SDN-Based Load Balancing for Multi-Path TCP

Austin Jerome, Murat Yuksel, Syed Hassan Ahmed, and Mostafa Bassiouni
Department of Electrical & Computer Engineering, University of Central Florida, Orlando, FL 32816, USA.
Emails: aus27@knights.ucf.edu, murat.yuksel@ucf.edu, sh.ahmed@ieee.org, bassi@ucf.edu

***Abstract*—In this paper, a novel load-balancing technique for local or metro-area traffic is proposed in mesh-style topologies. The technique uses Software-Defined Networking (SDN) architecture with virtual local area network (VLAN) setups typically seen in a small-to-medium enterprise networks. This was done to provide a possible solution for the load-balancing dilemma of congestion and under-utilization of network links. The transport layer protocol Multi-Path TCP (MPTCP) coupled with IP aliasing is used, which enables formation of multiple subflows depending on the availability of IP addresses at either ends and helps to divert traffic in the subflows across all available paths. The traffic formed of each subflow would be forwarded across the network based on Hamiltonian 'paths' that are created in association with each ingress/egress switch directly connected to hosts. Our study shows the advantages of using MPTCP for load balancing purposes in SDN architectures and provides a platform for using VLANs, SDN, and MPTCP for traffic management.**

I. Introduction

Network traffic load balancing has been a critical challenge faced by network administrators. The ability to balance traffic across two or more links has been getting harder as the networks are increasing in size and speed. A major recent change to the networking practice, Software-Defined Networking (SDN), advocates techniques to facilitate the design, delivery and operation of network services in a deterministic, dynamic and more scalable manner [1]. It assumes the introduction of a high level of automation in the overall service delivery and operation procedures. Basically, SDN involves a physical separation of the control plane and the forwarding plane [2] and argues for handling complex tasks such as load balancing in the control plane with a centralized view of the network. The functional separation benefits of the SDN technology enable networks to be directly programmable, and more agile as abstracting control from the forwarding plane lets administrators dynamically adjust network-wide traffic flow to meet the changing needs of the network.

However, SDN still is quite some strides away from maturing in load-balancing. For example, Zhou et al. [3] discuss problems that could exists in the link between the controller and the switch in case of overwhelming traffic. The authors point out that even if we deploy switches and their controllers very carefully, it's difficult for controllers to adapt to the changing traffic load. This could affect resource utilization. It is essential to balance a load across different controllers in any networking cluster instead of a static network configuration. Load balancing at the controllers' end is a common obstacle which SDN administrators face. Supporting the argument, Yannan Hu et al. [4], while making use of the OpenFlow protocol for communicating to switches, report on issues with load balancing between SDN controller and switches. They proposed an architecture called BalanceFlow for wide-area OpenFlow networks which can partition control traffic load among different controller instances in a more flexible way. Both of these studies on balancing load between controllers and switches give approaches and solutions to solve the problem with that perspective in mind. However, the problem of balancing load from switch to switch, especially when there are multiple links/paths available to move from source to destination is still an open issue.

With the advent of Multi-Path TCP (MPTCP) [7] offering multiple flows, the load-balancing problem becomes more complicated. In this paper, we focus on the SDN-based load-balancing problem within the context of local and metro-area networks utilizing Virtual LANs (VLANs) [11]. We use SDN and MPTCP in combination and make use of additional paths from source to destination other than just the shortest or the one which is already being used. Accordingly, MPTCP would balance out the load in the paths between each of its subflows. Keeping this in mind, our key contributions are as follows:

- To design and develop an SDN framework that allows exploration of how MPTCP could be beneficial for network administrators who are responsible for establishing VLANs which make use of OpenFlow.
- To give statistics that would give a fair distinction on how much improvements MPTCP brings to load balancing on mesh style topologies.
- To develop heuristics which would help direct MPTCP flows across the network in an efficient manner that would further help in improving load balancing and network utilization across the network.

The rest of the paper is organized as follows: Section II describes our SDN-based framework for VLAN load balancing with randomly generated Hamiltonian Paths (HPs). In Section III, we further present a heuristic for load balancing via the HPs to forward the packets. Section IV showcases performance evaluations followed by the conclusions and future work.

