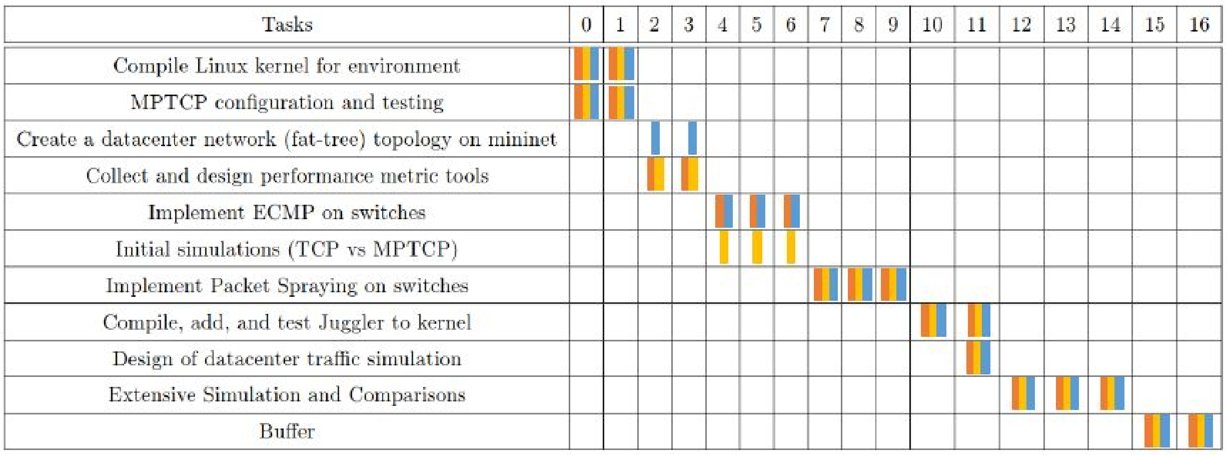
CNL 06 - Improving Datacenter Performance through Path Diversity

Charles Joshua Alba, Kyle Dominic Gomez, Rene Josiah Quinto



Tasks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Compile Linux kernel for environment	🟧🟨🟦	🟧🟨🟦															
MPTCP configuration and testing	🟧🟨🟦	🟧🟨🟦															
Create a datacenter network (fat-tree) topology on mininet			🟦	🟦													
Collect and design performance metric tools			🟧🟨	🟧🟨													
Implement ECMP on switches					🟧🟦	🟧🟦	🟧🟦										
Initial simulations (TCP vs MPTCP)					🟨	🟨	🟨										
Implement Packet Spraying on switches								🟧🟨🟦	🟧🟨🟦	🟧🟨🟦							
Compile, add, and test Juggler to kernel											🟧🟨🟦	🟧🟨🟦					
Design of datacenter traffic simulation												🟧🟨🟦					
Extensive Simulation and Comparisons													🟧🟨🟦	🟧🟨🟦	🟧🟨🟦		
Buffer																🟧🟨🟦	🟧🟨🟦

Project Schedule

Shown above is the division of tasks across the 16 week period. Each color represents a member of the group.

No major deviations from the gantt chart occurred. Some tasks were lengthened to accommodate unforeseen problems but other tasks were shortened to compensate.

In total, the project is ahead of schedule by a week - Packet spraying has been implemented on the switches used in the network. In fact, some tests using packet spraying have been obtained. The next task would be to compile a Linux kernel with both MPTCP and Juggler installed.

As of the moment, there are no foreseeable challenges that aren't reflected on the chart above. Deviations from the indicated schedule may be done through swapping around of tasks between members of the group to compensate for any unforeseen issues.

Halfway Point Deliverables

The following were the promised deliverables for the halfway point:

- Mininet script that generates a custom datacenter network (fat-tree) topology
- Initial experiment testbed (Mininet topology, MPTCP-enabled network)
- Performance metric scripts/tools (throughput, flow completion time)
- Software switches capable of ECMP
- MPTCP performance metrics (throughput, flow completion time) for a fat-tree topology
- TCP performance metrics (throughput, flow completion time)

All promised deliverables have been accomplished and have been reflected in the half-way documentation specifically as revisions and results in the Methodology part.

To expound, the scripts can be seen in the appendix of the Halfway Documentation.

Final Deliverables

- Software switches capable of Packet Spraying
- Juggler-enabled end-hosts in experiment testbed
- MPTCP performance metrics (throughput, flow completion time) for a fat-tree topology with switches running either ECMP or Packet Spraying, and end-hosts with or without Juggler enabled.

In addition to those listed above, a task for the group would be to create a statistical analysis of the data to confirm the hypothesis of the study.

Improving Data Center Network Performance through Path Diversity

Undergraduate Project Proposal

by

Charles Joshua Alba
2013-06878
B.S. Computer Engineering

Kyle Gomez
2013-25650
B.S. Computer Engineering

Rene Josiah Quinto
2013-14854
B.S. Computer Engineering

Adviser:

Professor Roel Ocampo
Professor Isabel Montes

University of the Philippines, Diliman
March 2018

Abstract

Improving Data Center Network Performance through Path Diversity

Datacenter networks allow the operation of Internet services by partitioning requests to multiple servers, and aggregating results back to form the response sent to the users. This requires servers within the datacenter to communicate efficiently and rapidly. Services, differentiated by their workload profiles, require a network that can guarantee high throughput, and low flow completion time. To achieve these, this paper focuses on Multipath TCP (MPTCP) as a transport protocol in the context of datacenters, by making better use of the network topology. In addition, changes in the packet forwarding protocol within network switches and the Generic Offload Receiver (GRO) layer within network servers will be employed to make up for MPTCP's undesired behavior within datacenters such as network link hotspots. With this, we hypothesize an increase in throughput and decrease in the flow completion time. We will test this through a comprehensive analysis of different testbeds with varying parameters. Initial tests in a smaller network topology show that throughput between two end-hosts in a network increases which also implies that network utilization also increases. It was also observed that the overhead of establishing multiple subflows in an MPTCP connection using the TCP handshake penalizes short flows.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To keep up with increasing demand for online services, requests and operations are usually serviced by partitioning tasks into multiple servers within a datacenter. These servers are then arranged in a special topology to ensure quick intercommunication between each other, regardless of packet stream length or size. As a consequence, servers have multiple paths between each other. There lies a promising performance benefit by taking advantage of this path diversity, in which the traditional transport protocol, TCP, cannot provide.

1.1 Servicing Increasing Demand with Datacenter Networks

Several companies and institutions (Google, Amazon, and Microsoft, to name a few) provide a wide variety of online services such as video streaming, online gaming, VoIP, file sharing, and simple web searching. These online services usually differ in their workload profiles [1, 2]—faster connections result to better performance for file sharing services, while connection stability is the main concern for video streaming services.

The demand for online services has been increasing steadily due to their popularity. Moreover, the competitive market have pushed for innovations to improve services, resulting in increased computational load [3]. To keep up with demand that increases in both quantity and complexity, companies need to expand and upgrade their resources, increasing costs [4, 5]. One way to meet this demand is to distribute service requests to multiple servers, then aggregating responses to form the final result (also known as the partition/aggregate design pattern) [6]. Infrastructures built with this demand distribution property are called datacenter networks (DCNs).

Apart from meeting the increasing demand, this provides several benefits. Firstly, by having redundant servers, services can be maintained even during partial network maintenance

[3]. Secondly, an organized and redundant network infrastructure eases management [3]. Finally, distributing requests to different servers increases server utilization, proving to be more cost-effective and scalable [7].

1.2 Connection-Rich Nature of Datacenter Network Topologies

Partitioned requests are distributed among servers within a datacenter (end-hosts) through packet streams (flows). Since majority of the flows stay within a datacenter [8], the network topology must guarantee a speedy and reliable connection between local end-hosts. These properties can be characterized with sufficient interconnection bandwidth [9, 10] and fault-tolerance [11].

Requests must be serviced quickly and aggregated back to the user, which means that the server response delays also add up and affect the performance of the service [6]. End-hosts also need to keep their internal data structures fresh [6] through frequent large transfers of data. Therefore, datacenters must also guarantee low latency for short flows, and high throughput to service large flows [6, 2].

Current datacenter network topologies guarantee the previous characteristics through different approaches. Switch-centric topologies [10, 12] rely solely on switches and routers to forward traffic across the network, which ensures speedy delivery due to specialized hardware at the expense of scalability. Server-centric topologies [13, 14] allow servers to forward packets as well, however software routing may be slow and can affect host-based applications [15].

Topologies can be also designed to be modular [14] or hierarchical [12] to increase network capacity through scalability [9]. This introduces multiple redundant links and paths between servers within the network, establishing a connection-rich nature [2, 16].

1.3 Better Network Utilization through Multiple Paths

Host-to-host connectivity within a datacenter was originally facilitated using the Transmission Control Protocol (TCP) [6, 17]. TCP ensures robustness in the delivery of packets between end-hosts in the presence of communication unreliability [18].

TCP connections make use of a single network path between end-hosts [19]. Since it is typical for two end-hosts in a datacenter network to have multiple paths, TCP is unable to fully utilize the available paths between end hosts [20], and may not be able to achieve optimal network utilization.

To make use of the available resources in a path-diverse network, Multipath TCP (MPTCP) was proposed as an experimental protocol by the IETF. MPTCP uses parallel TCP connections [21, 22], which increases throughput and resiliency of the network [20].

1.4 Outline of the Document

The rest of the document is presented as follows. The next chapter will discuss different improvements suggested for MPTCP based on different metrics such as reliability, and a global view of the network. Additionally, some issues in unmodified MPTCP are also discussed and studied in depth. Chapter 3 discusses the problem statement, objectives, and scope and limitations of the project. Section 4 and 5 discuss the steps taken to create the project.

Chapter 2

Review of Related Work

While MPTCP promises performance benefits such as increased throughput between two hosts, it suffers from problems which hinder or negate its advantages. The nature of datacenter network traffic and topology presents two potential setbacks for MPTCP. First, short packet streams (short flows) might experience a higher latency due to MPTCP's connection establishment mechanism. Second, large packet streams (large flows) might suffer from lower throughput due to MPTCP's congestion and subflow mechanism. In addition to the presented solutions to these problems, a closer look is given to switch-based packet spraying, combined with host-based reordering resiliency, as a potential improvement over MPTCP.

2.1 Connection Establishment of MPTCP Penalizes Short Flows

MPTCP, being designed as an extension for TCP [21], inherits its congestion control mechanisms. One of which is Slow Start [23], where each subflow starts sending a limited amount of packets, then exponentially increases its sending rate until a link capacity/threshold is reached. Exceeding link capacity can result to packet loss due to switch overflows, making the receiver fail to acknowledge the packet. This subsequent action will cause the sender to retransmit the packet due to its internal retransmission timeout, or due to receiving three duplicate acknowledgements (TCP Fast Retransmit) for a previously sent packet, in addition to reducing the sending rate of the sender.

Since majority of the flows on a datacenter are short flows [24] and are of a bursty nature [6], MPTCP would cause these flows to stay in the Slow Start phase [25], where the sending rate is initially limited, then is increased drastically. Therefore, when short flows lose packets due to congestion, retransmission is necessary, and this increases the flow completion time [26].

In addition, MPTCP also inherits TCP's handshake protocol, in order to create and establish subflows between end-hosts. Since MPTCP cannot discern between short and long flows, it cannot decide as to how many subflows should be established to optimally send packets across the network. Therefore, a mismatch in the number of subflows, specifically for short flows would further increase the flow completion time due to the extra packets required to establish a connection.

To minimize the flow completion time for short flows, connections can start by spraying packets over multiple paths. Connections then can switch back to MPTCP upon reaching a certain amount of packets [26], to make use of MPTCP's benefits over long flows such as increased throughput and better congestion control. However, the introduction of packet spraying in addition to using MPTCP would open this specific implementation to both problems associated with each.

2.2 Decreased Throughput due to Redundant Retransmissions

MPTCP uses Equal-Cost Multipath (ECMP), a switch-based algorithm, to forward packets through the network [17]. ECMP bases its forwarding on a 5-tuple (source IP, destination IP, source port, destination port, protocol number) [27] which is hashed and used to forward packets to paths. This means that all data belonging to one subflow will strictly follow one network path. Selected network paths by ECMP may traverse links that have already been chosen by another subflow, which will effectively decrease the maximum available capacity of the link. Areas in the network that are close to exceeding link capacity are called hotspots. Colliding subflows may exceed the link capacity for any given link in a datacenter network, resulting in full switch queues. Full switch queues will then lead to dropped packets, which in turn, decrease mean network utilization [9].

Duplicating packets across multiple paths (bicasting/multicasting) in an MPTCP connection can be used to mitigate the decrease in throughput due to dropped packets [28]. In full bicasting, each connection transmits a copy of all packets sent across multiple subflows, while selective bicasting only duplicates packets that are sent as a retransmission of lost packets.

However, the increase in throughput for individual subflows brought about in both full and selective bicasting may not be optimal in a datacenter. This is, in part, attributed to the duplicating nature of bicasting wherein redundant data is sent through the network's links and consumes resources for other subflows. In addition to redundant data packets, control packets such as the ACK packets in TCP add to network traffic. This added traffic to the network can be perceived as a loss in throughput.

