Providing Software Support for Reconfigurable Networks

ABSTRACT

Workload patterns in Data Center Networks (DCNs) vary by time. As such, deployment of reconfigurable topologies to handle these varying workloads is a promising approach. Network efficiency can be improved by tailoring the network topology to the workload, which can reduce the average path length thus supporting more flows simultaneously. However, without proper software support, physically altering a network is very disruptive to active flows and results in significant periods of poor network utilization.

We propose a software support framework for reconfigurable networks that reduces the amount of disruption to existing flows during topology reconfiguration. This framework deconstructs the planned topology changes (link additions and removals) into a series of smaller steps where each step only affects a small fraction of existing flows. It also proactively reroutes flows before and after each step to avoid packet losses due to topology reconfiguration and allow existing flows to take advantage of newly added links. We evaluate our prototype using `iperf` and find that without software support, there is a window during which new links may be under-utilized or not used at all due to congestive collapse. In contrast, our prototype system is able to ensure those links remain usable, and quickly increase overall network efficiency after the reconfiguration.

1. INTRODUCTION

Reconfigurable networks have been proposed to reduce network hotspots in datacenters and handle increasing bandwidth and latency requirements of emerging large-scale distributed applications. The underlying technologies used to provide network reconfigurability vary widely, spanning millimeter wave wireless transmitters with adjustable antennas [1] to optical interconnects with MEMs-based circuit switches [2, 3, 4]. Early efforts propose augmenting instead of replacing a traditional static network with a reconfigurable network [3, 4]. The traditional static network ensures connectivity between all end-points, while the reconfigurable network provides direct high-bandwidth connectivity between selected end-points. Although this approach requires deploying two networks and limits reconfigurability, it is effective at masking circuit switch latency and, more importantly, it simplifies flow management across the two networks because only a small number of selected flows are affected during reconfiguration.

However, recent designs [2] have proposed reconfigurable network-only deployments that can potentially allow dramatic network topology changes. Cloud providers can take advantage of this degree of reconfigurability to ensure that virtual instances belonging to the same tenant are tightly connected even when they are located on physical machines spread across different racks in the datacenter. Companies that currently reuse their client facing compute infrastructure for offline analytics during off-peak hours can also dramatically retailor their network topologies for each workload. However, by allowing such dramatic network topology changes without having a separate static network to provide all-pairs connectivity, it is possible to create temporary network partitions that can cause agreement-related problems for some distributed applications [5, 6]. Even in the absence of network partitions, dynamic topology changes can create significant disruptions to existing flows as the links they traverse are torn down and replaced by new links.

In this paper, we introduce Roundabout, a combined flow management and link reconfiguration framework that takes as inputs the final network topology and a maximum acceptable amount of network disruption during reconfiguration, and generates a series of reconfiguration steps and flow table changes that meets the network disruption requirements while minimizing total reconfiguration time. Roundabout limits network disruption by ensuring that each reconfiguration step, which consists of removing and adding two or more network links, only affects a small number of existing flows, by creating temporary alternative routes for affected flows before a reconfiguration step, and by migrating flows on to the new links after a reconfiguration step.

To provide a practical demonstration of the importance of reconfiguration planning and support, we

present the results of an emulated network that experiences a topology change. Our evaluation demonstrates that, without proper planning, the topological changes cause existing flows to be completely cut-off and only careful tuning allows them to recover in a timely manner. However, we contrast the results of an unplanned reconfiguration with a prototype implementation of Roundabout and validate its ability to accomplish the task of reconfiguration without adverse effects to the existing flows.

The rest of this paper is organized as follows: Section 2 provides an overview of the existing literature on reconfigurable network topologies and software defined networks. Section 3 describes the issues created by performing topology changes without software support, and Section 4 outlines the Roundabout approach to proactively handling network reconfigurations. We experimentaly demonstrate the important of software support in Section 5, and provide concluding remarks in Section 7.

2. BACKGROUND AND RELATED WORK

In this section we briefly look at the evolution of reconfigurable networks and the role of Software Defined Networking in our implementation.

2.1 Reconfigurable Networks

Traditional DCNs use a rigid structure and rely on over-provisioning the network [7, 8, 9] in order to avoid traffic bottlenecks and provide network guarantees. Traffic patterns on a 1500 server production data center [10] show that a small percentage of ToR switches handle large volumes of traffic, rendering the heavily provisioned network largely underutilized.

