A Tale of Two Topologies: Exploring Convertible Data Center Network Architectures with Flat-tree

Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira,
Dingming Wu, Xin Sunny Huang, T. S. Eugene Ng
Rice University

ABSTRACT

This paper promotes convertible data center network architectures, which can dynamically change the network topology to combine the benefits of multiple architectures. We propose the flat-tree prototype architecture as the first step to realize this concept. Flat-tree can be implemented as a Clos network and later be converted to approximate random graphs of different sizes, thus achieving both Clos-like implementation simplicity and random-graph-like transmission performance. We present the detailed design for the network architecture and the control system. Simulations using real data center traffic traces show that flat-tree is able to optimize various workloads with different topology options. We implement an example flat-tree network on a 20-switch 24-server testbed. The traffic reaches the maximal throughput in 2.5s after a topology change, proving the feasibility of converting topology at run time. The network core bandwidth is increased by 27.6% just by converting the topology from Clos to approximate random graph. This improvement can be translated into acceleration of applications as we observe reduced communication time in Spark and Hadoop jobs.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Physical topologies**; **Data center networks**;

KEYWORDS

Convertible data center networks; Clos networks; Random graph networks

ACM Reference format:
Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, T. S. Eugene Ng. 2017. A Tale of Two Topologies: Exploring Convertible Data Center Network Architectures with Flat-tree. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098837

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4653-5/17/08...$15.00
https://doi.org/10.1145/3098822.3098837

1 INTRODUCTION

In this paper, we appeal for rethinking the design of data center network architectures by introducing the concept of convertibility. *Convertibility is a network's ability to change between multiple topologies with different characteristics.* This change should be completely managed by software, without involving human labor for rewiring the physical devices. With the power of convertibility, it is possible for the first time to build a data center that can function with different network architectures to combine the benefits of conventionally incompatible worlds. Our proposal is rooted in the recent trends in the development of data center networks.

The first trend is the continuous efforts towards two mutually exclusive goals for data center network design: easy implementation vs. good performance. These efforts are reflected in the enthusiasm for Clos networks in industry and random graph networks in academia. Clos, or multi-rooted tree, is the *de-facto* standard data center network architecture because of easy implementation [38, 39]. Figure 2b shows an example Clos network. The central wiring between switches in adjacent layers is relatively easy to manage, and the network can be expanded to arbitrary size by adding stages. Bandwidth oversubscription can occur at any switch layer to save cost. Modular Pods are adopted as building blocks to further ease network deployment and management. However, Clos networks have suboptimal throughput, as traffic needs to traverse up and down the network hierarchy and the resulting inefficiency exacerbates oversubscription.

In contrast, random graphs are proven to have optimal throughput for uniform traffic [40, 41]. Without rigid structures, switches are more directly connected at shorter path lengths. If implemented using the same switches and servers as a Clos network, a random graph can provide richer bandwidth and effectively alleviate the oversubscription problem. This lack of structure also enables regional random graphs to be constructed, i.e. a set of smaller local random graph networks interconnected by random wiring into a large global network. However, the neighbor-to-neighbor wiring between random switch pairs is complicated, making real-world implementation a daunting task.

Closely related to these conflicting stances is the second trend of stagnation in the emergence of new data center network architectures. Back in 2008 and 2009, the research community proposed a number of interconnection networks as the data center fabric, fat-tree [12], DCell [24], BCube [23], and HyperX [11] being the famous examples. However, there has been no breakthrough ever since. In the design space, these architectures fall between Clos and random graph at the

extremes of the scale. They attempt to find the right middle ground between easy implementation and good performance by tuning the degree of hierarchical vs. flat structure, central vs. neighbor-to-neighbor wiring, etc. Yet, the performance of a network depends on the traffic pattern; each topology has the sweet spot for particular workloads [40]. Measurement studies of data center traffic show that data center services result in very different traffic locality [30, 38] and that cluster sizes in multi-tenant clouds vary significantly [13–15]. It is hard to use a one-size-fit-all topology to address the heterogeneous and ever-changing service needs in data centers.

Relaxing the constraint of fixed topologies, the third trend is the advent of configurable data center networks that create ad-hoc links as demanded by the traffic pattern. Some solutions provide a local remedy for fixed topologies by adding a small number of connections to alleviate hot spots [19, 22, 25, 26, 43, 51], while others create a flexible network core for small-scale networks [3, 16, 17, 33, 34]. On one hand, these works demonstrate it is technically mature to change the network topology by software at run time. On the other hand, the scalability limitation remains to be addressed.

Based on the above evidence, we make the bold claim that it is time to build convertible data center network architectures. The concept of convertible network is fundamentally different from existing proposals with link flexibility. First, it aims to achieve network-wide topology change in large-scale data centers. The scalability of many previous works is constrained by a centralized device that enables flexibility, such as 3D MEMS [16, 19, 43, 44, 48] and WDM ring [3, 33, 34]. To overcome this weakness, in our proposal the enabling devices are placed across the network in a decentralized manner. Second, instead of adding extra bandwidth to the network, a convertible network rearranges the network structure to utilize existing bandwidth resources more efficiently. Third, rather than incremental topology evolution according to the instantaneous traffic pattern, a convertible network changes the intrinsic characteristics of the topology to fit the requirements of different workloads throughout their lifecycle.

We experiment with this concept by designing and implementing the flat-tree[1] prototype architecture, which can convert between a Clos topology and random graphs at different scales. Clos has rich intra-rack bandwidth and thus is suitable for traffic with strong rack-level locality. Random graph is a perfect match for network-wide uniform traffic, but it may have suboptimal performance for skewed traffic or small cloud tenant clusters. Therefore, regional random graph and global random graph should be used to adapt to different cluster sizes. The examples in Section 2.1 show the advantage of each topology given different traffic patterns. Moreover, such a design has the potential to preserve easy implementation from the Clos network, making practical deployment of random graphs achievable.

[1] The name "flat-tree" captures the dual nature of the proposed architecture. It can function as approximate random graphs ("flat" networks) and Clos (multi-rooted "tree"). It is as easy to implement as a "tree" network and has good performance as "flat" networks.

Flat-tree leverages inexpensive small port-count converter switches to convert topologies dynamically. By changing the configurations of the converter switches, cables are rewired to different outgoing connections, as if they were unplugged and replugged manually. Flat-tree takes a pragmatic approach to start from a Clos network and addresses challenges of flattening the tree structure to approximate random graphs. Specifically, how to equalize switches in different layers and relocate servers from edge to aggregation and core switches? How to break the hierarchy and connect the network core and edge directly? How to enable connections between switches in the same layer at minimum wiring complexity?

Flat-tree inherits the merits of packaging and wiring from Clos networks. It adopts the modular Pod design. Additional hardware and wiring are packaged in Pods, leaving the same external connectors as a Clos counterpart. Pods are connected to core switches with a customized regular wiring pattern. Adjacent Pods are interconnected through multi-link side connectors to allow simple neighbor-wise wiring.

Flat-tree can approximate random graphs at different scales, ranging from a Pod, to a subnetwork comprising multiple Pods, to the entire network. It can also function as Clos, which benefits applications that require rich equal-cost redundant links, predictable path length, and rack-level locality. Flat-tree can operate in hybrid mode: the network is organized into functionally separate zones each having a different topology. Workloads are placed into suitable zones to optimize their performance. As the workloads change, the network can be reorganized to adapt to the new requirements.

We discuss design options for the control plane and present the implementation details given the current technology. To exploit the link diversity in flat-tree, we adopt k-shortest-path routing [50] and MPTCP [45], whose deployment in large-scale data centers is an open challenge. The enormous number of paths lead to explosion of network states. We propose an architecture-specific addressing scheme to aggregate IP addresses and use SDN-based source routing to relieve state-keeping at the switches. Packet-level simulations show that given various traffic patterns on flat-tree networks of different scales, the pragmatic implementation of k-shortest-path routing and MPTCP achieves comparable throughput to optimal routing from linear programming.

To further evaluate the practical performance of flat-tree, we run packet-level simulations given real traffic traces from several production data centers each carrying different services. The results show that flat-tree is able to optimize for diverse workloads with different topology options. We implement a flat-tree prototype on a 20-switch 24-server testbed and run Spark and Hadoop applications with different topologies. The traffic reaches the maximal throughput only 2.5s after a topology change, proving the feasibility of converting the topology at run time. The network core bandwidth is increased by 27.6% just by converting the topology from Clos to approximate random graph. This improvement can be translated into acceleration of applications as we observe reduced communication time in Spark and Hadoop jobs.

Table 1: Throughput of clustered traffic normalized against the minimum value in the compared architectures

Cluster Size	Fat-tree	Random Graph	Two-stage Random Graph
8	1.91	1	1.16
30	1	1.38	1.65
100	1	1.59	1.17

2 MOTIVATING EXAMPLES

2.1 The Case for Convertibility

Two reasons contribute to the diversity of data center workloads. First, enterprise data centers may deploy different services that have different traffic characteristics [30, 38]. For instance, the Facebook data centers with different services show different locality features. The Hadoop site has rack-level locality, while the web and cache sites have Pod-level locality [38]. Second, in public clouds, the virtual tenants have different sizes and traffic patterns [13–15]. For example, in a Microsoft data center, the mean tenant size is 79 VMs and the largest tenant has 1487 VMs [15, 49]. In this subsection, we use a simple example to motivate the necessity of using different network topologies to serve different workloads.