2.3 Diffusing Network Hotspots through MPTCP with Packet Spraying

As previously mentioned, network hotspots create areas of congestion in datacenter networks, throttling the throughput of the connections that pass through these hotspots. One way to avoid this is to use packet spraying [29]. Packet spraying spreads a sequence of packets through different candidate ports instead of forwarding packets of a single flow through a chosen port according to its hashed flow identifier as seen in ECMP. This scheme prevents the collision of multiple paths onto specific links (hash collision), and spreads the packets in the hopes of evenly distributing the load through all relevant links of the network.

However, distributing packets of a flow into different paths can lead to out-of-order packets if the paths have significantly different end-to-end delays [29]. End-hosts receiving packets that do not arrive in the order that they were sent may falsely perceive that the network is lossy and request for retransmissions [18]. The sending end-host will receive these retransmission requests and will falsely perceive that there is congestion in the network, cutting the congestion window in half which will then result in the reduction of throughput.

One way to avoid false congestion due to packet reordering is to adjust the time threshold of the end-hosts before they request for retransmission [26], effectively making end-hosts more tolerant to delays in the network. However, an improper configuration (mismatch) of this threshold will result in an increased delay before sending retransmission requests in actual congestion experienced in the network, greatly decreasing the flow completion time and penalizing short flows.

To increase the resiliency of the end-hosts against out-of-order packets while avoiding a threshold mismatch, one solution would be to increase the buffer size of the Generic Receive Offload (GRO) layer of the end-hosts [30]. The GRO layer works underneath the transport layer, and is responsible for bundling packets to decrease computational overhead versus individual per-packet processing. Increasing the buffer size enables it to accept packets that arrive earlier than those preceding it thereby increasing its resilience towards out-of-order packets. This is based on the assumption that, in a datacenter, delay differences between paths are significantly less than in non-datacenter networks.

2.4 Drawbacks Caused by MPTCP with a Global View

With Software Defined Networking (SDN), a global view of the network state can be provided to a central controller [31, 32, 33]. This enables network protocols to make use of global network information, such as link utilization and end-to-end delays [34, 35], to create more informed decisions. SDN approaches have introduced several benefits to MPTCP, such as increased throughput and network resiliency, by controlling the number of subflows and scheduling the routes of different connections to avoid both network hotspots and out-of-order packets [34].

These benefits, however, comes at the price of controller overhead [34]. SDN controllers need to communicate regularly with the network switches in order to issue forwarding rules to adapt with the situation, causing delays. Since most congestion events that happen in datacenter networks are because of traffic bursts that only lasts for a few microseconds (microbursts) [36], controller overhead becomes significant, and so SDN implementations cannot respond fast enough. Due to controller overhead, SDN implementations can also increase the flow completion time of connections, penalizing short flows. Moreover, SDN implementations generally do not scale with a large number of connections as the controller overhead also increases with the number of active connections [34].

Chapter 3

Problem Statement and Objectives

3.1 Problem Statement

Common topologies in datacenter networks exhibit symmetry and path diversity [2]. This ensures that multiple paths of equal costs exist between two end-hosts [29]. In addition, an ideal datacenter network must guarantee high throughput for large flows, and low latency for short flows [6, 2].

Multipath TCP (MPTCP), an experimental protocol by the IETF, is a TCP extension that uses multiple paths over many connections [21]. By using multiple paths simultaneously, MPTCP aims to increase network utilization, specifically throughput, and increase ”resilience of connectivity” [22]. This is done by employing Equal Cost Multi-Path (ECMP) [17], a switch-based forwarding algorithm. ECMP hashes subflows into separate paths using the packet's header information. However, hotspots in the network may occur when flows are hashed to paths with one or more links overlapping with each other [9], usually exceeding the link capacity. Because of this, the network experiences a drop in network utilization due to sender backoff [29].

Random packet spraying, an alternative to ECMP, can be used to resolve hotspots as it allows for a greater granularity for load balancing [2]. However, this can result in reordered packets at the receiver, triggering false retransmissions upon receipt of three duplicated packet acknowledgements [20], in addition to drastically reducing sender throughput [37]. This can be minimized by dynamically changing the retransmission threshold. However, a positive mismatch on the threshold may mean a slower response time for actual packet drops [37].

Minimizing retransmission due to out-of-order packets can be done by supplementing the function of the Generic Receive Offload (GRO) layer to fix the order of packets [1] in addition to merging packets together. This not only reduces computational overhead compared to per-packet processing, but makes the end-hosts more tolerant to packet reordering, and thus reduces retransmissions. However, reordered packets must arrive within a short delay of each other since the switch queues are limited and timeouts are smaller.

Also, to maintain backwards compatibility with TCP, MPTCP also suffers from the complexity in connection establishment [29] and slow start [23], which in turn penalizes short flows through higher flow completion times. To maintain lower flow completion times and lower latency for short flows, a specified amount of packets are sprayed through multiple paths initially before switching to MPTCP [26]. However, this means that it inherits problems from both implementations altogether.

Better network utilization can be also achieved using Software-Defined Networking [34, 35] with MPTCP. With a global view of the network, it can better utilize path diversity, have a better gauge on the number of subflows per connection, and ideally minimize the receipt of out-of-order packets. This solution benefits large data flows, but due to the added controller overhead, it may penalize short flows. In addition, it may also have some scalability issues [35].

To combat delay and lower throughput caused by multiple timeouts on wireless networks, retransmission redundancy was implemented using full and partial bicasting over MPTCP [28]. This may not necessarily be effective in the datacenter setup as redundant packets may cause more false retransmissions.

We hypothesize that an increase in overall throughput and network utilization in MPTCP can be achieved through implementing packet spraying in the network instead of ECMP. In addition, creating reorder-resilient end hosts will combat the negative effects of packet spraying such as decreased throughput. By comparing the results we see from different experiments, we strengthen the basis for considering a reorder-resilient network for future datacenter networks, as well as potentially contribute to the growth of MPTCP as an experimental protocol.

3.2 Objectives

The objectives of this work are as follows: First is to experimentally prove that MPTCP benefits large flows, but penalizes short flows. Next, understand and prove that network hotspots occur due to ECMP-based switches. Lastly, observe and analyze the effects of packet spraying switches, as well as reorder-resilient hosts, to minimize network hotspots, and in turn benefit both short and large flows.

3.3 Scope and Limitation

The project will focus on datacenter networks, and assume ideal working conditions. More specifically, this project does not consider the possibility of switch failures, host failures, link damages that could cause degradation of performance or even disconnections. Among datacenter topologies, only fat-tree topologies and possible variants will be considered.

While the nature and topology of datacenter networks are highly distinct from the vast majority of the Internet, or wide area networks (WANs), the results and observations presented in this paper may potentially apply to WANs as well.

As this paper relies heavily on experiments done through network simulations, this project cannot guarantee the realization of actual or realistic datacenter network traffic, but tests will be made to mimic certain datacenter network conditions such as worst-case scenarios.

Chapter 4

Methodology

Considering that Multipath TCP (MPTCP) penalizes short flows due to unnecessary overhead due to connection complexity, as well as creates hotspots due to ECMP routing, the metrics to be observed should be flow completion time, and end-host throughput, for short and large flows respectively.

To characterize the behavior of the routing protocol, the end-hosts, and the network, we execute a number of tests varying different parameters related to each. The control group would have end-hosts running TCP without any kernel or network stack modifications.

4.1 Behavioral Considerations

MPTCP is available as an ns-3 model [38] or a kernel patch [39]. Juggler [1], which modifies the GRO function to fix packet ordering, is only readily available as a kernel patch. Preference is then given to kernel patches, assuming that there are little to no conflicts between both MPTCP and Juggler.

Switches, on the other hand, can be described with its normal forwarding algorithm, which stores only one next-hop for all destinations in its routing table. However, the switches must also be capable of Equal Cost Multi-Path (ECMP), and Packet Spraying (PS), which require a routing table storing multiple next-hops. In consideration, the switch must be capable of the three previously described algorithms, with preference to easier switch customization.

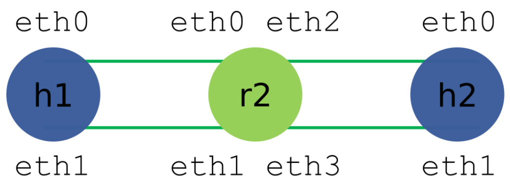


Figure 4.1: Mininet topology for MPTCP tests. Links are of 10 Mbps speed. Topology copied from [40]

4.2 Preliminary Work

To further understand the current implementation of MPTCP on servers, smaller tests were used and characterized. As the protocol is still in active development, a lot of parameters are available to play around it, resulting in different performances. Future work might focus on automatically setting these parameters to optimally cater to the network activity, completely transparent to the user.

4.3 Setting Up the Test Environment

Initially, three separate testbeds were created using `Ubuntu 16.04.3` with a Linux kernel version of `4.10.0-28-generic`.

The preliminary testing topology was based from an MPTCP laboratory experiment of an SDN class available online [40]. An overview of the topology can be found in Figure 4.1. To make things easier, router `r2` is actually a host functioning as a router[1] having it preconfigured to forward packets. This should be replaced with a configurable router in the actual experimentation.

4.4 Managing Multipath TCP Subflows

Multipath TCP handles the discovery, setup, and termination of subflows through the heuristics of a path manager [18]. We consider three of the four available path managers MPTCP provides [41].

First, the default path manager does not initiate nor announce different IP addresses, but will still accept subflows from end-hosts that do. Next, the full mesh path manager creates flows using all combinations of interfaces (device within end-host to connect to links [42]) of both

[1]From this point onwards, we use the terms switch and router interchangeably to mean devices that do layer 2 forwarding and layer 3 routing.

Routing Protocol	Path Manager	Initiates subflow creation	Number of used IP address pairs	Number of used port number pairs	Number of subflows
TCP	None	N/A	1	1	1
MPTCP	default	No	As sender: 1 As receiver: based on sender	As sender: 1 As receiver: based on sender	As sender: 1 As receiver: based on sender
MPTCP	fullmesh	Yes	All, combination	Configurable, default is 1	`No. of source IP-port pairs * No. of destination IP-port pairs`
MPTCP	ndiffports	Yes	1	Corresponds to number of subflows	Configurable, default is 2

Table 4.1: Comparison of MPTCP path managers and TCP.

Routing Protocol/Path Manager	Sender throughput (in Mbits/sec)
TCP	9.78
Default MPTCP	9.43
Ndiffports MPTCP	9.46
Full mesh MPTCP	19.1

Table 4.2: Comparison of sender throughput between MPTCP path managers and TCP. The topology used was described in Figure 4.1. Throughput values were taken using command line tool *iperf*.

end-hosts (i.e. assuming all interfaces correspond to unique IP addresses, two end-hosts with three interfaces each will have 9 subflows). Finally, the n-different-ports path manager allows the control over the number of subflows in a connection through the use of different ports. The fourth path manager called binder, isn't considered as it was designed for mobility of devices, a trait not present in datacenter networks.

In summary, we see the comparisons of MPTCP path managers in Table 4.1. To serve as a control group, TCP was characterized as well.

Considering the topology described in Figure 4.1, assuming all links are rated at 10 Mbps (i.e., the minimum supported transmission rate of both devices connected to the link is 10 Mbps), we expect to see TCP, default MPTCP, and ndiffports MPTCP to have a sender rate of 10 Mbps, whereas Full Mesh MPTCP can have a sender rate of up to 20 Mbps. This is because the first three would only use 1 IP address pair, and thus is limited to 1 subflow. But, the Full Mesh MPTCP establishes four subflows, because both hosts have two available interfaces, fully using all possible links. This was validated experimentally and the results are shown in Table 4.2.

Routing protocol/Path Manager	FCT (Mean, in ms)	FCT (Standard Deviation, in ms; Relative Standard Deviation)
TCP	7.86	1.53 (19.51%)
default MPTCP	8.39	1.10 (13.11%)
full mesh MPTCP	10.5	3.86 (36.70%)

Table 4.3: Mean, standard deviation, and relative standard deviation of flow completion time for different MPTCP path managers and TCP.

However, using more subflows may sometimes come at a cost. For short flows, opening multiple subflows may not be necessary, as the packets may have already been sent even before a new subflow has been established. To prove that MPTCP does indeed introduce some overhead for short flows, the flow completion time was measured between TCP, Default MPTCP, and Full mesh MPTCP.

This experiment once again uses the topology in Figure 4.1. The left host (`h1`) will request for the web page from the right host (`h2`), which has a web server running. The web page to be fetched is a default directory index page, and because the web server sends a stream of small packets to complete the request, it counts as a short flow. The tests were ran 10 times with TCP, default MPTCP, and full mesh MPTCP, and the flow completion time (FCT) were noted. We defined the flow completion time as the time difference between the first `SYN` packet up to the last `ACK` packet for the last `FIN` packet.

A summary of results are shown on Table 4.3. Here, we see that TCP has the fastest flow completion time. As default MPTCP works much like TCP, but has an added establishment overhead over TCP, it has a slower flow completion time. Lastly, since full mesh MPTCP opens multiple subflows while transmitting data, packets for connection establishment of other subflows compete for the same available links, resulting in the slowest flow completion time. In addition, since flows are terminated like TCP connections, extra packets are sent to terminate all open subflows before finishing.

As the topology considered for the previous experiments is relatively simpler compared to actual datacenter topologies, we hypothesize that the observed behavior will still hold true, and the effects are amplified to a certain extent.

4.5 Testbed Setup

Experiments and tests were ran on a SuperMicro bare metal server with the specifications listed below.