To address this issue, a hybrid of static and reconfigurable optical networks has been proposed [11, 12, 13, 10]. Reconfiguring a hybrid network involves rerouting flows away from the optical links to the packet switched network core [12, 11]. Recent work [14] describes a purely circuit switched network that supports greater bisection bandwidth compared to hybrid and static networks [14]. Pure circuit switched networks provide the reconfigurability without the added cost of maintaining an additional network incurred in hybrid networks.

A prominent use case for reconfigurable networks is adding new clients to an existing network [15] while maintaining a high degree of network utilization and throughput. In a case study on VM placement [15], experimental results comparing smart VM placement on traditional networks to dumb placement on reconfigurable networks show that the latter is more efficient in terms of computation time and throughput.

While these results [14, 15] are encouraging, the primary drawback of reconfigurable networks is their unsuitability for latency sensitive applications due to the fact that the network can get partitioned during reconfigurations [14]. In order to become truly viable, they must have the necessary software support that manages reconfigurations without disrupting network traffic.

2.2 Software Defined Networking

Separation of data and control plane provided by programmable network protocols such as OpenFlow [16] naturally lends itself to creating reconfiguration-aware networks. We use Floodlight [17], a Software Defined Network controller, to implement a network that is aware of changes in topology *a priori*, and is able to avoid network disconnects by acting on this information.

3. MOTIVATION

In this section we outline the problems with reactive reconfiguration and argue the need for software support. We then describe Roundabout, our prototype to manage reconfigurations proactively.

3.1 Under-utilization

Reconfiguration of the network involves the removal of existing links. Without some form of software support, this will appear to the switches as link failures and any flow using a removed link will be temporarily severed. The switches then need to find an alternative path (possibly with the help of an SDN controller) for the broken flows. During the process of recovering from the link removal, TCP is experiencing dropped packets, which may trigger the congestion control algorithm to start performing back-offs. Additionally, there is no guarantee that the newly added links will actually be used once the reconfiguration is complete, because the flows have already been assigned a route through the network.

3.2 Network Partitions

Perhaps the most dangerous problem with performing reconfiguration naïvely is the possibility of temporarily causing a network partition. Many distributed protocols make the underlying assumption that network partitions are exceedingly rare. In the event of a partition, it is common to only allow a majority partition to continue to make progress and require the minority to rejoin the group at some later point. Such protocols will experience side-effects beyond the time it takes for the network to stabilize.

3.3 High tail-end latency

Network measurement statistics indicate that the majority of flows in most networks are mice flows [18] (short lived flows, transmitting a small number of packets). In a network experiencing heavy-tail traffic patterns, dropped packets caused by reactive reconfigura-

Algorithm 1 Multi-step Topology Reconfiguration

Input: T_i, T_f
Output: $S_1, S_2, ..., S_n$
1: $i \Leftarrow 1, T \Leftarrow T_i, U \Leftarrow (T_f \setminus T_i) \cup (T_i \setminus T_f)$
2: **while** $U \neq \emptyset$ **do**
3: $S_i \Leftarrow COMPUTE\_STEP(T, U)$
4: **if** $S_i = \emptyset$ **then**
5: **return** ERROR
6: **end if**
7: Perform re-routing and apply link update operations in S_i to T
8: $U \Leftarrow U \setminus S_i$
9: $i \Leftarrow i + 1$
10: **break**
11: **end while**

Algorithm 2 Compute Step

Input: T, U
Output: S
1: **for** $v_0 \in V(RSG(U))$ **do**
2: $v \Leftarrow v_0, last\_action \Leftarrow NIL, S \Leftarrow \emptyset, o = NIL$
3: **while** *true* **do**
4: **if** $last\_action = NIL$ **then**
5: $o \Leftarrow r((u, v)) \in U$
6: $last\_action \Leftarrow REMOVE$
7: **else if** $last\_action = ADD$ **then**
8: $o \Leftarrow r((u, v)) \in U$
9: $last\_action \Leftarrow REMOVE$
10: **else**
11: $o \Leftarrow a((u, v)) \in U$
12: $last\_action \Leftarrow ADD$
13: **end if**
14: $S \Leftarrow S \cup \{o\}, U \Leftarrow U \setminus \{o\}, v \Leftarrow u$
15: **if** $v = v_0$ **and** $last\_action = ADD$ **then**
16: **break**
17: **end if**
18: **end while**
19: **if** T is connected after applying $R(S_i)$ **then**
20: **return** W
21: **end if**
22: **end for**
23: **return** $\emptyset$

tions would result in large tail-end latency through the network. High tail-end latency is a serious concern for large networks, because even small periods of low performance affect a significant proportion of user bound traffic.

