Evaluation of DOT with Hedera

Vishal Chaudhary
Dept. of Computer Science
University of Waterloo
v3chaudh@uwaterloo.ca

Harris Rasheed
Dept. of Computer Science
University of Waterloo
h2rashee@uwaterloo.ca

***Abstract*— Although Software Defined Networking (SDN) is a concept that has been around for almost two decades, recent major developments have made it the subject of interest over the past few years. Since its birth in the mid-90s, SDN has led to inception of many Software-Defined X entities like Software-Defined Storage, Software-Defined Data Centers and Software-Defined Environment. Despite much research being done, Mininet, a Stanford-based project, which is limited to resources of a single machine, is still currently the de-facto standard in testbed emulation for SDNs. However, with the recent stable release of Distributed OpenFlow Testbed (DOT), a new OpenFlow testbed option has emerged for researchers which is efficient, reliable and scalable. It overcomes some significant scalability limitations of Mininet. In this paper, we take a closer look at the aspect of scalability, in a couple of interpretations, of DOT with the implementation of Hedera, a dynamic flow scheduler for data center networks.**

***Index Terms*—Hedera, DOT, Mininet, Software-defined networking.**

I. Introduction

In recent times, Software-Defined Networking (SDN) has emerged as an architecture that is dynamic, efficient, effective and adaptable, making it the ideal for today's data centers which are complex and saturated from a hardware perspective. SDN is based on the principle of decoupling the network control plane and data plane. This leads directly to a programmable control plane and makes underlying infrastructure abstracted from the network applications and services. Before the birth of SDN, data centers were using many different kind of hardware modules for different services like intrusion detection, fault tolerance etc. With the steep growth in the usage of internet over the globe in the last decade, the network traffic and its management has become a major issue for data centers. SDN simplifies this issue by replacing software service applications in place of hardware modules for all the services. Also, it uses these software modules to manage data plane through control plane. However, testing network applications remained unexplored till Stanford developed Mininet, a virtual testbed which can be used to test network applications. Since then, Mininet has been become the testbed of choice for testing network applications on SDN. But, nothing is perfect in this world.

Even though, Mininet is being used in all the SDN related areas for application testing, the scalability factor is still a question for Mininet. Mininet's physical power is limited to the single machine it is being run on. Because of this, the testing results of Mininet for large scale applications are unreliable and might differ significantly with the real world scenario. To overcome this limitation of Mininet, the David R. Cheriton School of Computer Science Graduate Networking lab of University of Waterloo innovated the idea of DOT (Distributed OpenFlow Testbed) which can be scaled over a distributed network and is capable of performing all the tasks that Mininet can. DOT is able to distribute and provision multiple computers based on the total physical resources of the hardware. To test a larger scale experiment, more hardware nodes would need to be added and deploy the desired topology. It also facilitates the user to add multiple controllers to the topology based on the requirement. As a result in this paper, our motive is to evaluate DOT`s efficiency, reliability and most importantly, scalability.

Now, for experimentation and evaluation purposes, we needed a system which can be used as a base to evaluate the scalability of DOT. We chose Hedera [1], a dynamic flow scheduler for Data centers that adaptively schedules a multi-staging switching fabric to effectively utilize network resources. It gathers flow information from switches in the network and computes non-conflicting paths for the flow and then instructs switches to re-route traffic accordingly. Hedera consists of two main parts i.e. ECMP (Equal Cost Multi-Path) routing and the implementation algorithm. The implementation algorithm are one of Global First-fit approach and the Simulated Annealing approach. Both are discussed in detail in later sections.

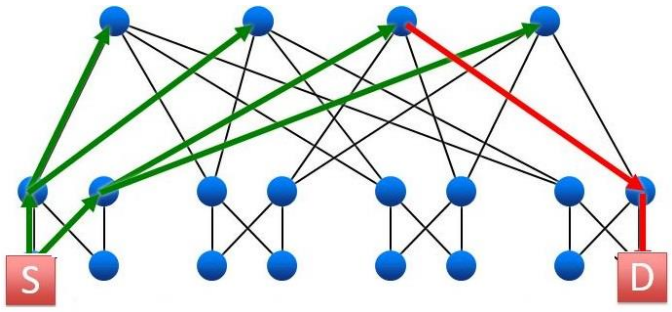


Figure 1: ECMP

Equal Cost Multi-Path is a routing algorithm which efficiently manages packet forwarding in a network grid where there are multiple paths between two hosts. Cost can be calculated based on available bandwidth, network traffic, number of switches and many other factors. In the topology, there are many equal cost paths going up to the core switches. It randomly allocates paths to flows using the hash of the flow and tries to keep long-lasting flows (Elephant flows: flows with size

more than 10% of the link capacity) on different paths. Collisions of elephant flows can have a strong negative effect on the network throughput and ECMP is responsible for minimising these kind of collisions. Moreover as shown in Figure 2, collisions may happen in two different ways namely upwards and downwards and ECMP needs to take care of both ways. Two flows are using the link between S1 and S2 creating an upward collision. Similarly, link between S3 and S4 is used by other two flows leading to a downward collision.

