On the Benefits of Using Multipath TCP and Openflow in Shared Bottlenecks

Marcus Sandri, Alan Silva, Lucio A. Rocha, Fabio L. Verdi
Department of Computing
Universidade Federal de São Carlos
{marcus.sandri, alansilva, larocha, verdi}@ufscar.br

***Abstract*—This paper focuses on evaluating the use of MPTCP to forward subflows in OpenFlow networks. MPTCP is a network protocol designed to forward subflows through disjointed paths. Modern networks commonly use Equal-Cost Multipath Protocol (ECMP) to split flows through distinct paths. However, even with ECMP enabled, subflows may be forwarded through the same path. MPTCP improves the multipath routing by setting subflows to be forwarded through distinct paths. As a consequence, the amount of subflows must be considered to evaluate the network throughput. In this paper, we design Multiflow to use MPTCP in OpenFlow networks. Our proposal is to improve the throughput in shared bottlenecks by forwarding subflows from a same MPTCP connection through multiple paths. We validate our approach in a testbed where shared bottlenecks occur in the link at the endpoints. The Multiflow improvement of the network performance is evaluated in experiments about resilience and end-to-end throughput.**

Keywords*—*Multipath-TCP, Openflow, Multiflow, SDN.

I. Introduction

Today's network research are focused on new routing techniques for improving end-to-end transfer [1], [2]. Also, transfer protocols have been redesigned for enabling multipath forwarding [3]–[5]. These techniques are used to improve the resilience and the network throughput. As an example, large WANs such as the Internet use best-effort routing to forward packets and traffic to the same destination tend to use the same route although there are multiple different paths to reach its endpoints. Another example is related to data centers that may use multiple pathways to forward data traffic inside its network. However, few protocols provide better alternatives to use different and disjointed paths to increase the throughput between servers [6]. Those issues motivate the use of network protocols to allow benefits of routing through multiple paths.

Recently, MPTCP was proposed as a TCP extention for multipath forwarding [7]. The main idea is to use a single TCP connection and separate it into many derivative flows, called subflows. A subflow is a conventional TCP connection derived from a main MPTCP connection. The network operator sets automated network rules to forward distinct subflows among each avaliable network interfaces card (NIC). When the host has multiple NICs and the subflows are forwarded through them, the end-to-end throughput is improved due to the sum of throughput. The idea is that each NIC be connected with different switch-ports. However, sometimes the host has only one NIC and the subflows are forwarded to the same path. We call such cases as shared bottlenecks. A shared bottleneck occurs when many subflows use a single network link, what implies in over-utilization of the bandwidth in these links. The problematic issue about MPTCP is when there is over-utilization of links due to the increase of subflows genereted by this protocol.

A common solution for splitting TCP flows in a network is to use the Equal-Cost-MultiPath (ECMP). Such protocol uses statistical hashing on the 5-tuple and is capable of forwarding subflows into distinct paths. However, at least two issues need to be considered: a) The flows between the source and the destination are forwarded only by the shortest path; b) the traffic split is used in very specific scenarios and there is no guarantee that subflows will be forwarded always through to the different disjointed paths due to the statistical behavior of the traffic split [8]. MPTCP solves these ECMP issues using subflows. When the network operator increases the number of subflows, the probability of the more disjointed paths be occupied increases.

In this paper our main proposal is to improve the throughput in shared bottlenecks by forwarding subflows from a same MPTCP connection through multiple paths. Our intention is to guarantee that the subflows of the same MPTCP connection are sent using different routes in the network. By doing this, the first advantage is to bring resilience to the original MPTCP connection since the subflows are spread in the network using disjointed paths. A second advantage is to increase the final throughput by distributing the subflows among several paths. Multiflow is then an approach to divide MPTCP subflows for Openflow networks. This fine-grained controll of every MPTCP subflow is only possible because of OpenFlow that allows to have a total programability of the network elements. Multiflow was designed as a component for the POX controller. This component maps multiple subflows from the same MPTCP connection to multiple pathways. We validate our approach in a testbed where shared bottlenecks occur in the links. The experiments were conducted to evaluate the feasibility and Goodput (end-to-end throughput) with Multiflow in an OpenFlow network.

The remainder of this paper is organized as follows: Section II describes the related works. Section III details the Multiflow approach. In section IV we perform the evaluation. Finally, section V concludes the paper and points to future directions.

II. Related Works

Early studies about MPTCP presented its utilization in data centers networks (DCN) [2]. Raiciu et al. [2] describe that

MPTCP has a good performance when used with multiple NICs when the routing is done using ECMP. However, the authors did not consider any issue about OpenFlow networks.