While it has been argued that the window of such disruptions can be reduced by adjusting the TCP retransmission timeout (RTO) [14], network providers cannot enforce this on client applications. Furthermore, regardless of TCP's RTO, affected flows still experience a window of zero bandwidth, because the network controller is always reacting to link changes. Given that network reconfigurations are planned changes, they should not cause major disruptions characteristic of network failures. To this end, software support for planned reconfiguration is necessary.

4. RECONFIGURATION PLANNING

We reduce the reconfiguration planning problem into two sub-problems. The first sub-problem involves determining a partial order of link updates to avoid network partitions and minimize the number of steps, where a step is a set of arbitrarily ordered link updates that can be applied to the topology in parallel. Given a sequence of steps, the second sub-problem is to modify flow entries in the switches to minimize the impact of the topology change on the existing flows. We assume that no additional ports on the switches are available for the reconfiguration and all switch ports are used before and after the reconfiguration.

We model the topology as an undirected graph T where the nodes $V(T)$ represent the switches and the edges $E(T)$ represent the links between switches. Every host in the network is always connected to one of the switches. The initial and final graphs are denoted by T_i and T_f respectively. For an edge $e = (u, v)$ where $u, v \in V(T)$, we use $a(e)$ to denote the addition of edge e and $r(e)$ to denote the removal of this edge. An edge update operation is $o(e) \in \{r(e), a(e)\}$.

4.1 Scheduling Link Updates

For the first sub-problem, we are looking to create a series of *steps* S_1 through S_n, where each step is defined as a set of edge addition, $A(S_i)$, and removal operations, $R(S_i)$ such that, when the steps are applied in order, will transform the graph T_i to T_f. Our solution approach, as illustrated in Algorithm 1, first determines the set of links that need to be added or removed (line 1), U, and calls $COMPUTE\_STEP$ to determine a feasible set of link operations that can be performed in parallel to form the first step. The step is then applied to generate an updated graph T, and each following step is determined and applied in the same way until $T = T_f$. For some starting and ending topologies, there does not exist a sequence of steps that will lead to a partition-free reconfiguration. In these cases, our algorithm will terminate when it reaches a configuration where the only next step is the null step 4.

The function $COMPUTE\_STEP$, as illustrated in Algorithm 2, determines the set of links to update in a step. It first picks a vertex from $RSG(U)$, which is the graph formed by the set of edges in the remaining link updates (U). Using this vertex as a starting point, it iteratively construct a walk, $(e_1, e_2, ..., e_l)$, over $RSG(U)$. In each iteration, the algorithm finds a link update per-

taining to the current vertex v (line 4 to 13) and appends it to the end of the walk. *last_action* records the type of action in the previous iteration and ensures that $o(e_i) \neq o(e_{i+1})$ $(1 <= i < l)$. This property helps satisfy the constraint that a port must be made available, by removing a link before attaching a new link to it. When the walk returns to v_0 following an edge addition, it terminates by returning the walk W. If no walk is found after iterating over all the vertices in $RSG(U)$, it returns an empty set.

We now show that Algorithm 2 is always able to find the next edge update in each iteration and finish the walk to find the set of link updates. Because the degree of a vertex remains the same after the reconfiguration, the number of edges removed from a vertex is the same as the number of edges added before the reconfiguration starts. Consider the state of vertex v at the beginning of an iteration in Algorithm 2 (line 4). If v has not been visited, then there must exist at least one edge addition and one edge removal pertaining to v; otherwise, v is not associated with any link updates, which contradicts to line 1 in Algorithm 2. It follows that we can always find a link update operation for unvisited vertices. Now consider the case when v has been visited. The algorithm guarantees that exactly one edge addition and one edge removal are included in the walk whenever it traverses v, except for the first vertex v_0. The number of edge additions and the number of edge removals pertaining to v will both decrease by 1 after the traversal. This means that the algorithm should always be able to find an edge removal associated with v when it visits v following an edge addition and vice versa, as long as there exists at least one edge update pertaining to v.