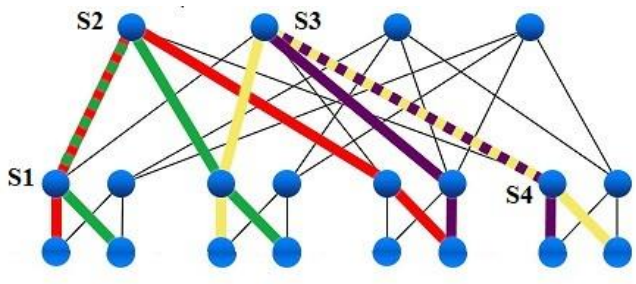


Figure 2: Collision in upward and downward flow

Another important concept need to be discussed in ECMP is about the fat trees. Fat tree [4] is a special variation of a tree, which has skinny links all over the network. Links are thinnest the bottom of the network i.e. between hosts and switches. The links become thicker in terms of bandwidth near the root of the tree. For the purposes of the project, we make use of fat graphs as opposed to trees. By definition [3], a tree is an undirected graph in which two vertices are connected by exactly one path. Hedera works on the ECMP concept which requires multiple paths between two vertices or hosts. Hence as shown in Figure 7, in the topology, there is a complex grid between all the switches whereas each host is connected to only one switch. Hence, we can say, it's a Fat graph, not a Fat tree.

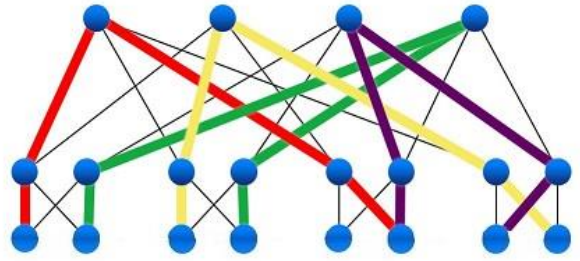


Figure 3: Collision less flows

Hedera can be implemented using two ways, by Global first-fit and by Simulated annealing. Both of these ways effectively use ECMP to make the network less congested by managing flow paths effectively. These methods are explained more in experiment design section.

The rest of this paper is organized as follows. We discuss the related work in the Literature Review section followed by a detailed system description. Then, we describe basic experiment design and evaluation using our prototype system. In the penultimate sections, we provide some discussion and future work for this study. At the end, we conclude our paper and acknowledge the contributors.

II. Literature Review

Today, there are many network emulators that are available which can be used to test SDN applications. Few of them are open and free for community-driven development whereas others are commercial and require a valid license for usage. For example, for academia purposes, Mininet is considered to be the best option available whereas HP Network simulator requires a license for commercial usage. In this paper, our main concentration is on DOT and Mininet but there are other network emulators available like EstiNet, MaxiNet, NS3 etc.

EstiNet [8] is a software tool for network planning, testing, education, protocol development and applications performance predictions. It operates differently to DOT and Mininet in certain ways. It makes use of the real-life Linux TCP/IP stack to generate high-fidelity simulation results [10]. It's also compatible with UNIX network monitoring tools which makes it more flexible. It supports both of the simulation mode and the emulation mode. All known open source OpenFlow controllers like NOX, POX and Floodlight can be used on the top of the controller node. EstiNet tries to overcome Mininet limitations. Mininet has lack of performance fidelity because it provides no guarantee that a host in Mininet that is ready to send a packet will be scheduled promptly to send the packet. Also, in Mininet there is no guarantee that all switches in network topology will forward packets at the same rate. The packet forwarding capability of a switch is unpredictable in Mininet. EstiNet overcomes these significant limitations of Mininet which makes it superior to Mininet. In IEEE communication magazine September 2013 edition, a paper [9] entitled "EstiNet OpenFlow Network simulator and emulator" was published which compared EstiNet with Mininet and NS-3 using their capabilities, performance and scalability. The paper concludes that EstiNet is more scalable, accurate and reliable than Mininet.

Apart from this, another paper was recently published in IEEE ISCC conference 2014, entitled "Comparison of SDN OpenFlow Network Simulator and Emulators: EstiNet vs. Mininet". In this paper, authors try to compare and evaluate the correctness, performance, and scalability of EstiNet OpenFlow simulator, EstiNet OpenFlow emulator and Mininet OpenFlow emulator over a set of grid networks. The authors used Floodlight OpenFlow controller without any modification to control the simulated OpenFlow switches. It was found that in EstiNet simulations, simulated results were always correct and repeatable whereas Mininet generated some strange results which were not repeatable and cannot be explained over a few network sizes. Also, it was concluded that Mininet requires more time for program launch, network setup and resource releasing when the network size is large. On the other hand, EstiNet emulator generated good performance and scalability and used less time to generate results.

MaxiNet [7] is a product of University of Paderborn, Germany, which extends the Mininet emulation environment to span the emulation across several physical machines. It also allows to emulate large SDN networks just like DOT. Although

the idea of MaxiNet is very similar to DOT, it uses different terminology and has different architecture when compared to DOT. MaxiNet runs a pool of multiple machines called workers. Each worker has a Mininet emulation which emulates a part of the whole network topology whereas DOT has its own architecture without any overlap with Mininet's. In MaxiNet, hosts and switches are connected using GRE tunnels across different workers. It also facilitates a centralized API for controlling the emulation. It provides support for POX. MaxiNet was released just three months after the first DOT release. There are a few more variety of OpenFlow testbeds that include single machine architecture or distributed ones such as OFELIA, PlanetLab, Emulab, GpENI [11].

As discussed above, EstiNet is one example that performs better in scalability than Mininet. This presents an area of future interest to investigate and compare its performance as a testbed with DOT. It also is able to deal with larger network deployments and provisioning more efficiently than Mininet. With these concerns, Mininet's popularity is somewhat questionable with the benefits that the newer testbed options are offering. Its widespread use could be attributed to it being better-known amongst researchers than the newer testbeds. Mininet was no doubt a popular option because of the advantages it offered over full system virtualisation, hardware testbeds and some simulators [5].