We construct a $k = 16$ fat-tree network [12], and use the same devices to form random graph and two-stage random graph networks [41]. The two-stage random graph network first forms a random graph in each Pod and takes the Pods as super nodes to form another layer of random graph together with core switches. Figure 2b, 2c and 2d show approximations of these topologies. To simulate intra-tenant communications in cloud data centers, we pack consecutive servers into clusters and create all-to-all traffic in each cluster. We measure the throughput following a well-adopted methodology [41], which assumes optimal routing and allocates bandwidth to flows using a linear programming solver.

Table 1 shows the normalized throughput with different cluster sizes. In the fat-tree network, each edge switch is connected to 8 servers, and there are 64 servers per Pod. 8-server clusters generate local traffic only, so fat-tree, without bottleneck in the network core, yields the highest throughput. Servers are distributed uniformly across all switches in the random graph. In the two-stage random graph, servers in each Pod are distributed uniformly across switches in the Pod, and core switches take no servers. As a result, the two-stage random graph has the second best performance since the traffic is served with better locality than in the random graph. For 30-server clusters, most of the traffic stays in Pods, so the two-stage random graph has the highest throughput. Random graph is particularly suitable for network-wide traffic because of the rich core bandwidth, so it performs the best for the cross-Pod traffic from 100-server clusters.

This example shows that different topologies perform better for different workloads, depending on the extent of locality they exhibit. We believe the network should be convertible between multiple topologies to adapt to different workloads. Our flat-tree architecture can work as a Clos network and can approximate random graph and two-stage random graph. The network can be configured to the topology that best suits the workload. In hybrid-mode, the flat-tree network is organized into functionally separate zones each having a different topology. Clusters of different sizes can be placed into suitable zones to optimize their performance. Our simulation experiments with real data center traffic in Section 5.2 demonstrate the performance advantage of each supported topology under different traffic.

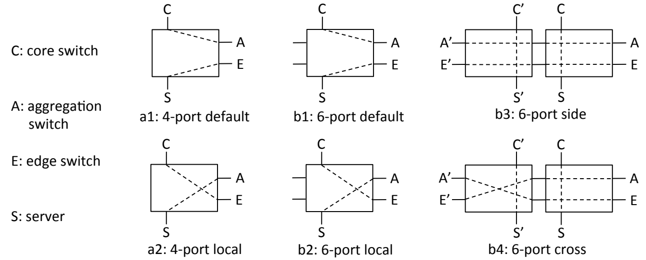


Figure 1: Converter switch configurations

2.2 Example Flat-tree Network

We use the simple flat-tree example in Figure 2 to demonstrate how to convert a Clos network to an approximate random graph. The gray lines represent original connections in the Clos Pod that need to be replaced by the dashed links in the flat-tree Pod. The most notable differences between Clos and random graphs are server distribution and the types of links. In Clos networks, servers are attached to edge switches only and all links are hierarchical, either between edge and aggregation switches or between aggregation and edge switches. All switches are equal in random graphs. Servers are uniformly distributed to the switches, and the links are between random switch pairs. So, the first step of conversion is to relocate servers to aggregation and edge switches and to diversify the types of links.

To save cost, we aim to achieve these goals using small port-count converter switches. We find 4-port and 6-port converter switches the minimum-scale switches to facilitate the required topology changes. As shown in the zoomed-in Pod, flat-tree breaks an edge-server link and an aggregation-core link in the Clos network, and connects the corresponding server, edge, aggregation, and core switches to a converter switch. Figure 1 illustrates the valid configurations of 4-port and 6-port converter switches. The "default" configuration enables the original Clos connections. The "local" configuration relocates the server to the aggregation switch and connects the core and edge switches directly. This change is local in the Pod.

4-port converter switches should not be used to relocate servers to core switches. If we connect the server and the core switch, the edge and aggregation switches must be connected as well, otherwise we waste a link. There are sufficient edge-aggregation links in the Pod, so this change fails to diversify the types of links. 6-port converter switches introduce side ports, through which two converter switches can be interconnected. The "side" and "cross" configurations both relocate servers to core switches, but connect edge and aggregation switches to their peers in different ways. We only allow 6-port converter switches in adjacent Pods to be interconnected for simple neighbor-to-neighbor wiring.

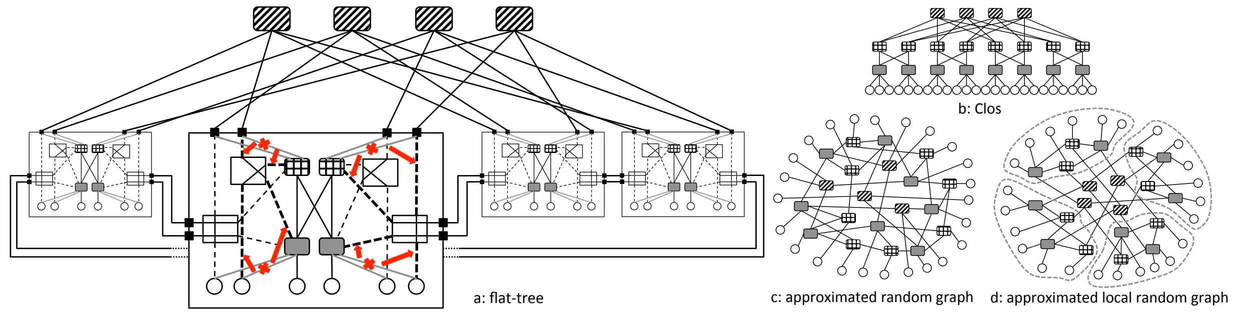


Figure 2: Example flat-tree network and some achievable topologies. Core switches in stripe, aggregation switches in grid, edge switches in shade, and servers as circles. Gray lines are connections in the original Clos network, which are replaced with the dashed links connected to converter switches to form flat-tree. The converter switches show the configuration for approximated random graph. Flat-tree uses a customized wiring pattern to connect Pods to core switches.

The number of 4-port and 6-port converter switches are determined by the layout of the Clos network. In Figure 2, each pair of edge and aggregation switches are connected to a 4-port converter switch and a 6-port converter switch, which show the approximate random graph configuration. Converter switches and the additional wiring are packaged in the Pod, keeping the same core connectors as a Clos Pod. The side connectors of 6-port converter switches are bundled as multi-link connectors to simplify inter-Pod wiring. Flat-tree Pods are connected to core switches via a customized wiring pattern (details in Section 3.2). In this example, the uplinks from Pods are swapped in different ways, so that servers are distributed uniformly across the core switches.

Flat-tree converts between multiple topologies with different converter switch configurations. Figure 2b shows the Clos network, when all converter switches take the “default” configuration. Figure 2c shows an approximate global random graph, with the 4-port “local” and 6-port “side” configurations. In practice, we can also use the 6-port “cross” configuration to swap connections. Figure 2d shows approximate local random graphs in each Pod. It is configured in a way that half servers are connected to the edge switches and half to the aggregation switches. In this example, we use 4-port “local” and 6-port “default” configurations. Flat-tree can also operate in hybrid mode, with different combinations of the above topologies each in a number of Pods.

This paper limits the discussion to one Pod layer connected by core switches. Flat-tree can be extended to multi-stages of Pods: the lower-layer Pods consider the edge switches in the upper-layer Pods as core switches; intermediate switch-only Pods take relocated servers from lower-layer Pods as their own servers. We leave the details to future work.

3 FLAT-TREE ARCHITECTURE

3.1 Flat-tree Pod

Figure 3 depicts a flat-tree Pod. Without loss of generality, we assume the number of edge switches is a multiple of the number of aggregation switches. There are d edge switches and d/r aggregation switches. We pair up each edge switch E_j with aggregation switch $A_{j/r}$ and connect them to n 4-port converter switches and m 6-port converter switches. n and m represent the maximum number of servers originally connected to an edge switch that can be relocated dynamically to aggregation and core switches. Whether to relocate them depends on the topology to be achieved. We place the converter switches evenly on the two sides of the Pod: those connected to $E_0 ... E_{d/2-1}$ locate on the left of the Pod and those connected to $E_{d/2} ... E_{d-1}$ locate on the right. This forms a $n \times d/2$ matrix of 4-port converter switches, i.e. blade A in figure, and a $m \times d/2$ matrix of 6-port converter switches, i.e. blade B in figure, on each side of the Pod.

For both types of blades, the converter switches in any row of column j on the left blade are connected to edge switch E_j and aggregation switch $A_{j/r}$, and those on the right are connected to edge switch $E_{j+d/2}$ and aggregation switch $A_{(j+d/2)/r}$. Each 4-port converter switch connects to a core switch and a server, so blade A has $n \times d/2$ core connectors and server connectors. Each 6-port converter switch has a pair of side connectors as well, so blade B has $m \times d/2$ core connectors, server connectors, and double side connectors. There may be remaining core connectors on the aggregation switches and server connectors on the edge switches. The total number of core connectors and server connectors are equal to those in a Clos counterpart. If d is odd, a middle converter switch can be on either side, but the side connectors of the 6-port converter switch are unused.

3.2 Pod-Core Wiring

In Clos, all Pod-core connections are between aggregation and core switches. Suppose each aggregation switch has h uplinks. As Figure 4a shows, aggregation switches with the same index i in different Pods are connected to the same group of h core switches via the aggregation connectors. Repeatedly for each Pod, this wiring pattern links the h connectors for each aggregation switch consecutively to core switches.

