A Simple Multipath OpenFlow Controller using topology-based algorithm for Multipath TCP

Chawanat Nakasan
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma Nara 630-0101, Japan
chawanat.nakasan.cb5@is.naist.jp

Kohei Ichikawa
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma Nara 630-0101, Japan
ichikawa@is.naist.jp

Hajimu Iida
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma Nara 630-0101, Japan
iida@itc.naist.jp

Putchong Uthayopas
Kasetsart University
50 Phahonyothin Rd., Lat Yao, Chatuchak
Bangkok 10900, Thailand
pu@ku.ac.th

ABSTRACT

Multipath TCP, or MPTCP, is a widely-researched mechanism that allows a single application-level connection to be split to more than one TCP stream, and consequently more than one network interface, as opposed to the traditional TCP/IP model. Being a transport layer protocol, MPTCP can easily interact between the application using it and the network supporting it. However, MPTCP does not have control of its own route. Default IP routing behavior generally takes all traffic through the shortest or best-metric path. However, this behavior may actually cause paths to collide with each other, creating contention for bandwidth in a number of edges. This can result in a bottleneck which limits the throughput of the network. Therefore, a multipath routing mechanism is necessary to ensure smooth operation of MPTCP. We created smoc, a Simple Multipath OpenFlow Controller, that uses only topology information of the network to avoid collision where possible. Evaluation of smoc in a virtual local-area and a physical wide-area SDNs showed favorable results as smoc provided better performance than simple or spanning-tree routing mechanisms.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Centralized networks; C.2.2 [**Network Protocols**]: Routing protocols

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

Networked systems, such as distributed database, computation, and file storage, have become more complex with increasing capacity. This evolution increases demands on the network. Many practices have been developed to improve their functionality or alleviate problems, such as *multi-homing* which connects a system to the Internet through multiple gateways. Another concept, *multi-site*, refers to the practice of distributing the system to multiple geographic locations. These practices have many benefits including locality, capacity, and redundancy. When these two concepts are used together, the sites of the networked system can be connected through wide-area network (WAN) by multiple paths.

However, having multiple paths between the sites does not mean both paths are always used. In a traditional network model, one application layer socket is supported by one transport-layer session [8], which is supported by a fixed pair of network-layer and link-layer endpoints [22]. *Multipathing* was developed to allow a host or network to utilize multiple paths at the same time. Multipathing has many benefits such as increasing maximum available bandwidth, balancing network load, and providing redundancy. Multipathing with commodity network medium can also be used as a low-cost alternative to using a single expensive network medium. Many multipathing solutions exist, with their own set of benefits and problems.

Multipath TCP (MPTCP) [10, 11] is a promising multipathing protocol which was developed as an extension to TCP. As an extension, MPTCP was designed to be backwards-compatible with current applications and networks. Additionally, being a transport layer protocol based on TCP, it has congestion control, making it useful in WANs which can be (relatively to LANs) unequal and unstable.

Even though MPTCP has many advantages, it also has a major shortcoming. As a transport layer protocol, it has no control over its own routing. Routing problems can arise if sites on a network are connected through a shared infrastructure and the routing system works in a legacy manner. Without the knowledge that MPTCP is being used, the network may route the multiple MPTCP "subflows" through the same path causing a bottleneck, while also leaving some other paths unused causing underutilization. When this is the case, the benefits of MPTCP would be limited due to the bottleneck and underutilization. Spreading MPTCP traffic across multiple paths become an important topic because its usefulness could be improved.

The role of a customizable routing mechanism that would be suitable for developing a routing mechanism for MPTCP would be easily filled in by OpenFlow [18], a software-defined network (SDN) protocol.

In this work, we aim to create a simple and efficient Open-Flow controller that splits and distributes MPTCP traffic through the network. This would increase bandwidth utilization of MPTCP in the network so that full capacity may be used. We have three core ideas behind our controller design. Firstly, the controller should be backwards-compatible with non-MPTCP traffic. Secondly, we strive to increase bandwidth available to an application. Finally, we attempt to explore various multipath routing strategies; one simple strategy is presented in this paper.

Our work is primarily targeted at large-scale, multi-homed multi-site systems connected together using OpenFlow. This class of systems include distributed storage, database, content delivery network (CDN), high-performance computing (HPC), or systems with disaster recovery (DR) sites, which are usually far away from the main site and connected through WANs. Since bandwidth in WAN is more limited compared to local-area networks, improved efficiency has greater impact in this kind of network.

