Datacenter Network Large Flow Detection and Scheduling from the Edge

Rui (Ray) Zhou
rui_zhou@brown.edu
Supervisor : Prof. Rodrigo Fonseca

Reading & Research Project - Spring 2014

Abstract

Today, datacenter topologies typically consist of multi-rooted trees with many equal-cost paths between every pair of hosts. Traditional protocols such as ECMP [1] or static per-flow hashing can cause substantial bandwidth loss due to the collision of large flows (a.k.a., elephant flows). In this report we present a fast dynamic flow scheduling system for detecting the large flows at the edge virtual switches in datacenter networks. Evaluation shows our real-time push notification approach performs significantly better achieving performance close to that of optimal scheduling with short lived elephant flows, comparing to traditional polling models with long polling intervals.

Elephant

1 Introduction

Studies [2] have shown that in datacenter networks the majority of flows tend to be short, whereas the majority of packets belong to a few long-lived large flows. The short flows (mice) are often related to bursty, latency-sensitive applications like search results, while the long-lived flows (elephants) are usually large transfers, such as backups or back-end operations.

The elephant and mice phenomenon has been addressed as an issue for network performance. Different applications have different requirements and constraints on the network resources. Elephant flows tend to fill network buffers end-to-end and introduce significant delay to the latency-sensitive mice flows that happen to share the same buffers, which leads to performance degradation of the network. In addition, state of the art hash-based multi-path routing methods (e.g., ECMP [1]) used in datacenters could hash multiple elephant flows onto one same link while leaving other links free and cause suboptimal network usage.

Therefore, it would be much desirable to handle elephant flows differently than mice flows. Doing so requires detecting elephant flows and signaling their existence. With Software-Defined Networking (SDN [3]), we can then inform a traffic engineering module at the controller level to route elephant flows properly.

Traditionally, the detection of elephant flows is achieved through periodically polling (e.g., Hedera [6]), or sampling technology (e.g., sFlow [4]). The Hedera approach uses five-second polling period, this level of granularity leads to possible network congestion between polls. Given current fast datacenter networks with 10Gps or even faster links, it is possible to drop many packets between polling intervals due to late detection of elephant flows. It is also possible that a short-lived elephant flow would stay in an undesired route during of its entire existence. sFlow [4] based sampling technology requires sending samples of all flows to a remote controller, which then determines the existence of elephant flows based on the samples. This approach can lead to considerable amount of sampling traffic, which may become extra burden to the already congested networks.

We may consider both of the aforementioned approaches a compromise to the fact that physical switches normally are equipped with powerful dedicated ASICs for data-plane switching processes, but weak CPUs for control-plane or any generic software defined tasks. Thus functionality like flow analysis and real-time push notifications are either impossible to program or too expensive to run within the switch. However, the emerging of virtual switches and the fact that they run in actual x86 boxes with powerful CPUs enables us to do more complex computation at the switch side. In this paper, we enable real-time notifications from the switch side, so the SDN controller will always get to know the elephant flows in real-time and response to re-schedule the optimal routes for all kinds of elephant flows as fast as possible.

2 Background

2.1 Datacenter Topology

We built our datacenter network based on the FatTree [5] topology. Comparing to the traditional datacenter topology "multi-rooted hierarchical tree" (Fig 1), the interconnection of the commodity switches gives us some advantages, such as: power saving, heat dissipation reducing. Most importantly, it gives us a high degree of available path diversity. Between any source-destination pairs there are $(k/2)^2$ equal cost paths each of which corresponds to a core switch, where k is the number of ports in each switch. In order to achieve the full bisection bandwidth, the default ECMP [1] approach is not sufficient due to the nature of collision by hashing. To reduce the chance of elephant flows collision, we need to use a more intelligent algorithm to utilize the SDN [3] architecture. Therefore, we apply the algorithm in Hedera [6].