When the algorithm visits the first vertex v_0 at the beginning, however, an edge removal pertaining to v_0 is selected. For v_0, the number of edge additions becomes greater than the number of edge removal by 1. It follows that the algorithm should be able to reach v_0 following an edge addition, no matter how many times v_0 has been visited before. Therefore, the walk will terminate. When $COMPUTE\_STEP$ exits, the number of edge additions and the number of edge removals for any vertex in T remains the same. Hence, the Algorithm should still be able to finish the walk when $COMPUTE\_STEP$ is called again.

While it is possible to examine all possible walks in T with a backtracking approach, it is more expensive and still ends up with an empty set when no feasible step can be found. Our approach is more efficient when there are multiple feasible solutions in each step.

4.2 Re-routing Flows

In this section, we will provide further details about re-routing step on line 7 in Algorithm 1. Our strategy first applies the link updates in $R(S)$ to T to prevent the routing algorithm from using them because the order of link updates is arbitrary. Since the links removals will reduce the overall link capacity in the network, our routing algorithm also attempts to balance the traffic across the links to minimize network congestion. The active flows in the network are re-routed before applying link updates in order to avoid temporary traffic loss. After link updates are complete, the strategy performs another re-routing step to enable the flows to leverage the new links.

5. EVALUATION

In this section we describe the hardware environments, explain how the experiments were conducted , and compare managed against unmanaged network reconfiguration.

5.1 Experiment Design

OSA motivates the need for reconfigurable networks by examining an 8-switch 3-cube network with 5 bidirectional flows, as shown in the top-left of Figure 1. It was pointed out that regardless of the routing scheme used, at least two of the flows must share a link. Hence it is impossible to improve the network throughput without making modifications to the physical connectivity of the switches. By reconfiguring the topology to that shown in the top-right of Figure 1, all of the flows are able to achieve full link bandwidth. We make our case by examining the behaviour of flows in two different transitions:

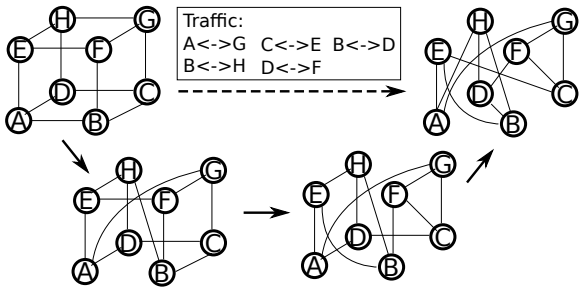

Figure 1: Multiple step vs single step reconfiguration.

- Reconfiguration-oblivious: This set of experiments emulates a network reconfiguration done without software support. The initial network configuration is setup, and iperf flows started. The network is then switched to the new topology in a single step.

- Reconfiguration-aware: This emulates a network reconfiguration with a rudimentary control mechanism. Similar to the previous case, the initial network is setup and flows started. Then the network is modified in a iterative series of link removals

and additions, creating an intermediate topology instance at each step. This is outlined as follows:

In both experiments, no new flows are introduced during the reconfiguration. We measured the aggregate throughput across all the flows in the system during the reconfiguration window.

5.2 Experimental Setup

To evaluate the impact of reconfiguration, we have emulated the aforementioned experiment using the combination of Mininet, OpenVSwitch, Floodlight, and iperf. The emulation was performed on an Ubuntu 14.04.0 LTS based server with 2 Intel Xeon E5-2630v2 2.60GHz 6-core CPUs and 256GB of DDR3 RAM. We configured each link to have 1 ms delay and 100 Mbps bandwidth. Each experiment ran for 60 seconds, and reconfiguration is applied at 20 second to allow flows enough time to converge. All flows are TCP flows started using `iperf3`.

For both experiments, we add the links before the experiments begin because we observe anomalous disruption to the active flows when the interfaces of new links are attached to the switches. The issue is acknowledged in the Mininet source code. We ensure these links are not used until the flow entries are updated.