III. System & Experiment Design

For the purpose of experiments and evaluation, we used a cluster of machines to run our DOT manager and deploy the network topology. As shown in the below Figure 4, the physical topology being used for our experiment includes two nodes (with IP address 10.0.0.12 and 10.0.0.18) connected to a DOT manager which has the Floodlight controller. The Floodlight controller runs on the DOT manager. Both machines are facilitated with 4 core CPUs, 8GB RAM and 80GB HDD storage. Performance of the system can be enhanced with a better configuration physical machines.

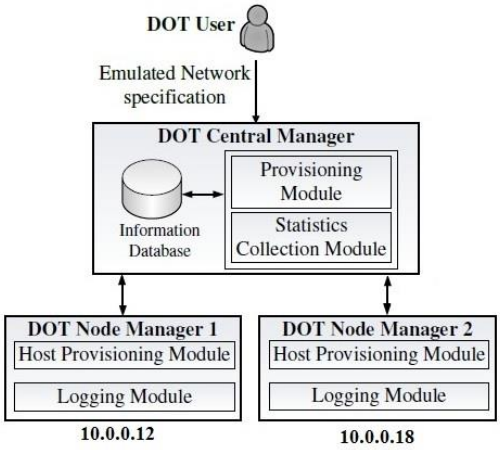


Figure 4: DOT Architecture

DOT manager is responsible for the allocation of network resources in order to emulate the network given by the DOT user in create_conf/Sig1.0_IB_Conf. The DOT manager has two distinct parts which together perform all the required activities and manages the DOT nodes. Those two parts are the provisioning module and statistics module. As the name suggests, provisioning module is responsible to read the DOT topology file and deploy that network on the physical machines. The statistics module is responsible for collecting data from nodes through the logging module. DOT nodes are responsible for following the DOT manager's instructions and to embed the virtual network inside the node. So, a DOT node may contain virtual switches, virtual hosts or machines that together form the required network topology. With the help of DOT node manager, allocation and configuration of required resources is done on that particular node. Whereas the logging module in the DOT node collects local statistical data from the network and pass it to the DOT Manager. This local statistical data may include throughput, resource utilization, delay, packet loss and OpenFlow messages.

Even though DOT Manager plays the main role in DOT functionality, it cannot run on its own. It requires a controller to handle everything inside the network from deploying the topology to routing messages within the network. Currently, there are many controllers available in the SDN market like POX, NOX, Big Network controller, Floodlight etc. POX is a controller written in Python that supports OpenFlow. It has a high-level SDN API including a queriable topology graph and support for virtualization. But, when compared to peers, POX is very slow and this was the reason we did not select POX for our prototype implementation. Floodlight, a Java-based OpenFlow controller developed at Stanford, is an easy-to-use controller. A huge amount of support and forums are available related to floodlight which makes it the best choice for our implementation. Also, Floodlight is a Java-based controller and our Hedera implementation is also Java-based which makes it easier to combine both together. Also, Floodlight performs faster than POX.

Now let's discuss the implementation of Hedera which would be used for evaluation purposes. Hedera is a dynamic scheduler which works on the principle of Equal Cost Multi Path routing so, in order to test different scenarios, we used a Fat tree [4] with multiple paths between any pair of source and destination. Fat tree, as discussed earlier, has multiple paths between any two hosts and these paths are used by ECMP to calculate the most efficient path between the hosts. As discussed earlier, there are two ways to implement Hedera namely global first fit and simulated annealing. Global first fit follows a greedy approach whereas simulated annealing follows an iterative approach. As per the definition, global first fit greedily chooses a path that has sufficient unreserved bandwidth and allocates flows accordingly. Speed of global first fit is O([Number of ports/number of switches]^2). On the other hand, simulated annealing iteratively tries to find a globally better mapping of paths to flows. Here the speed is O(number of flows). Comparatively, simulated annealing is more optimal than global first fit.

Global First Fit: Whenever any new flow is detected, global first fit linearly search all the possible paths between a source and destination. It detects the first path whose component can fit the flow and then places the flow on that path. Once decided, scheduler reserves the required capacity for that flow on the link corresponding to that flow. Then, the scheduler creates forwarding entries in the corresponding edge and aggregate switches. After the flow ends, all the entries and reserved capacities are timed out.

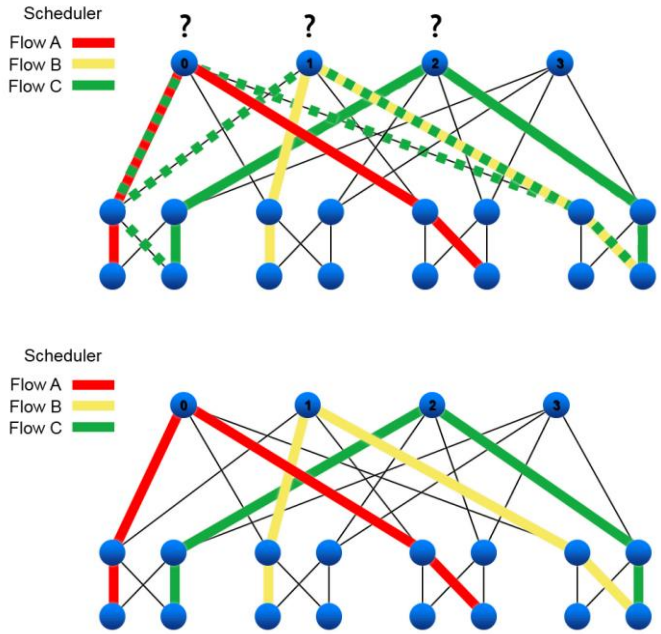
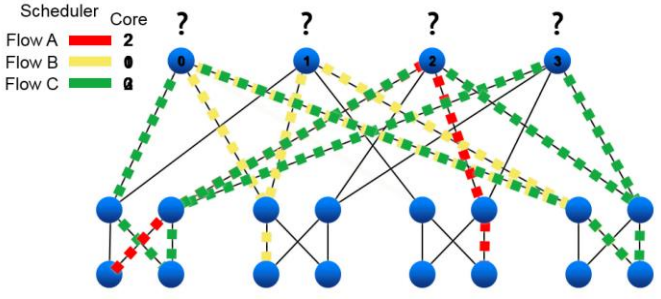