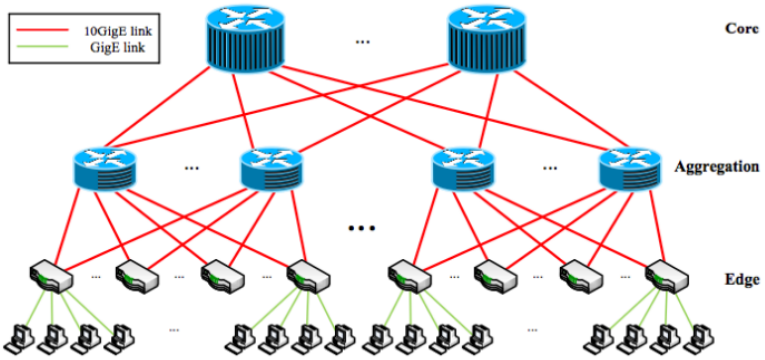

Figure 1: a common multi-rooted hierarchical tree, as shown in [6]

2.2 Hedera

Hedera was presented to solve the problem we mentioned in last paragraph. It is a dynamic flow scheduling system, in which an SDN controller periodically polls switch counters to identify elephant flows. Due to limitations in switch hardware, the polling in Hedera is done at five seconds intervals. Thus Hedera could not respond to emerging elephant flows in real-time. In our project (OVS-Hedera), We implemented Hedera demand estimation algorithm for estimating flow demands, and global first fit algorithm for scheduling based on an important assumption: all flows are network limited which we will describe in later chapters. Instead of five seconds polling, we use real-time elephant flow notifications.

2.3 OVS and Virtual Network

We use the Open vSwitch [8] as the edge switches and aggregation switches in our datacenter networks, in which we assume that all physical machines run hypervisors. This is common in cloud datacenters, and is advocated, for example, in [9]. Open vSwitch is a software switch which supports standard management interfaces and protocols and is also designed to enable massive network automation through programmatic extension. We choose a software-based switch due to the following reasons:

1. The virtual switches are more flexible and can implement all the complex functionality that interacts with the hosts. The core network infrastructure can instead focus on supporting the simple connectivity and data transfers between hosts.

2. The virtual switch can utilize the compute power from the hardware it is running on, which we expect to have much better computation ability than the control processors in common physical switches.

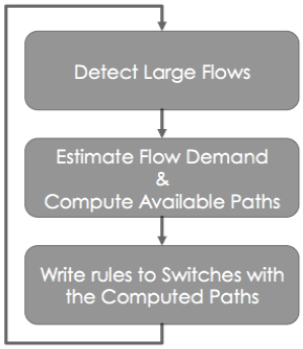

Figure 2: OVS-Hedera control loop, as shown in [7]

3 Architecture

We leave the details on machine setup in the next chapter. In a high level abstraction, there are three steps in our OVS-Hedera control loop, shown in Figure 2:

1. Detecting elephant flow at the edge switch
2. Estimating the natural flow demand
3. Scheduling and routing

3.1 Elephant Flow Detection

We created a module (Elephant flow monitor) in the Floodlight OpenFlow Controller [10] which listens to notifications sent from our OVS monitors. Each OVS monitor watches over activities on all OVSes within one physical machine. It acquires per-flow statistics from each OVS' flow table, measures throughput of each active flow, and notifies the controller when a flow's throughput exceeds a certain threshold, which is set to 10% of the maximum bandwidth of a link, i.e., 100Mbps threshold for 1Gbps links. Our OVS monitor is implemented as a Python script running on the same physical machine that OVSes run. OVS provides a command line interface through which we can send dump-flow requests. The OVS monitor script periodically queries this interface and parses the query result to get flow statistics. An alternative approach is to modify OVS source code to send a message to controller automatically when the flow tables that reside in OVS kernel space indicate that an elephant flow emerges. Although we have managed to make OVS to send such message, we find it difficult to customize the message content to provide enough information to controller for re-routing from within the kernel. Due to time limitation, we currently use the query script in our testbed.

```
ESTIMATE-DEMANDS()
1   for all i, j
2       M_{i,j} ← 0
3   do
4       foreach h ∈ H do EST-SRC(h)
5       foreach h ∈ H do EST-DST(h)
6   while some M_{i,j}.demand changed
7   return M

EST-SRC(src: host)
1   d_F ← 0
2   n_U ← 0
3   foreach f ∈ ⟨src → dst⟩ do
4       if f.converged then
5           d_F ← d_F + f.demand
6       else
7           n_U ← n_U + 1
8   e_S ← (1.0 − d_F) / n_U
9   foreach f ∈ ⟨src → dst⟩ and not f.converged do
10      M_{f.src,f.dst}.demand ← e_S

EST-DST(dst: host)
1   d_T, d_S, n_R ← 0
2   foreach f ∈ ⟨src → dst⟩
3       f.rl ← true
4       d_T ← d_T + f.demand
5       n_R ← n_R + 1
6   if d_T ≤ 1.0 then
7       return
8   e_S ← 1.0 / n_R
9   do
10      n_R ← 0
11      foreach f ∈ ⟨src → dst⟩ and f.rl do
12          if f.demand < e_S then
13              d_S ← d_S + f.demand
14              f.rl ← false
15          else
16              n_R ← n_R + 1
17      e_S ← (1.0 − d_S) / n_R
18  while some f.rl was set to false
19  foreach f ∈ ⟨src → dst⟩ and f.rl do
20      M_{f.src,f.dst}.demand ← e_S
21      M_{f.src,f.dst}.converged ← true
```

