (*e.g.*, hundreds per day [50]), which incurs very low volume of traffic for LS reporting. 2) Multiple switches simultaneously reply acknowledgments to the master when receiving LSes. However, taking the large topology in Fig. 1 as an example, even if all the switches send acknowledgments simultaneously, the whole traffic volume is less than 700KB (64B per acknowledgment), which is far below modern DCN switches' buffer capacity (*e.g.*, 9MB for Broadcom Trident-II chip [51]) and unlikely causes packet drops. Moreover, switches can add some random delay before sending acknowledgments, and such designed delay is only for the acknowledgment which does not increase the routing reaction time. For outcast traffic, masters do need to send a lot of LSes to a bunch of affected switches. However, this does not take much time, which can be done within 10s of ms even for ten thousand of switches (experiments in §VI-B).

Why a unique lead master to deliver LSes to all affected switches. A unique lead master simplifies the design and improves the efficiency. It is indeed possible to use multiple masters with each for different part of the switches according to some predefined job division. However, if multiple masters are used, any master (or any related components) failure may cause the failure of delivering LSes to some switches. This increases the failure possibility. Moreover, upon a master failure, it needs some renegotiation between other masters to redeliver the LSes for the failed one, which incurs consistency issue and delay. Primus uses a unique lead master, but employs a redundancy mechanism to avoid single point of failure.

Choice between in-band and out-of-band control-plane. Although we choose out-of-band way, in-band control is indeed possible for Primus. In-band control for centralized routing is a classical problem formerly originated from SDN. In the Internet scenario, there are three major challenges it is difficult to solve, *i.e.*, *1)* routing bootstrapping, *2)* data-plane failures affecting control-plane availability or correctness, *3)* data-plane traffic affecting control-plane performance. However, all these problems may be able to be addressed by Primus in the DCN environment.

Particularly, for problem *1)* and *2)*, each switch is pre-installed with base tables and routing entries (as introduced in "routing initialization" in Sec. III-E). As such, switches can also function properly even if they are totally or partially disconnected to masters during bootstrapping or data-plane failures. The LS redundancy schemes as well as the fact that the reporter will retransmit the LS until succeed can help switches finally get the latest LS through data-plane (if not fully partitioned). For problem *3)*, we can reserve dedicated hardware priority queues for control traffic in switches, such that the control is not delayed by the huge amount of data-plane traffic.

We choose the out-of-band way due to its simplicity in practice. For example, when building a large-scale DCN, it is easy to verify the topology connection and switch wiring using a dedicated control-plane network.

Primus for other topologies. Primus has certain requirements for the topology. *1)* The topology must be or able to be abstracted into a symmetric structure. *2)* Routing variations in this topology can be abstracted as several fixed patterns. Next, we will introduce how Primus can be adapted in other existing popular DCN topologies.

1) Primus for VL2. VL2 [2] is a scalable and reliable network architecture that designed for data centers, which provides a richly connected backbone, improving the ability of fault tolerance. Similar to fat-tree topology, VL2 is a three tier hierarchical model that comprises a core layer (also called intermediate layer in [2]), aggregation layer and access layer. If the links between core switches and the aggregation switches build a complete bipartite graph (*i.e.*, interconnected in a full-mesh), which provides better robustness than fat tree topology. ToR switches are in the lowest level, interconnect the underlay racks to the aggregation switches.

We can easily apply Primus to VL2 network architecture to improve the L3 routing performance and only require a minor change in affected paths calculation (§III-B). Assuming that there are n Core switches and p pods. Each pod contains k ToR switches and s Agg switches. In VL2, $s \times p$ Agg switches and n Core switches interconnect in a full-mesh, also the k ToR switches and s Agg switch in the same pod. Similar to the fat tree topology, from a switch's point of view, we can summarize the links as four types:

- *Type 1.* For upward link from ToR to Agg, it will affect $n \times (s \times p - 1) \times k$ path entries in total, *i.e.*, all the n paths stemmed from this Agg to all the $k \times p$ destinations.
- *Type 2.* For upward link from Agg to Core, it will affect $(s \times p - 1) \times k$ path entries in total, *i.e.*, the single path through this link to all the $k \times p$ destinations.
- *Type 3.* For downward link from Core to Agg, it will affect $s \times (p - 1) \times k^2$ path entries in total, *i.e.*, the single path through this link to all the k destinations in this pod.
- *Type 4.* For downward link from Agg to ToR, it will affect $n \times (p \times s - 1) \times k$ path entries in total, *i.e.*, the paths through this Agg to the single destination of the ToR.

VL2 can benefit from Primus but doesn't need to alter any components, *e.g.*, its address resolution, packet forwarding mechanism, directory system, *etc.*.

2) Primus for BCube. Unlike the switch-centric network structures (*e.g.*, fat-tree, VL2, *etc.*), BCube [52] is a kind of server-centric network structure, which is designed for shipping-container based modular data center (MDC). BCube comprises two types of network devices: commodity switches and servers that have multiple ports. BCube is a recursive network structure, and each server connects to multiple switches that belong to different levels. There are no direct network links between switches as which only connect to servers. $BCube_0$, which is simply composed of n servers and an n-port switch, is the minimum unit of BCube network structure. And n $BCube_0$s compose a $BCube_1$. More generally, a $BCube_k$ is constructed from n $BCube_{k-1}$s and n^k n-port switches.