Figure 5: Global First-Fit

Simulated Annealing: If any flow is detected, the scheduler will perform a probabilistic search to compute paths for the detected flow efficiently. Here the vital principle is to assign a single core switch for each destination host rather than for each flow. After each round, the final state is published to the switches which is used as the initial state for the next round. Because of this approach this technique is closer to optimal and requires less search space. And in this way this technique achieves good core-to-flow mapping.



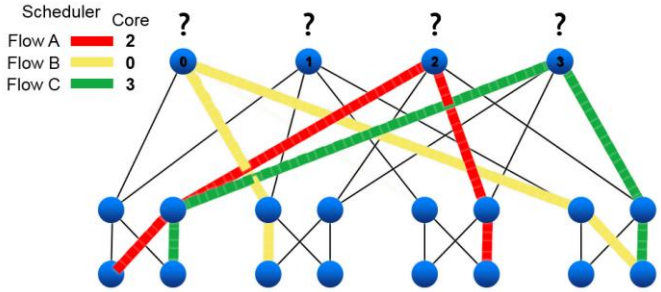


Figure 6: Simulated Annealing

Our Hedera implementation prototype uses global first fit approach for this project. As we have already discussed, global first fit is far from optimal when compared to simulated annealing approach, but in this project our motive is to evaluate the scalability of DOT using Hedera. So, all-in-all Hedera is just a base we are using to evaluate the performance of DOT.

After all this, let us discuss a very important part of system description i.e. our emulated network topology. With the constraint on the number of CPUs and number of cores, we emulated the network topology shown below in Figure 7. Our topology has 16 virtual machines (also known as hosts) and 14 switches deployed in different layers (like Edge, Aggregate and Core). As discussed earlier, Hedera requires a Fat tree in order to perform ECMP-based network routing, hence in this topology there are multiple paths between any two hosts. Each link between host and edge switch is of 100MB capacity, each link between edge switch and aggregate switch is of 200MB capacity and each link between aggregate switch and core switch is of 800MB capacity. We used Iperf [6] flows for emulating the traffic in the network. At the end, we will finish this section with the list of experiments we carried out.

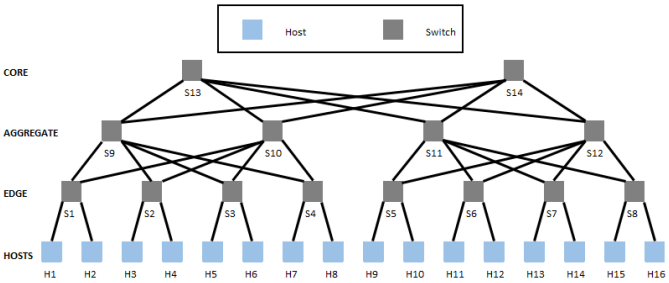


Figure 7: Simulated Network topology

Experiments: Our main motive behind this paper is to evaluate the performance of DOT. So, the most important part of our paper is the experiment and evaluation part which would demonstrate the significance of DOT. Hence, for the evaluation purpose, we divided our experiments into four categories:

1) Communication pattern experiments: In the first type of experiments, we try a few traffic patterns within the network and gather throughput data from the network which would tell us the packet forwarding capability of Hedera in DOT. We adopted the traffic pattern approach from the original Hedera paper which are as follows:

- *Stride:* A host with index *x* send to the host with index (*x* + *i*)*mod*(*num_hosts*). We performed Stride 1, 2, 4 and 8 in this experiment.
- *Staggered:* A hosts sends to another host in the same edge switch. We sent messages between pairs of hosts having the same edge switch at the same time.

2) Fault tolerance experiments: Here, we performed experiments which deal with the failure of different components of the network. We used node failure and link failure scenarios in this experiment.

3) Load based experiments: These experiments try to put as much load as possible on the network in different ways. We had covered traffic load experiment as a part of communication pattern experiments and with the limitation of our physical machines, we cannot perform reliable experiments for network size load. So, our main focus in this area is to put load on the machines by using multiple controllers.

4) Mininet based experiments: In this section, we mention our observations while using Mininet during the course of implementation.

IV. Evaluation

The evaluation section is divided to address the four different experiments as discussed in previous section. We used multiple Iperf flows in the first experiment to evaluate the efficiency of Hedera with Floodlight in DOT. For the second experiment, we only used one Iperf flow to demonstrate the fault tolerance capability of Hedera. In the last experiment we used very large number of Iperf flows to evaluate the load capacity of DOT. Now, we will be discussing each experiment in detail.

A. Experiment 1: Communication Pattern Experiments

The communication pattern experiment was adopted from [1] where authors used three different patterns. Due to some unexpected issues, we had to limit our communication pattern experiment to Stride and Staggered and hence will not be covering Random patterns. So, this experiment is divided in two different patterns namely: Stride and Staggered patterns.