Figure 3: Demand Estimation Algorithm from Hedera [6]

3.2 Demand Estimation

We assume the flows' behavior is limited by the same condition as in Hedera.

We expect a TCP flow's demand will grow naturally and eventually reach the full bandwidth of a link. It would also possibly become limited by the sender or the receiver NIC.

The natural demand of TCP flow is an important assumption for our demand estimator. The goal of demand estimator is to calculate the max-min fair bandwidth allocation for flows, in order to help our scheduler make intelligent decisions for re-routing.

The demand estimator takes a set of large flows, detected by the edge switches and performs iterations repeatedly to increase the capacity of flows from the sources and decrease exceeded capacity at the receiver until the flow converges.

Figure 3 is the algorithm of the demand estimator. The following is a simple example of how demand estimator works.

In Figure 4, sender A would like to send two flows to receiver X and Y. Under

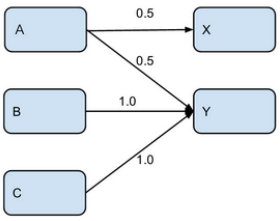

Figure 4: Initial Flow Demands

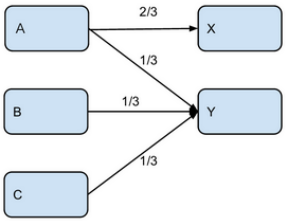

Figure 5: Converged Flow Demands

our assumption for fairness each flow can only share 50% of the total width since the flow grows and it is network limited. However the receiver Y limits the flows and for fairness, each flow can only share one third of the bandwidth. Therefore, after running the demand estimator the result will be like Figure 5, because the flows are eventually limited by both the receiver and the sender.

```
GLOBAL-FIRST-FIT(f: flow)
1   if f.assigned then
2       return old path assignment for f
3   foreach p ∈ P_src→dst do
4       if p.used + f.rate < p.capacity then
5           p.used ← p.used + f.rate
6           return p
7   else
8       h =HASH(f)
9       return p = P_src→dst(h)
```

Figure 6: Global First Fit Algorithm from Hedera [6]

3.3 Scheduling and Re-routing

Figure 6 is the global first fit algorithm. This algorithm picks the available route for each flow waiting to be placed greedily, searches all the available routes and find out the first path that has enough capacity to place our flow. In order to do so, the central controller provides two important features. First, it keeps the network topology in the memory. Secondly, it keeps track of the left-over capacity in all possible routes to the granularity of switch ports. After all those steps, our controller installs the rule on the switches where the flows pass through. Each of the rules is composed with a series of forwarding decision for the matching flow at each of the switches in the route. Those rules will keep in effect, unless no further packets matching the rules arrive at the switches. When a rule has been idle without affecting any packet for more than five seconds, it expires. We never revoke any flow rules manually.

One important thing to note here is that global first fit doesn't guarantee that all the flows are being placed. If there is no route with enough capacities left to place an elephant flow, we will have to let the default ECMP algorithm handle the flow.

4 Implementation

We have built a test bed for our project here in the system lab of Brown University. The testbed is highly virtualized. The instances/hosts in our test bed are virtualized x86 machines running Ubuntu 12.04 cloud image. Other than the core switches, all the edge switches and the aggregation switches are Open vSwitch instances. Finally all the switches instances are managed by one Floodlight OpenFlow controller, except for one common switch that connects our control plane.