BCube runs a protocol suite that specially developed for it, *e.g.*, a source routing protocol (*i.e.*, BCube Source routing, BSR) for routing, a packet forwarding engine as an auxiliary tool for packet processing. In BCube topology, there may be multiple available parallel routing paths. When a new flow comes, BSR firstly selects a default routing path and sends a probe packet to check whether this path is available. Meanwhile, the source server will send probe packets over multiple parallel paths to find the best path (*e.g.*, maximum bandwidth, minimum delay, *etc.*). Once the best path is selected, the source server switches the flow to the path. The mechanism of BSR will hurt the network transmission performance once the network failure occurs as information is not timely updated because BSR should send a probe packet to capture the information of link/server failure.

Primus can help BCube make rapid responses to network condition change (*e.g.*, link up/down, link bandwidth change, *etc.*), as a plug-in unit. The reason for the low efficiency of BSR in BCube are as follows: *1)* BSR will calculate a routing path set for every flow using its algorithm and select a default path from the path set. Note that BSR is unaware of the availability of the selected path. It requires BSR sending a probe packet to confirm the validity of this routing path. If it is unavailable, the time and bandwidth resources will be wasted during this period, which decreases the network transmission effectiveness. *2)* BSR uses the information that probe packet return to choose the best path from the path set, *i.e.*, BSR will send a probe packet for each path in the path set, the intermediate node or destination node will return a response message that carry some useful information (*e.g.*, bandwidth, delay, *etc.*). BSR determines which routing path is the best using different metrics, then the BSR switch the default path to the best. *3)* BSR depends on the response of the probe packet sensing network link failure. Although BSR will send probe packet to perceived network failures or dynamic network conditions periodically (*e.g.*, every 10 seconds), it still has a strong lag in the fast changing network condition.

We can use Primus to speed up the routing reaction of BSR. Primus completes work such as detecting accessibility in advance, thus avoiding the drawbacks of using probe packets. It can provide a global network perspective for BSR, which can bring the following benefits: *1)* BSR can quickly select a best routing path based on the global network perspective, avoiding sending probe packets for each routing path in the path set, which saves time and network bandwidth. *2)* Primus can help BSR achieves rapid reaction to changes in network conditions (*e.g.*, link up/down) because Primus keeps very short reaction time when network conditions changed.

V. Implementation and Testbed Setup

A. Primus Implementation

We have a complete implementation of Primus with 3128 lines of C++ code (available at [18]), based on Linux-machine masters/switches and Ruijie white-box switches [19] with SONiC [53] switch OS installed.[13] Primus works as a daemon process on each switch and master. The switch implementation mainly consists of three components, *i.e.*, *link monitor*, *link-state updater/receiver*, and *routing calculator/updater*. Link monitor monitors a switch's NIC (switch ports) status through Linux `epoll` events.[14] Link-state updater/receiver reports/receives/forwards LSes and other control messages through long-lived-TCP-based main channel and UDP-based redundancies. Each LS is formatted into a 52B data structure. Routing calculator/updater uses the table structures described before to calculate routing, and updates the Linux kernel routing table through `rtnetlink`. Master communicates with multiple switches through multiple TCP threads based on `epoll` event loop. The election protocol between masters is based on an existing implementation of Raft [54]. Although we have not used high-performance networking stacks (*e.g.*, DPDK [55]) for now, the current Primus implementation offers performance good enough even for a very large network (results in §VI). Integrating Primus implementation with high-performance networking stacks will be our future work.

B. Testbed Setup and Methods Compared

We build a prototype testbed consisting of 11 Linux-virtual-machine based switches (Ubuntu 16.04.4, kernel 4.12) and 3 B6510-48VS8CQ Ruijie switches (SONiC.201803.release.0, Kernel 3.16.0-5). The prototype data-plane topology is shown in the black and bold part of Fig. 1 (switch 1.1/2.1/2.2 are physical Ruijie switches). Note that we cannot virtualize these physical switches into more logical ones because currently SONiC does not support VRF (Virtual Routing and Forwarding) [56]. All the VM switches are connected through virtual switches with 1Gbps links, hosted on 4 Dell R720XD physical servers (Intel Xeon CPU E5-2620, 96GB memory). Ruijie switches are connected through a 1Gbps link between each other and between physical servers. Each VM uses two dedicated CPU cores and 1GB memory. Four extra VMs (same configuration) are used as the Primus (lead/backup) masters, connected with switches with an out-of-band 1Gbps control-plane switch (one control-plane access link per master). We use a server (Dell R720XD) to act as the out-of-band control plane and deploy the masters (in VMs) on that server. All switches and masters are connected through a 1Gbps 24-port control-plane switch. Due to the simple small-scale control-plane network, we pre-configure static routing entries for the control-plane routing.

We implement Firepath in our testbed based on the available information in its paper [9]. Since Firepath has neither published enough details nor provided its implementation, we choose Yen's k-SPF algorithm [14] for its routing calculation, and Raft [17], [54] for its LSDB synchronization and master reelection, which are the most classic and widely used ones in practice. We also compare Primus with BGP in our testbed, using the BGP implementation of Quagga Routing Software Suite v1.2.4 [57]. The BGP routing advertisement timer and connection recovery timer are set to be 1s and 4s in all the rest experiments, respectively, which is based on private conversations with operators in charge of one of the largest production DCNs in China. We rely on Quagga's Linux interface monitoring scheme to detect local link failure in BGP. We do not compare with link-state protocols (*e.g.*, OSPF [48]) as they are not used in current large scale DCNs, because of high overhead to maintain and broadcast the whole network's link-states among all switches.

In Primus, three UDP redundancies are generated for each LS report and delivery. The heartbeat period and reelection timeout in our Raft among masters are 5s and 30s, respectively. Since Firepath has to sync LSDB when processing each LS, we set those two Raft timeouts to shorter values of 1s and 5s in Firepath, respectively. We have tried shorter timers, however, they caused Raft leader oscillation. 1s and 5s are the shortest values which are stable in our testbed. The base RTT in our testbed is less than 1ms on average and $\sim$10ms in tail, and the TCP minRTO is set to 60ms. Unless explicitly specified, all the rest experiments use above testbed and settings.