In flat-tree, as shown in Figure 3, there are 3 types of core connectors. Core switches can be connected 1) to servers via blade B connectors, 2) to edge switches via blade A connectors, and 3) to aggregation switches via aggregation connectors. The Pod-core wiring determines the distribution of servers and different types of links (to an edge or aggregation switch) across the core switches, thus affecting how closely flat-tree approximates a random graph.

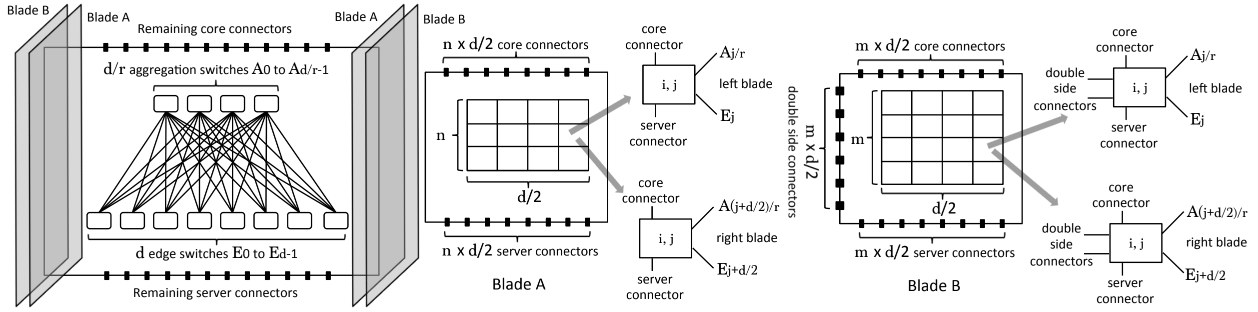


Figure 3: A flat-tree Pod. A pair of edge switch E_j and aggregation switch $A_{j/r}$ connected to n 4-port and m 6-port converter switches. Converter switches are placed evenly on both sides as matrices. Blade A and B has 4-port and 6-port converter switches respectively.

As each aggregation switch corresponds to r edge switches, the h aggregation connectors in Clos are replaced with $n \times r$ blade A connectors, $m \times r$ blade B connectors, and $h - m \times r - n \times r$ aggregation connectors. The Clos wiring pattern is based on aggregation switches, each connected to h core switches. Since flat-tree has edge-core connections, its wiring pattern should be based on edge switches. Each edge switch corresponds to n blade A connectors, m blade B connectors, and $h/r - m - n$ aggregation connectors, which connects to overall h/r core switches.

We offer two wiring options, shown in Figure 4b and 4c. Connectors corresponding to the edge switches with the same index j in different Pods are connected to the same group of h/r core switches. Both wiring patterns connect the group of core switches consecutively to blade B connectors, followed by blade A connectors and aggregation connectors. They rotate in different ways across Pods. Pattern 1 packs blade B connectors continuously Pod by Pod throughout the set of core switches. Pattern 2 moves them forward by one more core switch as the Pod index grows. Both patterns wrap around within the group.

Physically, we suggest wiring Pod 0 first, by linking every m blade B connectors, n blade A connectors, and $h/r-m-n$ aggregation connectors in turn to core switches consecutively. We start from the left blades and move on to the right blades, until all connectors in the Pod are consumed. In this process, we mark the mapping between each edge switch and the corresponding group of h/r core switches. For the following Pods, connectors corresponding to each edge switch are connected to the marked h/r core switches according to the rotating patterns.

These wiring patterns have the following properties:

Property 1: For both wiring patterns, servers are distributed uniformly across the core switches.

Property 2: For both wiring patterns, the core switches have an equal number of links of the same type.

Flat-tree maintains structure to ease implementation, so servers and links must be permuted by wiring. These properties ensure that flat-tree well approximates random graphs.

Because these patterns follow straightforward rules, they have low wiring complexity. Pattern 1 has better performance, because a core switch does not connect to servers from adjacent Pods at the same time, thus it takes advantage of side connections between adjacent Pods to the greatest extent. Yet when h/r is a multiple of m, different Pods are likely to repeat the same pattern, thus reducing the wiring diversity. In this case, pattern 2 is more favorable. Our previous paper contains evaluation of these wiring patterns (Figure 5 in [47]).

3.3 Inter-Pod Wiring

For adjacent Pods p and $p+1$, the 6-port converter switches on the left blade B of Pod $p+1$ are connected to those on the right blade B of Pod p by the side connectors. Recall from Figure 3 that the converter switches in the same column connect to the same pair of edge and aggregation switches. We want to connect an edge/aggregation switch to as many different switches as possible in the adjacent Pod, so we design a shifting wiring pattern such that the converter switches in the same column of the right Pod are connected to converter switches each in a different column of the left Pod. Specifically, let i and j be the row and column of the converter switch matrices, converter switch $\langle i, j \rangle$ on the left of Pod $p+1$ is connected to converter switch $\langle i, (d/2-1-j+i)\%(d/2) \rangle$ on the right of Pod p, which represents the converter switch in the same row i and in the column i slots shifted from the mirrored column $d/2-1-j$. We want the converter switches to be interconnected by different configurations, so we have both peer-wise and edge-aggregation connections across Pods. If i is even, they take the 6-port "side" configuration (in Figure 1); if i is odd, they take the 6-port "cross" configuration. To streamline the connection of adjacent Pods, the side connectors on the same side of a Pod are bundled as a multi-link connector that integrates this wiring pattern.

3.4 Server Distribution

In a random graph, servers are distributed uniformly across the switches, because the random links roughly connect the switches in a uniform manner. Yet flat-tree maintains structures, e.g. the Clos connections between edge and aggregation switches, core switches connected to the Pods, though using customized wiring patterns, and the neighbor-to-neighbor wiring restricted to adjacent Pods. The path length of switch pairs is not uniform for flat-tree, so we should place servers intelligently to leverage the shorter paths in the network.

Recall that 6-port converter switches can relocate servers to core switches, and 4-port ones can relocate servers to aggregation switches, so the server distribution is determined by the choice of m and n. Because flat-tree aims at converting

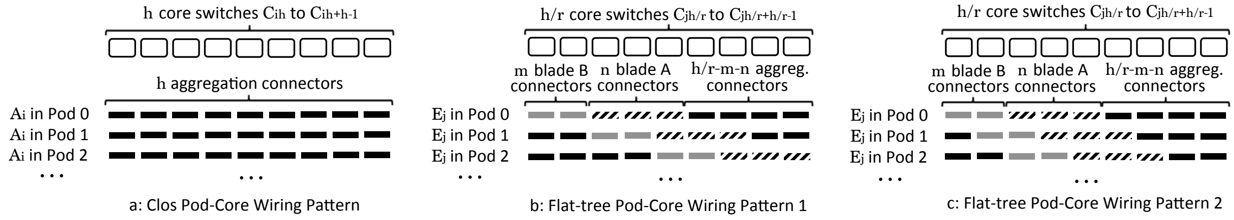


Figure 4: Pod-core wiring for the same set of connectors across Pods. All connectors are on aggregation switches in Clos; flat-tree has 3 types of connectors on blade A, B, and aggregation switches, enabling core-server, core-edge, and core-aggregation connections respectively.

generic Clos networks, which may have very different layouts, it is difficult to pre-define the m and n values for optimal transmission performance. We suggest a profiling scheme: under the preferred Pod-core wiring pattern described in Section 3.2, vary m and n until they result in the shortest average path length over all server pairs. The sensitivity test for this approach is in our prior paper [47].

3.5 Operation Modes

Global: Flat-tree approximates a network-wide (or global) random graph in the "global" mode. 6-port converter switches take either the "side" or the "cross" configuration (Figure 1 b3 or b4) depending on their row index in the matrix as described in Section 3.3. 4-port converter switches take the "local" configuration (Figure 1 a2).

Local: Flat-tree approximates a two-stage (or local) random graph in the "local" mode. It first forms random graphs in each Pod and takes the Pods as super nodes to form another layer of random graph together with core switches. 6-port and 4-port converter switches take the "local" configuration (Figure 1 a2 and b2) to relocate half servers to aggregation switches. Any remaining 6-port converter switches take the "default" configuration (Figure 1 b1).

Clos: Flat-tree functions as a Clos network by default. All converter switches take the "default" configuration (Figure 1 a1 and b1).

Hybrid: Flat-tree can be configured in the unit of a Pod, so it can have arbitrary combinations of the above three topologies each in a number of Pods. The converter switch configurations follow the rules in their corresponding mode.

3.6 Cost Analysis

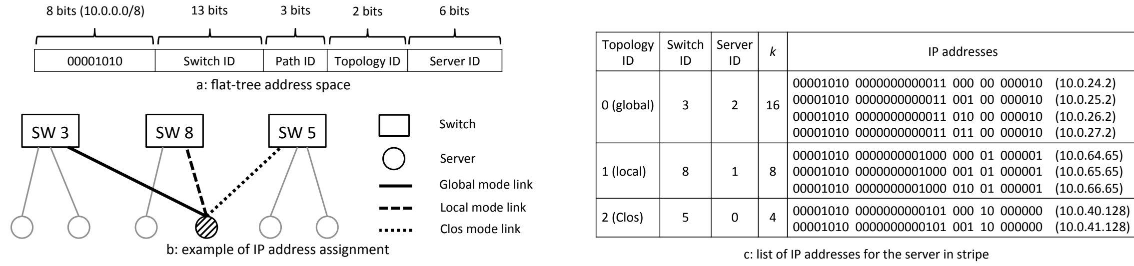
Because of their simple functionality, converter switches in flat-tree can be realized by passive circuit switches. The choice of specific switching technology depends on the existing devices already deployed in the data center. If the data center has copper cables in place, crosspoint switches whose per-port cost is as low as $3 [31] can be used. Converter switches split some cables into two parts. Because crosspoint switches are passive devices, cables connected to a converter switch do not need active elements. If manufactured properly, the cost of two cables each with only one active element at the packet switch end is equivalent to the cost of the original cable.