II. SDN-Based VLAN Load Balancing

A. Hamiltonian Paths with IP Aliasing

The main idea is to maximize the potential of MPTCP and improve network utilization by coupling it with IP Aliasing. Consequently, more than one IP address can be associated with a network interface. Then by using the SDN format of forwarding, we can accordingly divert where the subflows are to be forwarded in the network. To achieve this, we create Hamiltonian paths which are associated to each switch connected directly to the hosts in our topology.

Hamiltonian paths span all the switches in the topology, and that guarantees reachability. The typical learning switch

978-1-5386-5979-3/18/$31.00 ©2018 IEEE

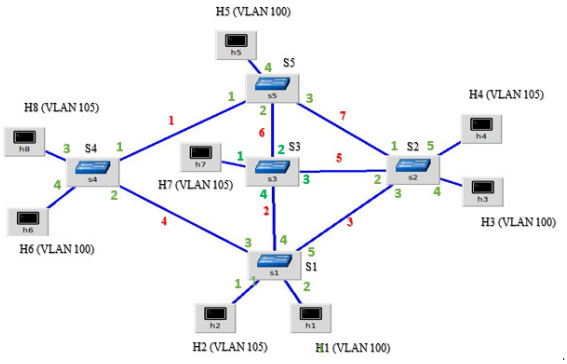

Fig. 1: Topology 1

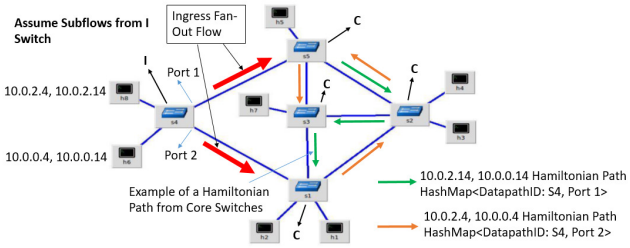

Fig. 2: Ingress Fan-Out

algorithm establishes a spanning tree among the switches to achieve that guarantee. Yet, spanning trees are less flexible than Hamiltonian paths in terms of placing each flow to a separate, potentially non-shortest, path. Since, for load-balancing, we desire to take non-shortest paths for some flows, we choose to use Hamiltonian paths as a building block. It is possible to expand our framework to spanning trees as well.

An additional issue is how MPTCP utilizes the additional paths existing between sender and receiver while controlling the congestion on each path? The solution to this question lies in congestion control algorithms like Linked Increase Algorithms (LIA) [10], Opportunistic Linked Increase Algorithm (OLIA) [8], and Balanced Adaptation Linked Increase Algorithm (BALIA) [9] which are used during traffic flow across the paths in an MPTCP connection. As per RFC 6356, these new congestion control algorithms are necessary for multipath transport protocols such as Multipath TCP and traditional single path congestion control algorithms (e.g., Additive Increase Multiplicative Decrease) have problems in a multipath context. In this paper, we use the LIA algorithm for our experiments. We will next explain our framework via an example topology.

1) Topology Setup: Lets take an example of a sample topology given in Fig. 1. The topology is created using Mininet [12] which is a popular network simulating tool. The SDN controller, which is the FloodLight SDN controller, will be running on a separate Ubuntu Linux virtual machine and will listen for OpenFlow messages from switches that are trying the connect to it. The Static Entry Pusher REST application [13] is used to push flows to switches which forward packets from their source to destination. The Static Entry Pusher application is part of the Floodlight Controller and is already activated upon setting up the Floodlight Controller. In the topology, we see that there are 5 switches and 8 hosts. Each host is placed in different VLANs mentioned in the diagrams. Port Numbers where links are connected to the switched and the link numbers that connect switches are marked in green & red respectively.

2) Hosts and IP Aliases: MPTCP allows for multiple TCP streams called as MPTCP subflows to form between sender to receiver depending on IP availability in both machines. This means that if there is more than one IP address associated with a host, MPTCP starts up another TCP session using the additional IP with the destination machine's IP if the destination machine is also MPTCP capable. Thereby, to make complete use of the traits of MPTCP, IP aliasing is used to associate multiple IP addresses with each host. The amount of IP addresses associated with a host would be dependent on the number of links passing out of the first switch which is directly connected to the host. This switch is called the Ingress Switch, I, of the host and all other switches adjacent to the switch is called the Core switch, C, of the host. For example, $H1$ in Fig. 1 is connected to $s1$, which is $H1$'s I switch. $s1$ has three links passing on to the C switches, $s2$, $s3$ and $s4$. Thereby, $H1$ would have two IP alias addresses associated to it and one original IP address giving it a total of three IP addresses which would be used during the creation of subflows when MPTCP is active in any connection.