The first work about MPTCP and Openflow is described by van der Pol et al. [9]. These authors perform a set of experiments using MPTCP as a transfer protocol and OpenFlow for Traffic Engineering (TE). Nevertheless, instead of our work, these experiments do not use MPTCP tokens for routing. The authors use the ECMP and the TRILL protocols for distributing flows. Additionally, these experiments use OpenFlow for discovering the network topology, applying rules and routing for each subflow using a distinct VLAN. We observed that the interest to integrate MPTCP and OpenFlow has increased in the literature. Sonkoly et al. [10] use a testbed to show that an OpenFlow network is capable of improving MPTCP by choosing the best path for the first subflow. The paper describes the strict correlations between the deployment of the first subflow and the network performance.

The MPTCP problems related to shared bottlenecks are described by Ming et al. [6]. The authors investigate the effects of shared bottlenecks in datacenters, and propose a new congestion control scheme called Equaly-Weighted Congestion Control. This scheme provides low computational cost to optimize the useful available bandwidth. EW-MPTCP increases the network bandwidth performance when used together with a single TCP connection. Zhou et al. [11] describe ECMP performance when the topology is fault-free, regular and balanced. Nevertheless, authors show that such scenario is far from real networks.

III. Multiflow Approach

This section describes our approach to improve the throughput in shared bottlenecks by forwarding subflows from a same MPTCP connection through multiple paths. We begin describing the functionalities of MPTCP. Then, we explain how MPTCP can be used with Openflow network.

A. MPTCP

MPTCP is a set of extensions to the regular TCP, which allows users to spread their traffic across potentially disjointed paths. MPTCP discovers the number of paths available to a user, establishes the paths, and distributes traffic across these paths through the creation of separate subflows [12].

As illustrated in Fig. 1, in MPTCP, each subflow is a standard TCP connection with its own sequence number space. An MPTCP level sequence number space is based on a Data Sequence Number (DSN), adding numbers bytes at the MPTCP level. A single MPTCP send buffer and a single MPTCP receive buffer are shared among all subflows, while each subflow has its own receive buffer to hold subflow level out-of-order data (since each subflow TCP receiver must deliver subflow level data in-order to the MPTCP receive buffer) [13].

When an application writes a stream of bytes to an MPTCP send buffer, MPTCP numbers each byte with a DSN. When bytes are then sent on a particular subflow, they are encapsulated into TCP-PDUs with MPTCP information placed in the TCP option field. When a TCP-PDU is received in-order at subflow level, the payload is delivered to the MPTCP receive buffer immediately. The MPTCP level cum-ack number, called DATA ACK, advances if the delivered data are also in-order at the MPTCP level [14].

The subflow receiver cum-acks those delivered data by a regular TCP cum-ack, and places the current DATA ACK in the TCP option field. An application consumes in-order data from the MPTCP receive buffer. Currently, an MPTCP sender only frees data from the MPTCP send buffer when they have been cum-acked by DATA ACK received on any subflow.

Data often arrive out-of-order at an MPTCP receive buffer because of loss and/or asymmetric RTTs of the subflows. In a heterogeneous network when different subflows have different characteristics (i.e., loss and RTT), the amount of out-of-order data arriving at the MPTCP receiver side can be significant [15].

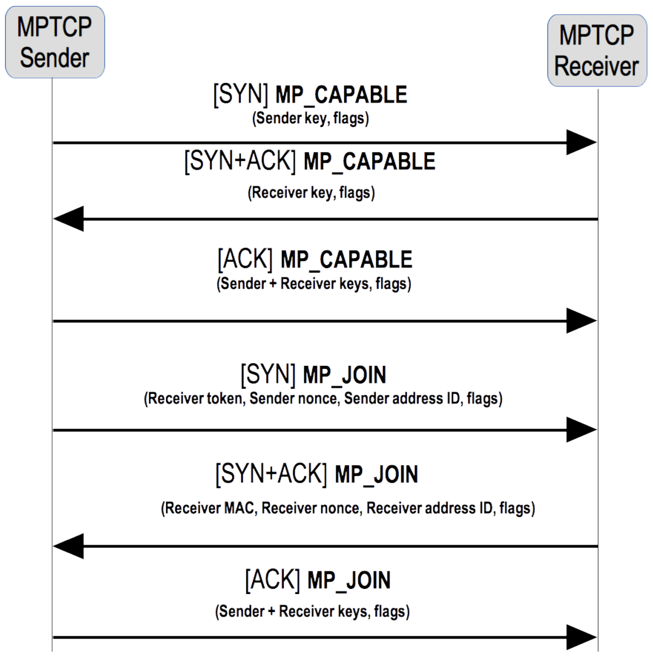

MPTCP Sender
MPTCP Receiver
[SYN] MP_CAPABLE
(Sender key, flags)
[SYN+ACK] MP_CAPABLE
(Receiver key, flags)
[ACK] MP_CAPABLE
(Sender + Receiver keys, flags)
[SYN] MP_JOIN
(Receiver token, Sender nonce, Sender address ID, flags)
[SYN+ACK] MP_JOIN
(Receiver MAC, Receiver nonce, Receiver address ID, flags)
[ACK] MP_JOIN
(Sender + Receiver keys, flags)

Fig. 1: MPTCP sequence diagram

Basically, an MPTCP session starts in the same way, with one change: the initiator adds the new MP_CAPABLE option with sender key and flags to the SYN packet. If the receiving host supports MPTCP, it will add that option to its SYNACK reply with receiver key and flags; the two hosts will also include cryptographic keys in these packets for later use. The final ACK (which must also carry the MP_CAPABLE option) establishes a multipath session using sender key, receiver key and flags.