5.3 Reconfiguration-Oblivious

In our first experiment we perform the reconfiguration of the network without any planning or software support. The results of this reconfiguration oblivious case can be observed in Figure 2. At the 20 second mark, when the reconfiguration is performed, all of the flows in the network drop to 0 Mbps because at least one of the links each flow traverses is removed.

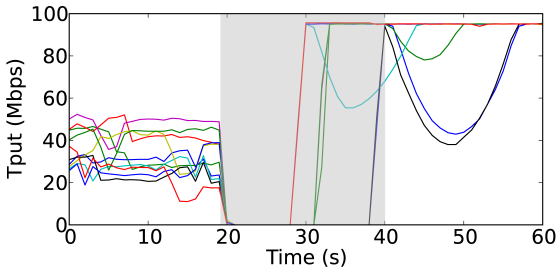

Figure 2: Reconfiguration Oblivious.

In a reconfiguration-oblivious environment the routing rules are changed reactively. To avoid dropping packets, the switch and controller must react to the link remove before any new packets arrive that are assigned to the removed link. While theoretically possible, our practical observations demonstrate that packets are dropped before the controller is capable of responding to the series of link removals. To make matters worse, Floodlight does not remove rules from the switches in response to link down events. Instead, as flows continue to send packets, the switch simply drops them until the rule's idle timeout expires, at which point it will request a new rule for the flow in question.

Floodlights default idle timeout for rules is 5 seconds, which is a logical choice for static networks as flows that exhibit periods of inactive are not uncommon, and, by having a sufficiently long idle timeout, the controller does not have to repeatedly install the same rules for bursty flows. In a reconfigurable environment, the 5 second idle timeout interacts very poorly with TCP. By the time the rules have expired, TCP has experienced multiple dropped packets and backed off to the point that it waits tens of seconds between retransmits. The result is in an artificially elongated transition period, which is highlighted by the shaded area of Figure 2.

There are a number of techniques we could employ to reduce the number of dropped packets when a link is removed. For example, the idle timeout of the flow table entries could be reduced so that the new flows are installed in a timely manner. However, any system tuning used to improve the response of a link removal maintains the fundamental flaw of reactively handling a planned event. The TCP flows should not need to be carefully tuned for reconfiguration, at the cost of potential performance degradation outside of the reconfiguration. With proper software support, no packets should be dropped because the reconfiguration should not occur without some form of preparation by the central controller that is responsible for the reconfiguration in the first place.

5.4 Reconfiguration-Aware

In contrast to the reconfiguration oblivious case, when the reconfiguration is performed in conjunction with software support, as shown in Figure 3, we observe a fast and smooth transition from the initial topology to the final topology. The primary differentiating factor of the reconfiguration aware system is that it proactively moves flows off of the links before they are removed. The benefits of proactive routing are made most clear by the fact that none of the flows drop to 0 Mbps before, during or after the reconfiguration.

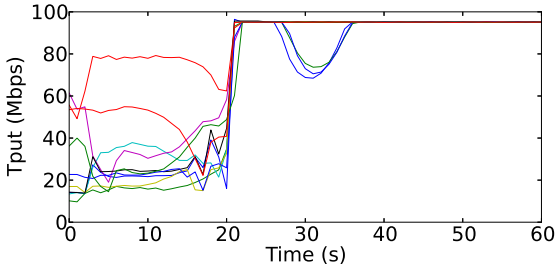

Figure 3: Reconfiguration Aware.

In important enabling factor of effective software support is breaking the reconfiguration into multiple steps. In the working example, only 4 of the 12 links are shared between the two topologies. If the requested reconfiguration was performed all at once, there would be a period of time when the network was partitioned into 3 separate sub-graphs. Such partitions make it impossible to avoid packet drops, which we have demonstrated cause lasting performance issue.

The bottom of Figure 1 shows the two intermediate steps that we use in the reconfiguration aware experiment. Breaking the reconfiguration into a sequence of steps the flows can be shifted off of links as they are removed, and onto new links as they are added, and at no point is there a partitioning of the network.