3) Ingress Fan-Out: The subflows created would be forwarded out of the I switch based on flows pushed through the Static Entry pusher. Also, each subflow associated with an IP of the host would be forwarded out through different ports going to the next core switch. For example, as discussed above, $H1$ would have three IP addresses, subflows associated with each of the three IPs would be forwarded out through three different ports. This means that if the IP addresses are 10.0.0.1, 10.0.0.11 and 10.0.0.21, then subflows associated with 10.0.0.1 as source address would be forwarded out of the I switch from the port which leads to the C switch $s2$. The subflows associated with 10.0.0.11 as source address would be forwarded out of the I switch from the port which leads to the C switch $s3$ and the subflows associated with 10.0.0.21 as source address would be forwarded out of the I switch from the port which leads to the C switch $s4$. This phase of the procedure is called the 'Fan-Out' phase which is forcing the MPTCP subflows to utilize different links enroute to the destination. Every connection from a host would have a fan-out phase once the packets reach the I switch which would be advantageous in balancing traffic during subflow creation in the MPTCP process. Fig. 2 is an example diagram showing the Ingress fan-out phase from Host $H8$ which would be associated with two IP addresses due to two links exiting out of its I switch towards the C switches.

4) Subflow Identifiers: Once packets from a host leave the I switch and move towards destination through C switches, they follow Hamiltonian paths which would be preset for the host and its TCP streams once they reach the C switches. A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once. Another important property to Hamiltonian graphs is that all Hamiltonian graphs are biconnected, however every biconnected graph need not be Hamiltonian. All topologies that are going to be used in this paper would satisfy the Hamiltonian graph property.

Moreover, due to the fan-out mechanism implemented for each connection going out of the ingress switch, each IP address from hosts connected to the I switch would have a common egress port that takes traffic to the C switches. Using such an idea, we can conclude that whenever traffic from an I switch leaves towards the C switch, the traffic can move out from any one of its outgoing ports which is directly connected to one of the core switches and thereby the identifier element for all the host's traffic moving outwards from its I switch is the pair of `Datapath` ID of the I switch and the outgoing port number associated with one of its source address. Thereby using these flow identifiers, we can create a HashMap associated to all hosts in the topology where the HashMap would consist of the `Datapath` ID of the switch and an outgoing port which would be the egress port that takes traffic from the host to one of the C switches in the topology. Fig. 2 shows how the paths for each of hosts on I switch would be around the topology. Here we see 10.0.2.14 belonging to Host H8 and 10.0.0.14 belonging to Host H6 having the same path around the network due to having the same I and same outgoing port in the fan out phase and we also see 10.0.2.4 belonging to Host H8 and 10.0.0.4 belonging to Host H6 having the same path due to similar reasons.

5) VLAN-Specific Hamiltonian Paths: The above concept can be further improved for better load balancing in the network, when the VLAN setups are considered. This is done by creating separate paths for every host belonging to different VLANs, which are connected to a particular I switch (refer Fig. 1). This ensures further segregation of paths for better load balancing and also monitors subflows through links based on VLANs present in the topology which would give network administrators more control in the network.

Again, we can compute the number of paths that would be associated to hosts connected to any I switch. Here, the average space complexity of the flow tables in the switches would be $O(Vnk)$ where V is the number of VLANs present among all the hosts connected to the I switches, n is the number of I switches and k is the average number of outgoing ports that takes traffic from an I switch to the C switches.

With the number of paths that would need to be created for the topology based on the above discussion known, now the question is how the paths are to be created. The most important factor in the creation of paths is to balance out load across each link of the topology that all paths would use. This means that on creating paths, there could be a likelihood that high number of paths utilize one particular link in the topology which could overload that link and create an imbalance. To avoid this, we create a range (upper bound and lower bound) of flows that a link can have by inspecting the standard deviation of the count of flows traversing a link.

III. Load Balancing Heuristics

Moving further, to systematize the path finding process for each subflow, we associate a Hamiltonian path traversing the C switches to each HashMap associated to hosts connected to an I switch. This means that once a packet belonging to a HashMap associated with a host leaves its I switch towards a C switch, there would be flow identifier entries in that C switch which can be associated to a path for this HashMap. Using this flow identifier entry, the C switch takes the flow's packets towards other C switches and finally to its destination. The diagram in Fig. 2 shows example Hamiltonian paths of host H8 and host H6 connected to I switch S4 and their subflow identifiers: $HashMap <$ `Datapath`$ID : S4, Port1 >, HashMap <$ `Datapath`$ID : S4, Port2 >$.