A new subflow is added to the MPTCP connection by sending a SYN packet with receiver token, sender nounce, sender address id and flags using the MP_JOIN option; it also includes information on which MPTCP session is to be joined. After that, the receiving host replies with SYN+ACK that includes a receiver MAC, receiver nounce, receiver address id and flags. The data is transfered with ACK segment, and includes the sender key, receiver key and flags for connection enablement [15] as shown in Fig. 1.

The previously exchanged cryptographic keys are used to prevent such attacks from succeeding and in our work its used too for packet indexing. If the receiving server is amenable to adding the subflow, it will allow the establishment of the new TCP connection and add it to the MPTCP session [16].

Once a session has more than one subflow, it is up to the systems on each end to decide how to split traffic between them (though it is possible to mark a specific subflow for use only when any other no longer work). A single receive window applies to the session as a whole. Each subflow looks like a normal TCP connection, with its own sequence numbers, but the session as a whole has a separate sequence number; there is another TCP option (DSS, or "Data Sequence Signal") which is used to inform the other end how data on each subflow fits into the overall stream [17], [18].

Each MPTCP connection is identified by a unique token which is generated by the sender when receiving the receiver key. The token is obtained by the cryptographic hash of the receiver's key that was exchanged in the initial MP_CAPABLE handshake. After having the cryptographic hash, the token is generated by truncating the most significant 32 bits. Such token is then included in the MP_JOIN option to identify every MPTCP connection [7].

B. Multiflow Component

The Multiflow Component is a component for OpenFlow networks. An OpenFlow network is divided into two operational planes, as shown in Fig. 2: The data-plane is where packets are travelling through switches by using match/rule structure. In a match/rule structure, the match is used for mapping a packet to some rule. When a match cannot find any rule for some match, the packet is forwarded to the control plane. In the control plane, an OpenFlow controller processes the packet and signalizes the match/rule to the switches. As the OpenFlow controller is virtualized and programmable we can program match/rules to the switch mesh.

An MPTCP packet typically uses the TCP header to encapsulate control messages. Then, when an application communicates with a server, control messages are exchanged for trying to establish an MPTCP connection. Multiflow identifies control messages looking at the token which identifies subflows of the same MPTCP connection. Furthermore, as those packets are created as a TCP extension, it is possible to create OpenFlow rules by using its TCP port numbers (5-tuple). With this, we check if the packet belongs to the same MPTCP connection and then we set rules for spreading them into disjointed paths.

In our approach, we design a component called Multiflow for routing subflows into disjointed paths. Multiflow component is designed in a POX controller. The POX controller is an Openflow controller developed in Python programming language. POX uses OpenFlow 1.1 to establish communication

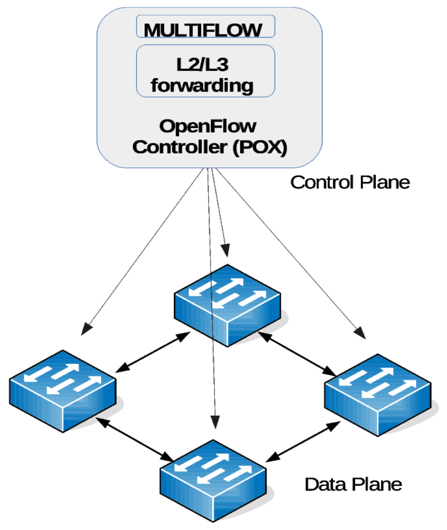


Fig. 2: Multiflow Architecture.

between datapaths in the OpenFlow switches. Fig. 3 shows the Multiflow component in the POX architecture.

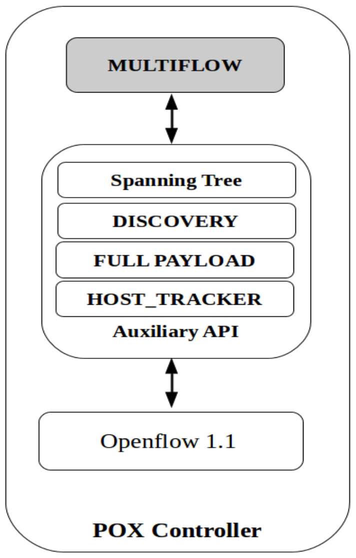


Fig. 3: Multiflow component in the POX Architecture.

