Primus: Fast and Robust Centralized Routing for Large-scale Data Center Networks

Guihua Zhou[1], Guo Chen[1*], Fusheng Lin[1], Tingting Xu[1], Dehui Wei[1], Jianbing Wu[1],
Li Chen[3], Yuanwei Lu[2], Andrew Qu[2], Hua Shao[4], Hongbo Jiang[1]
[1]*Hunan University,* [2]*Tencent,* [3]*Huawei,* [4]*Tsinghua University*

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488689

***Abstract*—This paper presents a fast and robust centralized data center network (DCN) routing solution called Primus. For fast routing calculation, Primus uses centralized controller to collect/disseminates 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 ∼1200x and ∼100x shorter convergence time than current distributed protocol BGP and the state-of-the-art centralized routing solution, respectively.**

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]–[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], [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 are 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 (possibly) the first and only published work that has successfully designed/implemented/operated a full routing protocol and system using centralized control for DCNs (other works are not a full routing protocol. 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 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?** Apparently, using a centralized controller to directly calculate the whole network's routing will be slow and not scalable. 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 using shortest-path first (SPF) algorithms. 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 $O(kn(m+n\log n)$ [13][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 time of routing connectivity loss to a rack of servers in Firepath (Table 4 in [9]).
- **How to gracefully handle control-plane failure?** In case of controller failure, Firepath runs multiple backup controllers, each maintaining an LSDB of the whole

*Corresponding author.

[1]We note that there exist some optimizations to the k-SPF algorithm (*e.g.*, [14], [15]). However, their calculation time still grows fast as the topology scales larger, which is undesired for scalability.

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.*, [16]) 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 norm in the control-plane network, since there are 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 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 does not really "calculate" the routes. 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 masters and little redundant traffic to tolerate control-plane failures**, which incurs low overhead in normal, meanwhile achieves 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 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 [17]). Our Primus master implementation runs on Linux machines, and Primus switch implementation can run both on Ruijie white-box switches [18] 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 has three orders of magnitude better 99_{th} and 99.5_{th} percentile performance compared to those using BGP, and ∼100x better compared to Firepath, respectively. We believe that Primus has set a new performance milestone for building centralized routing for large scale DCNs.

II. Related Work

Other centralized routing control: Besides Firepath, there are also previous works (*e.g.*, [2], [19]–[26]) using centralized control to address part of the routing problems in DCN. However, they are *not complete routing protocols*. For example, [21]–[23] use centralized controller 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.*, [22] is implemented on top of BGP). [19] takes the same idea of using centralized controller 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, [19] did not address problems discussed in §I-B. [2], [20] utilize centralized control to translate between physical and virtual addresses for network virtualization. [24]–[26] build underlying system (*e.g.*, switch 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.

Improved distributed routing protocol: Many works, *e.g.*, [27]–[29], try to improve BGP using various techniques. Although big improvements have been achieved, the intrinsic

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

[3]"Master" and "controller" are exchangeably used 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 §III-E for details.

drawbacks in distributed routing still remain to be open questions (*e.g.*, convergence still requires seconds to minutes) [4], [28]. RIFT [30] utilizes the preknowledge of the DCN fat tree topology to simplify the routing and limit the broadcasting area. However, staying as a distributed protocol, 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]–[8], [31]–[37]) aiming to provide fast data-plane connectivity recovery before routing convergence. Fast rerouting (FRR) techniques (*e.g.*, [31]–[33]) 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]–[8], [35] focus on fast data-plane recovery in DCN. However, they either require significant changes to physical topology [6], [35], or may incur temporary routing loops or use non-shortest bounce-back paths [7], [8], neither are desired in DCN. 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.*, [38]–[40]), which is different from intra-DCN environment.

III. Primus Design

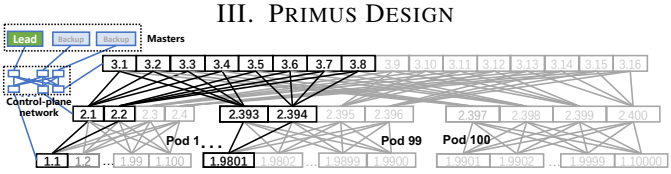


Fig. 1. Primus over an example DCN.

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 [41]) 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 [7].

Each switch is preconfigured 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 uniquely ascending ID per link. Whenever receiving a new LS, the master will deliver the LS to all the possibly affected switches 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).