An important property of all paths in the topology is that each path should be allowed to visit a node, which is a switch in this case, only once. This is done to avoid loops in the topology which is also the main property of Hamiltonian paths. Going by this approach, the number of paths that all hosts connected to any I switch will be the number of flow identifier entries each switch must maintain. This space complexity can be abbreviated into $O(nk)$ where n is the number of I switches in the topology and k is the number of outgoing ports in an I switch to the C switches in the topology which corresponds to each HashMap having a path associated to it. The question here is how does the packet reach its destination host once it reaches the switch which contains the connection straight to the host. Here, the packet must come out of the port that is directly connected to its destination. Priorities for flows within the switch in its flow table is made use off to overcome this. For all switches that contain direct connections to hosts, there are flows set up in switches which match to the destination host's IP address and these flows have a higher priority compared to the other flows in the flows table. Thus, when the packet first reaches any switch, it checks if there is direct connection to the host from this switch through the destination flow, and if not, it continues the direction that the path flows forward it to.

Algorithm 1: Random Hamiltonian Path Selection Algorithm

Output: Random Path P_i
Input : Graph G, Starting Node H_0

```
1  Procedure randomHPath(G,H_0)
                    // Setup and find possible Hamiltonian paths.
2      PathsList = Call pathHSeup (G, H_0)
3      Random RNG = new Random()
4      Idx = RNG.nextInt(PathsList.Size())
5      return P_i=PathsList.Get(Idx)
6  end
1  Procedure pathHSeup(G, H_0)
2      path = new array[G.length()]
3      path[0] = H_0
                                  // Find position of H_0 in G.
4      colNo = findCol(G, H_0)
5      Paths = Call findHPath (G, colNo, 0)
6      return Paths
7  end
1  Procedure findHPath(G, colNo, pathPos)
2      Recursively find paths starting from H_0
3      Backtrack through the latest path to find the next
         path PList contains all paths
4      return PList
5  end
```

A. Random Creation of Hamiltonian Paths

As presented in Algorithm 1, we first create Hamiltonian Paths (HPs) randomly for each existing HashMap in the

topology using the principle that a switch in the topology can be visited only once and each time a packet leaves its host towards the destination there must be an Ingress Fan-Out phase from the Ingress switch. The process for random path creation would be computed in the controller's end, and hence, this would not take up processing memory in the switches end as the switch would only be responsible for saving up flows in the flow table. Further, randomly generating paths ensures that the load is already spread out across multiple links.

Let's take an example using HPs in Fig. 1. A set of randomly created HPs are given in Table I. Taking H1 as an example in the topology, H1 would have three random paths associated with it due to three links exiting its *I* switch. As we see in these three paths, the *I* switch of host H1 is switch S1. 10.0.0.1 is the original IP address of H1 while 10.0.0.11 and 10.0.0.21 are aliased IP addresses created using IP aliasing for the purpose of MPTCP and path creation. We also notice that each node or switch is visited just once. Similar to this, a sample set of randomly generated HPs for all other hosts are listed in the table with their IP addresses.

TABLE I: Randomly Generated Hamiltonian Paths in Topology 1

Possible Hosts	Ingress Switch	Random HPs
H1(10.0.0.1, 10.0.0.11, 10.0.0.21)	S1	S1 → S3 → S2 → S5 → S4 S1 → S2 → S3 → S5 → S4 S1 → S4 → S5 → S2 → S3
H2(10.0.2.1, 10.0.2.11, 10.0.2.21)	S1	S1 → S3 → S2 → S5 → S4 S1 → S2 → S3 → S5 → S4 S1 → S4 → S5 → S3 → S2
H3(10.0.0.2, 10.0.0.12, 10.0.0.22)	S2	S2 → S1 → S4 → S5 → S3 S2 → S3 → S5 → S4 → S1 S2 → S5 → S3 → S1 → S4
H4(10.0.2.2, 10.0.2.12, 10.0.2.22)	S2	S2 → S1 → S3 → S5 → S4 S2 → S3 → S1 → S4 → S5 S2 → S5 → S3 → S1 → S4
H5(10.0.0.3, 10.0.0.13, 10.0.0.23)	S5	S5 → S2 → S3 → S1 → S4 S5 → S4 → S1 → S2 → S3 S5 → S3 → S2 → S1 → S4
H6(10.0.0.4, 10.0.0.14)	S4	S4 → S1 → S3 → S5 → S2 S4 → S5 → S3 → S1 → S2
H7(10.0.2.3, 10.0.2.13, 10.0.2.23)	S3	S3 → S1 → S4 → S5 → S2 S3 → S2 → S1 → S4 → S5 S3 → S5 → S2 → S1 → S4
H8(10.0.2.4, 10.0.2.14)	S4	S4 → S1 → S3 → S5 → S2 S4 → S5 → S3 → S1 → S2