The Multiflow component begins to listening the Discovery component. This is necessary because Multiflow uses this component to build the topology in a proactive mode, i.e., before any packet arrival. In parallel, Multiflow requests the Full Payload for MPTCP packets. Additionally, the POX Controller uses the Host_tracker component for detecting which port a given host is connected at each switch. Then, it is possible to prepare the Multiflow component to begin the process of the MPTCP protocol. We also use the Discovery component to find out all data-paths, switch-ports and their connections by using the Spanning-tree protocol (STP). Again, the Host_tracker component detects the availability of the hosts.

Multiflow component is divided into two main algorithms.

The first algorithm is responsible for finding the shortest path for MP_CAPABLE and send the match/rules to the switches. The second algorithm is called when an MP_JOIN arrives on the controller and is responsible for distributing MP_JOINS into disjointed paths. In fact, this algorithm is responsible for effectively routing the subflows of the same MPTCP connection through different paths.

In Algorithm 1, we consider the 5-tuple as a set of the IP source, IP destination, source port, destination port and protocol type. A Flow_entry is a new packet which has no match/rule in the switch and then is sent to the controller. The OpenFlow rules are messages with match/rules to the switch.

First of all, we need to separate MP_CAPABLE and MP_JOIN from each other. This is necessary because only MP_JOIN uses the token for identifying the connection. Then, when there is more than one MPTCP application which uses multiple MPTCP connections coming from the same host, all MP_CAPABLEs are forwarded through the same path. Then, the basic job of Algorithm 1 is to detect an MP_CAPABLE message and forward it to the destination host.

```
  input  : Flow_entry
  output: OpenFlow rules
1 if Flow_entry is tcp.syn then
2     if tcp.option == MPTCP_ID then
3         if MPTCP.option == MP_CAPABLE then
4             add_of.match(5-tuple)
5             add_of.action (default_route(match))
6         end
7     end
8 end
```
Algorithm 1: MP_CAPABLE inspection and forwarding.

In fact, Multiflow is interested only in dealing with TCP connections, more specifically with MPTCP connections. Then, UDP packets as well as traditional TCP flows are not of interest for the Multiflow component and are forwarded using any other protocol in the network. This means that all Flow_entry in the Multiflow component is a TCP header.

In order to identify an MP_CAPABLE message, Multiflow filters the SYN from the TCP header (line 1). It then identifies if the TCP SYN has the MPTCP header populated MPTCP_ID (line 2) by observing if the field "kind" is 30 (decimal) as the MPTCP IANA assigned value. In positive case, Multiflow inspects the MPTCP option field for identifying if the packet is an MP_CAPABLE or MP_JOIN (line 3). In case of being an MP_CAPABLE, Multiflow gives a default route to it (lines 4 and 5).

Algorithm 2 is responsible for identifying MP_JOINs and is considered the core of the Multiflow component. When an MP_JOIN arrives (line 1), Multiflow must route this subflow using a different and disjointed route from the previous subflows of the same MPTCP connection. This means that the Multiflow must know which routes the already established subflows of the same MPTCP connection are in use. For solving this, we use the token that comes with each MP_JOIN message (line 3) and associate it with a topology specific for each MPTCP connection. Then, each MPTCP connection will

```
1  if MPTCP.option == MP_JOIN then
2      if hash.get(raw_token) == None then
3          token ← mp_join.raw_token
4          best_path ←dijkstra($S, D, Topology$)
5          add_of.match(5-tuple)
6          add_of.action(best_path(match))
7          Topology ← del_path(best_path, Topology)
8          hash(token) ← Topology
9      end
10     else
11         newTopology ← hash.get(token)
12         best_path ←dijkstra($S, D, newTopology$)
13         add_of.match(5-tuple)
14         add_of.action(best_path(match))
15         Topology ← del_path(best_path, Topology)
16         hash(token) ← Topology
17     end
18 end
```
Algorithm 2: MP_JOIN inspection.

have its own topology and all the subflows of the same MPTCP connection (identified by having the same token) will use such individual topology. Fig 4 shows the idea with two different MPTCP connections (MPTCP1 and MPTCP2) each having two subflows, named SF1.1, SF 1.2, SF2.1 and SF 2.2 where SF stands for subflow and the indexes identify the MPTCP connection and the number of the subflow.