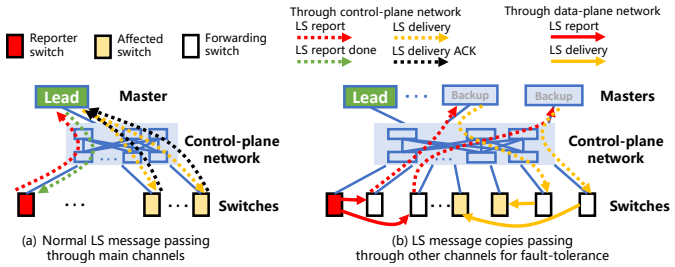


Fig. 2. Primus's fault-tolerant link-state updating scheme.

Next, we introduce in details how Primus's design achieves *1) quick routing calculation (§III-B), and 2) efficient fault-tolerance (§III-C)*, 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], [19], [42]. 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.

[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.

[7]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 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.

Note that Primus also works for fat tree with more layers, and can be easily adapted to other topologies (see §III-E).

TABLE I
SNAPSHOT OF PATH 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	...

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 §III-E). 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 §V-A). Note that this table is an internal data structure used by Primus routing calculation 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 precalculate 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.*, 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 $k \times (p-1)$ 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 which 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 §V-A).

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. If we organize the entries in the path table in the order shown in Table I (*i.e.*, in the order by the first hop, then by the destination, then by the second hop), *Type 1: 1* indicates that link 1.1→2.1 affects the continuous 40000 ($4 \times 100 \times 100$) entries starting from entry 1; *Type 2: 401* indicates that link 2.1→3.1 affects in total 9900 (100×99) entries, which includes the first of every 4 entries starting from entry 401.

C. Handling Control-Plane Failure

The reliable LS report scheme described in §III-A ensures the network have an eventual *consistent view* to the latest LS change, without relying on states maintained on 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.

[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.

Slight redundancy for speed: We use multiple backup masters. As Fig. 2 shows, there is one lead master, which is active to collect/disseminate all LS messages from main channels, and multiple backup masters, which work as hot standbys and process redundant LS messages. Each switch is preconfigured with the static address of each backup master.[10] 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)). Those 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 transfered 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 forms 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 responds 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: We detect the main channel status through periodical hellos. With 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 [16]) 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 §III-E). Note that the consensus protocol is decoupled with normal routing reaction upon LS changes, so it can run in a very low frequency, incurring low overhead.

D. 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 preconfigured with the base DCN topology and all other masters' static addresses. When a master initializes or reboots from 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

[10]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.

[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.

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 in low-cost, we simply use existing distributed protocols (*e.g.*, OSPF [43]) for routing in the control-plane network.

Centralized routing controllability/manageability. Benefited from the table data structures listing all the routing paths, it is natural and easy to control/manage each switchs routing by manipulating its path entries (*e.g.*, customizing path weights). Also, it is convenient to visualize network failures according to collected global LSes, and easy to find wiring error based on location ID reported by each switch. We have built several centralized control/management applications in our testbed (details omitted due to space limitation).

E. Discussions and Limitations

Master state loss. Since we use stateless master to achieve fast routing reaction upon complex failures, it may loose global LSes (although not likely). This may temporarily affect above centralized control/management functionalities (but not routing correctness). However, 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 normal network routing reaction to reduce overhead. It is our future work to make such advanced functionalities react fast to network changes similarly as normal routing processing, which is out of this paper's scope.

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 physically only non-shortest paths exist (*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 at 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 [44] traffic to the master, but neither of them will cause performance issue: 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 [45]), 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 [46]) 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 thousands of switches (experiments in §V-B).

Primus for other topologies. Since the basic idea of table-lookup routing calculation is based on fixed routing change patterns in preknown topology, it is easy to adapt Primus to fat-tree with more layers and other topologies (*e.g.*, cube topologies [47]), because we also can extract predefined routing change rules. Details on these topologies are beyond the scope of this paper. Primus's table-lookup routing calculation may not be applicable to DCNs using random topology without regularities (*e.g.*, [48]).

IV. Implementation and Testbed Setup

A. Primus Implementation

We have a complete implementation of Primus with 3128 lines of C++ code (available at [17]), based on Linux-machine masters/switches and Ruijie white-box switches [18] with SONiC [49] 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

[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.*, [41]).

on `epoll` event loop. The election protocol between masters is based on an existing implementation of Raft [50]. Although we have not used high-performance networking stacks (*e.g.*, DPDK [51]) for now, the current Primus implementation offers performance good enough even for a very large network (results in §V). 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) [52]. All the VM switches are connected through virtual switches with 1Gbps links, hosted in 4 Dell R720XD physical servers (Intel Xeon CPU E5-2620, 96GB memory). Ruijie switches are connected through 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).