2. BACKGROUND
2.1 Software-defined Network
OpenFlow is an SDN protocol which allows network traffic control and management from the centralized OpenFlow Controller. Centralization allows the network elements to be programmed to add or remove any switching or routing rules in its *flow table*. While OpenFlow provides programming flexibility to the network and allows many concepts such as QoS or traffic engineering to be realized, it cannot modify communication pattern between the end hosts. In traditional TCP/IP protocol suite, only one route or path is used per connection. This limits the maximum bandwidth to only one path, and not that of the entire network. This limitation cannot be circumvented by OpenFlow.

2.2 Multipathing
Multipathing allows us to use more than one path in one logical connection, increasing bandwidth utilization, improving redundancy and stability, as well as allowing seamless handovers in certain environments. It is especially useful in multi-homed systems, which see a recent upward trend. Multipathing has been attempted from many perspectives

for various purposes. We can roughly classify them into three general classes based on TCP/IP model layer as follows.

2.2.1 Application Layer
In the application layer, multipathing can be done by creating multiple sockets from the application. One notable example, GridFTP [4, 3], uses multiple TCP streams in parallel to improve performance along the network topology [13] by extending FTP to support parallel streams. However, application layer multipathing can be error-prone [5] and hard to maintain [8]. This is because the task of working with the multiple paths and flows will fall upon the application, which is not aware of the many mechanisms that are already available and working in the transport layer [17]. For example, the application may not be aware of unequal paths and continue to push equal data to both sockets, causing traffic to stay behind in the slower path.

2.2.2 Network Layer
Equal-Cost Multipath (ECMP) [24, 15] is a multipathing mechanism that allows multiple paths to be used on the network layer when they have equal costs. While it is simple and efficient as it can quickly select paths based on the packet header, TCP is not aware of ECMP. Some ECMP path selection and hashing strategies (such as simple round-robin) may cause packets to arrive out-of-order or unbalance the network, prompting TCP to retransmit as multiple duplicate ACKs may be received, resulting in decreased network performance [8]. Additionally, ECMP is designed for *equal*-cost networks and therefore will not work when path costs are not equal, such as in WANs, due to variation in bandwidth and latency.

2.2.3 Transport Layer
Transport layer is more informed about each path's conditions than the applications [5] and also more aware of high-level connections than the network layer. One prominent example is Stream Control Transmission Protocol (SCTP) [23]. However, while SCTP is also capable of multipathing, the feature is aimed for redundancy, not bandwidth utilization. Additionally, middleboxes such as network address translators (NATs) are not aware of SCTP and may block it. Applications also need to explicitly use SCTP because it is a completely new protocol, presenting a further compatibility problem [10].

To address this compatibility problem, protocols such as *concurrent TCP (cTCP)* [8] and *M/TCP* [21] (along with others mentioned in [6]) are based on or compatible with TCP. Among these protocols, MPTCP is one of the most promising as it has rich features, backwards-compatibility with TCP, and extensive research, including a Linux kernel implementation [20].

2.3 MPTCP
Multipath Transmission Control Protocol, or *MPTCP* is an extension to *TCP* at the transport layer that utilizes multiple paths between two network endpoints by stripping data into multiple *subflows*. Each subflow behaves like a TCP flow, with its own congestion control, send and receive windows, and so on. MPTCP interface for applications is a
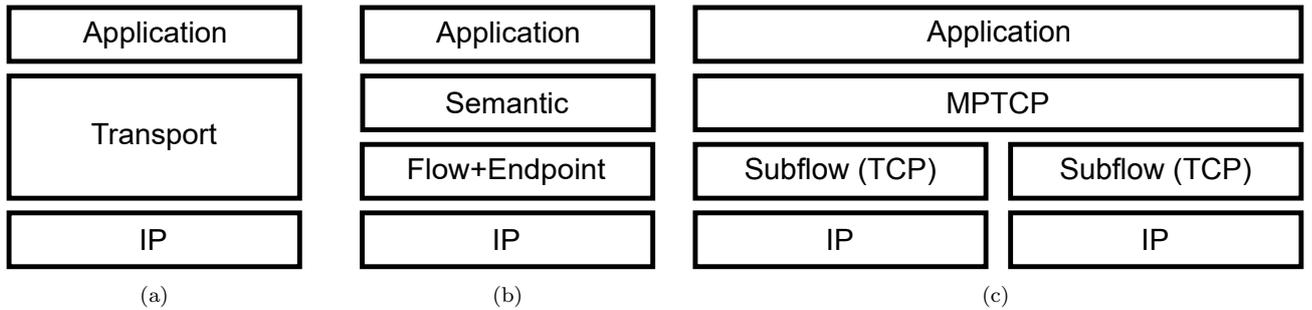
Figure 1: Relationship between the traditional TCP/IP network model and derivation that led to the principle design of MPTCP. (a) shows the regular TCP/IP model, (b) is the decomposition of transport layer which is the basis of MPTCP, and (c) is the actual MPTCP implementation of this model.