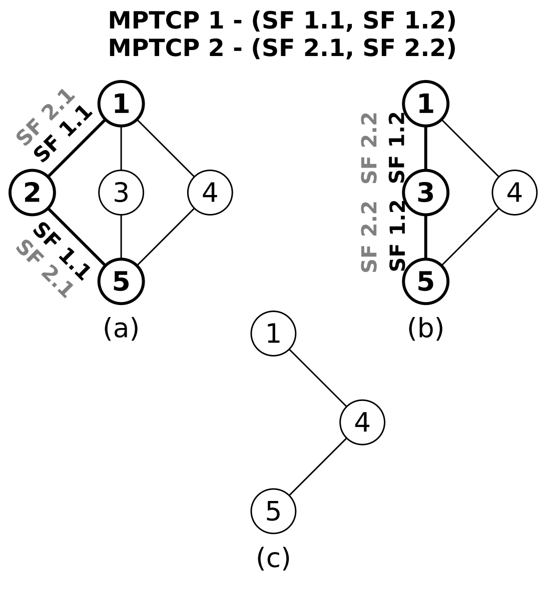


Fig. 4: Topology Virtualization with Multiflow.

Initially, the original topology is used for every new MPTCP connection. Then, suppose that the SF1.1 gets in the controller. Since this is the first subflow, the Multiflow uses the original topology as shown in Fig. 4 (a) and finds the shortest path between MPTCP client and MPTCP server (line 4). Suppose that the path chosen for forwarding SF1.1 is through switch 1, 2 and 5. The Multiflow sets the route for such subflow using OpenFlow rules (lines 5 and 6) and prunes (line 7) the route just used resulting in the topology shown in Fig. 4 (b). Such new topology is then stored associating it with the token which identifies this MPTCP connection (line 8). Now suppose that the second subflow of the same MPTCP connection arrives, say SF1.2. Multiflow verifies that already exists sublflows of this MPTCP connection (line 2) in the network and uses the pruned topology of Fig. 4 (b) (line 11) for finding a new disjointed path for SF1.2 (line 12). Again, Multiflow prunes the route just used resulting in the new topology of Fig. 4 (c). By doing this on and on, we guarantee that subflows of the same MPTCP connection use different disjointed paths.

Now consider that SF 2.1 gets in the controller which means that the first subflow of a new MPTCP connection (MPTCP2) is coming. For this subflow, the original topology is used (Fig. 4 (a)). So, the Multiflow finds the shortest path, sets the OpenFlow rules and prunes the route just used so that the next subflows of this MPTCP connection does not use this path. This algorithm is repetead as each new subflow gets in the controller.

Note that by applying the algorithm explained above, subflows of the same MPTCP connection will be allocated in disjointed paths (for resilience and throughput) and subflows of different MPTCP connections are allocated possibly in the same paths. Also observe that the idea of pruning the topology give the notion of having a virtual topology being manipulated for every MPTCP connection. It can be seen as having several different logical topologies over the single physical topology.

Analysing a little deeper the scenario of Fig. 4, we see that the original topology has 3 disjointed paths meaning that only three subflows for each MPTCP connection can be forwarded disjointedly. If a fourth subflow gets in the network, no disjointed path will be found. For solving this, Multiflow starts again with the original topology and allocates the fourth subflow in one of the shortest paths available. This will clearly configure subflows of the same MPTCP connection in the same paths. However, as we proved in the evaluation done in the next section, the throughput of a same MPTCP connection only increases if there are enough disjointed paths for allocating the different subflows in different paths. Then, it is not reasonable to have more subflows than the quantity of disjointed paths.

In the following Section, we show some experiments and evaluations with Multiflow.

IV. Experimental Evaluation and Results

The Multiflow architecture has been implemented and tested to validate our approach. The Multiflow was developed and tested with OpenFlow 1.1 and MPTCP v.88. We used a testbed compiled with MPTCP kernel in Ubuntu 13.10 and Mininet 2.1.0. Mininet simulator uses the physical machine kernel for setup the kernel in its hosts. All simulations have been done by using the following hardware configuration: one Intel i7 with 8 CPU cores (8 threads) and 8GB RAM. We restrict this machine to be only for our experimentation. We set a link-bandwidth of 10Mbps in Mininet to evaluate all the experiments.

A. *Throughput Evaluation*

In this section we demonstrate the throughput of MPTCP when used together with Multiflow. In our experiments, we used the topology shown in Fig. 5, which has three disjointed paths composed by OpenFlow switches. We used an MPTCP peer (client - server) connected to the switches 1 and 5 respectively. There is also two machines in the network responsible for sending an UDP background traffic of 7Mbps using the route 1, 3 and 5.

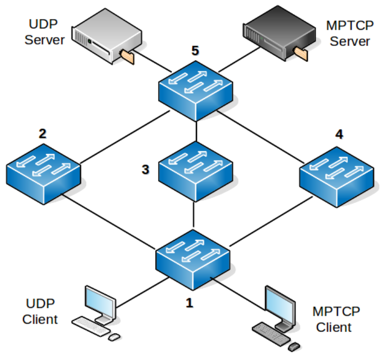


Fig. 5: Experimental Topology.