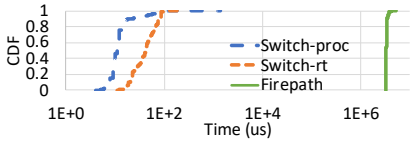
We implement Firepath in our testbed based on the available information in its paper [9]. Since Firepath has neither published enough details nor provided its implementation, we choose Yen's k-SPF algorithm [13] for its routing calculation, and Raft [16], [50] for its LSDB synchronization and master 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 [53]. 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 [43]) 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 osculation. 1s and 5s are the shortest value which are stable in our testbed. The base RTT in our testbed is less than 1ms in average and ∼10ms in tail, and the TCP minRTO is set to 60ms. Unless explicitly specified, all the rest experiments use above testbed and settings.

V. 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 (§V-A). 2) We test the overall routing processing time (including master and switch) under various network scales (§V-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.

A. Switch Processing



(a) Processing time.

Memory consumption	
Link Table	1.33 MB
Path Table	7.68 MB

(b) Memory consumed.

Fig. 3. Processing time and memory consumed in a switch, in a network having 10K ToRs, 400 Aggs and 16 Cores as shown in Fig. 1.

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. 3(a) 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 $\sim 10^4$-10^5 higher than Primus. Note that in Fig. 3(a) 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. Fig. 3(b) 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

[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. §VI shows actual routing recovery time and this part only focuses on the routing calculation speed.

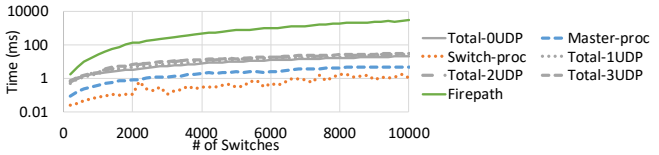


Fig. 4. Overall routing processing time as the network scale grows.

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 a 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. 4 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. Eeach 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 routing processing time since those processes run in parallel and independently.

[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.

VI. 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 (§VI-A). 2) Micro-benchmarks detailedly examine Primus's reaction time to network changes and how well Primus handles control-plane failures (§VI-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 [45], 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.

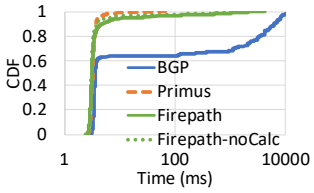


Fig. 5. Application's job completion time under random data-/control-plane failures (failure pattern derived from real measurements).

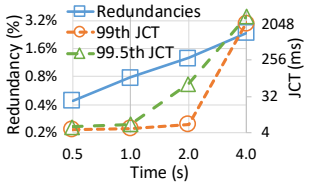


Fig. 6. Fraction of LSes received from redundancies and the app's tail job completion time, as the control-plane failures become severer.

Results: Fig. 5 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

$\sim$1.05s, respectively, and the worst case is $\sim$4.1s, which is $\sim$100x slower than Primus. This is due to the long connectivity loss caused by slow routing reaction (slow routing calculation and LSDB sync between masters). 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 §VI-B1). In BGP, the JCT is $\sim$11.9s in 99_{th} percentile and $\sim$12.9s in 99.5_{th} percentile, which is $\sim$1226x higher than Primus.

B. Micro-benchmark

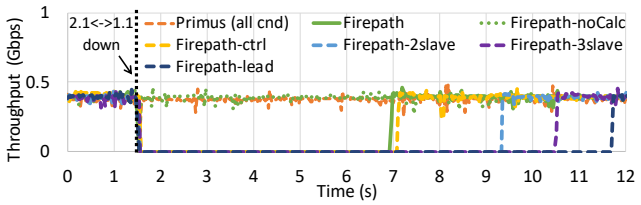


Fig. 7. Routing recovery time upon network changes. We also inject various control-plane failures along with the data-link failure. "-ctrl" denotes reporter switch's control-plane link failure. "-2slave" and "-2slave" denote 2 or 3 slave masters' control-plane link failure. "-lead" denotes lead master failure.

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 slave masters' control-plane links ("-2slave" and "-2slave"), 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 $\sim$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 $\sim$2-3.5s more, because slave masters' 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 reelection in Firepath. However, Primus well 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 §VI-A, but increase the lasting time for each control-plane link failure to evaluate the efficiency of Primus's LS redundancy schemes. Fig. 6 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 $\sim$0.4% to $\sim$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.

VII. 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 $\sim$1200x and $\sim$100x faster than BGP and the state-of-the-art centralized routing solution Firepath, respectively. As future works, we plan to explore more advanced centralized control applications (*e.g.*, application-aware DCN routing) based on Primus.