VI. Evaluation on Routing Processing

First, we evaluate the basic performance of routing processing: 1) We test the processing time and memory consumption on a real SONiC white-box switch when dealing with Primus routing in a very large network topology (§VI-A). 2) We test the overall routing processing time (including master and switch) under various network scales (§VI-B). For all experiments, we compare Primus with Firepath under the same topology scale (and same hardwares). We do not compare with BGP in this part since it is difficult to accurately emulate BGP's performance under large scale with only a few equipments.

[13]From the OS user's point of view, SONiC is almost the same as Linux except for some switch-specific network configurations. Our code can run both on Linux and SONiC.

[14]Linux can get those events from underlying detection schemes (*e.g.*, [44]).

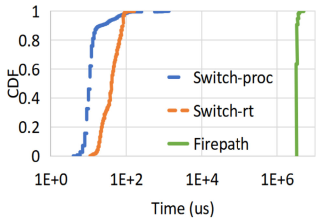


Fig. 4. Processing time in a switch, in a network having 10K ToRs, 400 Aggs and 16 Cores as shown in Fig. 1.

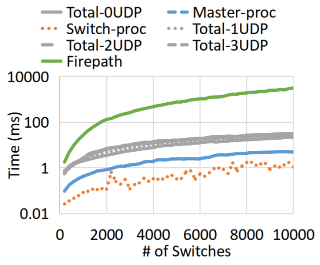


Fig. 5. Overall routing processing time as the network scale grows.

TABLE III

MEMORY CONSUMED IN A SWITCH, IN A NETWORK HAVING 10K TORS, 400 AGGS AND 16 CORES AS SHOWN IN FIG. 1

Link Table	Path Table
1.33MB	7.68MB

A. Switch Processing

Setup: We start a Primus switch process on a Ruijie physical switch, and install the link table (∼40K entries) and path table (∼160K entries) for the whole topology shown in Fig. 1 (10K ToRs, 400 Aggs and 16 Cores). We locally generate a random LS change to this switch for 10K times, and measure the routing processing time (from receiving the LS to updating SONiC kernel routing table entries).

Results: Fig. 4 shows the CDF of the switch's whole processing time in Primus ("Switch-proc") and in Firepath ("Firepath"), and the time for updating switch kernel routes in Primus ("Switch-rt"), for each LS change. Our smart table-lookup makes the whole processing time down to 11us in 50th percentile and 110us even in 99th percentile. The time for updating kernel routes in the physical switch is about 41us in 50th percentile and 92us in 99th percentile. However, due to high computation complexity in such large DCN topology (Fig. 1), it always takes more than 3s for routing calculation in Firepath,[15] which is $\sim10^4$-10^5 higher than Primus. Note that in Fig. 4 curve "Switch-rt" is higher than curve "Switch-proc" because not every LS change triggers a switch kernel route change. Particularly, due to our routing merge schemes (§III-B), Primus switch only changes a kernel route entry when all the paths of a next routing hop fail or a next routing hop has one path back after all its paths fail. Table III shows that the link table and path table only consume 1.33MB and 7.68MB memory, respectively, which can even fit into the caches of modern CPUs.

B. Overall Routing Processing

Setup: We connect two physical Dell servers directly together through two 25Gbps NIC ports. We run one Primus master process on one server machine, and multiple Primus switch processes on the other server machine to emulate multiple physical switches. The master process uses 9 sending threads and 2 receiving threads with each thread binding to a dedicated CPU core (hyper-threading disabled). Both the Primus master and switches are installed with the complete data structures for the whole topology shown in Fig. 1 (10K ToRs, 400 Aggs and 16 Cores). We vary the number of switch processes from 200 to 10K with step length of 200. At each step, one switch reports a random LS change to the master, and waits the master to deliver LS updates/receive acknowledgments/reply the response, for 10K times. For each LS, the master will deliver it to all the switch processes thus to evaluate the overhead of socket sending/receiving. Since all switch processes run on the same physical machine, only one switch process will actually perform the table-lookup calculation and change the Linux kernel routes. All other switch processes only directly replies an acknowledgment to the master after receiving an LS update. We evaluate: 1) the overall processing time (*from* the LS generated at the reporter switch *to* the master's response returned to the reporter switch), 2) the master processing time (*from* the LS received at the master *until* the master finishes sending all the LS updates through socket API), and 3) switch processing time (*from* the LS update received at the switch which calculates and changes routes *until* it finishes sending the acknowledgments to the master).

Results: Fig. 5 shows the average time of Primus's overall processing ("Total-xxx"), Primus's master processing ("Master-proc"), Primus's switch processing ("Switch-proc"), and Firepath's overall processing, as the number of switch processes grows. Each point is the average of 10K runs and Primus's master/switch processing time is evaluated when generating zero UDP redundancies. As the results show, even when controlling 10K switch processes, Primus' total routing processing time is only ∼22ms when we generate zero UDP redundancies ("Total-0UDP"), and only ∼26ms/29ms/32ms when generating 1/2/3 UDP redundancies[16] ("Total-1UDP/2UDP/3UDP") for each LS report/delivery, respectively. Compared to Firepath's results under 10K switches (which is close to the 4s time reported in their paper [9]), Primus's routing processing time is up to ∼148.3x (0UDP) and ∼98.8x (3UDP) faster. In Primus, for 10K switches, the master takes only about 5ms with most time spent in socket `send()/recv()` (detailed breakdown not in the figure), and the average processing time of the switch is only about 1ms. Note that there is a big gap between the overall processing time and the master/switch processing time. The gap is due to the processing time of all the 10K switch processes running on the same physical machine. We cannot accurately measure how much they contribute to the overall

[15]We note that in Firepath some LS changes (*e.g.*, upward links) can use data-plane routing recovery such as ECMP, and do not need to wait for control-plane routing calculation. §VII shows actual routing recovery time and this part only focuses on the routing calculation speed.