complete drop-in replacement, meaning that the applications need not be modified. An MPTCP session would be created for each socket opened by the application. For example, opening 5 sockets to download a file from an HTTP server would result in 5 separate MPTCP sessions being opened for the download application.

By decomposing the transport layer into two sublayers, as shown from Figure 1a to Figure 1b [12], MPTCP can separately recognize end-to-end and point-to-point situations better than the traditional model by having the upper half, which is the MPTCP extension, manage the connections and subflows, while the lower half works with congestion and other matters in each subflow as in Figure 1c.

2.4 Our interpretation of *multipathing*

Although reviewed literature does not give a clear definition for the term *multipathing*, they generally agree that it means *creating multiple network connections or sessions between a pair of hosts, with the connections or sessions traveling through different paths (when available) across the network.* We will use this meaning in our work.

3. DESIGN OF MPTCP ROUTING MECH-ANISM

Two actions are necessary to route MPTCP traffic through the network using multiple paths. First, we need to know which subflows belong to which instance of MPTCP. Second, we also need to decide which paths would be used and when. These actions are further discussed in the following two subsections.

3.1 Identifying MPTCP subflow group on the network layer

As stated above, we need to identify which subflows belong to the same MPTCP instance. Since all MPTCP information is encoded as TCP options, not headers, it is impossible to simply use OpenFlow's normal matching methods to identify MPTCP subflow grouping. Therefore, a method to identify the subflows from the OpenFlow controller's perspective is necessary. In order to do so, special information beyond IP addresses and TCP port numbers are required.

Fortunately, MPTCP exchanges all needed information during the initial MPTCP subflow establishment (using MP_CAPA
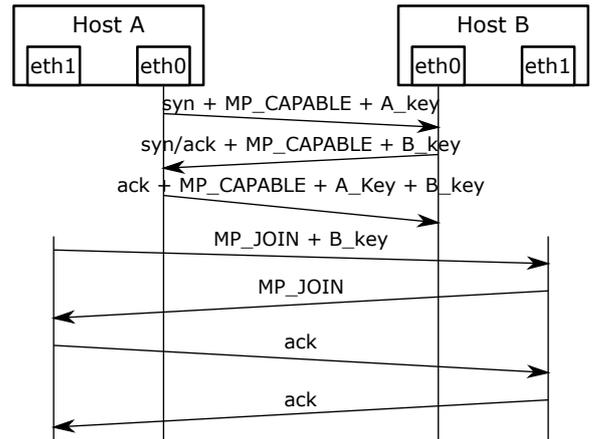

Figure 2: MPTCP handshake process, annotated with a partial list of TCP and MPTCP option fields used in our work.

BLE TCP option) and subsequent subflows (using MP_JOIN option). MPTCP relies on *keys* and *tokens* to identify a connection endpoint which is unique for each connection and host. We can use this identification information to find which subflows belong to which MPTCP connection[1]. When MPTCP creates a new instance for the first time, each host sends its own key to the other host. When a host establishes an additional subflow, it (A) will send the other party (B)'s key to identify an MPTCP session it (A) wishes to join. As this process uses different IP address and TCP port pairs, an OpenFlow packet_in message will be sent from an OpenFlow switch to the controller, which would use this information. This process is illustrated in Figure 2 and is used as a basis for flow detection and grouping in our routing algorithm.

3.2 Finding and using multiple paths

Apart from correctly identifying MPTCP subflows, we need to know which paths each flow should take. We aim to create a suite of MPTCP and OpenFlow working cooperatively in the same system implementation. In this work, OpenFlow would find optimal *path sets*, a collection of paths that lead

---

[1]In MPTCP, keys are later hashed, truncated, and called *tokens*. As we do not work on the full process of MPTCP, we will not care about the differences between these terms. *Key* will be used throughout this paper for simplicity.

a packet from one host to another, and decide which path an MPTCP subflow should use. This mechanism involves multiple stages: first we analyze the packet and gather or match information with the database, we may need to find a new path set if necessary, then the path set must be applied to new MPTCP subflows as they are created. By cycling through the different paths in a path set, MPTCP subflows can be distributed to multiple paths.