Stride: In this type of communication pattern, traffic is instructed to flow with different level of switches. Every host *x* communicates with every other host (*x* + *i*)*mod*(*number_of_hosts*) and throughput is calculated. For this experiment, we used four flows of 100 second time span each of 800MB size, which were using different levels of switches (like edge, aggregate and core) and then their throughput was collected. All four stride throughputs are shown in below Figure 8 – 12. Flow 1 was started and after 15 seconds, flow 2 was introduced. Similarly, after 30 seconds and 45 seconds, flow 3 and flow 4 were introduced in the network topology.

As shown in the stride figures below, the blue-colored lines are plotted using the original Floodlight without any added modules and changes. The orange-colored lines are plotted using Floodlight with Hedera implemented. It was observed Floodlight with Hedera performed better as compared to normal Floodlight. In Stride 1, both plots are very close to each other. The reason can be that very little of the network was overlapping for flow 1 with other flows as it was communicating within the same edge-node. On the other hand, in Stride 2, 4 and 8, Floodlight with Hedera outperformed basic Floodlight. This might be because of the ECMP algorithm inside Hedera that manages the flows and tries to keep all flows with least overlap (using same links). There were few instances when Floodlight with Hedera performed worse than basic floodlight. For example, in stride 8, at 79th second Hedera implementation throughput was less than basic Floodlight. This might be because of some indefinite entities because this behavior was not consistent every time.

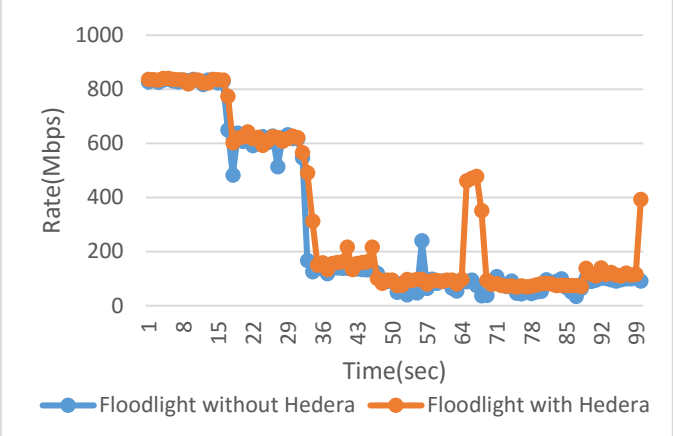


Figure 8: Stride 1 throughput

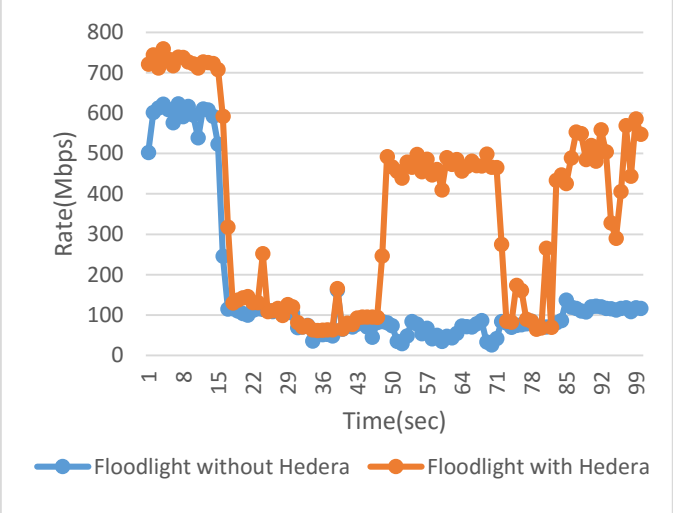


Figure 9: Stride 2 throughput

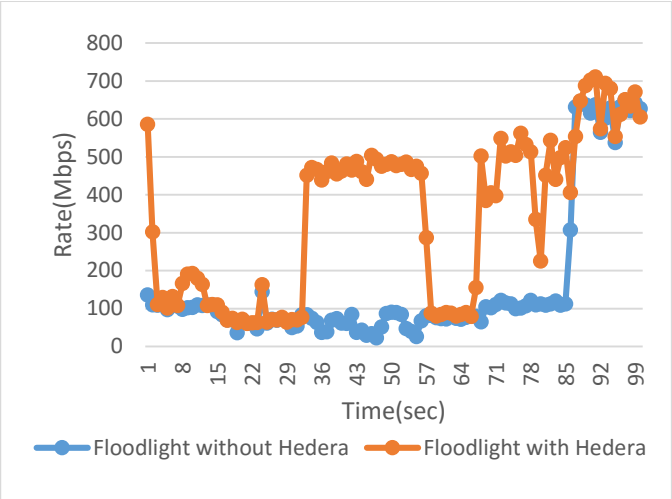


Figure 10: Stride 4 throughput

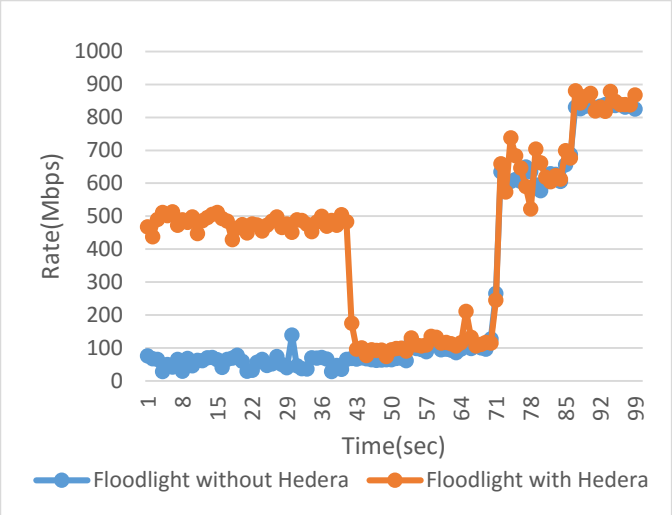


Figure 11: Stride 8 throughput

A really important thing to note here is the path that flows followed in basic Floodlight and in Floodlight with Hedera. Considering topology shown in Figure 7, in basic Floodlight, we started flow 1 which used H1-S1-H2 path and flow 2 used H1-S1-S9-S2-H3 path. And flow 3 also used same links (which are being used by flow 1 and 2) i.e. H1-S1-S9-S3-H5. On the other hand, in Floodlight with Hedera, flow 1 and flow 2 performed exactly same as in basic Floodlight but, in Flow 3 ECMP module of Hedera changed the path and used a different path i.e. H1-S1-S10-S3-H5. In this way, as far as possible, Hedera tries to keep flows on different links without overlapping (links being used by multiple flows).