Many data centers nowadays use optical fibers for cross-rack connections. To avoid the cost of extra transceivers, optical circuit switches are sensible options for converter switches. Because converter switches have small port count, we can use low-cost switching technologies, such as 2D MEMS [46] and Mach-Zehnder switches [20], whose port count is limited to moderate scale due to losses from photonic signal crossings or other effects. The mass production cost of these technologies is dominated by packaging. While we are not able to project future costs precisely, we anticipate that the per-port cost will become reasonably cheap as photonic packaging technology advances. The difference between transmit power and receive sensitivity of commercial optical transceivers can be over 8dB [7], which easily overcomes the insertion loss of most optical switches. Amplifiers are thus not needed.

4 CONTROL SYSTEM

Because a data center is administered by a single authority, we follow the recent trend of using a centralized network controller for global network management. Flat-tree has several operation modes with pre-known topologies, which designate a fixed set of configurations for the converter switches. The controller changes the topology by configuring the converter switches, via specific control mechanisms depending on the realization technology. For instance, most optical switches can be programmed via a software interface. The converter switch configurations for different flat-tree modes can be hard-coded into the controller.

For flat-tree Clos mode, we can use ECMP [28], two-level routing [12], or customized SDN routing with pre-computed paths [39]. The study on random graph network [41] suggests using k-shortest-path routing [50] and MultiPath TCP (MPTCP) [45]. We adopt this approach for flat-tree global mode and local mode, because they approximate random graph and two-stage random graph respectively. However, this prior study [41] failed to consider the overwhelmingly large number of network states in a real implementation. According to our experience with our testbed implementation, under the SDN paradigm, the number of Openflow rules easily exceeds the capacity of commercial SDN switches for a very small-scale random graph network. So, another major scalability concern of random graph networks is from the overhead of the control system, besides messy cable deployment. Therefore, even deploying a static random graph network requires a re-design of the control plane. In this section, we propose a control system with a manageable number of network states for flat-tree, and this solution can be easily applied to static random graph networks. Note that the main contribution of this paper is the flat-tree network architecture, and we acknowledge there may be alternative control plane designs.



Topology ID	Switch ID	Server ID	k	IP addresses
0 (global)	3	2	16	00001010 0000000000011 000 00 000010 (10.0.24.2) 00001010 0000000000011 001 00 000010 (10.0.25.2) 00001010 0000000000011 010 00 000010 (10.0.26.2) 00001010 0000000000011 011 00 000010 (10.0.27.2)
1 (local)	8	1	8	00001010 0000000001000 000 01 000001 (10.0.64.65) 00001010 0000000001000 001 01 000001 (10.0.65.65) 00001010 0000000001000 010 01 000001 (10.0.66.65)
2 (Clos)	5	0	4	00001010 0000000000101 000 10 000000 (10.0.40.128) 00001010 0000000000101 001 10 000000 (10.0.41.128)

c: list of IP addresses for the server in stripe

Figure 5: Illustration of the addressing scheme. "a" shows the IP address fields in flat-tree. In the "b" example, the server in strip connects to switch #3, switch #8, and switch #5 respectively in the global, local, and Clos mode, where the number of concurrent paths, or k, is chosen to be 16, 8, and 4. The IP addresses assigned to this server are shown in "c". All these addresses for every flat-tree topology mode are preconfigured on the server.

4.1 MPTCP

MPTCP has been standardized and widely used in academia and industry [21]. The kernel implementation has been released [2]. MPTCP establishes subflows via multi-homing: the end hosts using multiple IP addresses to distinguish paths. In flat-tree, servers have one uplink only, so we must associate multiple IP addresses to a single NIC. IP aliasing gives the solution by setting multiple virtual network interfaces. These virtual interfaces are linked to the physical interface by default, so traffic with different IP addresses can be forwarded by the physical interface.

The full-mesh option in MPTCP allows subflows with different combinations of the source-destination IP address pairs. For instance, with 2 IP addresses on both the sender and the receiver, we obtain $2 \times 2 = 4$ subflows. Therefore, the number of IP addresses per server is the square root of the number of concurrent paths, or k in k-shortest-path routing. Not all subflows are needed sometimes. For example, 8-shortest-path routing requires 3 IP addresses per server, thus creating one extra subflow. In such case, a straightforward workaround is to limit the routing logic to the necessary subflows only, and MPTCP will not allocate traffic to subflows with no end-to-end reachability.

This simple way of assigning IP addresses defines a flat address space, which may be inefficient considering the great number of servers in a large data center. The property of MPTCP to send traffic only with routable addresses gives the freedom for more intelligent addressing mechanisms. Generally, address assignment depends on the structure of the network and serves for the ease of routing. This task is particularly difficult for flat-tree, which has completely different network structures and routing paths for each topology. We propose a customized addressing scheme specific to the flat-tree architecture in the next subsection.

4.2 k-Shortest-Path Routing

In k-shortest-path routing, there are k routes for every source-destination server pairs. A critical consequence of the enormous number of paths is the explosion of the network states. For efficient routing, every transit switch needs to be configured with the forwarding rules of the k paths for all server pairs. Let n and N be the number of servers and switches in the data center and L be the average path length, the average number of network states per switch is $\frac{n^2 \times k \times L}{N}$. For a large data center, this number can easily reach tens of million, far exceeding the storage and processing capacity of switches. k-shortest-path routing requires matching both the source and destination IP addresses, and traditional ways of aggregation, such as destination IP lookup or prefix matching, do not readily work. A switch may forward packets for the same receiver to different ports, because they need to take different routes. Servers can be relocated to different switches under different flat-tree topology modes, making the definition of common prefix very challenging. We need novel approaches to factoring down the number of network states.

4.2.1 Addressing. We have two important observations from the flat-tree architecture and from an extensive analysis of the computed k-shortest paths in the network.

Observation 1: A server is connected to one and only one ingress/egress switch, regardless of the fact that it may be relocated to a different ingress/egress switch as the topology changes. So, there is no path diversion between servers and the connected ingress/egress switches.

Observation 2: The number of equal-cost paths is small[2] in the approximate random graph flat-tree creates. The k-shortest paths between server pairs are nearly deterministic, with uncommon exception of ties. So, the k-shortest paths between ingress and egress switches almost capture the full set of selected paths between source and destination servers.

Given these observations, it is promising to conduct prefix matching on the ingress/edge switch level. This way, the average number of network states per switch is reduced from $\frac{n^2 \times k \times L}{N}$ to $\frac{S^2 \times k \times L}{N}$, S being the number of ingress/egress switches. Usually 20 to 40 servers are connected to a top-of-rack switch (ToR) in a data center, so the number of network states can be reduced by a factor of 400 to 1600.

As discussed previously, the major difficulty in flat-tree is server mobility. To guarantee common prefix for servers under the same ingress/egress switch, we need a different set of IP addresses for each flat-tree topology mode. Because we aim to change the network topology at run time by software,

[2]Having few equal-cost paths does not imply poor failure resiliency. Like random graph networks, flat-tree can and should use paths of different lengths for high throughput. It has been established that throughput degrades more gracefully in random graph networks than in fat-tree under failure [41]. Because flat-tree approximates random graph networks, we expect flat-tree to be resilient to failure as well, although more thorough evaluations are left to future work.

it is infeasible to reset the server IP addresses manually for each topology. Thanks to the property of MPTCP to send traffic only with routable addresses, we can preconfigure all possible IP addresses for each topology onto the servers and let the network controller dynamically load the routing logic for the subset of addresses particular to the topology in use.

Our definition of the address space is shown in Figure 5a. We assume IPv4 addresses and allocate IP addresses within the private 10.0.0.0/8 block. The first 13 bits after the fixed heading octet represent the switch ID of the ingress/egress switch. In flat-tree, all switches may serve as an ingress/egress switch. We associate each switch with a unique ID, which is not changed with the conversion of topology. This 13-bit field allows for 8196 switches, which is sufficient for a large-scale data center. The next 3 bits are for the path ID in the k-shortest paths. As aforementioned, MPTCP distinguishes paths by different combinations of IP addresses between server pairs. This 3-bit field allows for 8 addresses at sender/receiver and thus supports $8^2 = 64$ concurrent paths at most, covering the range of k most data centers will use. The next 2 bits are used to specify the 3 possible flat-tree topologies. The rest 6 bits show the server ID under the ingress/egress switch. Because of the limited IPv4 address space, we cannot afford to assign a unique ID for every individual server. So, these IDs are reused for servers under different ingress/egress switches. This 6-bit field supports 64 servers per switch, which is enough for the 20 to 40 servers per ToR in most data centers. By this address assignment, we match the /24 prefix at the ingress/egress switches. This addressing scheme can be easily extended to IPv6 addresses, which even support globally unique server IDs.