3.2.1 Path Set Calculation Algorithm

Algorithm 1 Algorithm to find a path set to route from S1 to S2 in network graph G

Require: graph $G(V, E)$
Require: $S1, S2 \in switches$
 $G(V, E) \leftarrow NetworkTopology(switches, links)$
 $PrimaryPath \leftarrow \text{shortest\_path}(G, S1, S2)$
 $AltPaths \leftarrow \text{all\_simple\_paths}(G, S1, S2) - PrimaryPath$
 $AltPaths \leftarrow AltPaths$ sorted by number of edges shared with PrimaryPath ascending, by length of path ascending
 $PathSet \leftarrow PrimaryPath + AltPaths$
 return $PathSet$

Algorithm 1 describes a simple method to find a path set for multipath use. When supplied with a network topology graph and the source and destination switches, the algorithm chooses one shortest path as the *primary path*. The remaining paths are sorted and prioritized to minimize path sharing with the primary path, and then by path length. We used the shortest path and all simple path functions from `networkx`[14] package. The path sets are stored as Python `itertools.cycle` object, which allows us to easily cycle through all paths inside.

3.2.2 Collecting and Managing MPTCP Subflows

To maintain the states of MPTCP subflows, we use three tables to store and match the subflows and assign them to routes by Algorithm 2:

1. `pending_capable` table stores information of first SYN packets sent by the MPTCP initiator using `MP_CAPABLE` message. It maps the IP address and TCP port to the initiator's hash and also stores the path set from the initiator to the listener.

2. `pending_join` table does a similar function for subsequent subflows created by `MP_JOIN` messages.

3. `mptcp_connections` table stores established MPTCP connections. Once an entry in the previous two tables is matched by a reply packet (TCP ACK), that entry is removed from its original table and the path set will be stored here. It maps a destination's key to the source's key and the path set from source to destination.

4. IMPLEMENTATION OF SMOC: SIMPLE MULTIPATH OPENFLOW CONTROLLER

To achieve our goal of solving the multipath bottleneck problem by using OpenFlow to route MPTCP, we implemented the algorithms described in Subsection 3.2.1 in our controller, Simple Multipath OpenFlow Controller (smoc). The core of smoc is based on POX, a well-known OpenFlow
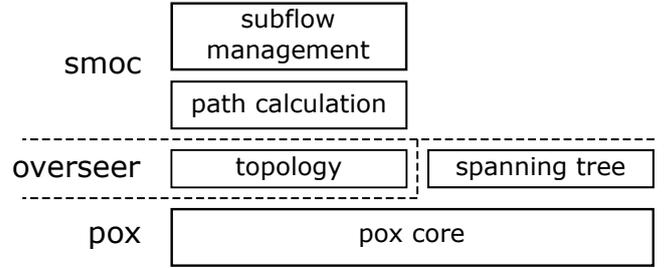

Figure 3: Components of the `smoc` controller, based on POX framework and Overseer's topology management modules.

framework. POX was chosen due to its modularity which means new features can be rapidly developed. Topology management and path management features are based on Overseer [27, 26] which is also an OpenFlow controller based on POX. Overseer's original purpose is to optimize routing based on characteristics of applications. To serve its purpose, Overseer has well-designed topology management and path management features, which also form the basis for smoc.

Path finding is assisted by the `networkx` Python package while path selection is based on Algorithm 1. smoc uses Algorithm 2 to manage all subflows and incoming packets. Underlying maintenance functions such as spanning tree management and OpenFlow protocol are handled by POX and Overseer.

To route flows, we maintain a list of pending and connected sessions. When we receive a new connection handshake message, we calculate a new path set and add both the information of the connection initiation and the path set to the pending list. When the pending connection is responded, we calculate another path set for the reverse direction and move everything to the connected sessions list. Any subsequent connections would only require a lookup in the connected sessions list to find an appropriate path.

5. EVALUATION AND RESULTS

We evaluated `smoc` against POX's original spanning tree controller (henceforth, POX S-T). With this controller, all MPTCP traffic would be confined to a single path even if multiple paths actually exist in the network. We chose this controller because it is based on the same framework and architecture, and spanning tree is commonly used to prevent loops in network topology. However, spanning tree eliminates any sort of multiple paths that exist at the network topology level. This means POX S-T always produces a single path between any pair of hosts. Being based on the same technology as smoc, all basic program libraries would be the same. This makes POX S-T suitable for an experimental control.