4.1 Hardware Architecture and Software Stack

We constructed our testbed based on our first OpenStack [11] cluster at the system lab of Brown university. As shown in 7, our OpenStack test bed includes four powerful blade servers as compute and storage nodes, each features an Intel(R) Xeon(R) CPU E5-2407 with four CPU cores and eight true threads. Each of the compute node has 16GB of memory. Every compute node runs an instance of the Nova-Compute service, which is the primary service to start and maintain virtual machines in OpenStack. One key highlight of our compute servers is that each of them are equipped with eleven network interfaces. Other than one IPMI interface that is not exposed to the system, the rest ten interfaces are all available to the operating system and the virtual machines running in the node. In our lab we were able to utilize PCI-pass through methods to assign the physical interfaces to the virtual machines for other projects. In our project arrangements, each aggregation OVS instance is connected to two physical NICs, through which the OVS instances were able to connect to the physical core switches. By providing substantial physical NICs, our virtualized testbed would not suffer insufficient network speed that

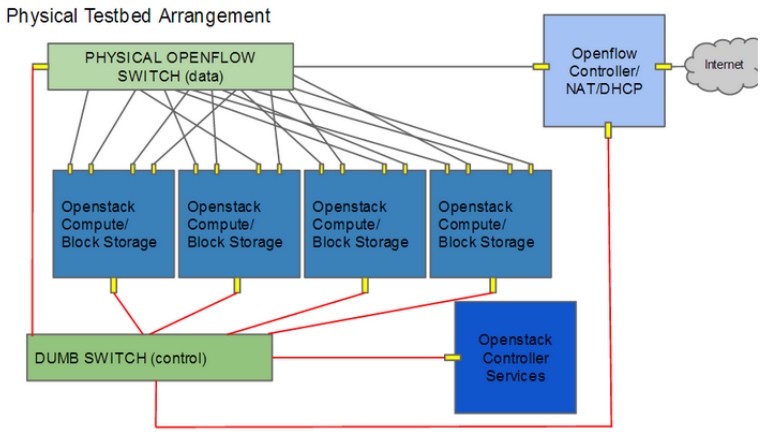

Figure 7: This graph shows our arrangement of physical testbed. The green boxes represents switches. The blue boxes represent physical machines. Yellow pieces are interfaces of note(we ignore the obviously existing interfaces/ports on the switches). The gray lines are Ethernet wires for the data plane, and the red lines represent the wires that are used in our controller plane. Note that the OpenStack control plane and the OpenFlow control plane shares the same network/subnet.

happens in purely virtualized networks. We dedicate one moderate machine as our OpenStack Controller, which runs services including Keystone(Identity Service), Glance(Image Service) , Horizon(Web Dashboard) and a NTP server to sync the time in the cluster. All the compute nodes and the controller are connected by the control plane switch. Besides the dedicated OpenStack controller, we also have a network controller. This network controller provides an instance of Floodlight OpenFlow Controller, a DHCP server as well as an access point to the internet with NAT service enabled. The core data-plane OpenFlow switch's management interface is also connected to the control plane switch, through which the core data plane OpenFlow switch is managed by the Floodlight controller instance running in the network controller node. The DHCP server in the network controller assigns static IP addresses to all the OpenStack nodes, and provides Internet access to them as well. Finally the Network controller node also connects to the data-plane OpenFlow switch's data-plane interfaces, which enables the remote access to virtual machines running in our OpenStack compute nodes. All of the machines run Ubuntu 12.04, and the version of our OpenStack is Havana.