Figure 5b shows an example of the address assignment. The server in stripe is connected to 3 different ingress/egress switches under different flat-tree modes. The servers under the same ingress/egress switch are ordered from left to right, so the server ID in the global, local, and Clos mode is 2, 1, and 0 respectively. The number of concurrent paths, or k, can be different under each mode, because each topology may have optimum transmission performance with a different k. In this example, k equals 16, 8, and 4 for each topology, so we need 4, 3, and 2 IP addresses accordingly. Figure 5c lists the allocated IP addresses according to our addressing scheme. All these addresses for every flat-tree topology are preconfigured on the server at deployment time.

One possible problem is the overhead of MPTCP probing unused IP addresses for potential paths. In our small testbed with 4 concurrent paths, as shown in Section 5.3, we implement this addressing mechanism (6 addresses per server, 2 for each topology) as well as the naive address assignment (2 addresses per server, no unnecessary addresses). We observe no noticeable difference in throughput between the two approaches. Whether the overhead is a valid concern in large data centers is the direction of future work.

4.2.2 Source Routing. A common solution to relieving state management at switches is source routing [29, 35, 42]. Segment routing is a natural fit to this request in SDN [6]. In segment routing, the k-shortest-path routing algorithm can be implemented in the Path Computation Element (PCE), an equivalent of the centralized network controller, which enforces per-route states only at ingress switches. It relies on the MPLS [36] and IPv6 architecture. The ingress switch encodes the hops of a path as a stack of MPLS labels. The transit switches forward packets by dumb matching of the label on top of the stack and pop it upon completion.

Table 2: List of flat-tree topologies for evaluating the control system. Abbreviations: Edge Switch (ES), Aggregation Switch (AS), Core Switch (CS), Upstream Port (UP), Downstream Port (DP), Oversubscription Ratio (OR).

ID	#ES (#UP,#DP)	#AS (#UP,#DP)	#CS (#DP)	OR at ES	OR at AS	#Server
topo-1	128 (8,32)	128 (8,8)	64 (16)	4	1	4096
topo-2	72 (6,24)	72 (6,6)	36 (12)	4	1	1728
topo-3	128 (8,64)	128 (8,8)	64 (16)	8	1	8192
topo-4	128 (8,32)	64 (16,16)	32 (32)	4	1	4096
topo-5	128 (16,32)	128 (8,16)	64 (16)	2	2	4096
topo-6	128 (16,32)	64 (32,16)	32 (32)	2	2	4096

Not all data centers have the MPLS and IPv6 forwarding fabric, so we provide an alternative solution in the better recognized OpenFlow paradigm. Source routing is not supported in OpenFlow by default. From the literature of workarounds [29, 35, 42], we pick a readily deployable approach without modification of the OpenFlow protocol [29]. We encode the path, represented as a list of next-hop output ports, into the source MAC address and use TTL as the location pointer in the path. Flat-tree is a small diameter network, where paths traverse less than 3 switches on average [47]. The 48-bit MAC address is able to hold 6 hops for switches having as many as 256 ports, which is sufficient for the need of the network. OpenFlow 1.3 allows matching arbitrary bits of a given field [4]. We can thus concatenate the transit hops in the MAC address and let intermediate switches match different bits using a mask depending on the TTL. For instance, if TTL equals 253 (third hop), we apply the mask 0x00:00:ff:00:00:00 on the MAC address and match the extracted bits to all possible 256 ports to decide the right output port. This way, we need an entry per TTL per output port. So, the number of OpenFlow rules on the transit switches is $D \times C$, where D is the diameter of the network and C is the switch port count. This number is at most a thousand, far below the capacity of an OpenFlow switch. These rules remain the same as the flat-tree topology changes, so they can be preconfigured statically.

With source routing, the number of network states per ingress/egress switch is reduced to $S \times k$. This number is at most a few tens of thousand, within the capacity of high-end OpenFlow switches [29]. There is large room for optimization to further bring down this number. For example, in public clouds, tenants request virtual clusters where only machines within the cluster talk to each other. In this case, we can set in-cluster routing logic, which involves a small number of ingress/egress switches. Traffic is skewed in many enterprise data centers [13–15, 30, 38]. We can use diverse paths (large k) for a small number of elephant flows, and simple paths (small k) for a large number mice flows.

4.3 Topology Conversion

The conversion delay of flat-tree topologies is determined by the switching delay of the converter switches and the delay

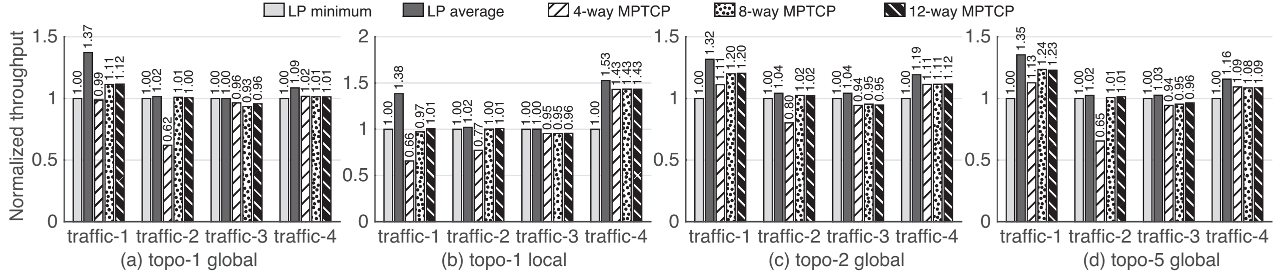


Figure 6: Average flow throughput normalized against LP minimum on selected flat-tree topologies

of changing the routing logic. Depending on the realization technology, the switching delay of converter switches ranges from several μs to hundreds of ms [3, 19, 20, 46]. The network controller takes roughly 1ms to add/delete a network state [27, 37]. Instead of streaming the states all from a single network controller, we can speed up the state distribution by having a set of controllers each managing a number of switches. We designate a logically centralized controller to maintain the global network graph. It observes link failures and updates the graph, which happens infrequently and does not cause heavy burden. The k-shortest-path routing algorithm is easily parallelizable, because the computation of paths between different nodes is independent. So, the distributed controllers can either work as dumb agents of the logically centralized controller and preload paths from it, or compute paths independently based on a consistent network graph. Following the trend of building customized switches for data centers [39], it is conceivable to push the computation to switches. This way, switches can update network states locally on simple signaling of topology change. The paths and the resulting network states can also be precomputed and stored into a table in memory to save the computation time. With these implementation options, we estimate the delay of changing the routing logic to be on the order of seconds.

From the network operator's perspective, topology conversion in flat-tree is similar to network upgrade. Network operators can plan when conversions should happen and are fully aware of the impact of the change. They can convert the topology gradually involving some of the network devices, so converter switches need not be coordinated to react all at the same time. Existing methods for updating or replacing a switch in the network, e.g. draining parts of the network incrementally before making the changes, can be used to avoid traffic disruption.

5 EVALUATION

Our prior study has demonstrated the theoretical performance benefits of flat-tree [47]. We compared its average path length to random graph networks, and ran a linear programming simulator to evaluate the throughput of synthetic data center traffic patterns assuming optimal routing. In summary, flat-tree well approximates random graph and two-stage random graph networks when functioning in global and local mode respectively: the difference in average path length is within 5% (Figure 5 and 6 in [47]) and the difference in throughput is less than 6% (Figure 7 and 8 in [47]).

In this paper, we answer several key questions about the flat-tree performance with more comprehensive and realistic experiments. Because the linear programming simulations ignore the overhead of practical routing and transport protocols, we first evaluate the performance of k-shortest-path routing and MPTCP to see how close the throughput they achieve is to the theoretical bound. We use the MPTCP packet-level simulator [1] and run extensive experiments given a series of synthetic traffic patterns on flat-tree networks of different layouts. Next, we feed the simulator with several data center traffic traces to understand the transmission performance under real settings. Finally, we implement flat-tree on a hardware testbed and run Spark and Hadoop jobs to measure the performance improvement to real data center applications.

5.1 Is k-shortest-path routing with MPTCP efficient enough?

We construct various flat-tree networks based on generic Clos networks with different parameter settings [18]. Table 2 lists the evaluated flat-tree topologies. We use topo-1 as the baseline topology and create other topologies by varying the network scale, oversubscription ratio, and arrangement of switches. topo-1 has 4:1 oversubscription at edge switches only. topo-2 is a proportional down-scale of topo-1. topo-3 is two times more oversubscribed at the edge than topo-1. topo-4 replaces the aggregation and core layers of topo-1 with fewer switches of larger port counts. topo-5 moves half of topo-1's oversubscription to the aggregation switch level. topo-6 replaces the aggregation and core switches of topo-5 with larger ones. These topologies capture the major variations in Clos networks. We have flat-tree function in both global and local mode for each topology.

A standard approach for evaluating routing in interconnection networks is to measure the throughput of flows given well-studied traffic patterns [18], so we use the following widely used synthetic traffic patterns to drive the simulation.

Permutation (traffic-1): every server sends a single flow to a unique server other than itself at random. This pattern creates uniform traffic across the network.

Pod Stride (traffic-2): every server sends a single flow to its counterpart in the next Pod. This traffic pattern creates heavy contention in the network core.

Hot spot (traffic-3): every 100 servers form a cluster, in which one server broadcasts to all the others. It simulates the multicast phase in many machine learning applications.