[16]In this speed testing experiment, we directly generate all the UDP redundancies to the one master process and it forwards UDP redundancies to switches, to emulate multiple backup masters.

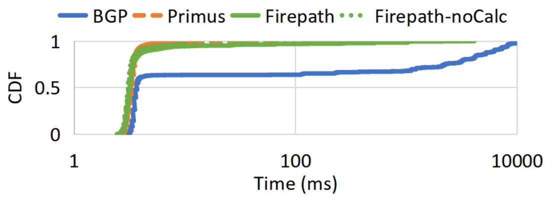


Fig. 6. Application's job completion time under random data-/control-plane failures (failure pattern derived from real measurements).

routing processing time since those processes run in parallel and independently.

VII. Evaluation on Routing Convergence

In this section, we evaluate Primus's performance on routing convergence, which includes two parts: *1)* Macro-benchmarks show how Primus's fast routing convergence can benefit the upper-layer applications (§VII-A). *2)* Micro-benchmarks examine in detail how long Primus reacts to network changes and how well Primus handles control-plane failures (§VII-B).

We install the whole large topology of Fig. 1 both in Primus's and Firepath's masters and switches, as such, although our testbed only contains 14 switches and 4 masters, the routing convergence time will be more close to the reality in the large topology. Since Firepath's routing calculation algorithm consumes significant amount of time under such large topology, we also evaluate a version of Firepath that only calculates route for our testbed's 14-switch topology (denoted as "Firepath-noCalc"). BGP runs directly on the 14-switch testbed and we do not manipulate its topology database scale.

A. Macro-Benchmark

Setup: We inject a partition-aggregate application to our testbed. Specifically, we start a server VM (under ToR 1.1) and three client VMs (under ToR 1.9801). For every second, the server round-robinly sends a small TCP single request to each of 3 clients and waits for a 2KB response from each client, which is a typical traffic pattern often existing in front-end data centers [7]. Meanwhile, we inject random link flapping failures to the network. Specifically, flapping failures (down and then up, each down time < 30ms) will randomly appear on each link (including control-plane links in Primus and Firepath). The time interval between flaps on each link obeys a log-normal distribution according to measurement results in [50], with the average interval being 100s. We measure the job completion time, which means all the 3 responses are received by the sender. The experiment is conducted 1000s.

Results: Fig. 6 shows the CDF of job completion time (JCT) in Primus, BGP and Firepath. Since Primus has much faster routing reaction time, the application is almost not affected by the link flaps. The 99_{th} and 99.5_{th} percentile JCT are only ∼9.4ms and ∼10.5ms, respectively, and the worst jobs are completed within ∼67ms (encounters one TCP RTO). In Firepath, the 99_{th} and 99.5_{th} percentile JCT are ∼1.02s and ∼1.05s, respectively, and the worst case is ∼4.1s, which is ∼100x slower than Primus. This is due to the long connectivity loss caused by the slow routing reaction (slow routing calculation and LSDB sync between masters).

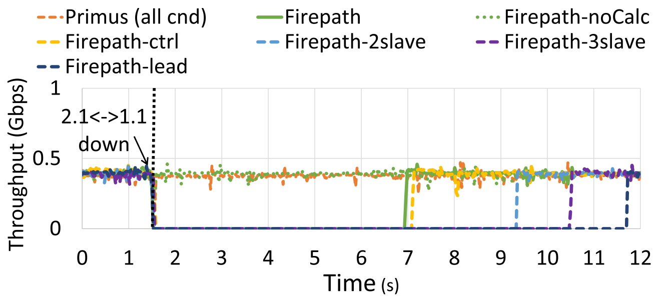


Fig. 7. Routing recovery time upon network changes. Primus' throughput never drops to zero (only to ∼10Mbps in one time bin) and recovers immediately, but others experience obvious recovery time except Firepath-noCalc. "-ctrl" denotes reporter switch's control-plane link failure. "-2slave" and "-3slave" denote 2 or 3 backup controllers' control-plane link failure. "-lead" denotes lead master failure.

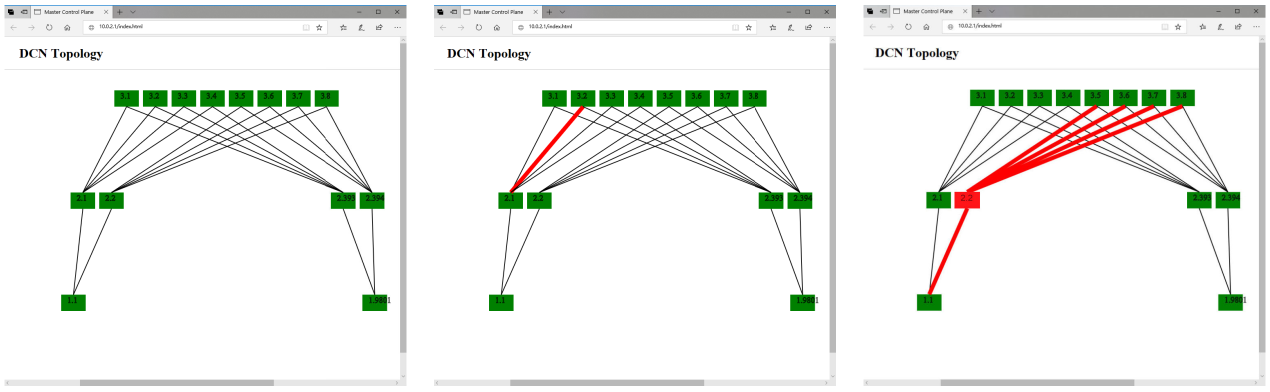
Even when Firepath has little routing calculation overhead ("Firepath-noCalc"), the 99.5_{th} percentile JCT is still above 1s due to its control-plane overhead (more details in §VII-B.1). In BGP, the JCT is ∼11.9s in 99_{th} percentile and ∼12.9s in 99.5_{th} percentile, which is ∼1226x higher than Primus.