4.2 Virtualized Test Bed Arrangements

To reproduce the testbed of Hedera, we have worked on two steps. First, we need to construct each pod of four VMs and four OVSes in each of our physical compute node. To do this, we first batch initialize four VMs in each compute node. Note that Nova-Network, the default network management service associated with Nova-Compute will bridge all the VMs in one compute node to one bridge, which requires us to unbound the tap device from it. After initialize the VMs, we create OVS instances with OVS provided CLIs, and attach the tap devices of VMs and

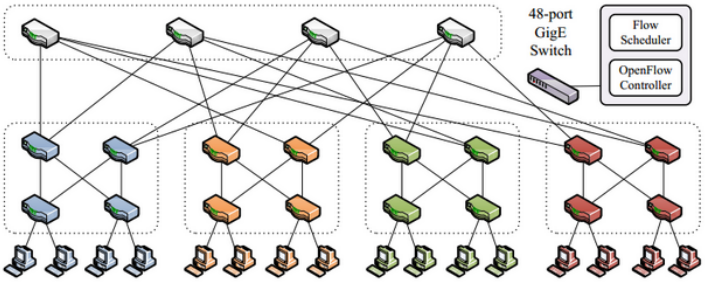

Figure 8: original Hedera [6] testbed, note that all the switches and hosts are physical existing.

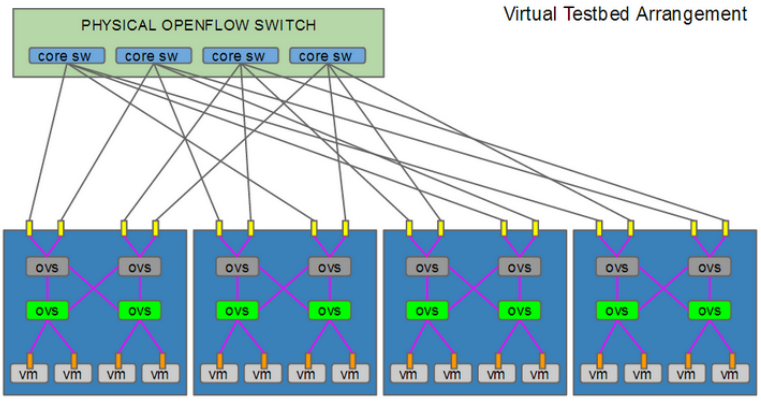

Figure 9: This graph shows our arrangement of the logical/virtualized testbed. It is a reproduction of Hedera's testbed based on OpenStack, OVS and Floodlight. The pale-green box represents the physical core switch, which is instructed by our Floodlight controller to work as four separated common switches(the core switches represented as small blue boxes). Every big blue box represents one machine that runs OpenStack Compute service, in each of them runs four VMs and four OVS instances. The green OVSes are the edge switches we monitor and the grey switches are the aggregation switches. The pink wires in between OVSes are linux virtual ethernet link pairs.

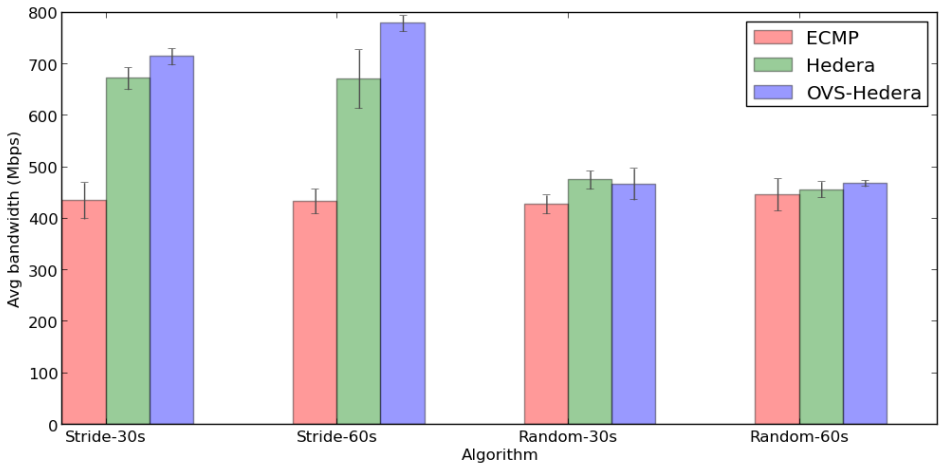

Figure 10: Comparison between different traffic engineering approaches, including default ECMP, Hedera like five seconds notification and OVS-Hedera's real-time notification

the physical NICs to its related OVS instances. Finally, the tricky part is to connect the OVSes. We utilize Virtualized Ethernet pairs to achieve such goal.