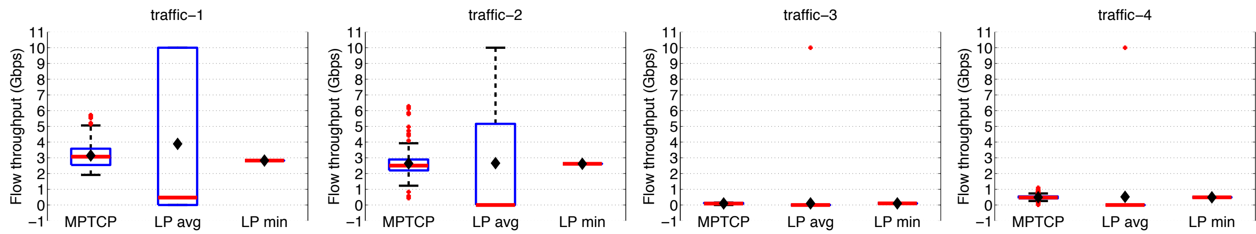


Figure 7: Box plots to show the distribution of flow throughput on the topo-1 topology under flat-tree global mode (topo-1 global). MPTCP uses 8 paths. The box contains the 25th to 75th percentiles of the data. The whisker lines extending above and below the box cover the data within 3 times the box range. The data in dots beyond the whisker are outliners. The bold line in the middle of the box shows the median and the diamond shows the average.

Many-to-many (traffic-4): every 20 servers form a cluster with all-to-all traffic. This traffic pattern simulates the shuffle phase in MapReduce jobs.

We also vary k, the number of concurrent paths in k-shortest-path routing, to evaluate the sensitivity of throughput against it. Given the above traffic, we compare the flow throughput from simulation to the optimum bandwidth allocation, which is the solution to the multi-commodity flow problem [32]. We make two linear programming (LP) formulations with different optimization goals: 1) maximizing the minimum flow throughput (denoted as "LP minimum") to achieve ideal load balancing; 2) maximizing the average flow throughput (denoted as "LP average") to achieve best network utilization.

Figure 6 shows the average flow throughput on selected topologies, and the topologies not shown have similar trends. We normalize against LP minimum for each evaluated method for readability, as throughput numbers are vastly different in scale. The number of concurrent paths, k, affects the MPTCP performance. If k is too small, the path diversity cannot be fully exploited, thus many links are under-utilized. In these experiments, 8 concurrent paths is sufficient, and larger k cannot improve the throughput further. This result is consistent with the performance of MPTCP and k-shortest-path routing in random graph networks [41].

k-shortest-path routing plus MPTCP reaches a reasonable middle ground between LP minimum and LP average. To scrutinize at the throughput of individual flows, we zoom in on topo-1 global mode and show the distribution of flow throughput with box plots in Figure 7. Neither LP minimum nor LP average is realistic. To balance the load among flows, LP minimum stops allocating residual bandwidth after it has successfully maximized the minimum flow throughput. LP average assigns some zero throughputs and some high or even full throughputs to maximize the network utilization. MPTCP balances between these extremes. It achieves higher average throughput than LP minimum, and the variance of flow throughput is smaller than LP average. Leveraging multi-paths and congestion control, MPTCP can dynamically adapt to the link utilization to get high throughput and maintain fair bandwidth sharing across flows.

From the above results, we conclude that k-shortest-path routing plus MPTCP is efficient enough. With the right choice of k, it constantly achieves comparable throughput to optimal bandwidth allocation from LP solutions given a set of traffic patterns on diverse flat-tree topologies. It balances between high network utilization and load balancing, which are important indicators of good performance in practice.

5.2 Does flat-tree handle real traffic well?

To evaluate the transmission performance of flat-tree with real data center traffic, we need a practical network topology. Large-scale data center network designs in recent years show the trend of non-blocking switch fabric with oversubscription only at the network edge [38, 39]. We follow this trend to use topo-1 as a representative flat-tree topology for the remaining simulations. We run traffic traces from 4 Facebook data centers each carrying different services. They are from the following two sources.

1) We obtain the one-hour trace in a Hadoop data center (denoted as Hadoop-1) from the Coflow benchmark [5], which contains aggregated rack-level traffic through a 1Gbps single-switch network core. Our flat-tree network uses 10Gbps links and has 8 uplinks per edge switch. For each rack-to-rack flow from the trace, we create 8 flows between servers under the source and destination edge switches to stress the switch uplinks and give 10 times the original traffic volume to each of the 8 flows to adjust the bandwidth difference.

2) We obtain traffic statistics for 3 other data centers (denoted as Hadoop-2, Web, and Cache) from the Facebook measurement study [38]. The full traces are not released, so we generate our own traces based on the publicly shared sampling data [8] and the reported results from the paper [38]. The source and destination servers of the flows are inferred from the sampling data. The flow size and the flow arrival rate are reverse-engineered from Figure 6 and Figure 14 in the paper. We omit inter-data-center traffic in the data.

The traffic has the following characteristics.

Hadoop-1: the trace reflects the shuffle phase of MapReduce jobs. The traffic does not have clear locality. We observe one-to-many, many-to-one, and many-to-many traffic involving a large number of machines network-wide.

Hadoop-2: different from the above trace, the traffic shows strong rack and Pod level locality. 75.7% of the traffic is intra-rack, and almost all the remaining traffic is intra-Pod.

Web: the traffic has strong Pod level locality. There is a tiny amount of intra-rack traffic. Around 77% of the traffic is intra-Pod, and the rest is inter-Pod.

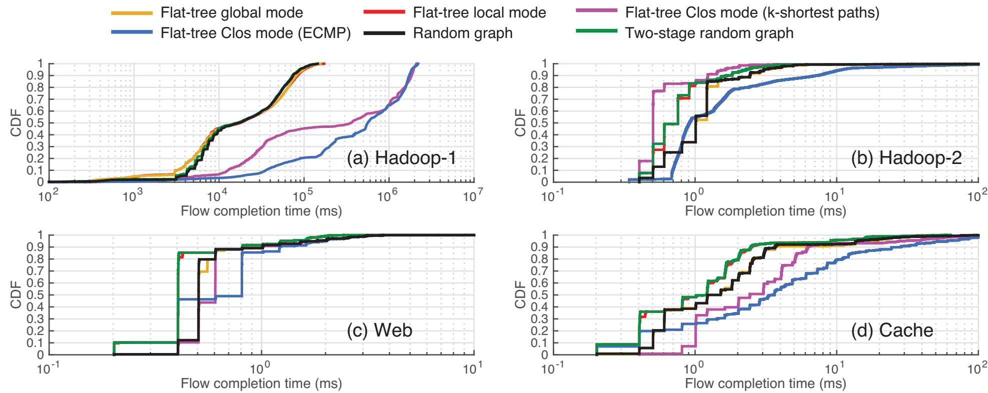


Figure 8: CDF of flow completion time in Facebook's Hadoop-1, Hadoop-2, Web, and Cache data centers

Cache: the traffic shows even stronger Pod level locality. There is almost zero intra-rack traffic. Around 88% of the traffic is intra-Pod, and the rest is inter-Pod.

Figure 8 shows the distribution of flow completion time of these different traffic traces. Regardless of the traffic, the performance of flat-tree in the global mode is close to that of the random graph, and the performance of flat-tree in the local mode is close to that of the two-stage random graph. It demonstrates that flat-tree well approximates random graphs of different scales given real data center workloads, which is consistent with the conclusion in the prior study [47].

In practice, Clos networks usually implement ECMP and TCP. For fair comparison, we simulate the flat-tree Clos mode with k-shortest-path routing and MPTCP as well to avoid the handicap of less efficient routing and congestion control mechanisms. As expected, the performance of flat-tree Clos mode with ECMP and TCP is remarkably worse than the other networks. For ECMP, the next hop at each switch is determined pseudo-randomly by header field hashing, so each TCP flow traverses only one of the equal cost shortest paths. Being unable to use multi paths concurrently is especially bad for large flows. In later discussions, we focus on the Clos mode with k-shortest path routing and MPTCP.

Most importantly, we observe that different modes of flat-tree are best suited for different types of traffic. In Figure 8(a), for the network-wide traffic, flat-tree global mode has an order of magnitude improvement over the Clos network. Flat-tree local mode has similar performance to the global mode for two reasons. First, the traffic is not intensive enough to saturate the links on these topologies, although the Clos network is already heavily congested. Second, there is a considerable amount of intra-Pod traffic in the network-wide traffic. Since the global mode has richer core bandwidth than the local mode, we expect greater benefit from the global mode given heavier traffic and more inter-Pod communications.

In Figure 8(b), the performance of flat-tree Clos mode is the best due to the large proportion of intra-rack traffic. Flat-tree local mode is the second best, because the topology handles intra-rack traffic relatively well and there is still around 24.3% intra-Pod traffic. Flat-tree global mode is not very efficient for traffic with strong locality. For traffic with Pod-level locality, as shown in Figure 8(c) and (d), flat-tree local mode has the best performance, followed by the global mode and the Clos mode. This result reflects the distribution of network bandwidth. The global mode has less intra-Pod bandwidth than the local mode, but the rich network-wide bandwidth makes it more efficient than the Clos network. The difference among topologies is more significant in Figure 8(d) due to stronger locality and higher traffic volume.