As shown in Figure 12 below, overall Floodlight with Hedera always performed better than basic Floodlight. Though as discussed earlier as well, both performed nearly equally well for stride 1 whereas in rest of the cases, Floodlight with Hedera outperformed basic Floodlight significantly.

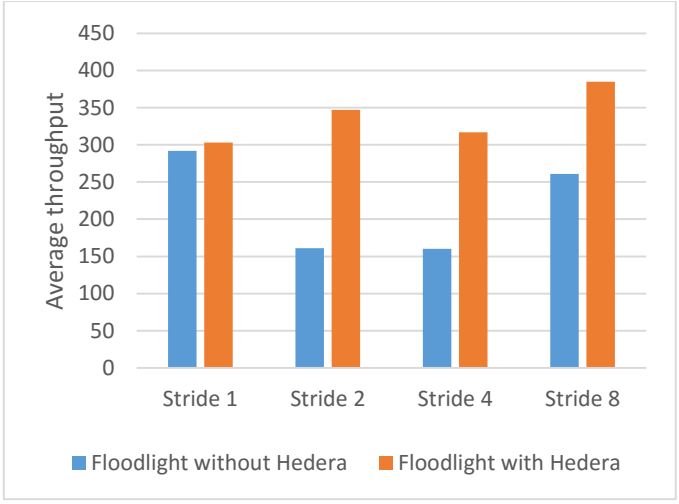


Figure 12: Overall Stride throughput

Staggered: In the staggered communication pattern, traffic is restricted to edge switches only. In this pattern, a virtual machine sends flows to another virtual machine in the same edge switch. So, basically, we sent messages between pairs of virtual machines having the same edge switch at the same time and gathered throughput of each staggered flow. In staggered pattern as well, we used similar approach as used in stride pattern i.e. 4 flows with 100 sec time span each of 800MB size. Introduction of each flow into the network was exactly same as in stride pattern i.e. flow 1,2,3 and 4 were introduced in the network at 0, 15, 30 and 45 seconds respectively.

We have plotted similar kind of graphs for staggered pattern as well. Blue-colored graph represents Floodlight without Hedera i.e. basic Floodlight and orange colored graph represents Floodlight with Hedera. As compared to stride graphs, the basic Floodlight graph follows Floodlight with Hedera graph more closely in the staggered pattern. The reason may be because staggered pattern is based on the same edge switch communication and hence very less/no link overlap (two flows using same link).