Table II shows the flow tables in Switch 1 of how it would look like depending on which switch would forward packets. Here, we see that there are three flow tables containing various flow entries. These flow entries are matched with the packets parameters and accordingly forwarded out of a certain port in the switch. Table 1 contains flow entries involving the Ingress Fan-out flows and Destination flows, that is if a host is directly connected to the switch. These contain highest priority and will be looked at first for a match to take an action. If none of these flows match, that would mean that the incoming packet is not an ingress fan out packet nor a destination packet for a host directly connected to that host. This would imply that this packet is heading towards a host directly connected to another switch. In this case, the VLAN ID of the packet would be matched depending on which the packet would be directed for a match in Table 2 or Table 3. Table 2 contains HashMaps for packets belonging to VLAN 100 and Table 3 contains HashMaps for packets belonging to VLAN 105. Accordingly, packets would be mapped to their respective HashMaps and passed on to the next switch. In this manner, flow tables would be created in all switches in the topology. Note that the flow table setup in Table II is for VLAN-specific HP configurations. Since we have two VLANs in our example, we have two additional tables (Table 2 and Table 3). For the case when the HPs are created in a non-VLAN-specific way, we do not need to match packets to their respective VLANs in the flows and thereby can set up our flows using one flow table which would contain matching HashMaps along with other flows like destination flows and ingress fan out flows.

TABLE II: Flow Tables in Switch 1 of Topology 1.

Match	Action	Priority
Flow Table 1		
Source IP → 10.0.0.1	Output:- Port 5	100
Source IP → 10.0.0.11	Output:- Port 4	100
Source IP → 10.0.0.21	Output:- Port 3	100
Source IP → 10.0.2.1	Output:- Port 5	100
Source IP → 10.0.2.11	Output:- Port 4	100
Source IP → 10.0.2.21	Output:- Port 3	100
Dest IP → 10.0.0.1	Output:- Port 1	100
Dest IP → 10.0.0.11	Output:- Port 1	100
Dest IP → 10.0.0.21	Output:- Port 1	100
Dest IP → 10.0.2.1	Output:- Port 2	100
Dest IP → 10.0.2.11	Output:- Port 2	100
Dest IP → 10.0.2.21	Output:- Port 2	100
VLAN ID → 100	Go to Table 2	80
VLAN ID → 105	Go to Table 3	80
Flow Table 2		
HashMap<DatapathID: S4, Port 1>	Output:- Port 4	50
HashMap<DatapathID: S4, Port 2>	Output:- Port 5	50
HashMap<DatapathID: S5, Port 1>	Output:- Port 5	50
HashMap<DatapathID: S5, Port 2>	Output:- Port 3	50
HashMap<DatapathID: S5, Port 3>	Output:- Port 3	50
HashMap<DatapathID: S2, Port 1>	Output:- Port 3	50
HashMap<DatapathID: S2, Port 2>	Output:- Port 3	50
HashMap<DatapathID: S2, Port 3>	Output:- Port 4	50
Flow Table 3		
HashMap<DatapathID: S4, Port 1>	Output:- Port 4	50
HashMap<DatapathID: S4, Port 2>	Output:- Port 5	50
HashMap<DatapathID: S3, Port 2>	Output:- Port 5	50
HashMap<DatapathID: S3, Port 4>	Output:- Port 3	50
HashMap<DatapathID: S2, Port 3>	Output:- Port 4	50
HashMap<DatapathID: S2, Port 1>	Output:- Port 3	50