We chose `iperf` as our benchmarking tool due to its simplicity. smoc was evaluated in two testbeds, a local- and a wide-area testbed, which represented different network environments.

In the local testbed, two topology configurations shown in (Figure 4) are modeled after a previous work from our re-

Algorithm 2 Algorithm to handle incoming MPTCP packets that trigger OpenFlow packet-in message

Require: *packet*
Ensure: *route* to route the flow *packet* belongs to
  pending_capable ← hash((init_ip_port, listen_ip_port) → (init_key, pathset))
  pending_join ← hash((init_ip_port, listen_ip_port) → (listen_key, pathset))
  mptcp_connections ← hash(dst_key → (src_key, pathset))
  if *packet* is `MP_CAPABLE` message **then**
    if (*packet.dst_ip_port*, *packet.src_ip_port*) in *pending_capable* **then**
      recvkey, ABpathset ← *pending_capable*[(*packet.src_ip_port*, *packet.dst_ip_port*)]
      BApathset ← find new pathset
      add *recvkey* → (*packet.sendkey*, *ABpathset*) to *mptcp_connections*
      add *packet.sendkey* → (*recvkey*, *BApathset*) to *mptcp_connections*
      delete key (*packet.dst_ip_port*, *packet.src_ip_port*) from *pending_capable*
      return *BApathset*.next()
    else
      ABpathset ← find new pathset
      add (*packet.src_ip_port*, *packet.dst_ip_port*) → (*packet.sendkey*, *ABpathset*) to *pending_capable*
      return *ABpathset*.next()
    end if
  else if *packet* is `MP_JOIN` message **then**
    if (*packet.dst_ip_port*, *packet.src_ip_port*) in *pending_join* **then**
      sendkey, ABpathset ← *pending_join*[(*packet.src_ip_port*, *packet.dst_ip_port*)]
      recvkey, BApathset ← *mptcp_connections*[*sendkey*]
      delete key (*packet.dst_ip_port*, *packet.src_ip_port*) from *pending_join*
      return *BApathset*.next()
    else
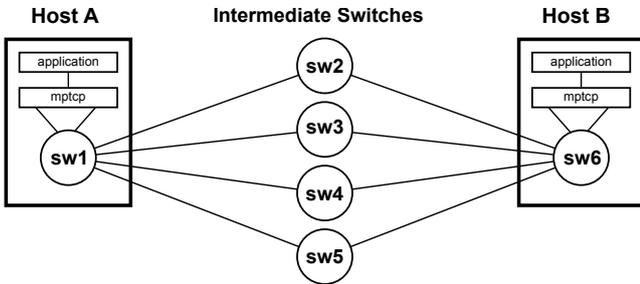   *sendkey, ABpathset* ← *mptcp_connections*[*packet.recvkey*]
      add (*packet.src_ip_port*, *packet.dst_ip_port*) → (*sendkey*, *ABpathset*) to *pending_join*
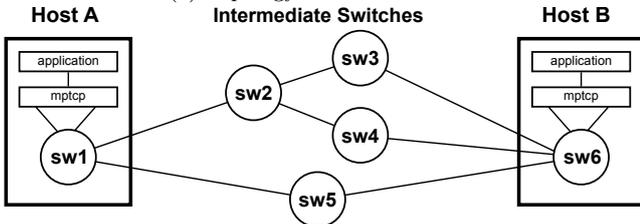   **return** *ABpathset*.next()
    end if
  end if
  return shortest path as *route* (otherwise)

(a) Topology 1 on local testbed

(b) Topology 2 on local testbed

Figure 4: Topology configuration of the local testbed. The switches are Open vSwitch installed on virtual machines. Each virtual machine is hosted on a separate physical host. Links between the switches are limited to 100 Mbps.
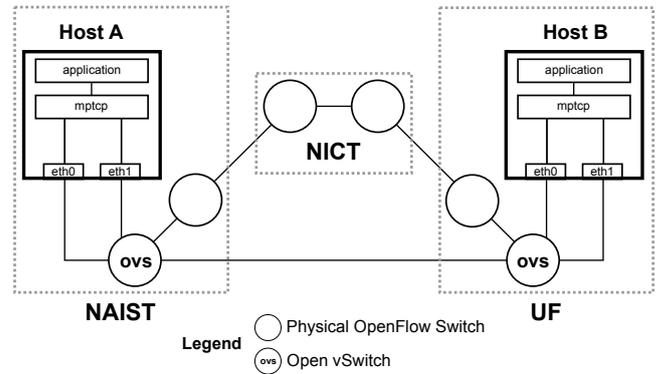

Figure 5: Testbed implementation in PRAGMA-ENT. The hosts are installed as virtual machines on the two sites.

search group [16]. Topology 1 (Figure 4a) has four isolated paths, while Topology 2 (Figure 4b) has paths partly sharing a link.