B. Micro-Benchmark

1) Routing Reaction Time Upon Network Changes: Since there is no precise global clock among switches in our testbed, we evaluate the accurate routing reaction time by monitoring data transmission throughput. Specifically, we inject a UDP flow sending infinite data at average rate 400Mbps from a host below switch 1.9801 to a host below switch 1.1, going through path 1.9801→2.393→3.1→2.1→1.1. During data transmission, at moment 1, we tear down link 2.1↔1.1. We measure the real-time receiving throughput at the receiver in the time bin of 30ms, to see how long it takes for the routing protocol to react. We also inject various control-plane failures simultaneously with link 2.1↔1.1 failure. Specifically, we generate transient failure to switch 2.1's control-plane link ("-ctrl"), 2 or 3 backup controllers' control-plane links ("-2slave" and "-3slave"), and lead master ("-lead").

Fig. 7 shows the throughput dynamics of the UDP flow. After link 2.1↔1.1 down, it takes about 5.4s for Firepath to recover the routing connectivity. This is mainly due to its slow k-SPF algorithm (since Firepath-noCalc is almost not affected by this single data-plane failure). The throughput in Primus is similar to Firepath-noCalc, which never drops to zero (only one time bin drops to ∼10Mbps) in the time bin of 30ms. Note that counting UDP throughput in such small time bins is not so accurate which may be affected by various factors such as OS scheduling, but the levels are valid.

When we inject simultaneous control-plane failures, Firepath takes more time to recover the routing connectivity. Particularly, Firepath-ctrl takes about 100ms more, since it encounters a TCP RTO because the reporting LS is dropped by switch 2.1's control-plane link failure. Firepath-2slave and Firepath-3slave takes ∼2-3.5s more, because backup controllers' link failure blocks LSDB synchronization, and lead master takes one or more heartbeat timeouts to finally replicate the LSDB before it can disseminate the LS change. For lead failure, it takes even more time (about 5s) for lead



(a) Normal. (b) Link 2.1↔3.2 down. (c) Switch 2.2 down.

Fig. 8. Snapshots of the routing fault localization tool.

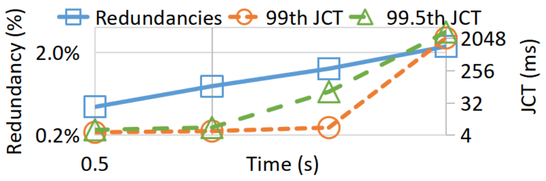
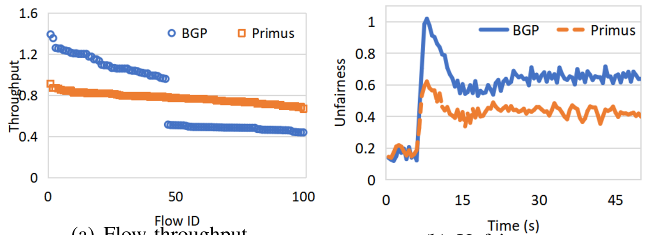


Fig. 9. Fraction of LSes received from redundancies and the app's tail job completion time, as the control-plane failures become severer.



(a) Flow throughput. (b) Unfairness.

Fig. 10. Dynamic WCMP.

reelection in Firepath. However, Primus will tolerate all these kinds of control-plane failures, keeping routing recovery time within ms level benefited from its redundancy schemes. Since the results are similar for Primus under all conditions, we do not draw them on the figure and use "Primus (all cnd)" to represent all these cases for simplicity.

2) *Anatomy of Primus's Redundancy Efficiency:* We conduct the same experiment as in §VII-A, but increase the lasting time for each control-plane link failure to evaluate the efficiency of Primus's LS redundancy schemes. Fig. 9 shows the fraction of LSes first received from UDP redundancies (among all the LSes received from TCP main channels and UDP redundancies) and the application's 99_{th} and 99.5_{th} percentile job completion time (JCT), as the control-plane failures lasting time grows from 0.5s to 4s, respectively. Results show that the fraction of LSes got from UDP redundancies (before TCP main channels) grows from ∼0.4% to ∼2.4%, as the control-plane network failures become severer. This helps Primus to maintain the 99_{th} and 99.5_{th} percentile JCT under 10ms and 70ms when control-plane link failure lasts for 2s, which are significantly better than Firepath's results even when its failures only last for <30ms. Even when the failure lasting time grows to 4s in such small network, Primus keeps the 99_{th} and 99.5_{th} percentile JCT under 2.2s and 3.2s.

VIII. Evaluation on Routing Controllability

In this section, we build the two example applications described before to show the benefits of Primus's fine-grained control and centralized view to the whole network's routing.