B. Rerouting Hamiltonian Paths for Load Balancing

Given randomly created HPs, we need to balance out the subflows on each path as much as possible to equally spread the traffic across the topology. Considering the three HPs from Table I, we see that the link between S4 and S5 is present as a route taken for all three HPs, thereby, this link has 3 subflows passing through it just based on these paths. The flow count is bi-directional since the layer 2 protocol between switches is assumed to be an Ethernet-like protocol. Likewise, the link between S1 and S2 is just being used once which is on the second path, thereby this link has 1 subflow passing through it. In this manner, we can calculate the number of subflows passing through all the links in this topology assuming all hosts are active and sending traffic. We create a table shown in Fig. 3(a) showing the number of subflows passing through each link in this topology. The port numbers to the links that are connecting the switches are given in Fig. 1.

Once the table in Fig. 3(a) is created, we observe the number of flows going through each link. As a first step, we try to reduce the maximum load on any link. If we observe that a link is the *only* maximally loaded link (e.g., link 4 in the table), we try to reroute the paths which are going through that link so that they avoid that link thus reducing the load

Link	Number of Flows
1	15
2	10
3	9
4	17
5	10
6	12
7	15

(a)

Link	Number of Flows
1	15
2	10
3	9
4	16
5	10
6	12
7	16

(b)

Link	Number of Flows
1	15
2	12
3	10
4	15
5	14
6	10
7	12

(c)

Fig. 3: Number of Flows per Link in Topology 1

on that link. While rerouting the paths, we avoid paths which would be forced to use that link due to the fan-out phase of the process and modify the routes of other paths. We keep selecting the link with the maximum count of flows and rerouting those paths traversing that link until we get to a point where we have at least two links having the same maximum number of flows as shown in a table Fig. 1(b) for the same topology. Here in this table, we notice that links 4 and 7 have the same amount of maximum flows, which is 16, passing through them.

Next, we calculate the standard deviation of the flow counts on each link to get an upper bound and lower bound for flows that can pass through each link. The range obtained for the above topology after calculation of the standard deviation was 10-15 where 10 is the minimum flow count that can traverse a link and 15 is the maximum flow count. Once the bounds are obtained, we keep rerouting the obtained paths iteratively until we have met the requirements of each link in the topology having the flow count within the targeted range. After rerouting the paths within this requirement, finally, we come up with the table shown in Fig. 1(c) which meets the lower and upper bound requirements to balance out the flows across each link in the topology. In the table, all the links have their flow counts as per the bounds calculated. Once this is calculated and now that the paths are set, we push flows to the switches in Mininet using Floodlight's Static Flow Pusher. After this is done, the hosts would then communicate with MTPCP using the paths which are set up in each of the switches.

This load balancing heuristic is a rudimentary approach and provides one possible solution to this problem and, in our experimentation, we have done this manually. However, this entire process can be automated and the processing could be done in the controllers end while after processing the controller would push the obtained optimal paths to the switches thus saving processing memory in the switches and using it only to store flows in flow tables. One intuitive way could be to randomly pick one of the paths on the maximally loaded link, l_m, and try rerouting it by calling Algorithm 1 with the input graph G not including the link l_m.

IV. Performance Evaluation

We present the simulation results of the proposed scheme in comparison to the regular TCP. We used VLAN specific path creation for hosts and paths, which do not consider the number of unique VLAN hosts connected to an Ingress switch. We also measured the performance using standard MPTCP without the use of any path creation. Collectively, we compared the following three techniques against TCP: 1) *VS*: MPTCP with