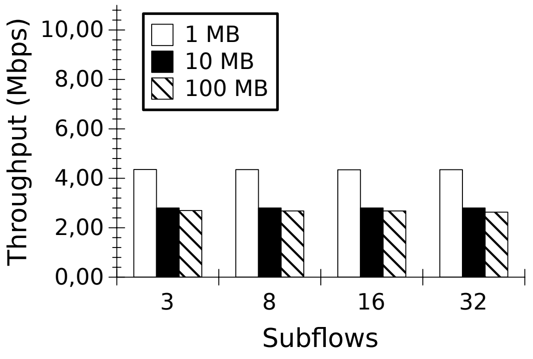
The MPTCP machines will transfer files of 1MB, 10MB and 100MB. For each file, we have only one MPTCP connection with 3, 8, 16 and 32 MPTCP subflows. The idea in this experiment is to compare pure MPTCP (without distributing the subflows through disjointed paths) versus MPTCP with Multiflow (distributing flows through disjointed paths).

Fig. 6 (a) shows the scenario in which all the MPTCP subflows are sent using the same route (switches 1, 3 and 5). Note that this route is the same route of the UDP traffic. It is interesting to observe that the results of Fig. 6 (a) depict that as we increase the number of subflows, the throughput of MPTCP connection does not increase. This is an expected result since there is no use of different paths for the subflows. Now, observe Fig. 6 (b) which shows the results when we use Multiflow. Here, the subflows are routed using the three disjointed paths and the throughput increases for the transfer of 10MB and 100MB. This occurs because the MPTCP congestion control increases the window in the paths with better throughput (route 1, 2, 5 and route 1, 4, 5).

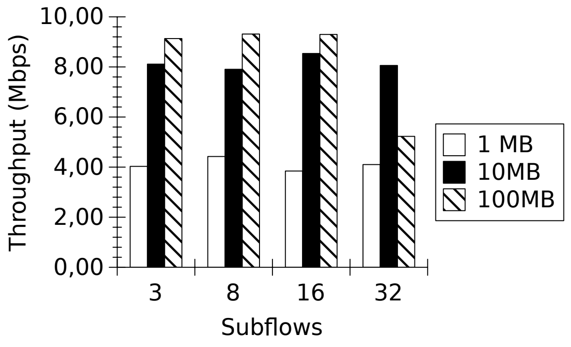
Also, it is possible to see that as we increase the number of subflows, there is no gain in the throughput. More than that, when we have 32 subflows, the throughput starts to decrease due to the overhead of having to deal with many open subflows. So, the conclusion with this test is two fold: first, routing MPTCP subflows using different routes increases the throughput; second, increasing the quantity of MPTCP subflows without increasing the quantity of disjointed paths does not help to increase the throughput and can even get worse results. Then, based on this result, it would be reasonable to avoid the creation of more MPTCP subflows than the quantity of available disjointed paths.

B. *Bottleneck Sharing Evaluation*

In this section, we evaluate the benefits of Multiflow when multiple MPTCP peers are present in the network. The purpose



(a) MPTCP subflows using the same route.



(b) Multiflow using disjointed paths.

Fig. 6: MPTCP and Multiflow working together.

of this experiment is to compare pure TCP connections with Multiflow. We used the topology of Fig. 7, which makes use of three disjointed paths and has 3 MPTCP peers (client - server), connected to switch 1 and 5 respectively. In this scenario, there is no UDP traffic.

For this experiment, we again transfer files of 1MB, 10MB and 100MB but instead of open 3, 8, 16 and 32 MPTCP subflows, we just open 3 subflows for each MPTCP connection due to the conclusion we had in the previous section (shown in Fig. 6 (b)).

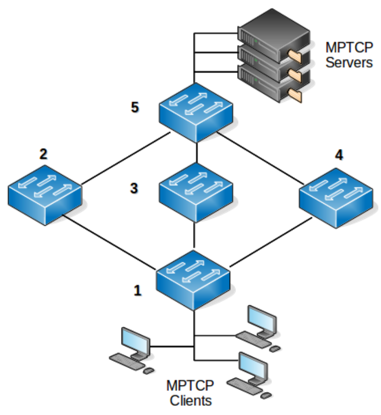


Fig. 7: Openflow Network Topology: 3 MPTCP Servers and 3 MPTCP Clients.

Fig. 8 (a) shows the results for sending TCP traffic (no MPTCP) using the route composed by the switches 1, 3 and 5. Note that as the number of TCP connections increase, the throughput decreases since such flows are sharing the same route. Fig. 8 (b) shows the scenario with Multiflow. In this case, each peer opens one MPTCP connection and three subflows for each connection are created, totalizing 9 subflows. The subflows are sent using the three disjointed paths available in the topology applying the Multiflow algorithm for separating subflows of the same MPTCP connection in different paths. The first advantage of using Multiflow here is the resilience in case of failures. In the scenario where Multiflow is not used, if a link or device fails in route 1, 3, 5, all the TCP connections will be disrupted.