A. Routing Failure Visualization

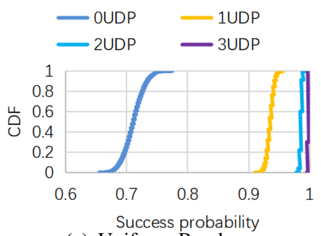
First, we build a simple network management tool to help operators to visualize routing problems. Specifically, we build an HTML5 web page using Canvas element [58] to visualize the current status of the whole network's link-state. The current link-state is fetched from the Primus master node using JavaScript in a real-time manner, with a periodical fetching interval of 3 seconds. Fig. 8 shows the snapshot of our visualized pages for three example cases, *i.e.*, normal, link 2.1↔3.2 failure, and switch 2.2 failure, respectively (red color refers to failure).

All path failure information has already been collected. Compared to other tools, failure localization tools built on primus only need to visualize this information.

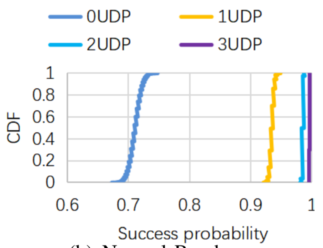
B. Dynamic WCMP

Next, leveraging the master's centralized view, we extend our Primus routing with WCMP, which dynamically adjusts each path's ECMP weight based on its current bandwidth capacity. In this experiment, we change the topology of our testbed a little, enabling ToR switch 1.2 and 1.9802, and removing Core switch 3.3, 3.4, 3.7, 3.8, thus to form a full-bisection bandwidth network with no over subscription. ToR 1.1 and 1.2 both have two clients below them, respectively (similarly, ToR 1.9801 and 1.9802 each have two servers below them). We inject 25 long TCP flows sending data from each client to each server (100 flows in total between the four pairs of client/server). During data transmission, we tear down link 3.1↔2.1 to evaluate the performance of WCMP.

Fig. 10(a) shows the throughput of each flow (in descending order) after routing converged after link failures. The throughput is normalized to the fair-share throughput of each flow when no failures happen. Since BGP has no dynamic WCMP, about half of the flows going through switch 2.1 share only one upward link after link 3.1↔2.1 fails, so they only



(a) Uniform Random.



(b) Normal Random.

Fig. 11. Probability of successfully collecting an LS and delivering it to all the affected switches, under different network failure distribution.

get about half of the throughput. However, in Primus, after the link fails, the master recalculates the path weights and set switch 1.1 and 1.2 to spread flows to the second-layer Agg switches with weights of 1:2. As such, all the flows fairly share the three working upward links of switch 2.1 and 2.2, so each flow gets about 3/4 throughput. Fig. 10(b) shows the dynamics of the unfairness between flows before/during/after the link failure event. The unfairness is defined as the ratio of the standard deviation to the average of all flows' throughput (measured in 0.5s time bin). Results show that Primus can quickly calculate the right WCMP weights and route flows accordingly after link failure, so the unfairness is much lower than in BGP, whose paths still use the unbalanced ECMP weights before. Note that the unfairness both in Primus and BGP soars up temporarily after the failure because the throughput of some flows traversing on the failed link drops to zero before the routing recalculation finishes. The unfairness after failure is higher than normal state even we spread flows using WCMP, because the same number of flows share a less number of links, so the competition is severer and TCP throughput is not as stable as before.

IX. Evaluation on Primus' Overhead

In this section, we evaluate the efficiency of the LS redundancy scheme based on the theoretical model derived before through simulation, and evaluate the efficiency of topology update scheme through testbed experiments. The overheads of UDP mechanism and dynamic WCMP are evaluated. All experiment settings are the same as in Sec. VI-B. Time bin in Sec. IX-D is 30ms.

A. Effectiveness of the LS Redundancy

We evaluate the robustness of the LS redundancy system with 0/1/2/3 UDP backup channel(s), respectively. We assume that each link/switch has a fixed failure probability, whose value randomly distributes between 0.001% and 1%.[17] We evaluate two failure distributions, *i.e.*, uniform random and normal random.

Fig. 11 shows the results of running 10K times (in CDF) in a 10K-switch topology as shown in Fig. 1, under uniform random and normal random distribution, respectively. When using 3-UDP backup channels, the mean success probabilities are 99.668% and 99.667%, which means the there is only about 0.333% possibility that a reporter switch has to retransmit the LS upon it detects a local link change.

[17]The actual failure probability would be lower in reality [50].

TABLE IV
Bandwidth Usage of LS Redundancy

	1UDP	2UDP	3UDP
Mean(kb/s)	34.68	67.95	101.03

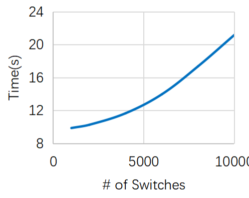


Fig. 12. Time to update base topology for networks with different sizes.

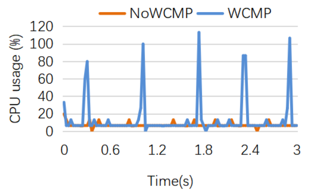


Fig. 13. CPU usage of Dynamic WCMP.

B. Bandwidth Usage of LS Redundancy

Here, we evaluate the traffic comes from the LS redundancy system with 0/1/2/3 UDP backup channel(s), respectively. We assume that the failure probability of link/switch follows a uniform random distribution, which is valued from 0.001% and 1%.(*i.e.*,up to 1000 links/switches fail per day in 10K+ topology). Table. IV shows the results. Even with 3UDP redundant channels, the average new traffic in the network is only 101.03KB per second.

C. Base Topology Update

We evaluate the time to initialize (update) the base topology in this section. Specifically, starting from the network that only contains 1000 switches (10 pods), we gradually add switches to the complete topology with 1000 switches (100 pods). We measure the time that it takes to update the base topology for all the switches in the network. Fig. 12 shows the results. It only takes about 10s to 22s to update the base topology for the whole network.