These simulation results of real data center traffic on a practical data center topology validate the design purpose of flat-tree. Flat-tree can be configured into different modes to optimize traffic with different locality features, i.e. Clos mode for rack-level locality, local mode for Pod-level locality, and global mode for no locality. If the network is used for a different service, the network topology can be easily reconfigured to adapt to the new traffic. This flexibility in topology is particularly useful for public clouds where the service requirements are constantly changing. For a production data center like Facebook with integral parts of different services, flat-tree can be used in the hybrid mode with various service-specific zones, interconnected by the network core for inter-zone communication. When the services are reorganized, the network zones can be repartitioned to accommodate the change of needs.

5.3 Is flat-tree implementable?

To explore the feasibility of implementing flat-tree in practice, we build the example network in Figure 2 on a hardware testbed. As shown in Figure 9, it consists of 5 48-port packet switches, one 192-port 3D-MEMS optical circuit switch (OCS), and 24 servers each with 6 3.5GHz dual-hyperthreaded CPU cores and 128GB RAM. All links are 10Gbps. The first 4 packet switches are partitioned into edge and aggregation switches in each Pod, and the last packet switch is partitioned into the 4 core switches. The converter switches are logical partitions of the OCS. To make the testbed more manageable, we connect servers to converter switches via an extra hop on packet switches.

We implement k-shortest-path routing and MPTCP for all 3 flat-tree topologies. k is set to 4 as it yields the best performance in the simulation of this network. We realize the addressing scheme as described in Section 4.2.1. Our

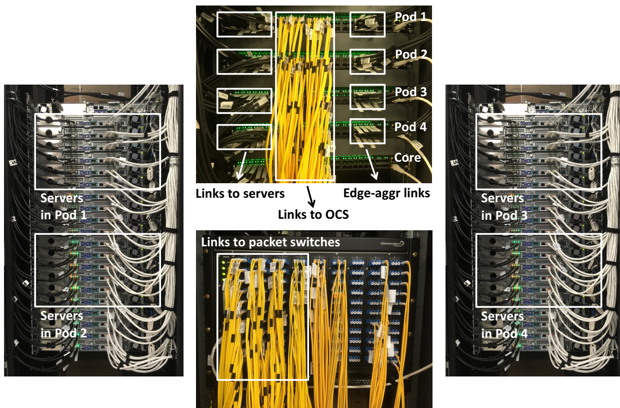


Figure 9: A testbed implementing the flat-tree example in Figure 2

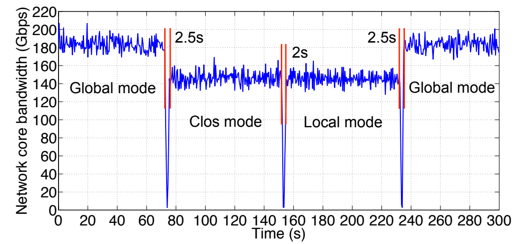


Figure 10: Summation of iPerf throughput every 0.5 second on the testbed with different flat-tree modes. Every server sends iPerf traffic to its counterparts in the other Pods to saturate the network core. Traffic adapts to the topology change in 2 to 2.5 seconds.

packet switches use a legacy OpenFlow 1.0 image. It does not support arbitrary bits matching of a field, which is necessary for source routing as shown in Section 4.2.2. So, we conduct prefix matching for the source and destination IP addresses on the switches a path traverses. The maximum number of OpenFlow rules per switch under each topology is 242, 180, and 76 respectively. The difference is due to the different number of ingress/egress switches in each topology. With source routing, these numbers will be significantly less.

We demonstrate the functionality of the testbed with an iPerf throughput experiment. On every server, we send iPerf traffic to the 3 servers with the same index in the other 3 Pods. This traffic pattern enables the measurement of the core bandwidth in the network. iPerf is set to update the flow throughput every 0.5 second. Throughout the 5-minute experiment, we change the network topology to different flat-tree modes. We add up the throughputs of individual flows to obtain the real-time bidirectional core bandwidth.

Figure 10 plots the variation of core bandwidth as we change the network topology. The local mode and the Clos mode have around 145Gbps average total throughput. Compared to the Clos mode, the local mode rearranges servers within Pods only, so there is no change to the core bandwidth. Our testbed is 1.5:1 oversubscribed, so the Clos network has $24\times$ 10Gbps/1.5 = 160Gbps total bandwidth. This result shows that the overhead of MPTCP and k-shortest-path routing is within 9.38% of the bandwidth, which is reasonable as the MPTCP packet processing lays extra burden on CPU and k-shortest-path routing is not perfect. The average total throughput in the global mode is around 185Gbps. With the power of convertibility, the network core bandwidth increases by 27.6% in this small testbed. We envision greater improvement in real data centers with a larger number of switches and more flexibility of conversion.

From the figure, we observe that iPerf reaches the maximum throughput in 2s to 2.5s after a topology change. Table 3 shows the accurate measurement of the conversion delay by the control software. The delay can be broken down into the time for reconfiguring the OCS, deleting old OpenFlow rules, and adding new OpenFlow rules. The rule deletion and installation delay are proportional to the number of rules for the topology before and after conversion. Our implementation has room for improvement. First, the legacy switches process rules more slowly than the main-stream technology [27, 37]. Second, the packet switches and the OCS are configured sequentially, and this can be easily parallelized. Even with these artifacts, the network topology can be converted in roughly 1s and the application adapts to the topology change in another 1.5s.

5.4 Does convertibility benefit applications?

Most data center applications are computation-oriented, internode communications serving the purpose of exchanging intermediate computation data. For this reason, the behavior of data transmission is influenced by many factors in the computation framework. For instance, read/write data serialization/deserialization adds to the end-to-end data transmission time; imperfect synchronization of computation nodes disorganizes traffic patterns; garbage collection may block communications, etc. In our testbed, converting the network topology from the Clos mode to the global mode improves the core bandwidth by 27.6%. However, with all these overheads from the computation framework, whether the bandwidth increase can be translated into acceleration of data center applications is yet another question.

We answer this question by running Spark and Hadoop, the most widely used computation frameworks, on our testbed. Among the 24 servers, we set the first server as the master node and all the other servers as slave nodes. We change the network topology and compare the end-to-end data read time under different flat-tree modes. The characteristics of the jobs are as follows.

Spark broadcast: we run Word2Vec, the iterative machine learning job for document feature extraction. In each iteration, the master node broadcasts the updated model to all workers. We choose the "torrent" option for the broadcast operation to distribute the data in the BitTorrent fashion. Spark promotes in-memory computation, so the data to be transmitted is readily available in memory, although data serialization and deserialization are needed.

Hadoop shuffle: we run the Sort job on Tez [9], a variant of Hadoop MapReduce. It has a heavy shuffle phase, where all the nodes as mappers send data to a subset of nodes as

Table 3: Conversion delay of the experiment in Figure 10

Topology	Configure OCS	Delete rule	Add rule	Total
Global	160ms	477ms	644ms	1281ms
Local	160ms	202ms	482ms	844ms
Clos	160ms	635ms	209ms	1004ms

reducers. We store the data on a RAM disk to prevent the hard drive being the bottleneck of data read/write.

Figure 11 shows the average end-to-end data read time and the duration of the communication phase for the above two applications. The end-to-end data read time includes the time for data serialization and deserialization. In the Spark broadcast application, flat-tree global mode reduces the average data read time by 10% and reduces the broadcast phase duration by 16% compared to the Clos topology. In the Hadoop shuffle application, the reduction in the average data read time and in the shuffle phase duration are 10.5% and 8% respectively. With this visible difference, we conclude that the improvement of network topology can be reflected in the application performance. The global mode only slightly outperforms the local mode, because their network structures are not hugely different at this small scale (Figure 2c vs. 2d). The topologies of these two modes will become less alike as the network scale grows, thus we expect more considerable performance improvement to applications from the change of topology in a large-scale data center.

6 RELATED WORK

Flat-tree is distinguished from other data center network architectures such as [10–12, 23, 24, 41] by its convertibility. Each of these fixed topologies has sweet spots for particular traffic patterns [40], whereas flat-tree is able to convert the topology to adapt to different workloads. These architectures have varying implementation complexity and performance properties. With the power of convertibility, flat-tree can be implemented more easily like a Clos network and has better performance like random graph networks.

Flat-tree also goes beyond the recent proposals of configurable data center network architectures. One group of works creates ad-hoc links at run time to alleviate hot spots [19, 22, 25, 26, 43, 44, 48, 49, 51]. Another group constructs an all-connected flexible network core with high bandwidth capacity [3, 16, 17, 33, 34]. However, these solutions are constrained by the port count of central switches when enabling configurability [16, 19, 43], the number of optical wavelengths that can be reused [3, 16, 17, 33, 34], or the interference and attenuation of wireless signals [22, 25, 26, 51]. Due to these scalability concerns, only a small number of connections can be added as a local remedy or the size of the network is limited to a small scale. Flat-tree is the first architecture to realize globally convertible data center networks at large scale. It is fundamentally different from these previous work. First, instead of adding new links to the network, it repurposes existing links to increase the total bandwidth with more efficient topologies. Second, rather than using central switches, it distributes a set of small port-count converter switches across the network to spread convertibility. Third, converter switches simply pipe out data packets through wired channels, making technologies for multiplexing signals or maintaining

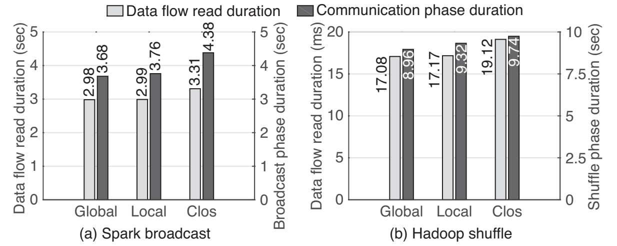


Figure 11: Average data flow read duration (left y-axis) and average communication phase duration (right y-axis) in the Spark broadcast and Hadoop shuffle applications under different flat-tree topology modes

signal intensity unnecessary. Fourth, besides reconfiguring switch-to-switch links as done in other proposals, flat-tree also reconfigures server-to-switch links to facilitate greater flexibility in the network structure.