Note also that the throughput is basically the same in both scenarios (with and without Multiflow). However, when we observe the numbers, the throughput for 100MB when using Multiflow is higher than when Multiflow is not used, 4,67Mbps and 4.06Mbps, respectively. Then, by observing this experiment the conclusion is that Multiflow is useful for longer flows also known as elephant flows. For short flows, the performance in terms of throughput does not increase.

C. Assyptotic and Setup Time Evaluation

We also analysed the assymptotic time of Multiflow as well as the RTT of the JOIN message (setup time) when using Multiflow versus STP.

Fig. 9 shows the best, worse and medium cases in terms of time to run the Multiflow algorithm. The time considered is the time necessary only to parse the MPTCP packet. The time to set the OpenFlow rules in the network is not considered.

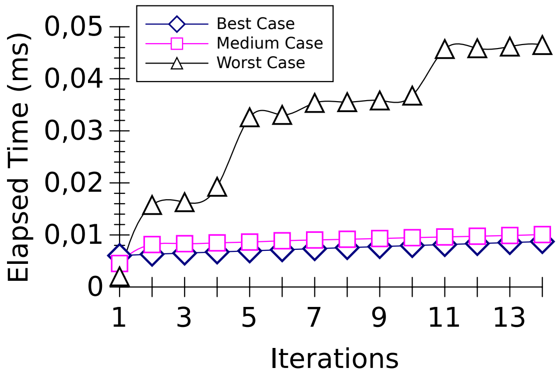
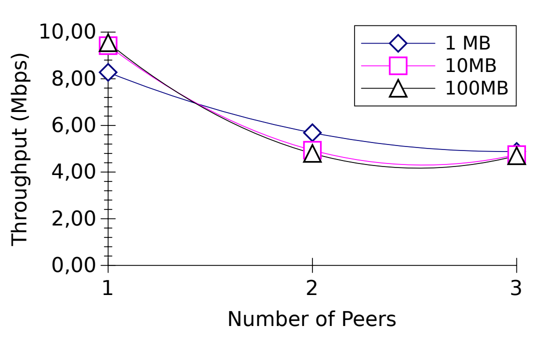
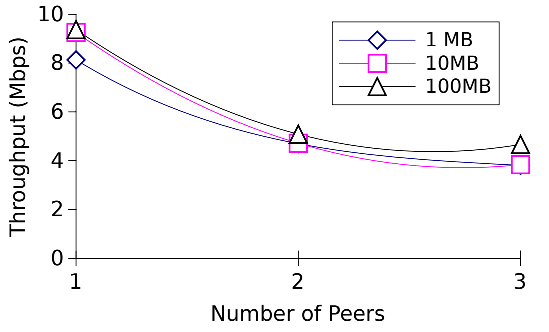


Fig. 9: Assymptotic analysis.

We can observe that the best case represents the scenario in which the TCP packet is not an MPTCP message and then Multiflow does nothing with this packet. The worst case is when the packet is a JOIN message. In this scenario, Multiflow needs to recover the virtual topology correspondent to that MPTCP connection (by using the token), find a new route,



(a) Three TCP connections using the same path (no Multiflow).



(b) Three MPTCP connections each having three subflows (with Multiflow).

Fig. 8: TCP Reno vs. Multiflow.

prune the new route just used and store this new virtual topology for further use. The medium case is when the packet is an MP_CAPABLE message by which Multiflow needs only to find a route for sending the message.

The setup time was calculated by taking the RTT necessary to a JOIN message using Multiflow versus STP. We used the STP that comes together with the POX controller. STP protocol creates proactively all the routes from every pair of hosts in the network. So, the expected result is that the setup time for STP is lower than the setup time for Multiflow. However, as well known, STP is a very simple solution for L2 forwarding and does not take into account any of the features that Multiflow considers. Fig. 10 depicts the numbers.

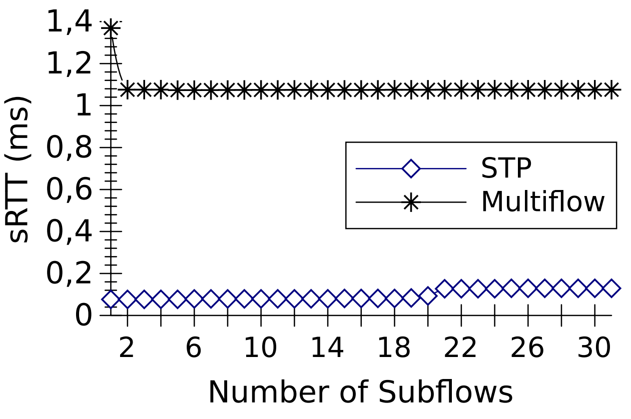


Fig. 10: Setup time.