The wide-area testbed (Figure 5) experiment uses an existing collaborative wide-area software-defined network project known as the Pacific Rim Applications and Grid Middleware Assembly Experimental Network Testbed (PRAGMA-ENT) [1].

5.1 Evaluation in virtual local-area SDN

We implemented our local-area testbed on a VMware vSphere environment using six virtual machines. Each virtual machine, containing MPTCP installation and Open vSwitch [2], were deployed to different physical host machines. The GRE connections established between each virtual machine are manually limited to 100 Mbps to ensure that our virtual environment has a stable and clear maximum level of bandwidth, allowing easier verification of MPTCP and our controller.

We obtained MPTCP kernel and utilities from [19]. The kernel provides MPTCP functionality while the other MPTCP utilities allow us to disable MPTCP on select interfaces to make sure that the experimental traffic does not "spill" into the management subnet. This MPTCP kernel comes with multiple options that can be set through the `sysctl` variables, allowing us to customize the subflow creation options and numbers. Some options allow an arbitrary number of subflows to be created, regardless of the actual number of interfaces of the machine.

Testing POX S-T and smoc produced results as shown in Figure 6. Without a combination of a multipath router and MPTCP, only one path could be used at a time and the test run showed only approximately 95 Mbps of bandwidth, slightly below the theoretical limit was 100 Mbps, was used. However, when smoc and MPTCP are used together, after a few seconds of delay in the controller, the measured bandwidth was increased to greater than 100 Mbps, indicating that multipathing was successful with this combination. Test results using Topology 1, shown in Figure 6a, indicate that all four paths between Host A and Host B were used, allowing the maximum throughput to reach up to 400 Mbps. Test results using Topology 2, shown in Figure 6b, the measured throughput reached the maximum aggregate bandwidth of 200 Mbps as configured.

5.2 Evaluation in physical wide-area SDN

Two virtual machines were used for the evaluation in wide-area SDN. One was deployed in NAIST (Nara Institute of Science and Technology), Japan. Another was deployed in UF (University of Florida). Two paths were used in this experiment. For the first path, NAIST and UF are connected through two physical OpenFlow switches provided by NICT (National Institute of Information and Communications Technology), Japan. For the second path, a GRE link was directly established over the Internet between NAIST and UF.

smoc outperformed POX S-T from the start, then continued to increase its throughput the test as shown in Figure 6c. It is noteworthy that since TCP increases window size slowly in wide-area networks due to long round-trip time, more experiment time is needed for smoc to reach the maximum bandwidth possible in the network. Specific to the test in PRAGMA-ENT, we used 12 `iperf` threads (-P 12) because a larger number of threads would saturate wide-area networks more fully. This means `iperf` produces more consistent values toward the maximum available bandwidth.
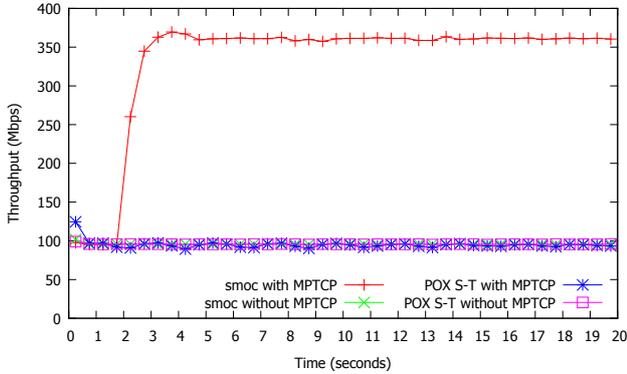
(a) Results of test on local testbed with Topology 1
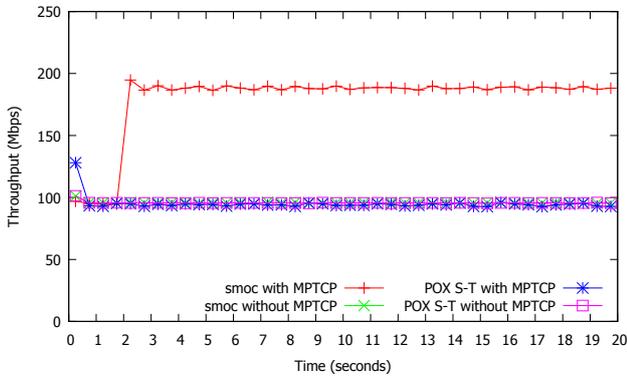
(b) Results of test on local testbed with Topology 2
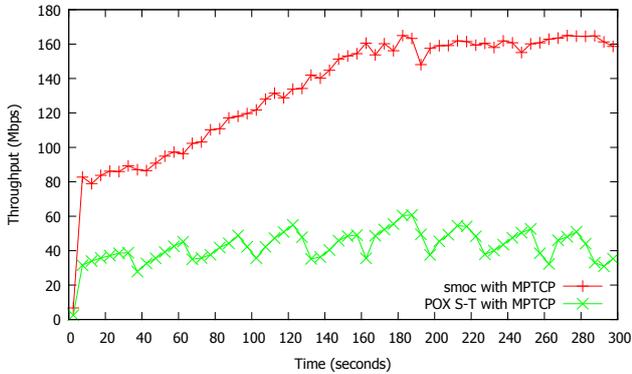
(c) Results of test on wide-area testbed with PRAGMA-ENT