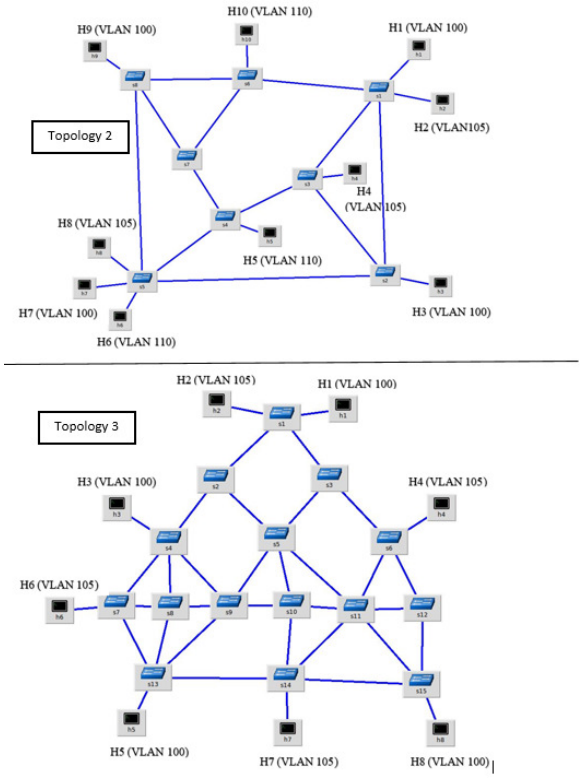

Fig. 4: Topology 2 and 3

VLAN-Specific Hamiltonian paths in place in the switches, 2) *NVS*: MPTCP with paths without considering the number of unique VLAN hosts which are directly connected to the switch, and 3) *MPTCP*: MPTCP without any paths incorporated.

Simulations were carried out in three different mesh like topologies involving client-server communication using the $iperf$ tool[1] in Linux. Topology 1 is the referred topology as shown in Fig. 1, which resembles an Enterprise-Level Local Area Network. While Topology 2 and 3 resemble a Mid-Size Metro-Area Network and Large Metro-Area or Datacenter Network as shown in Fig. 4. To send traffic from source to destination, Standard MPTCP and Regular TCP used a reactive SDN approach i.e. when a switch receives an unknown packet for the first time, it forwards it to the SDN controller as a PACKET-IN message. Then, the controller calculates the shortest path to the destination and sends that information back to the switch as a PACKET-OUT message. In the end, switch receives this forwarding information and forwards the packet towards the direction of the destination accordingly.

Furthermore, we divided experiments into three categories (i.e., LOW, MEDIUM, and HIGH) based on the amount of traffic load. A LOW loaded case refers few client-server communications, for example in a eight hosts case, there would be between two to four client-server communications and apart from this, there would be no ongoing parallel sessions, where one server can serve multiple clients. Whereas, a MEDIUM loaded case depicts servers serving multiple clients simultaneously and clients are allowed to connect multiple different

[1]A performance tool that is used for measuring the maximum achievable bandwidth on IP networks.

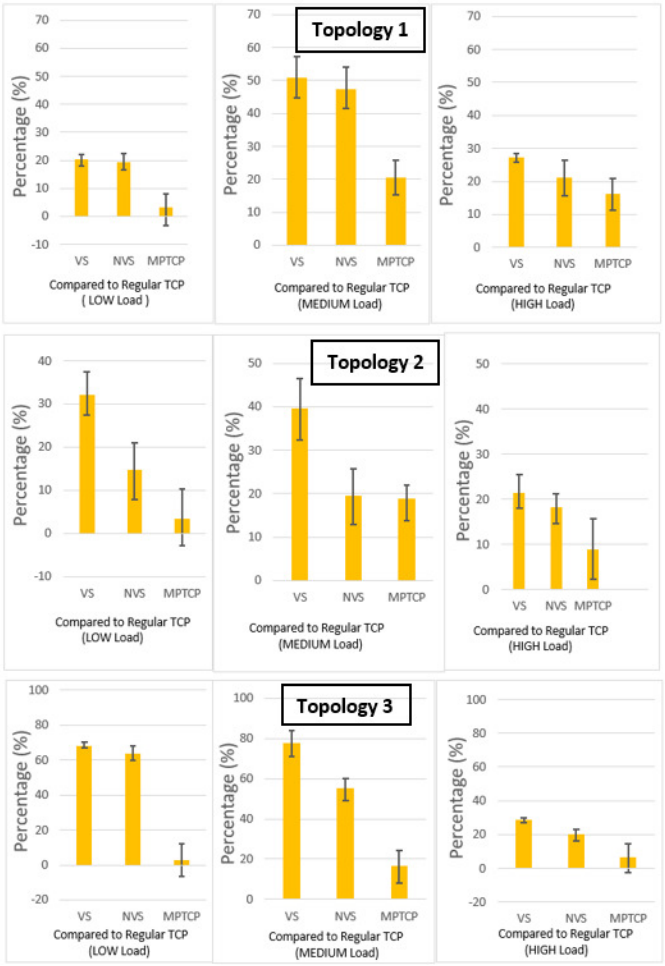

Fig. 5: Throughput Results for LOW, MEDIUM, & HIGH Load Cases

servers, which increases the number of ongoing TCP sessions across the network. Finally, a HIGH loaded case considers all hosts communicating with each other, either as a client or as a server, thus maximizing the traffic load to the maximum. Each $iperf$ session in any case between client and server was run for a fixed amount of time and equally to get a fair estimation of the overall bandwidth. For each case, we considered multiple separate simulation runs and showcase averaged results with the confidence interval of 90% approximately.