7 CONCLUSION

Flat-tree is the first effort towards building convertible data center networks. By converting between Clos and approximate random graph of various scales, it achieves the conventionally conflicting goals of easy implementation and good performance. Convertibility can be achieved by a set of small port-count converter switches distributed across the network. They have low cost and can be packaged into Pods to ease deployment. We find flattening Clos' tree structure does not require global rewiring. With regular wiring patterns between Pods and core switches and simple connections between adjacent Pods, we effectively approximate randomness in the network core and at the same time obtain low wiring complexity. Multi-path routing and congestion control are crucial to exploiting the path diversity in flat-tree, and we have shown that aggregation strategies can be applied to avoid an explosion of network states. Existing routing and transport protocols combined with our architecture-specific state aggregation schemes can balance between high network utilization and fair bandwidth sharing among flows. We explore the implementability of flat-tree using simulations with real data center traffic and a testbed implementation of the system. We observe flat-tree can optimize for diverse workloads with different topology modes, and it brings performance improvements to applications with greater core bandwidth. Flat-tree is merely one design point in the broad space of convertible data center networks. We believe our experience will motivate future studies on convertibility.

ACKNOWLEDGEMENT

We would like to thank our shepherd Mohammad Alizadeh and the anonymous reviewers for their thoughtful feedback. This research was sponsored by the NSF under CNS-1422925 and CNS-1305379, an IBM Faculty Award, and by Microsoft Corp.

REFERENCES

[1] 2009. MPTCP simulator. (2009). http://nrg.cs.ucl.ac.uk/mptcp/implementation.html
[2] 2009. MultiPath TCP - Linux Kernel implementation. (2009). http://multipath-tcp.org/pmwiki.php/Main/HomePage

[3] 2010. Plexxi. (2010). http://www.plexxi.com/
[4] 2012. OpenFlow Switch Specification, Version 1.3.0. *Open Networking Foundation* (2012).
[5] 2015. Coflow-Benchmark. (2015). https://github.com/coflow/coflow-benchmark
[6] 2015. Segment Routing: Prepare Your Network for New Business Models White Paper. *Cisco Technology White Paper* (2015).
[7] 2017. 40G short range transceiver. (2017). http://www.fs.com/products/17931.html
[8] 2017. Facebook Network Analytics Data Sharing. (2017). https://www.facebook.com/groups/1144031739005495/
[9] 2017. Tez. (2017). https://tez.apache.org/
[10] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. August 2010. Symbiotic Routing in Future Data Centers. In *SIGCOMM '10*. New Delhi, India, 51–62.
[11] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. November 2009. HyperX: Topology, Routing, and Packaging of Efficient Large-scale Networks. In *SC '09*. Portland, OR, 1–11.
[12] M. Al-Fares, A. Loukissas, and A. Vahdat. August 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM '08*. Seattle, Washington, USA, 63–74.
[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. August 2010. DCTCP: Efficient Packet Transport for the Commoditized Data Center. In *SIGCOMM'10*.
[14] T. Benson, A. Anand, A. Akella, and M. Zhang. January 2010. Understanding Data Center Traffic Characteristics. *SIGCOMM CCR* 40, 1 (January 2010), 92–99.
[15] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica. August 2012. Surviving Failures in Bandwidth-constrained Datacenters. In *SIGCOMM '12*. Helsinki, Finland, 431–442.
[16] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen. April 2012. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *NSDI '12*. San Jose, CA.
[17] K. Chen, X. Wen, X. Ma, Y. Chen, Y. Xia, C. Hu, and Q. Dong. 2015. WaveCube: A scalable, fault-tolerant, high-performance optical data center architecture. In *INFOCOM '15*. 1903–1911.
[18] W. Dally and B. Towles. 2003. *Principles and Practices of Interconnection Networks*.
[19] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. August 2010. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM '10*. New Delhi, India, 339–350.
[20] M. Fokine, L. E. Nilsson, Å. Claesson, D. Berlemont, L. Kjellberg, L. Krummenacher, and W. Margulis. 2002. Integrated Fiber Mach–Zehnder Interferometer for Electro-Optic Switching. *Optics Letters* 27, 18 (September 2002), 1643–1645.
[21] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. 2011. Architectural Guidelines for Multipath TCP Development. *RFC 6182* (2011).
[22] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P. A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper. August 2016. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM '16*. Florianopolis, Brazil, 216–229.
[23] C. Guo, G. Lu, Da. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. August 2009. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM '09*. Barcelona, Spain, 63–74.
[24] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. August 2008. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *SIGCOMM '08*. Seattle, WA, USA, 75–86.
[25] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. August 2011. Augmenting Data Center Networks with Multigigabit Wireless Links. In *SIGCOMM '11*. Toronto, 38–49.
[26] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. August 2014. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-space Optics. In *SIGCOMM '14*. Chicago, Illinois, USA, 319–330.
[27] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M Thottan. 2015. Measuring Control Plane Latency in SDN-enabled Switches. In *SOSR '15*. Santa Clara, CA, 1–6.
[28] C. Hopps. 2000. Analysis of an Equal-Cost Multi-Path Algorithm. *RFC 2992* (2000).
[29] S. A. Jyothi, M. Dong, and P. B. Godfrey. June 2015. Towards a Flexible Data Center Fabric with Source Routing. In *SOSR '15*. Santa Clara, CA, 1–8.
[30] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. November 2009. The Nature of Data Center Traffic. In *IMC '09*. Chicago, Illinois, USA, 202–208.
[31] S. Legtchenko, N. Chen, D. Cletheroe, A. Rowstron, H. Williams, and X. Zhao. 2016. XFabric: A Reconfigurable In-rack Network for Rack-scale Computers. In *NSDI'16*. Santa Clara, CA, 15–29.
[32] T. Leighton and S. Rao. November 1999. Multicommodity Max-flow Min-cut Theorems and Their Use in Designing Approximation Algorithms. *J. ACM* 46, 6 (November 1999), 787–832.
[33] Y. J. Liu, P. X. Gao, B. Wong, and S. Keshav. August 2014. Quartz: A New Design Element for Low-latency DCNs. In *SIGCOMM '14*. Chicago, Illinois, USA, 283–294.
[34] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. August 2013. Integrating Microsecond Circuit Switching into the Data Center. In *SIGCOMM '13*. Hong Kong, China, 447–458.
[35] R. M. Ramos, M. Martinello, and C. Esteve Rothenberg. 2013. SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow. In *LCN '13*. 606–613.
[36] E. Rosen, A. Viswanathan, and R. Callon. 2001. Multiprotocol Label Switching Architecture. *RFC 3031* (2001).
[37] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. 2012. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *PAM'12*. Vienna, Austria, 85–95.
[38] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. August 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM '15*. London, UK, 123–137.
[39] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. August 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*. London, UK, 183–197.
[40] A. Singla. *Designing Data Center Networks for High Throughput*. Ph.D. Thesis. University of Illinois at Urbana-Champaign.
[41] A. Singla, C. Y. Hong, L. Popa, and P. B. Godfrey. April 2012. Jellyfish: Networking Data Centers Randomly. In *NSDI '12*. San Jose, California, USA, 1–14.
[42] M. Soliman. 2015. *Exploring Source Routing as an Alternative Routing Approach in Wide Area Software-Defined Networks*. Ph.D. Dissertation. Carleton University Ottawa.
[43] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. August 2010. c-Through: Part-time Optics in Data Centers. In *SIGCOMM '10*. New Delhi, India, 327–338.
[44] H. Wang, Y. Xia, K. Bergman, T. S. E. Ng, S. Sahu, and K. Sripanidkulchai. 2013. Rethinking the Physical Layer of Data Center Networks of the Next Decade: Using Optics to Enable Efficient *-cast Connectivity. *SIGCOMM CCR* 43, 3 (July 2013), 52–58.
[45] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI'11*. Berkeley, CA, USA, 99–112.
[46] M. C. Wu, O. Solgaard, and J. E. Ford. 2006. Optical MEMS for Lightwave Communication. *Journal of Lightwave Technology* 24, 12 (December 2006), 4433–4454.
[47] Y. Xia and T. S. E. Ng. November 2016. Flat-tree: A Convertible Data Center Network Architecture from Clos to Random Graph. In *HotNets '16*. Atlanta, GA, 71–77.
[48] Y. Xia, T. S. E. Ng, and X. Sun. April 2015. Blast: Accelerating High-Performance Data Analytics Applications by Optical Multicast. In *INFOCOM '15*. Hong Kong, China, 1930–1938.
[49] Y. Xia, M. Schlansker, T. S. E. Ng, and J. Tourrilhes. 2015. Enabling Topological Flexibility for Data Centers Using OmniSwitch. In *HotCloud '15*. Santa Clara, CA.
[50] Jin Y. Yen. 1971. Finding the K Shortest Loopless Paths in a Network. *Management Science* 17, 11 (1971), 712–716.
[51] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. August 2012. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *SIGCOMM '12*. Helsinki, Finland, 443–454.