Mininet will be used to simulate traffic between virtualized hosts. Mininet uses the

Technical Specifications	Value
Processor	Intel(R) Core i7-3612QE
Processor Speed	2.10 GHz
Number of Cores	4
RAM	16 GB
Operating System	`Ubuntu 16.04.3 LTS`
Linux Kernel	`Linux version 4.9.60.mptcp`

Table 4.4: Testbed Technical Specifications.

configuration of the host operating system to simulate the hosts, which makes it easier to change the hosts' transport layer protocol.

MPTCP can be installed through compiling a modified kernel, or installation through a public apt-repository [43]. For the initial tests, the installation through the apt-repository is enough. However, when reorder-resiliency in the hosts will be necessary, a modified kernel shall be used.

In addition, `tshark`, and `git` were installed as helper tools.

4.5.1 Configuring MPTCP

To easily control the behavior of the host's transport protocol and MPTCP [41], the `sysctl` feature is used. The transport protocol can be changed through `net.mptcp.mptcp_enabled`. Furthermore, we can control the MPTCP path manager using `net.mptcp.mptcp_path_manager`. The path manager can be set to `default`, `ndiffports`, `fullmesh`, or `binder`, as discussed previously.

4.5.2 Setting up the Network Topology

For this experiment, the fat tree topology was chosen, a common and scalable data center network topology. Like all DCN topologies, the fat tree topology provides several equal cost paths between any two end-hosts. Moreover, because of the symmetry of the topology even with scaling, a mininet topology can be easily set up with unique addressing [10]. Mininet was used for the simulation of the nodes, and Python was used to construct the topology (See A.1) A $K = 4$ fat tree topology can be seen in 4.2, complete with the conventions used.

The following are the definitions and conventions followed in generating the topology.

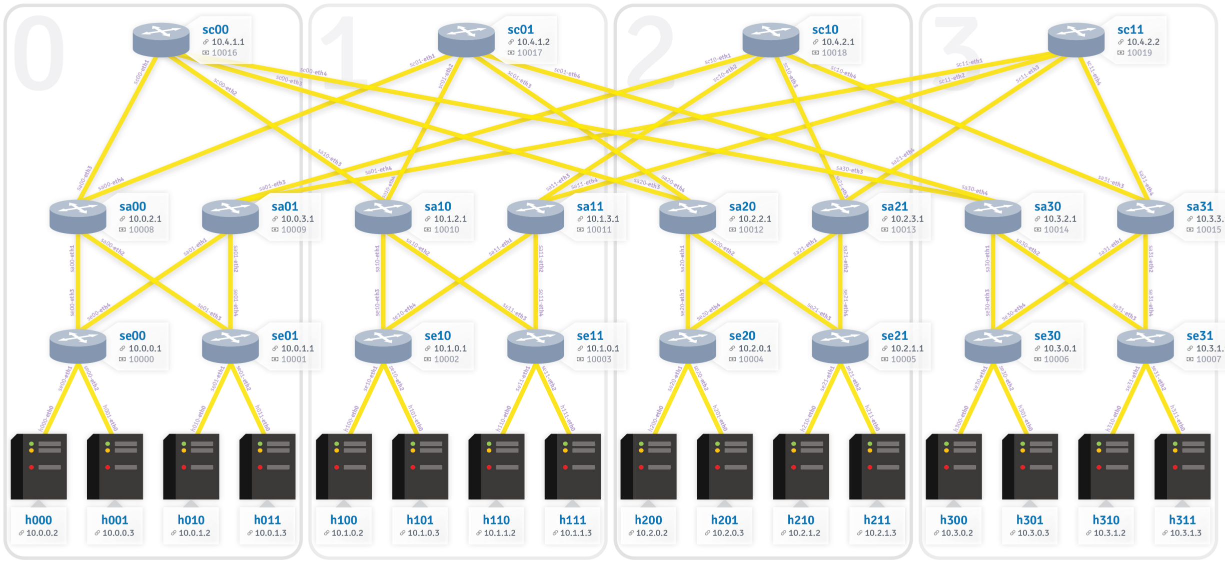


Figure 4.2: Fat tree (`k = 4`) topology in Mininet, including naming and addressing conventions.

4.5.2.1 Standard Topology Structure

For a given scaling parameter K, the topology is composed of K pods. Each pod is composed of $\frac{K}{2}$ aggregate routers and $\frac{K}{2}$ edge routers, with each aggregate router connected to all edge routers, and vice versa. Each edge router is connected to $\frac{K}{2}$ hosts, for a total of $(\frac{K}{2})^2$ hosts per pod. There are a total of $(\frac{K}{2})^2$ core routers, each connected to one aggregate router per pod.

4.5.2.2 Indexes and Names

The following numbering and naming conventions are used.

- The pods are numbered for 0 to $K-1$.
- Edge routers are named `se<pod><i>`, where `<pod>` is the pod id, and `<i>` is the edge id within the pod, ranging from 0 to $\frac{K}{2}-1$.
- Aggregate routers are named `sa<pod><i>`, where `<pod>` is the pod id, and `<i>` is the aggregate id within the pod, ranging from 0 to $\frac{K}{2}-1$.
- Core routers are named `sc<i><j>`, where `<i>` and `<j>` ranges from 0 to $\frac{K}{2}-1$. `<j>` is an identifier for all cores of the same `<i>`, and `<i>` determines which aggregate core it connects to for each pod. The significance of `<i>` and `<j>` are explained in the Links subsection.

- Hosts are named `h<pod><i><j>`, where `<pod>` is the pod id, `<i>` is the edge id of the edge router it is connected to, and `<j>` is the host id for all hosts connected to the i'th edge router.

4.5.2.3 IP Addresses

From the node names, we can map it directly to unique IP addresses [10].

- For hosts with names `h<pod><i><j>`, the IP address is `10.<pod>.<i>.<j+2>`.
- For edge routers with names `se<pod><i>`, the IP address is `10.<pod>.<i>.1`.
- For aggregate routers with names `sa<pod><i>`, the IP address is `10.<pod>.<i+(K/2)>.1`.
- For core routers with names `sc<i><j>`, the IP address is `10.<K>.<i+1>.<j+1>`.

4.5.2.4 Links

The following describes more precisely the connections between nodes[10].

- An edge router `se<POD><I>` is connected to hosts `h<POD><I><j>` for all $0 \leq j \leq \frac{K}{2} - 1$.
- All aggregate routers are connected to edge routers in the same pod, and vice versa. More precisely, an aggregate router `sa<POD><I>` is connected to edge routers `se<POD><j>` for all $0 \leq j \leq \frac{K}{2} - 1$.
- A core router `sc<I><J>` is connected to aggregate routers `sa<pod><I>` for $0 \leq pod \leq K - 1$.

4.5.2.5 Port Assignment

Each link connected to a router is assigned a port for that router. For the following descriptions, the ports are numbered from 0 to $K - 1$ (though in reality, it is usually numbered from 1 to K). The following describes the assignment for the p'th port for each type of router.

- For an edge router `se<POD><I>`, the first $\frac{K}{2}$ ports is assigned to host `h<POD><I><p>`, and the last $\frac{K}{2}$ ports is assigned to aggregate router `sa<POD><p-(K/2)>`.
- For an aggregate router `sa<POD><I>`, the first $\frac{K}{2}$ ports is assigned to edge router `se<POD><p>`, and the last $\frac{K}{2}$ ports is assigned to core router `sc<I><p-(K/2)>`.
- For a core router `sc<I><J>`, the ports are assigned to aggregate router `sa<p><I>`.

The interfaces are similarly assigned, but from `eth1` to `eth<K>` instead of 0 to $K - 1$.

4.5.2.6 Thrift Port

Thrift ports are assigned for each router, and can be used to communicate and debug with the router. The following describes how the thrift ports are assigned.

- The first thrift port is 10000, and is assigned to `se00`. Succeeding ports are increasing in increments of 1.
- The first $K * \frac{K}{2}$ thrift ports are assigned to edge routers `se00, ..., se0<(K/2)-1>, se10, ..., se<K-1><(K/2)-1>`.
- The next $K * \frac{K}{2}$ thrift ports are assigned to aggregate routers `sa00, ..., sa0<(K/2)-1>, sa10, ..., sa<K-1><(K/2)-1>`.
- The next (and last) $(\frac{K}{2})^2$ thrift ports are assigned to core routers `sc00, ..., sc0<(K/2)-1>, sc10, ..., sc<(K/2)-1><(K/2)-1>`.

4.5.3 Router Behavior

Apart from setting up the topology, we also need to define the behavior of the switches, i.e. given the destination and other metadata found in the packet header, the router behavior dictates which port the packet is forwarded. For this purpose, we used P4, a programming language that specifies how switches and routers process packets [44].

The P4 `behavioral-model` repository [45] was forked to get its target executables (e.g., `simple_router`) and modified the source to extend their functionalities (See Appendix A.2). Each behavior is defined by tables and actions coded in C++, and compiled to a JSON using the `p4c-bm` tool [46]. The JSON is fed to the target executable, and the tables are filled with entries during initialization.

4.5.3.1 Downstream Packets

For the following discussions, packets that are forwarded towards the core (host to edge, edge to aggregate, or aggregate to core) are considered to be upstream, and otherwise downstream. Since downstream packets in the fat tree topology have a unique path towards their destination, each router has only a single correct port to forward to per destination in the downstream path. Thus, for any router behavior, the packet's destination IP Address is matched with a longest prefix match to check if the packet is headed downstream, and if so forward it to the appropriate port. All bits are matched for edge routers, the first 24 bits for aggregate routers, and the first 16 bits for core routers.

For any router forwarding upstream, each of the $\frac{K}{2}$ upstream ports to choose from is a valid path. Thus, different router behaviors may differ in how packets are forwarded upstream. For this experiment, three router behaviors have been implemented - `Static`, `ECMP`, and `PS` (for packet spraying), with `Static` serving as the control. The packet forwarding scheme for each behavior is discussed in detail in the next subsections.

4.5.3.2 Static

For Static behavior, every destination is assigned to a random port with equal probability during initialization, and will not change during runtime (thus the name `Static`). Thus, all upstream paths are matched in the same table as downstream packets, but using all 32 bits. The P4 code for `Static` behavior can be found in Appendix A.3, and the `Static` table entry generation script can be found in Appendix A.4.

4.5.3.3 ECMP

For `ECMP` behavior, the flow metadata is hashed to determine the forwarding port. The flow metadata consists of the `source IP Address`, `source port`, `destination IP address`, `destination port`, and `protocol number`. The hash used is the CRC16 hash function, and outputs a 3-bit integer.

A 2-parameter matching is done on the table, the first parameter being the `destination IP Address`, and the second parameter being the output of the hash function. The first parameter is matched using a longest prefix match, and the second parameter is matched exactly. For more details, see the P4 code for `ECMP` behavior in Appendix A.5.

For each downstream port, 8 entries are inserted into the table, one for each 3-bit integer for the second parameter. Another 8 entries are inserted for upstream packets, one for each 3-bit integer for the second parameter, and with `0.0.0.0/0` as the first parameter (match anything). With this setup, downstream ports are matched with the appropriate entry regardless of the hash value, and those that aren't matched downstream are matched according to the hash value. Each 3-bit integer is then assigned to an upstream port in a random but fair manner i.e. all upstream ports have the same number of hash values assigned to it, but randomly shuffled. Since we can have at most 8 hash values, K is limited to at most 16, though this can be increased when necessary. For more details, see the `ECMP` table entry generation script in Appendix A.6.

4.5.3.4 PS

For `PS` behavior, the chosen upstream forwarding port is chosen uniformly at random, also known as Random Packet Spraying. According to the P4 Specifications Document [47], uniform random assignment on a field is a primitive operation, however it was found that the `simple_router` target in the P4 `behavioral-model` repository [45] did not support said operation. Thus, the source code was modified to extend support, and the target was recompiled. The modifications can be seen in Appendix A.2.

The `destination IP Address` is matched with a longest prefix match to forward downstream packets, similar to Static behavior. In addition, upstream packets are matched with the entry `0.0.0.0/0` (match all) and corresponds to a special action that assigns a uniformly random upstream port as the forwarding port. The P4 code for `PS` behavior can be found in Appendix A.7, and the PS table entry generation script can be found in Appendix A.8.

4.6 Test Sets and their Expected Behavior

To review, the goals of the tests are to measure the performance of the varying network configurations previously discussed. In particular, the study will focus on throughput and flow completion time.

For the purposes of this section, a topology with $K = 4$ will be considered. This means that there are 16 hosts, 16 switches in the aggregate, 4 in the core.

Initially, the tests were run on a silent network to establish a baseline for the tests. In future tests, a rudimentary form of traffic will be added to the network to simulate a busy datacenter.

4.6.1 Flow Types and Definitions

Three flow types are to be used and tested for these experiments: `query`, `short`, and `long` flows [6]. Considering the simulated link bandwidth (pegged at `20 Mbps`) and common flow definitions, the following sizes were used when referring to the different flows.

Since the ideal FCT is computed by simply dividing the flow size by the simulated link bandwidth without considering the additional overhead (connection handshake, termination, retransmissions, protocol-specific implementations), the measured FCT is expected to be larger.

Flow	Examples	Size	Ideal FCT
query	Quick, bursty connections	10 kiB	˜0.004096 s
short	Web page requests	500 kiB	˜0.2048 s
long	Streaming, VoIP, file hosting and transfer	25 MiB	˜10.485 s

Table 4.5: Flow definitions and examples, corresponding sizes, and ideal FCTs (based on a 20 Mb/s link bandwidth).