D. CPU Usage of Dynamic WCMP

We let the leader periodically send link changes to a switch running dynamic WCMP and measure the real-time CPU utilization. Fig. 13 shows the results. The results show that when running WCMP, the highest instantaneous CPU usage is 113.7%, which is ∼16.9x as not using WCMP.

X. Conclusion

We presented Primus, a centralized DCN routing protocol and system. Leveraging the regular DCN topologies, Primus simplifies the routing into centralized link-state management and simple table-lookup routing calculation. Moreover, through low-cost control-plane fault-tolerant schemes, Primus can keep very good performance even under complex control-plane failures. We made Primus's implementation publicly available. Testbed experiments show that Primus can significantly improve the routing convergence time, being ∼1200x and ∼100x faster than BGP and the state-of-the-art centralized routing solution Firepath, respectively. Moreover, Primus also keeps high routing controllability/manageability which can enable various advanced routing scenarios.

There are still some unresolved questions in this paper. For example, we discuss Primus for other topologies such as VL2 and BCube, but how can these topologies use the information collected by Primus to realize different routing policies such as non-shortest path routing? These are valuable problems that worth further studying.

REFERENCES

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

[2] A. Greenberg, J. R. Hamilton, N. Jain, and S. Kandula, "VL2: A scalable and flexible data center network," in *Proc. SIGCOMM*, 2009, pp. 51–62.

[3] G. D. Dutt, *BGP in the Data Center*. Sebastopol, CA, USA: O'Reilly Media, 2017.

[4] R. B. da Silva and E. S. Mota, "A survey on approaches to reduce BGP interdomain routing convergence delay on the internet," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2949–2984, 4th Quart., 2017.

[5] M. Yannuzzi, X. Masip-Bruin, and O. Bonaventure, "Open issues in interdomain routing: A survey," *IEEE Netw.*, vol. 19, no. 6, pp. 49–56, Nov. 2005.

[6] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, "Aspen trees: Balancing data center fault tolerance, scalability and cost," in *Proc. 9th ACM Conf. Emerg. Netw. Experiments Technol.*, Dec. 2013, pp. 85–96.

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

[8] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *Proc. NSDI*, 2013, pp. 113–126.

[9] 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.

[10] A. Greenberg et al., "A clean slate 4D approach to network control and management," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 5, pp. 41–54, Oct. 2005.

[11] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. V. D. Merwe, "Design and implementation of a routing control platform," in *Proc. NSDI*, 2005, pp. 15–28.

[12] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun.*, Aug. 2007, pp. 1–12.

[13] A. D. Ferguson et al., "Orion: Google's software-defined networking control plane," in *Proc. 18th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, pp. 83–98, 2021.

[14] J. Y. Yen, "Finding the K shortest loopless paths in a network," *Manage. Sci.*, vol. 17, no. 11, pp. 712–716, Jul. 1971.

[15] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[16] *OSPF Incremental SPF—Cisco*. Accessed: Dec. 11, 2019. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_ospf/configuration/15-sy/iro-15-sy-book/iro-incre-spf.pdf

[17] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX ATC*, 2014, pp. 305–319.

[18] *Primus Code Base*. Accessed: Apr. 20, 2020. [Online]. Available: https://github.com/GuihuaZhou/PrimusCode2.0

[19] *Ruijie Bare Metal Switches, B6510-48VS8CQ Switch*. Accessed: Dec. 15, 2019. [Online]. Available: https://www.ruijienetworks.com/products/switches/bare-metal-switches/b6510-48vs8cq-switch

[20] G. Zhou et al., "Primus: Fast and robust centralized routing for large-scale data center networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2021, pp. 1–10.

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

[22] T. Koponen et al., "Network virtualization in multi-tenant datacenters," in *Proc. NSDI*, 2014, pp. 1–15.

[23] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proc. NSDI*, 2010, pp. 1–15.

[24] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," in *Proc. SIGCOMM*, 2015, pp. 307–318.

[25] J. Zhou et al., "WCMP: Weighted cost multipathing for improved fairness in data centers," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, pp. 1–14.

[26] N. Gude et al., "NOX: Towards an operating system for networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[27] T. Koponen, M. Casado, N. Gude, and J. Stribling, "Distributed control platform for large-scale production networks," U.S. Patent 8 830 823, Sep. 9, 2014.

[28] *Open/R: Open Routing for Modern Networks*. Accessed: Aug. 23, 2018. [Online]. Available: https://engineering.fb.com/connectivity/open-r-open-routing-for-modern-networks/

[29] D. Pei et al., "Improving BGP convergence through consistency assertions," in *Proc. 21st Annu. Joint Conf. IEEE Comput. Commun. Societies*, Jun. 2002, pp. 1–11.

[30] 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.

[31] A. Bremler-Barr, Y. Afek, and S. Schwarz, "Improved BGP convergence via ghost flushing," in *Proc. 22nd Annu. Joint Conf. IEEE Comput. Commun. Societies (INFOCOM)*, Mar. 2003, pp. 1–11.

[32] A. Abhashkumar and K. Subramanian, "Running BGP in data centers at scale," in *Proc. NSDI*, 2021, pp. 65–81.

[33] A. T. Atlas Przygienda, A. Sharma, and J. Drake, *RIFT Routing in Fat Trees*, document RFC draft-przygienda-rift-05, 2018.

[34] A. Atlas and A. Zinin, *Basic Specification for IP Fast Reroute: Loop-Free Alternates*, document RFC 5286, 2008.

[35] S. Bryant, C. Filsfils, S. Previdi, M. Shand, and N. So, *Remote Loop-Free Alternate (LFA) Fast Reroute (FRR)*, document RFC 7490, 2015.