5 Evaluation

We have run a set of twelve tests on our test bed. During those tests, we have used three different kinds of push notification modes: 0.5 second monitoring and pushing as an representative of OVS-based real-time push notification model; 5 second monitoring and pushing as an representative of original Hedera polling approach; and finally the default ECMP model where there is no push notifications to the controller. Note that 0.5 second is the default interval that OVS post its counters from kernel to user space and make them available to our monitoring programs. In other words, 0.5 second is the smallest monitoring interval we can achieve with official unmodified OVS. Based on our limited test, we believe that modified OVS can co-operate with 50ms monitoring without affecting its performance.

Two different traffic patterns are used in the tests. The first one is Stride, in which a host with index x sends to the host with index (x + i)mod(num hosts). Note that in the stride pattern, we always make sure the elephant flows take the long route that includes the core switch, and there is always an optimal flow arrangement to achieve full speed in all the links. The second one is Random, in which a host sends to any other host in the network with uniform probability. In the random pattern, it is quite normal that no optimal solution exists to achieve full speed on every link.

Finally, we tested on two different kind of flow lengths: 30 seconds and 60 seconds. We did not test very short flows because in our extensive trials we observed that short flows often disappear before reaching the maximum speed possible. In which case the speed of short flows are highly affected by the TCP's sensitive nature to loss before it finally converges. We also do not test very long flows, because

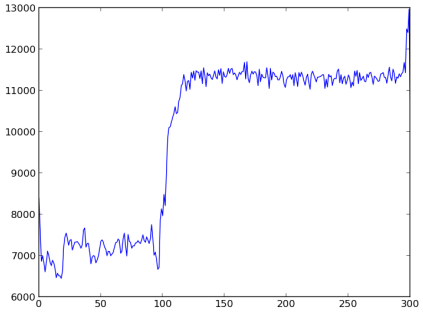

Figure 11: Total bandwidth before and after re-routing

longer flows blurs the OVS-Hedera's advantages on fast elephant flows detection. If a flow exists for a day, it does not matter much that we detect it five seconds earlier than Hedera.

Each of the 12 tests were carried out three times and the average as well as the standard deviation of each tests are calculated as the final results [Figure 10]. From the result, we can clearly observe that our OVS-Hedera performs better to find optimal solutions if one exists. When there is no possible optimal arrangement to use all available bandwidth, flow arrangements tend to fall back to ECMP.

We also performed a test to verify the performance of our algorithm, we picked 8 hosts from the total number of 16 hosts sending elephant flows to the rest(8 hosts left). If we use the default ECMP algorithm, the collision of the flows is nearly inevitable. However, if we use global first fit instead, according to 4.1 we would have 4 different paths for each source and destination pairs. Therefore, each of the flows should be able to be rerouted to a path that enables them to utilize the full bandwidth if our algorithm works as we expected. [Fig. 11] and [Fig. 12] are the results of the experiment. From the figures, we can see that in the beginning the average and total bandwidth shows that the collision did happened. After turning on the elephant flow monitor, the bandwidth increases rapidly. We can see that our OVS-Hedera works correctly, and the result is reliable.

6 Limitations and Future Work

Due to the time limitations, we did not fully evaluate our approach using the modified OVSes with monitoring intervals shorter than 0.5s. However, from our limited tests, we believe a sub 50ms monitoring interval is feasible in our testbed. Given a powerful enough testbed, there is no limitations on how close we can get to truly real-time elephant flow detection and push notifications.

Another major problem is that TCP flows are very sensitive to loss, and takes around ten seconds in jitters before reaching maximum speed in our test bed which

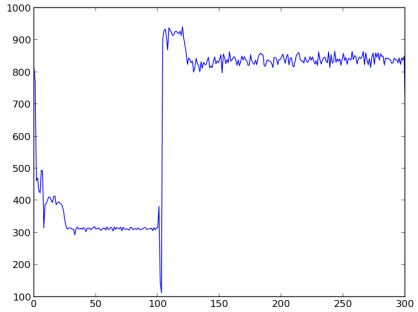

Figure 12: Bandwidth of a flow before and after re-routing

was a limitation to our experiment. Even when the optimal route has been assigned to such flow and it has been properly re-routed within half a second after being identified as elephant flow. We believe that a optimized testbed to minimize jitters could improve the evaluation results significantly.