Router Behavior	Juggler Modification	Transport Layer Protocol	MPTCP Path Manager	N (for `ndiffports`)	Maximum Number of Usable Paths
`static`	no	TCP	-	-	1
`ecmp`	no	MPTCP	`fullmesh`	-	1
`ecmp`	no	MPTCP	`ndiffports`	4	1
`packet spraying`	no	TCP	-	-	4
`packet spraying`	no	MPTCP	`fullmesh`	-	4
`packet spraying`	no	MPTCP	`ndiffports`	4	4
`packet spraying`	yes	TCP	-	-	4
`packet spraying`	yes	MPTCP	`fullmesh`	-	4
`packet spraying`	yes	MPTCP	`ndiffports`	4	4

Table 4.6: Test sets used, their configurations, and the maximum number of usable paths between two hosts.

4.6.2 Test Sets

In line with the objectives of the study, nine test sets are to be used, with each test set ran against the three flow types as discussed previously.

Test sets configured to use `ndiffports` as MPTCP's path manager will use $n = 4$, the maximum number of paths between two hosts in a fat tree topology.

$$Given\ k = 4,\ paths_{max} = (\frac{K}{2})^2 = (\frac{4}{2})^2 = 4 = n \tag{4.1}$$

The test set with TCP running with statically-configured routers will serve as the control group. TCP and MPTCP tests will observe MPTCP's penalty for `short` (and subsequently `query`) flows, and benefit for `long` flows. Lastly, tests using packet spraying routers and the modified Juggler kernel and if packet spraying and/or the Juggler modification will affect the results significantly.

4.6.3 Expectations

These expectations assume a network where only one host sends data at a time.

4.6.3.1 Flow Sizes and Switch Queues

Given the small sizes of both query and short flows, switch overflows happen less than with long flows. This limits the occurrence of packet retransmissions, guaranteeing a smaller FCT.

4.6.3.2 Path Utilization

Each test set is expected to have a maximum number of paths used described in 4.4.

A fat tree network with ECMP-based routers, and MPTCP hosts using the full mesh path manager, can only use one path since each host is using only a single interface to connect to the rest of the network. Meanwhile, random packet spraying (random PS) will be able to maximize all possible paths between two hosts, computed with the formula described above 4.1.

Generally, using multiple paths will lead to higher throughput, and therefore a smaller FCT compared to only using a single path. However, since these paths will eventually converge into the edge router a host is connected to, switch overflows might occur at the first phase of the flow, hurting throughput. These switch overflows will also trigger timeouts and retransmissions, which slows down FCT.

4.6.4 Test Initialization

A test runner was created to build the topological structure of the fat tree network, log the network behavior, inject commands to the hosts, and extract essential information (such as throughput and flow completion time) through the logged network behavior. More information is available at the source code at Appendix A.9.

4.6.5 Test Proper

Once the topology is built, the testing proper begins.

Each test set's results were obtained by measuring the throughput and flow completion time between the virtualized hosts in mininet. Each test is done with `query`, `short`, and `long` flows to obtain a metric on each test set performance for a specific network traffic in the data center.

Multiple server-client pairs were randomly chosen at the start of the test. This was done to obtain a thorough assessment of the different links in the network. The generation of server-client pairs is done such that each host will become a server and client at some point in the test.

Tests and subsequent throughput measurements were obtained through the use of `iperf`. `Iperf` is a command line utility that allows the measurement of throughput between two hosts in a network. In addition to its base use, `iperf` also allows the specification of specified payload sizes or specific files to be used as the payload. In this case, predefined payload sizes 4.5 were used to match tests sets with a corresponding flow type. This injection and extraction process is done in a Python script executed within the mininet CLI (see Appendix A.10).

Since `iperf` does not measure the flow completion time for the payload, packet captures were logged. However, these packet capture files are not saved to the disk completely up until the mininet instance is finished. Therefore, flow completion times were to be extracted after the iterations, and will be orchestrated through a Python script making use of `tshark` (see Appendix A.11).

Each test is executed five times to gather sufficient data.

4.7 Experimental Hypotheses for a Silent Network and Initial Results

For a silent network (i.e., only two hosts are actively communicating to each other at a time), we propose the following hypotheses for both throughput and FCT.

4.7.1 `Query` and `Short` Flows

Metric	Hypothesis	Reason
Throughput	`TCP > FM`	MPTCP exhibits prominent connection establishment complexity (remember that both setups use only one path)
Throughput	`(TCP > FM) << 4D`	`ndiffports - 4` compensates for throughput with more used paths, and the lack of switch overflows
FCT	`TCP < MPTCP`	MPTCP exhibits prominent connection establishment complexity
FCT	`(TCP < 4D < FM)?`	`ndiffports - 4` compensates for time, lacking switch overflows

Table 4.7: Comparative hypotheses for `query` and `short` flows.

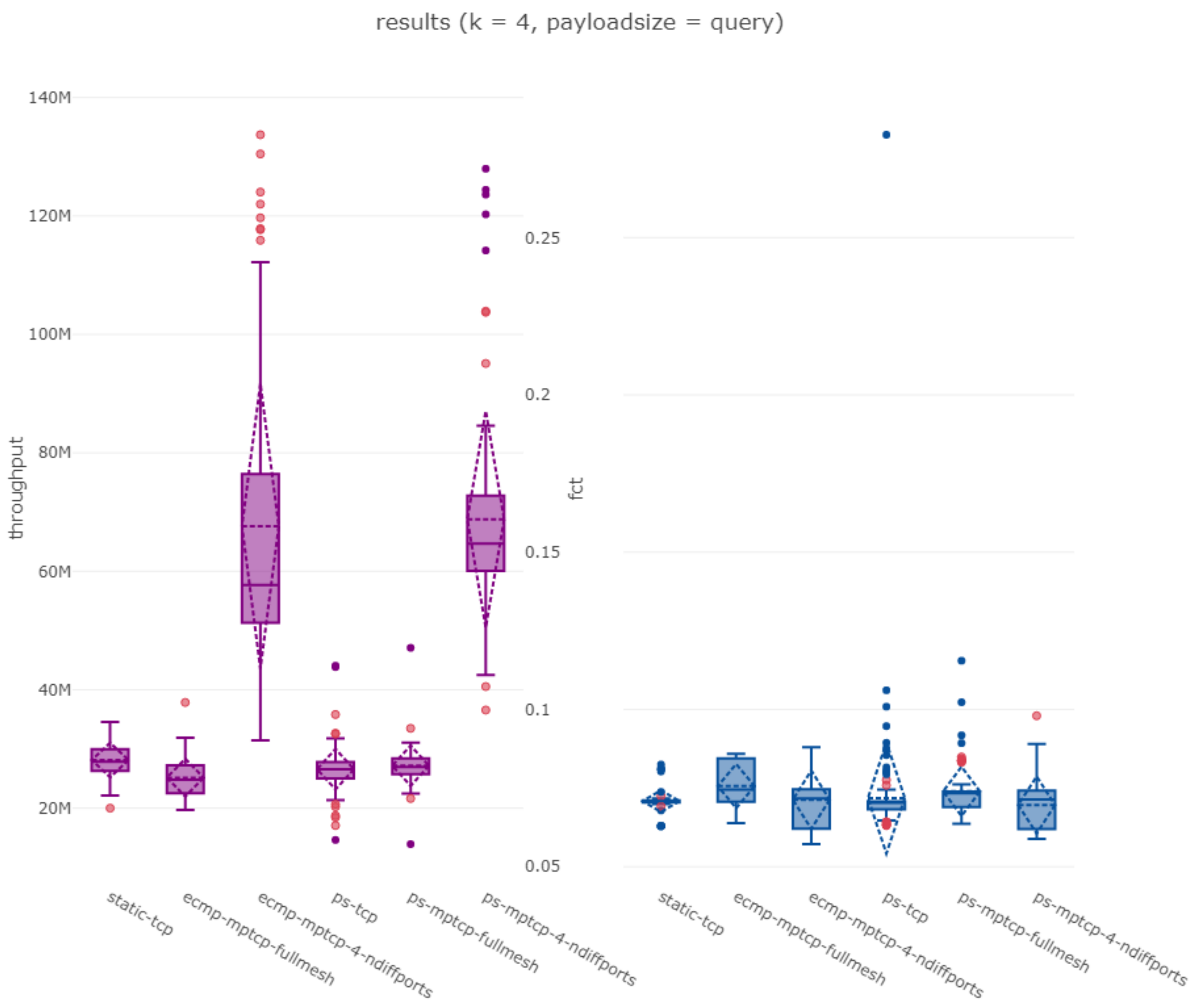


Figure 4.3: Throughput (Mbps) and FCT (s) box graphs for `query` flows with varying test set combinations.

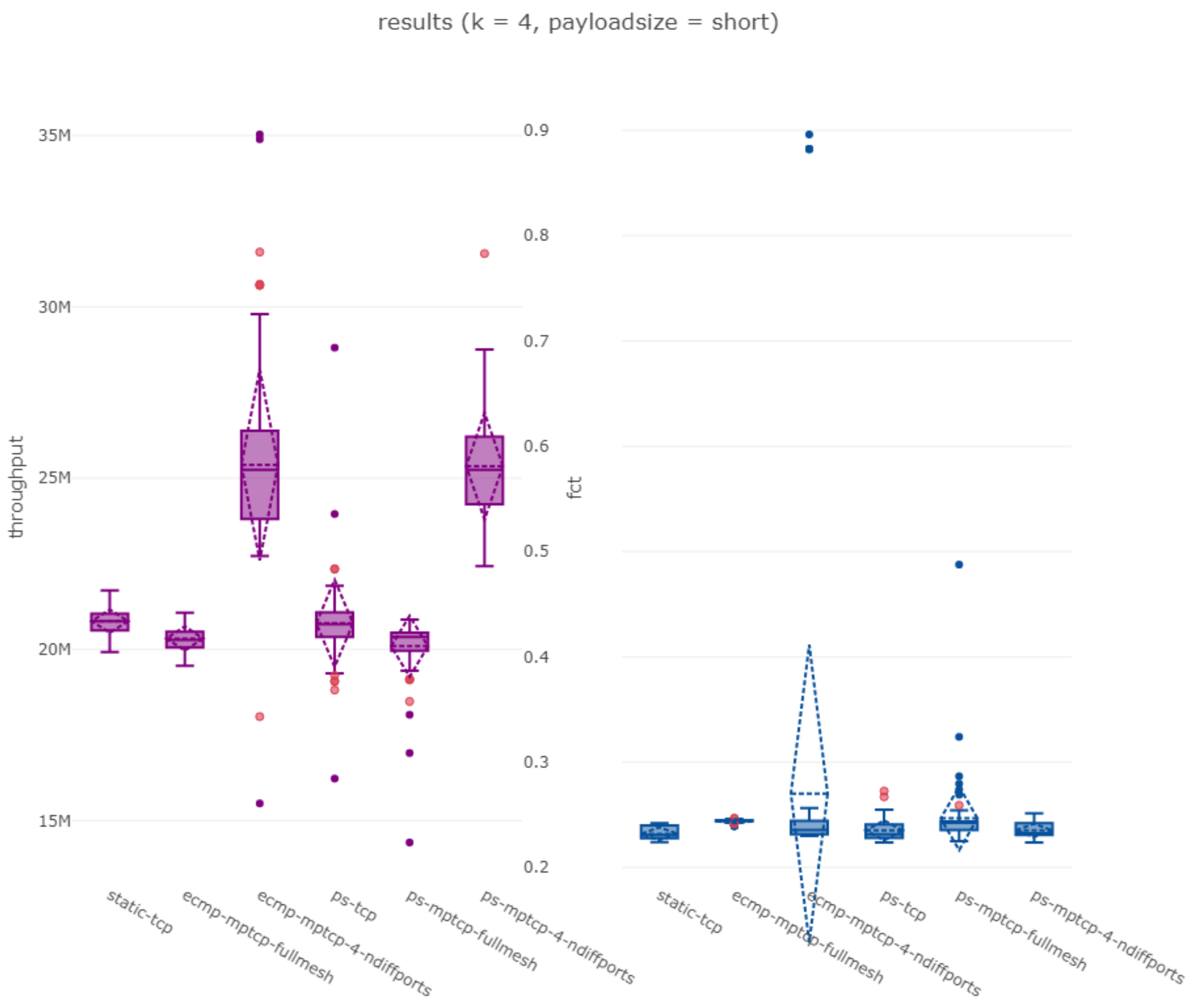


Figure 4.4: Throughput (Mbps) and FCT (s) box graphs for `short` flows with varying test set combinations.

4.7.2 `Long` Flows

Metric	Hypothesis	Reason
Throughput	`TCP > 4D >~ FM`	Switch overflows are now prominent, which reduces sender throughput. This allows TCP to get closer to the link bandwidth versus MPTCP.
FCT	`TCP < MPTCP`	Connection establishment complexity effects now diluted
FCT	`(TCP < 4D ~ FM)?`	This can be attributed either to computational complexity of ECMP switches, or congestion control

Table 4.8: Comparative hypotheses for `long` flows.