Acknowledgment

We thank Kai Chen, Xiaoliang Wang, Dan Pei and the anonymous INFOCOM reviewers for their helpful comments on improving this paper. This research was partially funded by the National Natural Science Foundation of China (No.61872132), the Fundamental Research Funds for the Central Universities, Training Program for Excellent Young Innovators of Changsha, Tencent and Huawei.

References

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

[2] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[3] Dinesh G. Dutt. *BGP in the Data Center*. O'Reilly Media, Inc., 2017.

[4] Ricardo Bennesby da Silva and Edjard Souza Mota. A survey on approaches to reduce BGP interdomain routing convergence delay on the Internet. *IEEE Communications Surveys & Tutorials*, 2017.

[5] Marcelo Yannuzzi, Xavier Masip Bruin, and Olivier Bonaventure. Open issues in interdomain routing: a survey. *IEEE network*, 2005.

[6] Meg Walraed-Sullivan, Amin Vahdat, and Keith Marzullo. Aspen Trees: Balancing Data Center Fault Tolerance, Scalability and Cost. In *CoNEXT*, 2013.

[7] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *NSDI*, 2013.

[8] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring Connectivity via Data Plane Mechanisms. In *NSDI*, 2013.

[9] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hoelzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[10] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM*, 2005.

[11] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus Van Der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, 2005.

[12] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[13] Jin Y Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 1971.

[14] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 1987.

[15] OSPF Incremental SPF - Cisco. 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.

[16] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, 2014.

[17] Primus Code Base. https://github.com/GuihuaZhou/PrimusCode2.0.

[18] Ruijie Bare Metal Switches, B6510-48VS8CQ Switch. https://www.ruijienetworks.com/products/switches/bare-metal-switches/b6510-48vs8cq-switch.

[19] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.

[20] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[21] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[22] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. *SIGCOMM*, 2015.

[23] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.

[24] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM*, 2008.

[25] Teemu Koponen, Martin Casado, Natasha Gude, and Jeremy Stribling. Distributed control platform for large-scale production networks, 2014. US Patent 8,830,823.

[26] Open/R: Open routing for modern networks. https://engineering.fb.com/connectivity/open-r-open-routing-for-modern-networks/.

[27] D. Pei, X. Zhao, L.Wang, D. Massey, A. Mankin, S. F.Wu, and L. Zhang. Improving BGP convergence through consistency assertions. In *IEEE INFOCOM*, 2002.

[28] Alex Fabrikant, Umar Syed, and Jennifer Rexford. There's something about MRAI: Timing diversity can exponentially worsen BGP convergence. In *INFOCOM*, 2011.

[29] Anat Bremler-Barr, Yehuda Afek, and Shemer Schwarz. Improved BGP convergence via ghost flushing. In *INFOCOM*, 2003.

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

[31] A Atlas and A Zinin. Basic specification for IP fast reroute: loop-free alternates. *RFC 5286*, 2008.

[32] Stewart Bryant, Clarence Filsfils, Stefano Previdi, Mike Shand, and Ning So. Remote loop-free alternate (LFA) fast reroute (FRR). *RFC 7490*, 2015.

[33] 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). *RFC 7811*, 2016.

[34] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *NSDI*, 2019.

[35] Guo Chen, Youjian Zhao, Hailiang Xu, Dan Pei, and Dan Li. F^2Tree: Rapid Failure Recovery for Routing in Production Data Center Networks. *IEEE/ACM Transactions on Networking (TON)*, 2017.

[36] Nate Kushman, Srikanth Kandula, Dina Katabi, and Bruce M Maggs. R-BGP: Staying connected in a connected world. *NSDI*, 2007.

[37] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. Swift: Predictive fast reroute. In *SIGCOMM*, 2017.

[38] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *SIGCOMM*, 2013.

[39] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[40] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *SIGCOMM*, 2018.

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

[42] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[43] John Moy. OSPF Version 2. *RFC 2328*, 1998.

[44] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82. ACM, 2009.

[45] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM*, 2011.

[46] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.

[47] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, 2009.

[48] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[49] SONiC: Software for Open Networking in the Cloud. https://azure.github.io/SONiC/.

[50] Qihoo360's implementation of the Raft consensus protocol. https://github.com/Qihoo360/floyd.

[51] DPDK: Data Plane Development Kit. https://www.dpdk.org/.

[52] Virtual Route Forwarding Design Guide - Cisco. https://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucme/vrf/design/guide/vrfDesignGuide.html.

[53] Quagga Routing Suite. http://www.nongnu.org/quagga/.