6. DISCUSSION

In this paper we have identified cases where the lack of software support for network reconfiguration can cause significant network disruption. However, in building our prototype implementation of Roundabout, we uncovered a new set of research challenges. In particular, we found that topology changes that are oblivious to existing flows can lead to network disruption even if the network is fully connected, and existing flows are rerouted proactively around the removed links. This is a difficult problem, because the objective is to provide the same performance for all of the flows, but with the use of less networking resources. In order to address this problem, we need to extend our existing planning algorithm to account for existing flows and the impact of rerouting these flows onto secondary links.

A related problem is determining the order of reconfiguration steps. Because the objective is to minimize the performance impact of reconfiguration, the ideal reconfiguration plan must take into account the total order of link additions and removals in relation to the existing flows to maximize their performance. In our planning algorithm, instead of returning a single sequence of steps, we can provide multiple sequences. This gives the flow controller the option to select a sequence that minimizes disruption.

Another possible optimization is to perform multiple reconfiguration steps in parallel in order to reduce the reconfiguration time. The trade-off comes from the impact on active traffic, which is forced to share a further reduced subset of links when multiple reconfiguration steps are realized at once. We believe that this parameter should be exposed to the network administrator and selected based on workload-related requirements.

7. CONCLUSION

Reconfigurable networks have the potential to improve efficiency by supporting more flows simultaneously, on a different topology. However, without software support taking advantage of planned link changes, we observed a period of time during which all affected flows suffer complete throughput loss. Regardless of how well the flows are tuned, a network reconfiguration without software support is will result in packet drops and network disruption.

In contrast, a solution that deconstructs a planned reconfiguration into a series of steps, and proactively reroutes flows before and after each step will limit throughput loss and maintain high network utilization. We believe that this is a critical enabler for making reconfigurable networks practical for use in large-scale datacenters.

8. REFERENCES

[1] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting Data Center Networks with Multi-Gigabit Wireless Links," in *SIGCOMM*, (Toronto, Ontario, Canada), 2011.

[2] K. Chen, A. Singlay, A. Singhz, K. Ramachandranz, L. Xuz, Y. Zhangz, X. Wen, and Y. Chen, "OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility," in *NSDI*, (San Jose, CA), 2012.

[3] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan, "c-Through: Part-time Optics in Data Centers," in *SIGCOMM*, (New Delhi, India), 2010.

[4] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers," in *SIGCOMM*, (New Delhi, India), 2010.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *SOSP*, (Stevenson, Washington), 2007.

[7] C. E. Leiserson, "Fat-Trees: Universal networks for hardware-efficient supercomputing," vol. 34, no. 10, pp. 892–901, 1985.

[8] A. G. J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. M. A., P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," in *SIGCOMM*, (Barcelona, Spain), 2009.

[9] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high

performance, server-centric network architecture for modular data centers," in *SIGCOMM*, (Barcelona, Spain), Aug 2009.

[10] S. Kandula, J. Padhye, and P. Bahl, "Flyways to de-congest data center networks," 2009.

[11] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan, "c-Through: part-time optics in data centers," in *SIGCOMM*, (New Delhi, India), August 2010.

[12] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," in *Proceedings of SIGCOMM*, (New Delhi, India), August 2010.

[13] N. Farrington, A. Forencich, G. Porter, P.-C. Sun, J. E. Ford, Y. Fainman, G. C. Papen, and A. Vahdat, "A multiport microsecond optical circuit switch for data center networking," *Photonics Technology Letters, IEEE*, vol. 25, no. 16, pp. 1589–1592, 2013.

[14] K. Chen, A. Singla, A. Singhz, K. Ramachandranz, L. Xuz, Y. Zhangz, X. Wen, and Y. Chen, "OSA: an optical switching architecture for data center networks with unprecedented flexibility," in *NSDI*, (San Jose, California), April 2012.

[15] Y. Xia, M. Schlansker, T. S. E. Ng, and J. Tourrilhes, "Enabling Topological Flexibility for Data Centers Using OmniSwitch," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, (Santa Clara, CA), USENIX Association, July 2015.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer @incollectionlechtenborger2009two, title=Two-Phase Commit Protocol, author=Lechtenbörger, Jens, booktitle=Encyclopedia of Database Systems, pages=3209–3213, year=2009, publisher=Springer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[17] "The Floodlight Project." `http://www.projectfloodlight.org/floodlight/`.

[18] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 115–120, ACM, 2004.