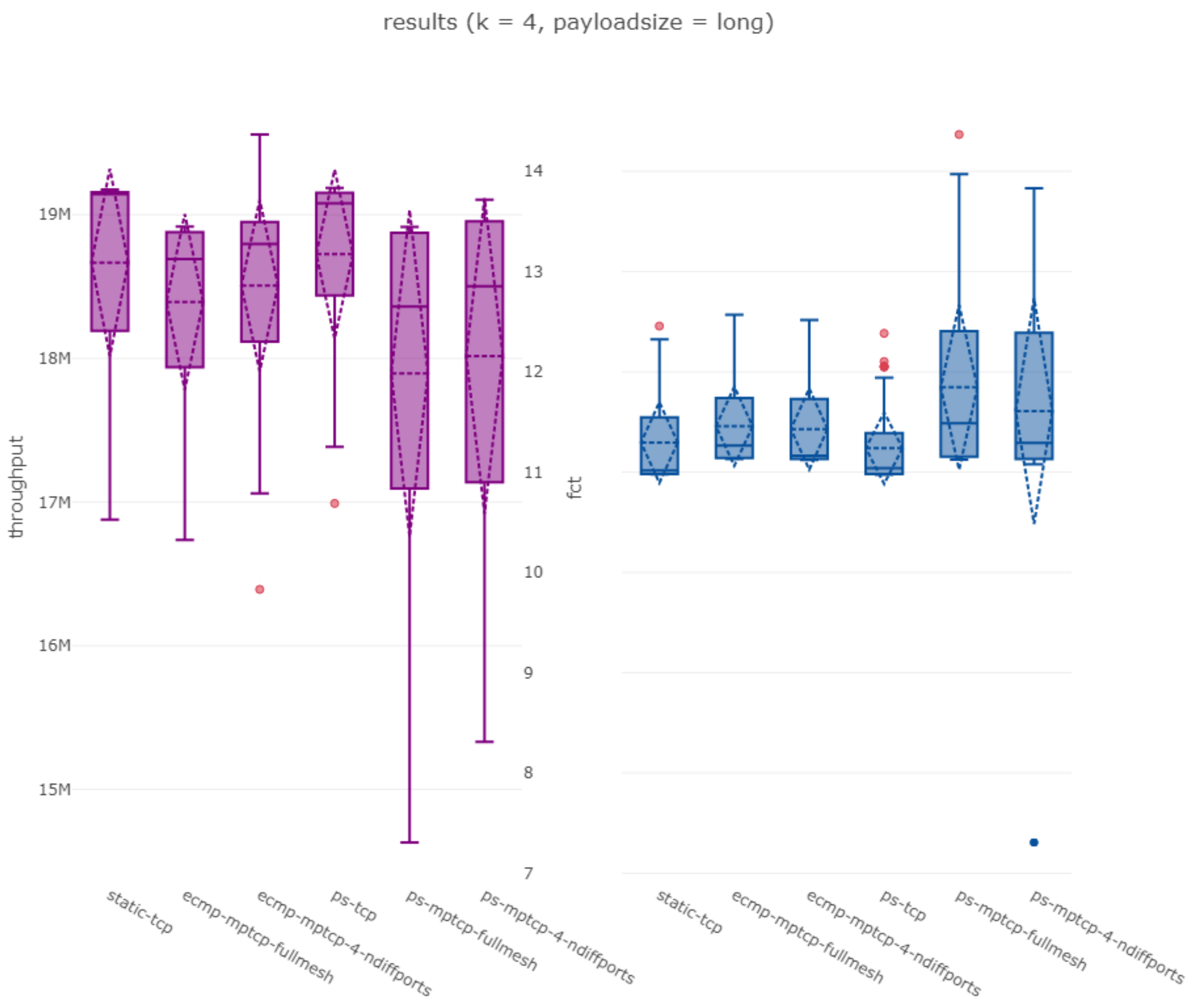


Figure 4.5: Throughput (Mbps) and FCT (s) box graphs for `long` flows with varying test set combinations.

Chapter 5

Project Schedule and Deliverables

5.1 Halfway Point Deliverables

The following are the deliverables for the project halfway point:

- Mininet script that generates a custom datacenter network (fat-tree) topology
- Initial experiment testbed (Mininet topology, MPTCP-enabled network)
- Performance metric scripts/tools (throughput, flow completion time)
- Software switches capable of ECMP
- MPTCP performance metrics (throughput, flow completion time) for a fat-tree topology
- TCP performance metrics (throughput, flow completion time)

5.2 Final Deliverables

The following are the final deliverables for the project:

- Software switches capable of Packet Spraying
- Juggler-enabled end-hosts in experiment testbed
- MPTCP performance metrics (throughput, flow completion time) for a fat-tree topology with switches running either ECMP or Packet Spraying, and end-hosts with or without Juggler enabled.

5.3 Gantt Charts

In addition to the following tasks, all researchers must document their work, and continue working on the paper for the entire research period.

Tasks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Compile Linux kernel for environment	■	■															
MPTCP configuration and testing	■	■															
Create a datacenter network (fat-tree) topology on mininet																	
Collect and design performance metric tools			■	■													
Implement ECMP on switches					■	■	■										
Initial simulations (TCP vs MPTCP)																	
Implement Packet Spraying on switches								■	■	■							
Compile, add, and test Juggler to kernel											■	■					
Design of datacenter traffic simulation												■					
Extensive Simulation and Comparisons													■	■	■		
Buffer																■	■

Table 5.1: Charles Alba's Tasks

Tasks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Compile Linux kernel for environment	■	■															
MPTCP configuration and testing	■	■															
Create a datacenter network (fat-tree) topology on mininet																	
Collect and design performance metric tools			■	■													
Implement ECMP on switches																	
Initial simulations (TCP vs MPTCP)					■	■	■										
Implement Packet Spraying on switches								■	■	■							
Compile, add, and test Juggler to kernel											■	■					
Design of datacenter traffic simulation												■					
Extensive Simulation and Comparisons													■	■	■		
Buffer																■	■

Table 5.2: Kyle Gomez's Tasks

Tasks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Compile Linux kernel for environment	■	■															
MPTCP configuration and testing	■	■															
Create a datacenter network (fat-tree) topology on mininet			■	■													
Collect and design performance metric tools																	
Implement ECMP on switches					■	■	■										
Initial simulations (TCP vs MPTCP)																	
Implement Packet Spraying on switches								■	■	■							
Compile, add, and test Juggler to kernel											■	■					
Design of datacenter traffic simulation												■					
Extensive Simulation and Comparisons													■	■	■		
Buffer																■	■

Table 5.3: Rene Quinto's Tasks

Bibliography

[1] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh, "Juggler: a practical reordering resilient network stack for datacenters," in *Proceedings of the Eleventh European Conference on Computer Systems*, p. 20, ACM, 2016.

[2] J. He and J. Rexford, "Toward internet-wide multipath routing," *IEEE network*, vol. 22, no. 2, 2008.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[4] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 50–61, ACM, 2011.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, pp. 63–74, ACM, 2010.

[7] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[8] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, ACM, 2010.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *NSDI*, vol. 10, pp. 19–19, 2010.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, ACM, 2008.

[11] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *ACM SIGCOMM Computer Communication Review*, vol. 39, pp. 39–50, ACM, 2009.

[12] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[13] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 75–86, ACM, 2008.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[15] B. Lebiednik, A. Mangal, and N. Tiwari, "A survey and evaluation of data center network topologies," *arXiv preprint arXiv:1605.01701*, 2016.

[16] S. Rost and H. Balakrishnan, "Rate-aware splitting of aggregate traffic," tech. rep., Technical report, MIT, 2003.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *ACM SIGCOMM Computer Communication Review*, vol. 41, pp. 266–277, ACM, 2011.

[18] J. Postel *et al.*, "Transmission control protocol rfc 793," 1981.

[19] J. F. Kurose, *Computer networking: A top-down approach featuring the internet, 3/E.* Pearson Education India, 2005.

[20] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural guidelines for multipath tcp development," tech. rep., 2011.

[21] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "Tcp extensions for multipath operation with multiple addresses," tech. rep., 2013.

[22] A. Ford, "Multipath tcp architecture: Towards consensus."

[23] W. R. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, pp. 51–62, ACM, 2009.

[25] R. Barik, M. Welzl, S. Ferlin, and O. Alay, "Lisa: A linked slow-start algorithm for mptcp," in *Communications (ICC), 2016 IEEE International Conference on*, pp. 1–7, IEEE, 2016.

[26] M. Kheirkhah, I. Wakeman, and G. Parisis, "Mmptcp: A multipath transport protocol for data centers," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pp. 1–9, IEEE, 2016.

[27] C. E. Hopps and D. Thaler, "Multipath issues in unicast and multicast next-hop selection," 2000.

[28] M. Fukuyama, N. Yamai, and N. Kitagawa, "Throughput improvement of mptcp communication by bicasting on multihomed network," in *Student Project Conference (ICT-ISPC), 2016 Fifth ICT International*, pp. 9–12, IEEE, 2016.

[29] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella, "On the impact of packet spraying in data center networks," in *INFOCOM, 2013 Proceedings IEEE*, pp. 2130–2138, IEEE, 2013.

[30] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 253–266, ACM, 2017.

[31] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[32] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[33] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[34] A. Hussein, I. H. Elhajj, A. Chehab, and A. Kayssi, "Sdn for mptcp: An enhanced architecture for large data transfers in datacenters," in *Communications (ICC), 2017 IEEE International Conference on*, pp. 1–7, IEEE, 2017.

[35] S. Zannettou, M. Sirivianos, and F. Papadopoulos, "Exploiting path diversity in datacenters using mptcp-aware sdn," in *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pp. 539–546, IEEE, 2016.

[36] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 225–238, ACM, 2017.

[37] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "Rr-tcp: a reordering-robust tcp with dsack," in *Network Protocols, 2003. Proceedings. 11th IEEE International Conference on*, pp. 95–106, IEEE, 2003.

[38] M. Kheirkhah, "Multipath tcp (mptcp) implementation in ns-3." Github, May 2014.

[39] U. catholique de Louvain, "Linux kernel implementation of multipath (mptcp)." Github, Nov. 2017.

[40] C.-H. Ke, "Multipath tcp test." Website, Dec. 2017.

[41] U. catholique de Louvain, "Multipath tcp - linux kernel implementation : Users - configure mptcp browse." Website, Aug. 2017.

[42] R. Coltun, D. Ferguson, J. Moy, and A. Lindem, "Rfc 5340, ospf for ipv6," *IETF. July*, vol. 24, 2008.

[43] U. catholique de Louvain, "MultiPath TCP - Linux Kernel implementation : Users - How To Install MPTCP browse."

[44] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[45] p4lang, “behavioral-model: Rewrite of the behavioral model as a C++ project without auto-generated code (except for the PD interface),” Mar. 2018. original-date: 2015-01-26T21:43:23Z.

[46] p4lang, “p4c-bm: Generates the JSON configuration for the behavioral-model (bmv2), as well as the C/C++ PD code,” Mar. 2018. original-date: 2015-08-07T15:48:22Z.

[47] p4lang, “p4lang/p4-spec,” Feb. 2018. original-date: 2015-08-14T08:45:53Z.

Appendix A

Code Snippets

These code snippets come from the researchers' git repository available at `https://github.com/MMfSDT/`

A.1 Topology Generator - Python Script

The full repository is available at `https://github.com/MMfSDT/mininet-topo-generator`
.

Listing A.1: mininet-topo-generator/topogen.py, written in Python

```
#!/usr/bin/env python

################################
#   topogen.py
#       Generates a scalable fat-tree topology.
#       Follows this syntax:
#           ./topogen.py
#               [--test path_to_test {none}]
#               [--post path_to_post_process_script {none}]
#               [--router router_behavior {static}]
#               [--pcap]
#               [--K K {4}]
#               [--proto tcp|mptcp {mptcp}]
#               [--pmanager fullmesh|ndiffports {fullmesh}]
#                   [--diffports num_diff_ports {1}]
#                   [--payloadsize query|long|short {short}]
#                   [--runcount num_counts {10}]
#                                         [--exec_path exec_path {../behavioral-model/targets/
    simple_router/simple_router}]
#                                         [--json_path json_path {./router/simple_router.json}]
#                                         [--cli_path cli_path {../behavioral-model/tools/runtime_CLI.
    py}]
#                                         [--tablegen_path tablegen_path {./router/tablegen_simple.py}]
#       Make sure to set env.sh first before proceeding.
################################

from mininet.net import Mininet
from mininet.cli import CLI
from mininet.link import Link, TCLink, Intf
from subprocess import Popen, PIPE
from mininet.log import setLogLevel
import os
import argparse
import sys
import subprocess
from random import randint
import imp

from router.p4_mininet import P4Switch, P4Host

# Handle arguments in a more elegant manner using argparse.

parser = argparse.ArgumentParser(description='Generates a scalable Fat-tree topology.')
parser.add_argument('--test', default=None, type=str, metavar='path_to_test', help='specify
    a test to run. defaults to None.')
parser.add_argument('--post', default=None, type=str, metavar='path_to_post_process_script'
    , help='specify a post-processing script to run. defaults to None.')
```

```
parser.add_argument('--router', default='static', type=str, metavar='router_behavior', help='configure␣switch␣behavior␣between␣static,␣ecmp,␣and␣ps.␣defaults␣to␣static.')
parser.add_argument('--pcap', action='store_true', help='dumps␣pcap␣files')
parser.add_argument('--K', default='4', type=int, metavar='num_ports', help='number␣of␣ports␣per␣switch.␣defaults␣to␣4.')
parser.add_argument('--proto', default='mptcp', type=str, metavar='tcp|mptcp', help='configure␣host␣protocol␣between␣tcp␣and␣mptcp.␣defaults␣to␣mptcp.')
parser.add_argument('--pmanager', default='fullmesh', type=str, metavar='fullmesh|ndiffports', help='specify␣a␣MPTCP␣path␣manager.␣defaults␣to␣fullmesh.')
parser.add_argument('--diffports', default=1, type=int, metavar='num_diff_ports', help='if␣--pmanager␣is␣set␣to␣ndiffports,␣set␣diffports␣here.␣defaults␣to␣1.')
parser.add_argument('--payloadsize', default='short', type=str, metavar='query|long|short', help='specify␣flow␣size.␣defaults␣to␣short.')
parser.add_argument('--runcount', default=10, type=int, metavar='num_counts', help='specify␣how␣many␣tests␣should␣be␣done␣per␣pair.␣defaults␣to␣10.')
parser.add_argument('--exec_path', default='../behavioral-model/targets/simple_router/simple_router', type=str, help='provide␣the␣path␣to␣the␣simple␣router␣executable')
parser.add_argument('--json_path', default='./router/simple_router.json', type=str, help='provide␣the␣path␣to␣the␣behavioral␣json')
parser.add_argument('--cli_path', default='../behavioral-model/tools/runtime_CLI.py', type=str, help='provide␣the␣path␣to␣the␣runtime␣CLI')
parser.add_argument('--tablegen_path', default='./router/tablegen_simple.py', type=str, help='provide␣the␣path␣to␣the␣table␣generator␣script')

args = parser.parse_args()

exec_path = args.exec_path;
json_path = args.json_path;
cli_path = args.cli_path;
tablegen_path = args.tablegen_path;

# Code proper.