We found that MPTCP with VLAN specific paths (VS) and MPTCP without VLAN specific paths (NVS) achieves better throughput ahead of standard MPTCP as compared to regular TCP. On average, for LOW load cases in all three topologies, we can see 40% increase as compared to the regular TCP in MPTCP with VLAN specific paths, whereas, over 32% increase is noted against regular TCP in MPTCP without VLAN specific paths. Further, the trends are similar in case of MEDIUM load for all the three topologies, in fact better performance was observed, as depicted that 55% increase compared to regular TCP in MPTCP with VLAN specific paths, whereas, we see over 40% increase compared to regular TCP in MPTCP without VLAN specific paths and over 18% increase in standard MPTCP compared to regular TCP. Lastly, for HIGH Load cases, we found 25% increase compared to regular TCP in MPTCP with VLAN specific paths, whereas, over 19% increase compared to regular TCP in MPTCP without VLAN specific paths. Also, 10% increase in standard MPTCP as compared to regular TCP is reported in Fig. 5. Conclusively, we state that applying Hamiltonian paths for load balancing via OpenFlow and MPTCP in local and metro-area networks is a worthy solution and can enhance the overall network performance.

V. Concluding Remarks

In this paper, we investigated MPTCP that can greatly improve load balancing and network utilization. Whereas directing MPTCP traffic with the use of VLAN specific paths and even without VLAN specific paths produces even better results in the overall throughput of the network. Also, the concept of IP Aliasing could assist with associating multiple IP addresses with any host which would help with the generation of additional MPTCP subflows using OpenFlow. Future work includes the automation of the path calculation process at the switch level that can be beneficial and cost effective.

Acknowledgement

This work was supported in part by National Science Foundation award 1647189.

References

[1] Mohamed Boucadair, and Christian Jacquenet. “Software-defined networking: A perspective from within a service provider environment.” RFC 7149, March 2014.

[2] E. Haleplidis, K. Pentikousis, Spyros Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou. “RFC 7426: Software-Defined Networking (SDN): Layers and Architecture Terminology.” RFC 7426, Jan 2015.

[3] Yuanhao Zhou, Mingfa Zhu, Limin Xiao, Li Ruan, Wenbo Duan, Deguo Li, Rui Liu, and Mingming Zhu. “A load balancing strategy of sdn controller based on distributed decision.” In IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 851-856, 2014.

[4] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. “Balanceflow: controller load balancing for openflow networks.” In IEEE 2nd International Conference on Cloud Computing and Intelligent Systems (CCIS), vol. 2, pp. 780-785, 2012.

[5] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. “Achieving high utilization with software-driven WAN.” In ACM SIGCOMM Computer Communication Review, vol. 43, no. 4, pp. 15-26, 2013.

[6] R. V. D. Pol, S. Boele, Freek Dijkstra, Artur Barczyk, Gerben van Malenstein, Jim Hao Chen, and Joe Mambretti, “Multipathing with MPTCP and OpenFlow.” In SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC), pp. 1617-1624, 2012.

[7] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. “TCP extensions for multipath operation with multiple addresses” RFC 6824, January 2013.

[8] R. Khalili, N. Gast, and Miroslav Popovic. “Opportunistic linked-increases congestion control algorithm for MPTCP.” CiteSeer, Feb 2013.

[9] A. Walid, Q. Peng, J. Hwang, and S. Low, “Balanced Linked Adaptation Congestion Control Algorithm for MPTCP.” Internet Draft, draft-walidmptcp-congestion-control-04, Internet Eng. Task Force, Fremont, CA, USA, Jan. 2015.

[10] C. Raiciu, M. Handley, and D. Wischik. “Coupled Congestion Control for Multipath Transport Protocols.” RFC 6356, Oct 2011.

[11] ANSI/IEEE Standard 802.1Q, “IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks”, 1998.

[12] ”Mininet”, http://mininet.org/ [Last Accessed: 27- Dec- 2017]

[13] ”StaticFlowPusher” http://bit.ly/2pKbuO9 [Last Accessed: Dec 2017]