Figure 6: Transient throughput between two hosts measured by `iperf` on different network topologies

6. DISCUSSION

In this section, we discuss the performance of our algorithm, issues with path installation delay in our controller, conditions of the test environment, and scalability of our solution.

6.1 Algorithm performance

We used a purely topological routing algorithm and generated path sets based on "minimum shared edges – minimum hops" basis. Whlie this is very simple to implement, only requiring a few calculations and no monitoring at all, the performance in real-world WANs may be debatable as the topology alone is not enough to effectively route flows through the best paths. One quick improvement that could be done to this controller is to use bandwidth-based routing by implementing a weighted graph and bandwidth monitoring to supply the graph with weights. Passive bandwidth monitoring was considered because we do not require the level of precision that could only be achieved by active monitoring. Any changes to the topology in real-time would be noticed by the management modules provided by POX and Overseer.

6.2 Path installation delay

We experienced 2-3 second delay in path installation as seen in Figure 6. This delay is caused by the path installation process by underlying POX modules. While this delay may be insignificant when a flow is long enough, it may impact short flows and cause scalability problems when handling a large number of flows. We need to find some way to improve the performance of the controller, such as shifting from the current reactive approach to a more proactive one which is more scalable [9] and has better performance. Some examples of proactive measures possible for smoc include anticipating and preinstalling secondary paths for additional subflows right after the first subflow is created, or storing a group of frequently-used path sets so they do not have to be calculated every time a new flow enters the network.

6.3 Test environment

While PRAGMA-ENT is a very good representation of WANs, the segment that we used consist of only two paths and a small number of switches. Even if these switches represented many more actual network elements, a more complex network could prove beneficial to the evaluation of our work. Additionally, testing with real-world applications would provide a realistic picture of our experiment. The high latency present in PRAGMA-ENT caused TCP flows to increase their window sizes more slowly. Shown in Figure 6c, it takes about 160 seconds for smoc's TCP flows to collectively increase their throughput to about 160 Mbps. This means spending more time with the test runs on high-latency networks should provide clearer results.

6.4 Scalability

Even though the multipath routing algorithm described in this work is adequate to efficiently route a set of subflows belonging to an MPTCP session through multiple paths, the smoc controller itself may have scalability problems. smoc is inherently centralized due to its use of OpenFlow. It has been studied that the number of flows that can be processed by the OpenFlow controller reduces at a quadratic rate with increasing number of switches, regardless of using proactive or reactive approach in routing [9]. As described earlier, reducing path installation delay by using proactive routing and reducing path set computation time can be some simple ways to mitigate (but not completely eliminate) the scalability problem by increasing the rate of flow processing. More involved methods include considering additional features in later OpenFlow versions, such as TCP flag matching introduced in version 1.5.0, to allow the switches to make more decisions on their own without invoking the controller. However, not all switches support the newest versions. We must consider the compatibility between the controller and the target environment carefully before upgrading the protocol version used in our controller.