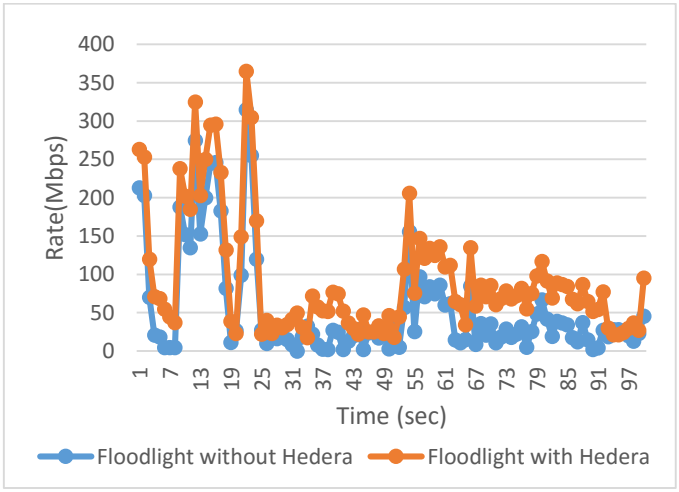


Figure 13: Staggered 0 throughput

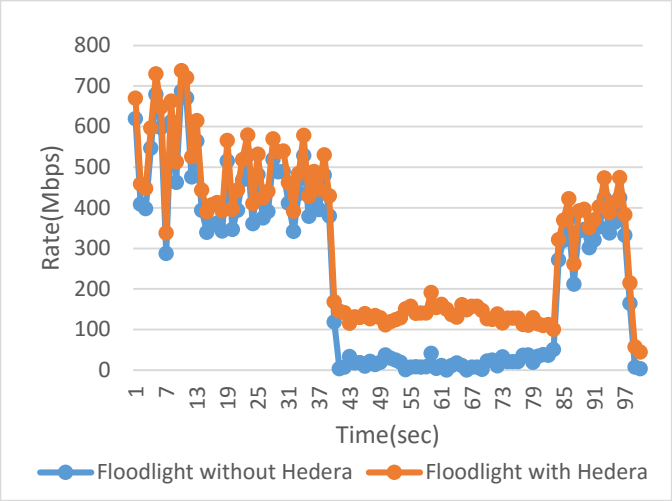


Figure 14: Staggered 1 throughput

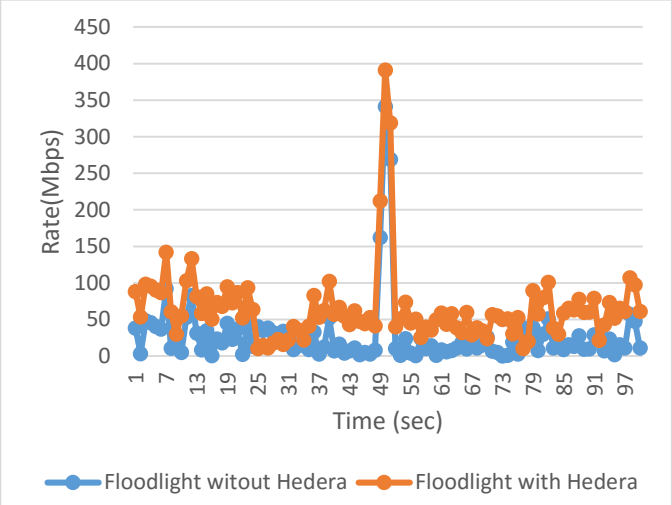


Figure 15: Staggered 2 throughput

As shown in Figure 16, throughput of basic Floodlight was always close to the throughput of Floodlight with Hedera. We found some really big throughput differences in stride pattern but the same was not observed in staggered pattern.

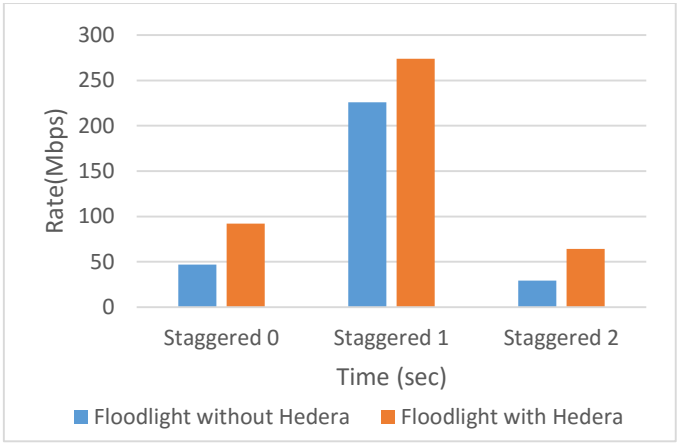


Figure 16: Overall Staggered throughput

B. Experiment 2: Fault Tolerance Experiments

In the Fault tolerance experiments, our main aim is to achieve Hedera functionality as discussed in [1] with the DOT testbed. In this experiment, we focused our experiment to link failure and switch failure. Here we only used one flow for this experiment. For this experiment, in the topology as shown in Figure 7, first we started a flow between H1 host and H9 host. Initially flow was using H1-S1-S9-S13-S11-S5-H9 path when the whole topology was provided to the flow. Then by using the *disconnect* command in DOT console (*./dot_console*), we disconnected H13 and H11 link. As we disconnected the link, the flow stopped and asked the Floodlight controller with Hedera about the next path. Floodlight sent this request to Hedera and it used ECMP to recalculate next best possible path viz. H1-S1-S9-S14-S11-S5-H9 path. In this way, Hedera helps traffic to overcome link failures. A similar kind of scenario is shown in Figure 17 below. Initially, a flow was using the green-colored path but when link failure happened on that link, the same flow used the blue-colored path to reach its destination.

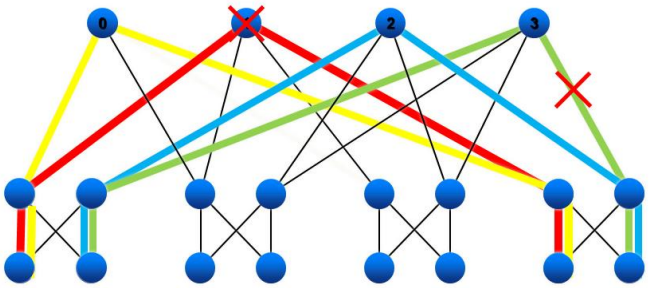


Figure 17: Node and Switch Failure scenario

In the second phase of this experiment, we tried to a simulate node failure scenario. Again using the same topology, we used the same flow with the same path as with the previous link failure case. This time instead of the link we disconnected the switch itself. After starting the flow, we disconnected the switch S13 because of which Hedera diverted the path of the flow towards S14 to reach destination host H9. A similar kind of example is demonstrated in Figure 17, where initially the flow

was using red-colored path but as switch 1 failed, Hedera diverted the path towards switch 0 following the yellow-colored path. In this way Hedera helps network to overcome node failure scenarios.

Though we have not performed any scheduler failure related experiments but, it is expected that in case of scheduler failure, switches are supposed to fall back to ECMP routing state.

C. Experiment 3: Load based Experiments

In this section of load based experiments, we tried to achieve full capacity of network using different ways. Load based experiments include increasing the network size, increasing traffic volume and increasing the number of controllers. Network size loading experiment needs more CPUs and more hardware resources which were not available to us. For load based on traffic size, we have already covered similar kind of experiment in experiment 1 where we simulated stride and staggered patterns by achieving maximum link capacity. So, our main focus in this experiment is to load the network by number of controllers. For this experiment, we used 4 random flows with 3 controllers with different functionality. The first controller is the basic Floodlight controller without any changes or extra modules. The second module is Floodlight with ECMP routing mechanism and the last controller is Floodlight with a Hedera implementation. We started basic Floodlight first and started four random flow between different pairs of hosts. Then, while the flows are occurring between source and destination, we started Floodlight with Hedera. At that moment, node 10.0.0.12 consisting maximum number of virtual resources went down and we could not perform further experiments. We tried different combination in next iteration but the results were same. Every time we tried running multiple experiments, our node with all the resources went down. The possible reason behind this could be the limited node capacity. This demonstrates a very significant drawback of distributed architecture.