```

```
if '__main__' == __name__:
        setLogLevel('info')
        net = Mininet(controller=None)
    #key = "net.mptcp.mptcp_enabled"
    #value = 1
    #p = Popen("sysctl -w %s=%s" % (key, value),
    #         shell=True, stdout=PIPE, stderr=PIPE)
    #stdout, stderr = p.communicate()
    #print "stdout=", stdout, "stderr=", stderr

        K = args.K                                                                   #
    Moved from argv[1] to args.K
        print "Generating topology for K =", K

        print "Naming convention"
        print "Host:               h<pod><i><j>"
        print "Edge switch:        se<pod><i>"
        print "Aggregate switch:   sa<pod><i>"
        print "Core switch:        sc<i><j>"

        host_ip = [[[
        '10.%d.%d.%d'%(pod,i,j+2)
        for j in range(K/2)]
        for i in range(K/2)]
        for pod in range(K)]

        host = [[[
        net.addHost('h%d%d%d'%(pod,i,j),
                cls=P4Host,
                ip=host_ip[pod][i][j])
    for j in range(K/2)]
    for i in range(K/2)]
        for pod in range(K)]



        port_offset  = 10000

        edge_port = [[
        pod*K/2+i + port_offset
        for i  in range(K/2)]
        for pod in range(K)]

        agg_port = [[
        pod*K/2+i + K*K/2 + port_offset
        for i  in range(K/2)]
        for pod in range(K)]

        core_port  = [[
        i*K/2+j + K*K + port_offset
        for j  in range(K/2)]
        for i  in range(K/2)]

        edge = [[
        net.addSwitch('se%d%d'%(pod,i),
                cls  = P4Switch,
                sw_path = exec_path,
                json_path = json_path,
                thrift_port  = edge_port[pod][i ],
                pcap_dump = args.pcap)
        for i  in range(K/2)]
        for pod in range(K)]

        agg = [[
        net.addSwitch('sa%d%d'%(pod,i),
                cls  = P4Switch,
                sw_path = exec_path,
                json_path = json_path,
                 thrift_port  = agg_port[pod][i ],
                pcap_dump = args.pcap)
        for i  in range(K/2)]
        for pod in range(K)]

        core = [[
        net.addSwitch('sc%d%d'%(i,j),
                 cls  = P4Switch,
                sw_path = exec_path,
                json_path = json_path,
                 thrift_port  = core_port[i ][ j ],
                pcap_dump = args.pcap)
        for j  in range(K/2)]
        for i  in range(K/2)]



        edge_ip  = [[
        '10.%d.%d.1'%(pod,i)
        for i  in range(K/2)]
        for pod in range(K)]

        agg_ip  = [[
        '10.%d.%d.1'%(pod,i)
        for i  in range(K/2,K)]
        for pod in range(K)]

        core_ip  = [[
        '10.%d.%d.%d'%(K,i+1,j+1)
        for j  in range(K/2)]
        for i  in range(K/2)]
```

```



    linkopt = {'bw': 10}

    #host to edge
    for pod in range(K):
        for i in range(K/2):
            for j in range(K/2):
                net.addLink(host[pod][i][j],edge[pod][i])

    #edge to aggregate
    for pod in range(K):
        for i in range(K/2):
            for j in range(K/2):
                net.addLink(edge[pod][i],agg[pod][j])

    #aggregate to core
    for pod in range(K):
        for i in range(K/2):
            for j in range(K/2):
                net.addLink(agg[pod][i],core[i][j])


    net.build()
    net.staticArp()
    net.start()

    #configure host forwarding
    for pod in range(K):
        for i in range(K/2):
            for j in range(K/2):
```

```
                        host[pod][i][j].setDefaultRoute('dev eth0 via %s'%(edge_ip[pod][i]))
                        # IPv6 messes with the logs. Disable it.
                        host[pod][i][j].cmd("sysctl -w net.ipv6.conf.all.disable_ipv6=1")
                        host[pod][i][j].cmd("sysctl -w net.ipv6.conf.default.disable_ipv6=1")
                        host[pod][i][j].cmd("sysctl -w net.ipv6.conf.lo.disable_ipv6=1")

    #get tablegen to initialize routing tables
    tablegen = imp.load_source('tablegen',tablegen_path).TableGenerator(
    K=K,
    port_offset=port_offset,
    verbose=True,
    cli_path=cli_path,
    json_path=json_path
)

    tablegen.init_all()

    print "\n\n*** Topology setup done."

    if (args.test is not None):
        if (os.path.isfile("kickstart_python.test")):
            print "*** Running test: {}\n\n".format(args.test)
            CLI(net, script="kickstart_python.test")
            print "*** Test done: {}\n\n".format(args.test)
        else:
            print "*** Skipping test file, it does not exist: {}\n\n".format(args.test)
    else:
        print "*** No test to execute."
```

```
            # The interactive cmd will now only run if there are no tests executed.
            print "\n*** To quit, type 'exit' or press 'Ctrl+D'."
            CLI(net)

    try:
            net.stop()
    except:
            print "\n*** Quitting Mininet."
```

A.2 Modified P4 - Simple Router Primitives - C++ Script

The full repository is available at `https://github.com/MMfSDT/behavioral-model`.

Listing A.2: behavioral-model/targets/simple_router/primitives.cpp

```
/* Copyright 2013-present Barefoot Networks, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * Antonin Bas (antonin@barefootnetworks.com)
 *
 */

```

```
#include <bm/bm_sim/actions.h>
#include <bm/bm_sim/core/primitives.h>
#include <cstdlib>
#include <ctime>

template <typename... Args>
using ActionPrimitive = bm::ActionPrimitive<Args...>;

using bm::Data;
using bm::Field;
using bm::Header;

class modify_field : public ActionPrimitive<Field &, const Data &> {
  void operator ()(Field &f, const Data &d) {
    bm::core::assign()(f, d);
  }
};

REGISTER_PRIMITIVE(modify_field);

class add_to_field : public ActionPrimitive<Field &, const Data &> {
  void operator ()(Field &f, const Data &d) {
    f.add(f, d);
  }
};

REGISTER_PRIMITIVE(add_to_field);

class drop : public ActionPrimitive<> {
  void operator ()() {
     get_field ("standard_metadata.egress_spec").set(511);
  }
};
```

```

REGISTER_PRIMITIVE(drop);

class modify_field_rng_uniform : public ActionPrimitive<Field &, const Data &, const Data
    &> {
        unsigned int g_seed = -1;
        bool seeded = false;

        // Used to seed the generator.
        inline void fast_srand(int seed) {
            g_seed = seed;
            seeded = true;
        }

        // Compute a pseudorandom integer.
        // Output value in range [0, 32767]
        inline int fast_rand(int bits) {
            g_seed = (214013*g_seed+2531011);
            return (g_seed>>16)&bits;
        }


        void operator()(Field &f, const Data &a, const Data &b) {
                if (!seeded)
                        fast_srand(time(0));
                Data d(fast_rand(b.get_int()));
                bm::core::assign()(f,d);
        }
};

REGISTER_PRIMITIVE(modify_field_rng_uniform);
```

A.3 Routers and their Table Generator Scripts - P4 and Python Scripts

The full repository is available at `https://github.com/MMfSDT/behavioral-model`.

Listing A.3: mininet-topo-generator/router/simple_router.p4

```
/* Copyright 2013-present Barefoot Networks, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *   http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
    }
}

parser start {
    return parse_ethernet;
}

#define ETHERTYPE_IPV4 0x0800

header ethernet_t ethernet;
header ipv4_t ipv4;

field_list ipv4_checksum_list {
        ipv4.version;
        ipv4.ihl;
        ipv4.diffserv;
        ipv4.totalLen;
        ipv4.identification;
        ipv4.flags;
        ipv4.fragOffset;
        ipv4.ttl;
        ipv4.protocol;
        ipv4.srcAddr;
        ipv4.dstAddr;
```

```
}


 field_list_calculation   ipv4_checksum {
     input {
          ipv4_checksum_list;
     }
     algorithm : csum16;
     output_width : 16;
}

 calculated_field  ipv4.hdrChecksum  {
     update ipv4_checksum;
}

parser parse_ethernet {
     extract(ethernet);
     extract(ipv4);
     return ingress;
}

action _drop() {
     drop();
}

action set_nhop(port) {
     modify_field(standard_metadata.egress_spec, port);
     modify_field(ipv4.ttl, ipv4.ttl - 1);
}

table ipv4_match {
     reads {
          ipv4.dstAddr : lpm;
```

```
    }
    actions {
        set_nhop;
        _drop;
    }
    size: 1024;
}

control ingress {
    if(valid(ipv4) and ipv4.ttl>0) {
        apply(ipv4_match);
    }
}

control egress {
}
```

Listing A.4: mininet-topo-generator/router/tablegen_simple.py

```
#!/usr/bin/env python

from random import randint
import subprocess

class TableGenerator:

    def __init__(self, K, port_offset, cli_path, json_path, verbose=False):
        self.host_ip = [[[
        '10.%d.%d.%d'%(pod,i,j+2)
        for j in range(K/2)]
        for i in range(K/2)]
        for pod in range(K)]

        self.port_offset = port_offset

        self.edge_port = [[
        pod*K/2+i + port_offset
        for i in range(K/2)]
        for pod in range(K)]

        self.agg_port = [[
        pod*K/2+i + K*K/2 + port_offset
        for i in range(K/2)]
        for pod in range(K)]

        self.core_port = [[
        i*K/2+j + K*K + port_offset
        for j in range(K/2)]
        for i in range(K/2)]

        self.verbose = verbose
        self.K = K
        self.port_offset = port_offset
        self.cli_path = cli_path
        self.json_path = json_path

        if self.verbose:
            print "Initialized TableGenerator with K=",K,", port_offset=",port_offset,", verbose=",verbose

    def edge_init(self):
        if self.verbose:
            print "Configuring edge routers"

        for pod in range(self.K):
            for i in range(self.K/2):
                if self.verbose:
                    print "Configuring se%d%d"%(pod,i)

                    cmd = ['table_set_default ipv4_match _drop']

                #downstream
                for j in range(self.K/2):
                    cmd.append('table_add ipv4_match set_nhop 10.%d.%d.%d/32 => %d'%(pod,i,j+2,j+1))

                #upstream
                for npod in range(self.K):
                    for ni in range(self.K/2):
                            if npod==pod and ni==i:
                                    continue
                            for nj in range(self.K/2):
                                    fwd = randint(0,self.K/2-1)
                                    cmd.append('table_add ipv4_match set_nhop %s/32 => %
d'%(self.host_ip[npod][ni][nj],fwd+self.K/2+1))

                p = subprocess.Popen(
                    [ self . cli_path , '--json', self .json_path, '--thrift-port', str(self.
edge_port[pod][i ]) ],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self .verbose:
                    print msg

def agg_init ( self ):
    if self .verbose:
        print "Configuring_aggregate_routers"

    for pod in range(self.K):
        for i in range(self.K/2):
            if self .verbose:
                print "Configuring_sa%d%d"%(pod,i)

            cmd = ['table_set_default_ipv4_match__drop']

            #downstream
            for j in range(self.K/2):
                cmd.append('table_add_ipv4_match_set_nhop_10.%d.%d.0/24_=>_%d'%(
pod,j,j+1))

            for npod in range(self.K):
                if npod==pod:
                    continue
                for ni in range(self.K/2):
                    for nj in range(self.K/2):
                                        fwd = randint(0,self.K/2−1)
                                        cmd.append('table_add_ipv4_match_
set_nhop_%s/32_=>_%d'%(self.host_ip[npod][ni][nj],fwd+self.K/2+1))

            p = subprocess.Popen(
                [ self . cli_path , '−−json', self .json_path, '−−thrift−port', str(self.
agg_port[pod][i ]) ],
                stdin=subprocess.PIPE,
                stdout=subprocess.PIPE,
                stderr=subprocess.PIPE)

            msg,err = p.communicate('\n'.join(cmd))
            if self .verbose:
                print msg

def core_init ( self ):
    if self .verbose:
        print "Configuring_core_routers"

    for i in range(self.K/2):
        for j in range(self.K/2):
            if self .verbose:
                print "\nConfiguring_sc%d%d"%(i,j)

            cmd = ['table_set_default_ipv4_match__drop']

            for pod in range(self.K):
                cmd.append('table_add_ipv4_match_set_nhop_10.%d.0.0/16_=>_%d'%(pod,
pod+1))

            p = subprocess.Popen(
                [ self . cli_path , '−−json', self .json_path, '−−thrift−port', str(self.
core_port[i][j])],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

def init_all(self):
    if self.verbose:
        print "Initializing all routers\n\n"

    self.edge_init()
    self.agg_init()
    self.core_init()
```