7 Related work

The datacenter topology was based on FatTree [5], our traffic engineering solution was based on the improvement of Hedera [6],where we replace the physical switch with OVS, and significantly increases the pulling rate from 5s to 0.5s (50ms with modified OVS).

Previous research such as MATE [13] and TeXCP [18] was focus on works on scheduling flows in the multi-path environment too, however they requires special supports form the switch since they will need the explicit congestion notification packets, which we would be easily done by Open vSwitch.

Projects like Ethane, 4D [15] share the same spirit with our projects, besides that the whole project of us was built on OpenFlow [16] protocol switches.

8 Conclusion

Nowadays, due to the rapid growth rate of the global users in datacenter network, the utilization rate of the bandwidth has become much more important. With significantly increased bandwidth of links in datacenter networks, even a short delay in the detection of elephant flows could result in big loss of the overall performance in the datacenter networks. Hedera completes ECMP in the sense that it can re-schedule long-lived elephant flows to optimal routes, but its significant five seconds polling interval falls slow for short-lived elephant flows. With the emerging utilization of OVSes at the edge of the networks, we are able to achieve real-time

push notification to network controllers from the switches. With such real-time notification, we can re-schedule even short-lived elephant flows quickly, and optimize the overall usage of the networks as fast as possible. As shown in our evaluations, our OVS-Hedera performs significantly better with short lived elephant flows to find the optimal scheduling solution. Given further elaboration into the details of our project, we have the confidence that real-time push notification model in our OVS-Hedera can become one of the most promising traffic engineering solutions for modern datacenter networks.

9 Acknowledgment

I would like to provide my most sincere thanks to Cheng-lun Chen and Yujie Wan for their extensive help during this project. Also I would like to thank Jeff Rasley and Rodrigo Fonseca for their huge support and guidance.

References

[1] Hopps, Christian E. "Analysis of an equal-cost multi-path algorithm." (2000).

[2] Blog post by Martin Casado, http://networkheresy.com/2013/11/01/of-mice-and-elephants/

[3] Feamster, Nick, Jennifer Rexford, and Ellen Zegura. "The Road to SDN." Queue 11.12 (2013): 20.

[4] Phaal, Peter, Sonia Panchen, and Neil McKee. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks. RFC 3176, 2001.

[5] Al-Fares, Mohammad, Alexander Loukissas, and Amin Vahdat. "A scalable, commodity data center network architecture." ACM SIGCOMM Computer Communication Review. Vol. 38. No. 4. ACM, 2008.

[6] Al-Fares, Mohammad, et al. "Hedera: Dynamic Flow Scheduling for Data Center Networks." NSDI. Vol. 10. 2010.

[7] Hedera illustration PPT, www.cs.uky.edu/ qian/CS685/Hedera_Xuzi.pptx, May 2014

[8] Pfaff, Ben, et al. "Extending Networking into the Virtualization Layer." Hotnets. 2009.

[9] Martín Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In Workshop on Hot Topics in SDN (HotSDN), 2012.

[10] The Project Floodlight, http://www.projectFloodlight.org/documentation/ , May 2014

[11] The project OpenStack official website, https://www.OpenStack.org/, May 2014

[12] Kandula, Srikanth, et al. "The nature of data center traffic: measurements & analysis." Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference. ACM, 2009.

[13] Elwalid, Anwar, et al. "MATE: MPLS adaptive traffic engineering." INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. Vol. 3. IEEE, 2001.

[14] Caesar, Matthew, et al. "Design and implementation of a routing control platform." Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005.

[15] Greenberg, Albert, et al. "A clean slate 4D approach to network control and management." ACM SIGCOMM Computer Communication Review 35.5 (2005): 41-54.

[16] McKeown, Nick, et al. "OpenFlow: enabling innovation in campus networks." ACM SIGCOMM Computer Communication Review 38.2 (2008): 69-74.

[17] Pfaff, Ben, et al. "Extending Networking into the Virtualization Layer." Hotnets. 2009.

[18] Kandula, Srikanth, et al. "Walking the tightrope: Responsive yet stable traffic engineering." ACM SIGCOMM Computer Communication Review. Vol. 35. No. 4. ACM, 2005.