D. Experiment 4: Mininet based experiments

We used the same topology for Mininet as we had in DOT which is shown in Figure 7. In Mininet, response time reduces with increase in size of the topology. We deployed 16 hosts and 14 switches in the network and it took 274 seconds compared to 169 seconds in DOT. Though we could not perform substantial experiments in Mininet, we tried using simple pings between different hosts within the topology. We observed both DOT and Mininet response time is quite close but in DOT, when we ping between two virtual machines available on different physical machines, the response time is significantly large than pinging within the same node.

V. Discussion & Future Work

After implementing Hedera in Floodlight and experiments in DOT with and without Hedera, we would say our experience with DOT has been satisfactory. There are few reasons why it was just satisfactory and not good.

First and for most, failure of nodes multiple times. There were days when we faced node failure 4-5 times in a day, majorly after we performed a set of experiments. The most frustrating part was when we were deploying our topology using 10.0.0.12 node and 10.0.0.18 node with maximum virtual resources being deployed on 10.0.0.12 node. Every time, the 10.0.0.12 node inevitably failed. By the end of our experimentation phase, we also saw 10.0.0.12 node failing without any significant reason as well. We have witnessed failure of node 10.0.0.12 just by running original Floodlight controller (with no added module or changes). Though DOT comes with distributed architecture, the chances of component failure also increases as compared to Mininet. For this project, we implemented four different combinations. Namely:

- Floodlight + DOT
- Floodlight + ECMP + DOT
- Floodlight + Hedera + DOT and
- Floodlight +Hedera + Mininet.

Due to hardware issues we could only perform experiments related to Floodlight + DOT and Floodlight + Hedera + DOT. We also wanted to add ECMP and Mininet results.

Second, very silly but very important from a developer's perspective, Mininet has a strong online community with forums and support available on internet whereas DOT, being relatively new, has very limited support and help available on internet.

Third, we did not find any major documentation available for DOT on its official website as well. In the case of Mininet, they have the API and tutorials available on the official website. For example, though DOT has a statistics module, it has no information provided on the website.

In this paper, our motive was to evaluate DOT using Hedera to compare it to Mininet. Due to hardware issues and constant node & server failures, we could not perform similar experiments in Mininet. We did not want to perform Mininet experiments on a local machine because the configuration of the physical machine would be different and results would not be accurate and reliable. Ultimately, we did not include Mininet experiments in this paper so, our first possible future work would be to perform same experiments in Mininet as well with the same configuration of servers and machines to get accurate results. Also, as we know Mininet can be used on a single machine and DOT is a distributed testbed so their comparison is relatively obvious. We found that apart from DOT, there are other distributed testbeds and emulators available. For example, MaxiNet which was developed by University of Paderborn is also a distributed testbed. Also, another testbed called OFELIA developed by ETH Zurich is also a distributed OpenFlow Testbed. So, comparing the performance of DOT with these testbeds would be much better and interesting.

VI. Contributions

We would like to thank Bastin Gomez for his help in times of technical difficulties. Aimal Khan and Arup Roy were both very helpful for their guidance on the direction of the experimentation and this paper. Extra special thanks for Arup for restarting the node multiple times during the experimentation phase when we dealt with hardware failure. A huge thanks to Prof. Raouf Boutaba for letting us implement this project on DOT as a part of the course and make the necessary resources available to us.

VII. Conclusion

DOT being a distributed architecture testbed is more scalable when compared to Mininet. We have seen in the prototype system that Mininet behaves extremely slow as we increased the network topology size whereas increasing the topology in DOT is very easy. But, we also observed that with increase in the network size, response time reduces and chances of failure of nodes increases. We also witnessed several node failures while performing the experiments which clearly shows that currently DOT is not in its best possible stable state. Hedera, a dynamic scheduler, which uses ECMP (Equal Cost Multi Path) for routing decisions in the network topology, couples with Floodlight really well. We observed Hedera outperforming original Floodlight implementation significantly when links are shared by multiple flows whereas difference in performance was very less when less or no links are shared. Overall, it provides better throughput and efficiency to the system and overcomes a few significant issues of distributed systems like fault tolerance.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat, Hedera: Dynamic scheduling for Data center networks.

[2] Arup Raton Roy, Md. Faizul Bari, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba, Design and Management of DOT: A Distributed OpenFlow Testbed

[3] Trees; http://en.wikipedia.org/wiki/Tree (graph_theory)

[4] Fat trees; http://en.wikipedia.org/wiki/Fat_tree

[5] Mininet: An Instant virtual network on your laptop; http://mininet.org/

[6] Iperf – The TCP/UDP bandwidth measurement tool; https://iperf.fr/

[7] MaxiNet: Distributed Software Defined Network Emulation; http://www.cs.uni-paderborn.de/?id=maxinet

[8] Estinet: OpenFlow Network simulator and emulator; http://www.researchgate.net/publication/260670458_EstiNet_openflow_network_simulator_and_emulator. September 2013 IEEE conference.

[9] Estinet: OpenFlow Network Simulator and Emulator; http://www.estinet.com/fckimages/1411701431891112745.pdf

[10 Estinet Technologies- Real applications protocols and performance; http://estinet.com/products.php?lv1=1&sn=2

[11] OFELIA: A Distributed OpenFlow Testbed; http://www.fp7-ofelia.eu/assets/Publications-and-Presentations/social-networks-workshopETHZJune2011.pdf