Listing A.5: mininet-topo-generator/router/ecmp_router.p4

```
/* Copyright 2013-present Barefoot Networks, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */


header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
        padding: 32;
    }
}

header_type tcp_t {
    fields {
        srcPort: 8;
        dstPort: 8;
    }
}

header_type routing_metadata_t {
    fields {
        hashVal: 3;
    }
}

header ethernet_t ethernet;
header ipv4_t ipv4;
header tcp_t tcp;
metadata routing_metadata_t routing_metadata;

 field_list ipv4_checksum_list {
        ipv4.version;
        ipv4.ihl;
        ipv4.diffserv;
        ipv4.totalLen;
        ipv4.identification;
        ipv4.flags;
        ipv4.fragOffset;
        ipv4.ttl;
        ipv4.protocol;
        ipv4.srcAddr;
        ipv4.dstAddr;
}

 field_list_calculation  ipv4_checksum {
    input {
        ipv4_checksum_list;
    }
    algorithm : csum16;
    output_width : 16;
}

calculated_field  ipv4.hdrChecksum  {
    update ipv4_checksum;
}

field_list  flow_id  {
    ipv4.srcAddr;
    tcp.srcPort;
    ipv4.dstAddr;
    tcp.dstPort;
    ipv4.protocol;
}

 field_list_calculation  flow_hash {
    input {
        flow_id;
    }
    algorithm: crc16;
    output_width: 3;
}

 calculated_field  routing_metadata.hashVal {
    update flow_hash;
}

parser start {
    extract(ethernet);
    extract(ipv4);
    extract(tcp);
    return ingress;
}

action _drop() {
    drop();
}

action set_nhop(port) {
    modify_field(standard_metadata.egress_spec, port);
    modify_field(ipv4.ttl, ipv4.ttl - 1);
}

table ipv4_match {
        reads {
                ipv4.dstAddr : lpm;
        routing_metadata.hashVal : exact;
        }
        actions {
                set_nhop;
        _drop;
        }
```

```
}

control ingress {
    if (valid(ipv4) and ipv4.ttl >0) {
        apply(ipv4_match);
    }
}

control egress {
}
```

Listing A.6: mininet-topo-generator/router/tablegen_ecmp.py

```
#!/usr/bin/env python

from random import shuffle
import subprocess

class TableGenerator:

        def __init__(self, K, port_offset, cli_path, json_path, verbose=False):
                self.host_ip = [[[
                '10.%d.%d.%d'%(pod,i,j+2)
                for j in range(K/2)]
                for i in range(K/2)]
                for pod in range(K)]

                self.port_offset = port_offset

                self.edge_port = [[
                pod*K/2+i + port_offset
                for i in range(K/2)]
                for pod in range(K)]

                self.agg_port = [[
                pod*K/2+i + K*K/2 + port_offset
                for i in range(K/2)]
                for pod in range(K)]

                self.core_port = [[
                i*K/2+j + K*K + port_offset
                for j in range(K/2)]
                for i in range(K/2)]

                self.verbose = verbose
        self.K = K
        self.port_offset = port_offset
        self.cli_path = cli_path
        self.json_path = json_path

        if self.verbose:
            print " Initialized TableGenerator with K=",K,", port_offset=",
port_offset,", verbose=",verbose

    def edge_init(self):
        if self.verbose:
            print "Configuring edge routers"

        for pod in range(self.K):
            for i in range(self.K/2):
                if self.verbose:
                    print "Configuring se%d%d"%(pod,i)

                cmd = ['table_set_default ipv4_match  drop']

                #downstream
                for j in range(self.K/2):
                    for p in range(8):
                        cmd.append('table_add ipv4_match set_nhop 
10.%d.%d.%d/32 %d => %d'%(pod,i,j+2,p,j+1))

                ports = []
                for p in range(self.K/2):
                    for j in range(16/self.K):
                        ports.append(p)
                shuffle(ports)


```

```
                    #upstream
                    for j in range(8):
                        p = ports[j]
                        cmd.append('table_add ipv4_match set_nhop 10.0.0.0/8
  %d => %d'%(j,p+1+self.K/2))

                    p = subprocess.Popen(
                        [self.cli_path, '--json', self.json_path, '--thrift-
port', str(self.edge_port[pod][i])],
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)

                    msg,err = p.communicate('\n'.join(cmd))
                    if self.verbose:
                        print msg

    def agg_init(self):
        if self.verbose:
            print "Configuring aggregate routers"

        for pod in range(self.K):
            for i in range(self.K/2):
                if self.verbose:
                    print "Configuring sa%d%d"%(pod,i)

                cmd = ['table_set_default ipv4_match  drop']

                ports = []
                for p in range(self.K/2):
                    for j in range(16/self.K):
                        ports.append(p)
                shuffle(ports)

                #downstream
                for j in range(self.K/2):
                    for p in range(8):
                        cmd.append('table_add ipv4_match set_nhop 10.%d.%d.0/24 %d => %d'%(pod,j,p,j+1))

                #upstream
                for j in range(8):
                    p = ports[j]
                    cmd.append('table_add ipv4_match set_nhop 10.0.0.0/8 %d => %d'%(j,p+1+self.K/2))

                p = subprocess.Popen(
                    [self.cli_path, '--json', self.json_path, '--thrift-port', str(self.agg_port[pod][i])],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

    def core_init(self):
        if self.verbose:
            print "Configuring core routers"

        for i in range(self.K/2):
            for j in range(self.K/2):
                if self.verbose:
                    print "\nConfiguring sc%d%d"%(i,j)

                cmd = ['table_set_default ipv4_match  drop']

                #everything is downstream
                for pod in range(self.K):
                    for p in range(8):
                        cmd.append('table_add ipv4_match set_nhop 10.%d.0.0/16 %d => %d'%(pod,p,pod+1))

                p = subprocess.Popen(
                    [self.cli_path, '--json', self.json_path, '--thrift-port', str(self.core_port[i][j])],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

    def init_all(self):
        if self.verbose:
            print "Initializing all routers\n\n"

        self.edge_init()
        self.agg_init()
        self.core_init()
```

Listing A.7: mininet-topo-generator/router/ps_router.p4

```
/* Copyright 2013-present Barefoot Networks, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */


header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
```

```
            fragOffset  :  13;
            ttl  :  8;
            protocol  :  8;
            hdrChecksum : 16;
            srcAddr : 32;
            dstAddr: 32;
      }
}

header ethernet_t  ethernet;
header ipv4_t  ipv4;

 field_list   ipv4_checksum_list  {
            ipv4.version;
            ipv4.ihl;
            ipv4. diffserv ;
            ipv4.totalLen;
            ipv4. identification ;
            ipv4.flags;
            ipv4.fragOffset;
            ipv4.ttl;
            ipv4.protocol;
            ipv4.srcAddr;
            ipv4.dstAddr;
}

  field_list_calculation   ipv4_checksum {
      input {
            ipv4_checksum_list;
      }
      algorithm : csum16;
      output_width : 16;
}
```

```

 calculated_field  ipv4.hdrChecksum  {
     update ipv4_checksum;
}

parser  start  {
     extract(ethernet);
     extract(ipv4);
     return  ingress;
}

action  set_nhop_random(port_cnt) {
     modify_field_rng_uniform(standard_metadata.egress_spec, 0,  port_cnt−1);
      add_to_field (standard_metadata.egress_spec, port_cnt+1);
      modify_field(ipv4.ttl,  ipv4.ttl  − 1);
}

action  set_nhop(port) {
      modify_field(standard_metadata.egress_spec, port);
      modify_field(ipv4.ttl,  ipv4.ttl  − 1);
}

table  ipv4_match {
            reads {
                        ipv4.dstAddr : lpm;
            }
            actions {
                        set_nhop;
            set_nhop_random;
            }
}

control  ingress  {
    if ( valid (ipv4)  and ipv4. ttl >0) {
        apply(ipv4_match);
    }
}

control  egress  {
}
```

Listing A.8: mininet-topo-generator/router/tablegen_ps.py

```
#!/usr/bin/env python

from random import shuffle
import subprocess

class TableGenerator:

        def __init__(self, K, port_offset, cli_path, json_path, verbose=False):
                self.host_ip = [[[
                '10.%d.%d.%d'%(pod,i,j+2)
                for j in range(K/2)]
                for i in range(K/2)]
                for pod in range(K)]

                self.port_offset = port_offset

                self.edge_port = [[
                pod*K/2+i + port_offset
                for i in range(K/2)]
                for pod in range(K)]

                self.agg_port = [[
                pod*K/2+i + K*K/2 + port_offset
                for i in range(K/2)]
                for pod in range(K)]

                self.core_port = [[
                i*K/2+j + K*K + port_offset
                for j in range(K/2)]
                for i in range(K/2)]

                self.verbose = verbose
```

```
            self.K = K
            self.port_offset = port_offset
            self.cli_path = cli_path
            self.json_path = json_path

            if self.verbose:
                print "Initialized TableGenerator with K=",K,", port_offset=",
    port_offset,", verbose=",verbose

    def edge_init(self):
            if self.verbose:
                print "Configuring edge routers"

            for pod in range(self.K):
                for i in range(self.K/2):
                    if self.verbose:
                        print "Configuring se%d%d"%(pod,i)

                    cmd = ['table_set_default ipv4_match set_nhop_random %s'%(
    self.K/2)]

                    #downstream
                    for j in range(self.K/2):
                        cmd.append('table_add ipv4_match set_nhop 10.%d.%d
    .%d/32 => %d'%(pod,i,j+2,j+1))

                    p = subprocess.Popen(
                        [self.cli_path, '--json', self.json_path, '--thrift-
    port', str(self.edge_port[pod][i])],
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

    def agg_init(self):
        if self.verbose:
            print "Configuring aggregate routers"

        for pod in range(self.K):
            for i in range(self.K/2):
                if self.verbose:
                    print "Configuring sa%d%d"%(pod,i)

                cmd = ['table_set_default ipv4_match set_nhop_random %s'%(self.K/2)]

                #downstream
                for j in range(self.K/2):
                    cmd.append('table_add ipv4_match set_nhop 10.%d.%d.0/24 => %d'%(pod,j,j+1))

                p = subprocess.Popen(
                    [self.cli_path, '--json', self.json_path, '--thrift-port', str(self.agg_port[pod][i])],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

    def core_init(self):
        if self.verbose:
            print "Configuring core routers"

        for i in range(self.K/2):
            for j in range(self.K/2):
                if self.verbose:
                    print "\nConfiguring sc%d%d"%(i,j)

                cmd = ['table_set_default ipv4_match set_nhop_random %s'%(self.K/2)]

                #everything is downstream
                for pod in range(self.K):
                    cmd.append('table_add ipv4_match set_nhop 10.%d.0.0/16 => %d'%(pod,pod+1))

                p = subprocess.Popen(
                    [self.cli_path, '--json', self.json_path, '--thrift-port', str(self.core_port[i][j])],
                    stdin=subprocess.PIPE,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE)

                msg,err = p.communicate('\n'.join(cmd))
                if self.verbose:
                    print msg

    def init_all(self):
        if self.verbose:
            print "Initializing all routers\n\n"

        self.edge_init()
        self.agg_init()
```

```
                    self . core_init ()
```

A.4 Test Runner Scripts - Bash and Python Scripts

The full repository is available at `https://github.com/MMfSDT/network-tests`.