[36] G. Enyedi, A. Csaszar, A. Atlas, C. Bowers, and A. Gopalan, *An Algorithm for Computing IP/LDP Fast Reroute Using Maximally Redundant Trees (MRT-FRR)*, document RFC 7811, 2016.

[37] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *Proc. NSDI*, 2019, pp. 161–176.

[38] G. Chen, Y. Zhao, H. Xu, D. Pei, and D. Li, "F^{2}Tree: Rapid failure recovery for routing in production data center networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 1940–1953, Aug. 2017.

[39] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, "R-BGP: Staying connected in a connected world," in *Proc. NSDI*, 2007, pp. 1–14.

[40] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever, "SWIFT: Predictive fast reroute," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 1–14.

[41] C. Y. Hong, S. Kandula, R. Mahajan, and M. Zhang, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 1–12.

[42] S. Jain, A. Kumar, S. Mandal, and J. Ong, "B4: Experience with a globally-deployed software defined WAN," in *Proc. SIGCOMM*, 2013, pp. 3–14.

[43] C.-Y. Hong et al., "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN," in *Proc. SIGCOMM*, 2018, pp. 1–9.

[44] D. Katz and D. Ward, *Bidirectional Forwarding Detection (BFD)*, document RFC-5880, 2010.

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

[46] C. Guo et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA: ACM, 2015, pp. 63–74.

[47] C. E. Hopps, *Analysis of An Equal-Cost Multi-Path Algorithm*, Internet Engineering Task Force, document RFC 2992, 2000.

[48] J. Moy, *OSPF Version 2*, document RFC 2328, 1998.

[49] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proc. 1st ACM Workshop Res. Enterprise Netw.* New York, NY, USA: ACM, Aug. 2009, pp. 73–82.

[50] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 350–361, 2011.

[51] Y. Zhu et al., "Congestion control for large-scale RDMA deployments," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 1–14.

[52] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, and Y. Shi, "BCube: A high performance, server-centric network architecture for modular data centers," in *Proc. SIGCOMM*, 2009, pp. 63–74.

[53] *SONiC: Software for Open Networking in the Cloud*. Accessed: Oct. 8, 2018. [Online]. Available: https://azure.github.io/SONiC/

[54] *QIHOO360's Implementation of the Raft Consensus Protocol*. Accessed: Apr. 13, 2020. [Online]. Available: https://github.com/Qihoo360/floyd

[55] *DPDK: Data Plane Development Kit*. Accessed: Feb. 9, 2019. [Online]. Available: https://www.dpdk.org/

[56] *Virtual Route Forwarding Design Guide—Cisco*. Accessed: Feb. 20, 2019. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucme/vrf/design/guide/vrfDesignGuide.html

[57] *Quagga Routing Suite*. Accessed: Mar. 3, 2019. [Online]. Available: http://www.nongnu.org/quagga/

[58] *HTML5 Canvas*. Accessed: Dec. 3, 2019. [Online]. Available: https://www.w3schools.com/html/html5_canvas.asp

Fusheng Lin received the master's degree from Hunan University, in 2022. He is currently with Tencent. His research interests include computer networking and networked systems.

Hongyu Wang received the B.S. degree from Guangxi University, China, in 2020. He is currently pursuing the master's degree with Hunan University, China. His research interests include computer networking.

Guo Chen (Member, IEEE) received the Ph.D. degree from Tsinghua University in 2016. He was a Researcher with Microsoft Research Asia from 2016 to 2018. He is currently a Professor with Hunan University. His current research interests include networked systems and with a special focus on data center networking.

Guihua Zhou received the B.S. degree from Xiangtan University, China, in 2018, and the M.D. degree from Hunan University, China, in 2021. He is currently a Software Engineer with Tencent. His research interests include data center networking.

Tingting Xu (Student Member, IEEE) received the B.E. degree from Hunan University, Hunan, China, in 2019. She is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Nanjing University, under the supervision of Prof. Xiaoliang Wang. Her research interests include programmable networks, data center networks, and network function virtualization.

Dehui Wei received the B.E. degree (Hons.) in computer science and technology from Hunan University, Changsha, China, in 2019. She is currently pursuing the Ph.D. degree with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications (BUPT). Her research interests include network transmission control and cloud computing.

Li Chen (Member, IEEE) received the B.E. degree (Hons.) in electronic and computer engineering with a minor in mathematics and the M.Phil. degree from The Hong Kong University of Science and Technology (HKUST) in 2011 and 2013, respectively. He is currently with the Zhongguancun Laboratory working on topics in systems, networking, and cybersecurity research.

Yuanwei Lu received the joint Ph.D. degree from the University of Science and Technology of China and Microsoft Research Asia in 2018. His research interests include data center networking and networked systems.

Andrew Qu is a highly experienced Network System Architect with over 20 years of expertise in designing and implementing distributed network systems, high-performance network ASIC, and hyper-scale data center network architectures. He has served as the Senior Leader for multiple technology leading companies like: Cisco, Tencent, Huawei, and Intel. He is currently the Senior Director of data center solution architecture.

Hua Shao received the doctor's degree from Tsinghua in 2022. He has over 20 years of experiences in hyperscale infrastructure research and development. He was with Tencent from 2013 to 2020 and was the Director of the Network Infrastructure Center, where he worked on network architecture and network systems. Since 2020, he has been the Head of the Infrastructure with Pinduoduo, responsible for design, development and operation of data centers, servers, and networking.

Hongbo Jiang (Senior Member, IEEE) received the Ph.D. degree from Case Western Reserve University in 2008. He was a Professor with the Huazhong University of Science and Technology. He is currently a Full Professor with the College of Computer Science and Electronic Engineering, Hunan University. His research interests include computer networking, especially algorithms and protocols for wireless and mobile networks.