Apart from the methods mentioned above, we could consider alternatives and modifications to OpenFlow, such as HyperFlow [25] and DevoFlow [7]. HyperFlow uses multiple synchronized OpenFlow controllers to communicate with each other and split the workload. Installing one HyperFlow controller per site may be more scalable than using a single OpenFlow controller for the entire network. On the other hand, DevoFlow, which is a significant modification to OpenFlow, aims to reduce workload on the controller by allowing additional actions on the switches, such as rule cloning and multipath support. These solutions would be able to improve scalability of many existing OpenFlow applications including smoc.

7. CONCLUSION

In this work we presented a simple multipath OpenFlow controller that routes MPTCP sessions by splitting them across multiple paths. Tests on both LAN and WAN SDN testbeds yielded positive results, indicating that our controller works as intended. No modifications to applications or host machines were made (only the kernel in the virtual machines), making our solution backwards-compatible with existing systems. We would find ways to improve its performance in future iterations of our work.

8. ACKNOWLEDGMENT

9. REFERENCES

[1] Scientic expeditions - pragma. http://www.pragma-grid.net/expeditions.php. Accessed: 2015-02-05.

[2] Open vSwitch. https://github.com/openvswitch/ovs, 2009.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing,*

page 54. IEEE Computer Society, 2005.

[4] W. Allcock, I. Foster, S. Tuecke, A. Chervenak, and C. Kesselman. Protocols and services for distributed data-intensive science. In *AIP Conference Proceedings*, pages 161–163. IOP INSTITUTE OF PHYSICS PUBLISHING LTD, 2000.

[5] S. Barré, O. Bonaventure, C. Raiciu, and M. Handley. Experimenting with multipath tcp. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 443–444, New York, NY, USA, 2010. ACM.

[6] B. Chihani and D. Collange. A survey on multipath transport protocols. http://xxx.tau.ac.il/pdf/1112.4742.pdf.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 254–265. ACM, 2011.

[8] Y. Dong, D. Wang, N. Pissinou, and J. Wang. Multi-path load balancing in transport layer. In *3rd EuroNGI Conference on Next Generation Internet Networks*, pages 135–142, May 2007.

[9] M. P. Fernandez. Comparing openflow controller paradigms scalability: Reactive and proactive. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 1009–1016. IEEE, 2013.

[10] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural guidelines for multipath TCP development. RFC 6182, RFC Editor, Mar. 2011.

[11] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP extensions for multipath operation with multiple addresses. RFC 6824, RFC Editor, Jan. 2013.

[12] B. Ford and J. Iyengar. Breaking up the transport logjam. In *ACM HotNets, October*, 2008.

[13] D. Gunter, R. Kettimuthu, E. Kissel, M. Swany, J. Yi, and J. Zurawski. Exploiting network parallelism for improving data transfer performance. In *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, pages 1600–1606. IEEE, 2012.

[14] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, Aug. 2008.

[15] C. Hopps. Analysis of an equal-cost multi-path algorithm. RFC 2992, RFC Editor, Nov. 2000.

[16] C. Huang, C. Nakasan, K. Ichikawa, and H. Iida. A multipath controller for accelerating GridFTP transfer over SDN. In *11th IEEE International Conference on e-Science*, 2015 (to appear).

[17] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 314–329, New York, NY, USA, 1988. ACM.

[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[19] C. Paasch, G. Detal, and D. Heidelberger. Multipath tcp. https://github.com/multipath-tcp, 2014.

[20] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[21] K. Rojviboonchai and A. Hitoshi. An evaluation of multi-path transmission control protocol (m/tcp) with robust acknowledgement schemes. *IEICE transactions on communications*, 87(9):2699–2707, 2004.

[22] T. Socolofsky and C. Kale. A TCP/IP tutorial. RFC 1180, RFC Editor, Jan. 1991.

[23] R. Stewart. Stream control transmission protocol. RFC 4960, Sept. 2007.

[24] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, RFC Editor, Nov. 2000.

[25] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, pages 3–3. USENIX Association, 2010.

[26] P. U-chupala, K. Ichikawa, H. Iida, N. Kessaraphong, and P. Uthayopas. Application-Oriented Bandwidth and Latency Aware Routing with OpenFlow Network. In *6th IEEE International Conference on Cloud Computing Technology and Science*, 2014.

[27] P. U-chupala, K. Ichikawa, P. Uthayopas, S. Date, and H. Abe. Designing of SDN-Assisted Bandwidth and Latency Aware Route Allocation. In *Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP)*, 2014.