Listing A.9: mininet-topo-generator/run.sh

```
#!/bin/bash

# 
    ############################################################

#   run.sh
#       Bootstrap a static topology.
#       Virtually follows topogen.py's syntax:
#           ./run.sh
#               [--test path_to_test {none}]
#               [--post path_to_post_process_script {none}]
#               [--router router_behavior {static}]
#               [--pcap]
#               [--K K {4}]
#               [--proto tcp|mptcp {mptcp}]
#               [--pmanager fullmesh|ndiffports {fullmesh}]
#               [--diffports num_diff_ports {1}]
#               [--payloadsize query|long|short {short}]
#               [--runcount num_counts {10}]
#       Make sure to set env.sh first before proceeding.
# 
    ############################################################


# Export all the prerequisite environmental variables for this process.
set -o allexport
source env.sh
set +o allexport

```

```
# Clean the mess Mininet makes from a failed exit silently.
sudo mn -c &> /dev/null

# Clean old .pcap traces in case of errors.
rm s*.pcap

# Find the test script, if indicated.
# Store arguments first as we're mutating it.
args=("$@")

# Loop through and find value assigned to "--test".
while [[ "$#" > 0 ]]; do case $1 in
    --test) test="$2"; shift;;
    --post) post="$2"; shift;;
    --router) router="$2"; shift;;
    --pcap) pcap="true";;
    --K) K="$2"; shift;;
    --proto) proto="$2"; shift;;
    --pmanager) pmanager="$2"; shift;;
    --diffports) diffports="$2"; shift;;
    --payloadsize) payloadsize="$2"; shift;;
    --runcount) runcount="$2"; shift;;
    esac; shift
done

# Restore arguments to original position.
set -- "${args[@]}"

# Set default arguments.
if [[ -z "$router" ]]; then router="static"; fi
if [[ -z "$K" ]]; then K="4"; fi
if [[ -z "$proto" ]]; then proto="mptcp"; fi
if [[ -z "$pmanager" ]]; then pmanager="fullmesh"; fi
if [[ -z "$diffports" ]]; then diffports="1"; fi
if [[ -z "$payloadsize" ]]; then payloadsize="short"; fi
if [[ -z "$runcount" ]]; then runcount="10"; fi

# It is necessary to create a source file for Mininet to parse.
#   This automatically generated file is at "./kickstart_python.test"
if [[ -z "$test" ]]; then
    # If "--test" wasn't given as an argument, remove existing source file.
    echo "No test to execute."
    rm kickstart_python.test &> /dev/null
elif [[ ! -f "$test" ]]; then
    # If "--test" file does not exist, remove existing source file.
    echo "File \"$test\" does not exist. Skipping."
    rm kickstart_python.test &> /dev/null
else
    # If "--test" file does exist, write the source file.
    echo "py execfile(\"$test\")" > kickstart_python.test
    echo "Running test: \"$test\""
fi

# Set the TOPO_JSON and TOPO_TABLEGEN paths accordingly.
if [ "$router" == "static" ]; then
    TOPO_JSON_PATH=$TOPO_JSON_SIMPLE_PATH
    TOPO_TABLEGEN_PATH=$TOPO_TABLEGEN_SIMPLE_PATH
elif [[ "$router" == "ecmp" ]]; then
    TOPO_JSON_PATH=$TOPO_JSON_ECMP_PATH
    TOPO_TABLEGEN_PATH=$TOPO_TABLEGEN_ECMP_PATH
elif [[ "$router" == "ps" ]]; then
    TOPO_JSON_PATH=$TOPO_JSON_PS_PATH
    TOPO_TABLEGEN_PATH=$TOPO_TABLEGEN_PS_PATH
else
    echo "run.sh: error setting up router: unknown value \"$router\""
    exit 1
```

```
fi

# Quit the script if it is run with a post-processing script (--post) without pcap-logging
    enabled (--pcap).
if [[ ! -z "$post" ]] && [[ -z "$pcap" ]]; then
    echo "run.sh: can't run post-processing script without --pcap"
    exit 1
fi

# Create the argument file for the test file in JSON.
echo "{" > ../network-tests/logs/args.txt
echo "\"router\": \"$router\"," >> ../network-tests/logs/args.txt
echo "\"K\": \"$K\"," >> ../network-tests/logs/args.txt
echo "\"proto\": \"$proto\"," >> ../network-tests/logs/args.txt
echo "\"pmanager\": \"$pmanager\"," >> ../network-tests/logs/args.txt
echo "\"diffports\": \"$diffports\"," >> ../network-tests/logs/args.txt
echo "\"payloadsize\": \"$payloadsize\"," >> ../network-tests/logs/args.txt
echo "\"runcount\": \"$runcount\"" >> ../network-tests/logs/args.txt
echo "}" >> ../network-tests/logs/args.txt

# Set the environmental variables before running the topology generator.
TOPO_EXEC_PATH=$TOPO_EXEC_PATH
TOPO_CLI_PATH=$TOPO_CLI_PATH

# Then finally run the generator, passing fully all arguments.
    ./topogen.py "$@" --exec_path $TOPO_EXEC_PATH --json_path $TOPO_JSON_PATH
    --cli_path $TOPO_CLI_PATH --tablegen_path $TOPO_TABLEGEN_PATH || exit 1

# Clean the mess again after exiting, silently.
sudo mn -c &> /dev/null

# Run the postprocessing file, then delete all traces.
if [[ ! -z "$post" ]]; then
```

```
    sudo -u $SUDO_USER ./$post "$@" || exit 1
    rm s*.pcap
fi
```

Listing A.10: network-tests/test.py

```
from random import sample, choice
from os import path, makedirs
from time import sleep
import json
from subprocess import Popen, PIPE

# Requirements
## Note that network-tests and mininet-topo-generator should be in the same directory.

directory = "../network-tests/logs/"
filepath = directory + "args.txt"

with open(filepath, 'r') as jsonFile:
        args = json.load(jsonFile)



# Network configuration:
print "***␣Configuring␣network"
## Protocol --proto [(mptcp),tcp]
key = "net.mptcp.mptcp_enabled"
val = 1 if args['proto'] == "mptcp" else 0
p = Popen("sysctl␣-w␣%s=%s" % (key, val),
                shell=True, stdout=PIPE, stderr=PIPE)
stdout, stderr = p.communicate()
print stdout[:-1]
if len(stderr) != 0:
        print stderr

## Path manager --pmanager [(fullmesh),ndiffports]
if args['proto'] == "mptcp":
        key = "net.mptcp.mptcp_path_manager"
        val = args['pmanager']
        p = Popen("sysctl -w %s=%s" % (key, val),
                            shell=True, stdout=PIPE, stderr=PIPE)
        stdout, stderr = p.communicate()
        print stdout[:-1]
        if len(stderr) != 0:
                print stderr

        ## Ndiffports --diffports [(1)-16]
        if args['pmanager'] == "ndiffports":
                key = "echo " + args['diffports'] + " | tee /sys/module/mptcp_ndiffports/
    parameters/num_subflows"
                p = Popen(key, shell=True, stdout=PIPE, stderr=PIPE)
                stdout, stderr = p.communicate()
                print "/sys/module/mptcp_ndiffports/parameters/num_subflows =", stdout
    [:-1]
                print stderr
                if len(stderr) != 0:
                        print stderr

        print ""

## Payload Size --payloadsize [(query),short,long]
### Addendum: Quarter size lonf due to test time
if args['payloadsize'] == "query":
        payloadSize = "10K"
elif args['payloadsize'] == "short":
        payloadSize = "500K"
elif args['payloadsize'] == "long":
        payloadSize = "25M"

## Run count --runcount [(10),N]
runCount = int(args['runcount'])

```

```
# Generate randomized sender/receiver pairs.
length = len(net.hosts)
client = sample(xrange(length), length)

server = []
for each in range(0, length):
        server.append(choice([x for x in client if x not in server]))

        while server[-1] == client[len(server) - 1]:
                server[-1] = choice([x for x in client if x not in server])

print "*** Servers: " + str(server)
print "*** Clients: " + str(client)
print ""

# Iterate through the previously generated server/client pairs.
entries = []
for server, client in zip(server, client):
        # Start iperf on server host (non-blocking).
        serverCmd = "iperf -s &> /dev/null"
        net.hosts[server].sendCmd(serverCmd)

        results = []

        sleep(0.1)
        print "Testing server-client pair " + \
                str(net.hosts[server]) + " " + str(net.hosts[client])
        for each in range(0, runCount):
                clientCmd = "iperf -c" + net.hosts[server].IP() \
                        + " -n " + payloadSize + " -y c -x CSMV"

                results.append(net.hosts[client].cmd(clientCmd))
                sleep(0.1)
```

```


            # JSON FOR LIFE
            entry = { 'server': str(net.hosts[server]), 'client': str(net.hosts[client]), 'results': [] }
            for each in results:
                        entry['results'].append({ 'throughput': int(each.split(",")[-1][:-1].strip()), 'fct': 0 })
            entries.append(entry)

            net.hosts[server].sendInt()
            net.hosts[server].monitor()

# Write it into json dump middle file.
filepath = directory + "mid.json"
with open(filepath, 'w+') as jsonFile:
            json.dump(entries, jsonFile)

print ""
print "Test complete."
```

Listing A.11: network-tests/postprocess.py

```
#!/usr/bin/env python

#
    #################################################################

#   postprocess.py
#       Takes in .pcap files generated within Mininet, as well as iperf throughput results.
#       Follows this syntax:
#           ./postprocess.py
#       Make sure to set env.sh first before proceeding.
#
    #################################################################


import json
import re
from datetime import datetime
from os import listdir, makedirs
from os.path import isfile, join, abspath
from shlex import split
from shutil import copy
from subprocess import check_call, check_output, Popen, PIPE
from sys import exit

def unique(list_):
    # Python cheat to get all unique values in a list.
    return list(set(list_))

def time():
    # Python cheat to get time from Unix epoch
    return int(datetime.now().strftime("%s")) * 1000

```

```
topopath = abspath(".") # TODO change this omg
# topopath = abspath("../original-captures/") # TODO change this omg
logpath = abspath("../network-tests/logs/")
standardtime = time()
pcappath = abspath("../network-tests/logs/pcaps/pcap-{}".format(standardtime))
midfile = join(logpath, "mid.json")
argsfile = join(logpath, "args.txt")
aggregatefile = join(logpath, "aggregate.json")

def copyPcapFiles():
    # Move .pcap logs from topopath to logpath.
    print("*** Copying Mininet .pcap dumps.")
    # Make pcap directory.
    try:
        makedirs(pcappath)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise

    for file in [join(topopath, f) for f in listdir(topopath) if isfile(join(topopath, f)) and re.search(r'.pcap$', f, re.M)]:
        copy(file, pcappath)

def getInterfaces():
    # Infer all available interfaces using this code.
    # Flow: Get all files if it ends with ".pcap", strip and get the interface name (sxdd-ethd), then get all unique values.
    return unique([re.search(r'^(s[eac]\d\d-eth\d)', f, re.M).group(1) for f in listdir(pcappath) if isfile(join(pcappath, f)) and re.search(r'.pcap$', f, re.M) and re.search(r'^(s[eac]\d\d-eth\d)', f, re.M)])

def mergePcapFiles(interfaces):
    # Merge all _in and _out interfaces.
    print("*** Merging _in and _out pcap files.")
    try:
        [check_call(split("mergecap -w {1}/{0}.pcap {1}/{0}_in.pcap {1}/{0}_out.pcap".format(interface, pcappath))) for interface in interfaces]
    except:
        print("*** Failed to merge pcap files. Assuming already merged. Skipping.")

def deleteExcessPcapFiles(interfaces):
    # Delete all _in and _out interfaces, we don't need them anymore.
    print("*** Deleting _in and _out pcap files.")
    try:
        [check_call(split("rm {1}/{0}_in.pcap".format(interface, pcappath))) for interface in interfaces]
    except:
        print("*** Failed deleting *_in.pcap files. Assuming already deleted. Skipping.")

    try:
        [check_call(split("rm {1}/{0}_out.pcap".format(interface, pcappath))) for interface in interfaces]
    except:
        print("*** Failed deleting *_out.pcap files. Assuming already deleted. Skipping.")

def convertServerToIP(server):
    parsed = [int(x) for x in re.search(r'^h(\d)(\d)(\d)', server, re.M).groups()]
    return '10.{}.{}.{}'.format(parsed[0], parsed[1], parsed[2] + 2)

def getClientInterface(client):
    parsed = [int(x) for x in re.search(r'^h(\d)(\d)(\d)', client, re.M).groups()]
    return '{}/se{}{}-eth{}.pcap'.format(pcappath, parsed[0], parsed[1], parsed[2] + 1)

def includeFCT(entries):
    print("*** Extracting FCT from .pcap files.")
    for index, entry in enumerate(entries):
        fcts = Popen(["sh", "-c",
                "tshark␣-qz␣conv,tcp,ip.addr=={}␣-r␣{}␣|␣sed␣-e␣1,5d␣|␣head␣-n␣-1␣|␣sort␣
    -k␣10␣-n␣|␣awk␣-F'␣'␣'{{print␣$11}}'".format(
                    convertServerToIP(entry['server']),
                    getClientInterface(entry['client'])
                )], stdout=PIPE).communicate()[0].splitlines()

        for index_2, result in enumerate(entry['results']):
            result['fct'] = fcts[index_2]
            entries[index]['results'][index_2] = result

    return entries

def processJSONFiles():
    entries = None
    aggregate = None
    args = None

    with open(argsfile, 'r') as jsonFile:
        print("***␣Reading␣args.txt␣file␣from␣test␣script.")
        args = json.load(jsonFile)
        args['timestamp'] = standardtime
        print("***␣Using␣the␣following␣test␣arguments:")
        print(json.dumps(args, indent=4, sort_keys=True))

    with open(midfile, 'r') as jsonFile:
        print("***␣Reading␣mid.json␣file␣from␣test␣script.")
        entries = json.load(jsonFile)

    try:
        print("***␣Reading␣aggregate.json␣file.")
        with open(aggregatefile, 'r+') as jsonFile:
                        aggregate = json.load(jsonFile)
    except (ValueError, IOError) as e:
        print("*** Creating new aggregate.json file.")
        aggregate = []
        with open(aggregatefile, 'w+') as jsonFile:
                json.dump(aggregate, jsonFile)

    entries = includeFCT(entries)
    aggregate.append({ "metadata": args, "entries": entries })

    with open(aggregatefile, 'w+') as jsonFile:
        print("*** Writing aggregate.json file with FCTs.")
        json.dump(aggregate, jsonFile)

if '__main__' == __name__:
    print("")
    copyPcapFiles()
    interfaces = getInterfaces()
    mergePcapFiles(interfaces)
    deleteExcessPcapFiles(interfaces)
    processJSONFiles()
```

Appendix B

Experimental Results

B.1 Raw Results (Test Sets 1-6)

The test sets were defined at 4.6. Each test set was executed against the three flow types, iterating over all hosts in the network. During each iteration, each host must act as a server and a client eventually. This is done five times, and the data is collected into a JSON file. The raw data is available at `https://mmfsdt-aggregate-server.now.sh/aggregate_mar_9.json`.