V. Conclusion and Future Work

In this paper, we show the benefits of using MPTCP and OpenFlow together in shared bottlenecks. With an OpenFlow architecture we could design the Multiflow controller for forwarding MPTCP subflows into disjointed paths. The first advantage of this is to have a fine-grained control for routing subflows of the same MPTCP connection in different routes. By doing this, resilience is obtained and the throughput increases for scenarios with elephant flows.

As mentioned in the paper, we observe that to open more subflows than the available disjointed paths does not help to get more throughput. In this paper, Multiflow blocks further JOIN messages beyond the number of disjointed paths in our topology. However, this is currently done in a very simple way and must be improved considering any topology. Then, Multiflow should be capable of discovering how many disjointed paths there exist for every pair of source and destination hosts in the network and then blocks JOIN messages if there are no disjointed paths. Further studies include this aspect and the comparison with ECMP, one of the most used protocols for datacenter networks.

Acknowledgment

We would like to thank Mr. Murphy McCauley for his several contributions for the development of this work. We also thank CAPES, CNPq and FAPESP for the financial support.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: Enabling Innovation in Campus Networks.”

[2] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley., “Improving Datacenter Performance and Robustness with Multipath TCP,” *In Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM ’11)*, 2011.

[3] M. Arye, E. Nordstrom, R. Kiefer, J. Rexford, and M. Freedman, “A Formally-verified Migration Protocol for Mobile, Multi-homed Hosts,” in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, Oct 2012, pp. 1–12.

[4] K.-J. Grinnemo, A. Brunstrom, and J. Cheng, “Using Concurrent Multipath Transfer to Improve the SCTP Startup Behavior for PSTN Signaling Traffic,” in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*, May 2014, pp. 772–778.

[5] B. Arzani, A. Gurney, S. Cheng, R. Guerin, and B. T. Loo, “Impact of Path Characteristics and Scheduling Policies on MPTCP Performance,” in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*, May 2014, pp. 743–748.

[6] L. Ming, A. Lukyanenko, S. Tarkoma, and A. Yla-Jaaski, “MPTCP Incast in Data Center Networks,” *Communications, China*, vol. 11, no. 4, pp. 25–37, April 2014.

[7] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, “TCP Extensions for Multipath Operation with Multiple Addresses,” *IETF RFC*, 2013.

[8] M. Chiesa, G. Kindler, and M. Schapira, “Traffic Engineering with Equal-Cost-Multipath: An Algorithmic Perspective,” in *INFOCOM, 2014 Proceedings IEEE*, April 2014, pp. 1590–1598.

[9] R. van der Pol, S. Boele, F. Dijkstra, A. Barczyk, G. van Malenstein, J. Chen, and J. Mambretti, “Multipathing with MPTCP and OpenFlow,” *High Performance Computing, Networking, Storage and Analysis (SCC 2012)*, 2012.

[10] B. Sonkoly, F. Nemeth, L. Csikor, L. Gulyas, and A. Gulyas, “SDN Based Testbeds for Evaluating and Promoting Multipath TCP,” in *Communications (ICC), 2014 IEEE International Conference on*, June 2014, pp. 3044–3050.

[11] J. Zhou, M. Tewari, A. K. Min Zhu, L. Poutievski, A. Singh, and A. Vahdat., “WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers.” *In Proceedings of the Ninth European Conference on Computer Systems (EuroSys ’14)*, 2014.

[12] S. Barré, C. Paasch, and O. Bonaventure, “Multipath TCP: From Theory to Practice,” in *NETWORKING 2011*. Springer, 2011, pp. 444–457.

[13] C. Diop, J. Gomez-Montalvo, G. Dugue, C. Chassot, and E. Exposito, “Towards a semantic and mptcp-based autonomic transport protocol for mobile and multimedia applications,” in *Multimedia Computing and Systems (ICMCS), 2012 International Conference on*. IEEE, 2012, pp. 496–501.

[14] A. Alheid, D. Kaleshi, and A. Doufexi, “An Analysis of the Impact of Out-of-Order Recovery Algorithms on MPTCP Throughput,” in *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, May 2014, pp. 156–163.

[15] C. Paasch and O. Bonaventure, “multipath tcp,” *Communications of the ACM*, vol. 57, no. 4, pp. 51–57, 2014.

[16] C. Paasch, R. Khalili, and O. Bonaventure, “On the Benefits of Applying Experimental Design to Improve Multipath TCP,” in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 393–398.

[17] S. Barré, O. Bonaventure, C. Raiciu, and M. Handley, “Experimenting with Multipath TCP,” *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 443–444, 2011.

[18] D. Wischik, “Multipath: A new control architecture for the internet: Technical perspective,” *Communications of the ACM*, vol. 54, no. 1, pp. 108–108, 2011.