Improving Data Center Network Performance through Path Diversity

Undergraduate Project Proposal

by

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

Kyle Dominic 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. To test this, we created a simulated datacenter environment wherein various tests were done to measure throughput, goodput, flow completion time, and mean network utilization. Different parameters were changed during the tests to gather data on their respective performance. These parameters are payload size, network traffic conditions, switch forwarding protocol, and reorder resiliency in hosts. Analyzing the data, the study found that packet spraying turned out to be beneficial for the network in terms of throughput, goodput, and mean network utilization. Reorder resiliency, on the other hand, resulted in a decrease in the performance of the network.

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. Since packet spraying can introduce an eventual decrease in throughput, we hypothesize that this can be mitigated by creating reorder-resilient end hosts. 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

Taking into consideration the penalties incurred on short flows from MPTCP connection establishment complexity, as well as the penalties incurred on long flows due to the network hotspots that arise as a consequence of ECMP routing, we chose to measure flow completion time, and end-host throughput and goodput, respectively. Mean network utilization was also measured to gauge the effectivity of each protocol in terms of how well the available network routes were used.

To properly attribute each change in the above metrics, several possible permutations of the routing behavior, transport protocol, flow types (or payload sizes) end-host pairs, and network conditions were tested. These metrics were extracted and compared against each other considering the varying parameters between each test.

4.1 Preliminary Work

Smaller tests using MPTCP were done to further increase the researcher's understanding of the protocol and its interaction with Mininet. MPTCP was confirmed to be working as indicated in the laboratory experiment of an SDN class [38]. An overview of the topology can be found in Figure 4.1. To simplify things, router `r2` is configured to forward packets, acting as a router[1]. All links are capped at a speed of 10 Mbps (i.e., the minimum supported transmission rate of both devices connected to the link is 10 Mbps).

[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.

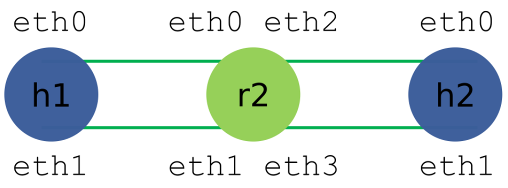

Figure 4.1: Mininet topology for MPTCP tests. Links are of 10 Mbps speed. Topology copied from [38]

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.

4.1.1 Managing Multipath TCP Subflows

In the preliminary setup and tests, it was found that MPTCP can be configured using different parameters. MPTCP handles the discovery, setup, and termination of subflows through the heuristics of a path manager [18]. We considered three of the four available path managers that MPTCP provides [39].

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 [40]) of both 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.

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*.

4.1.2 Validating the Behavior of MPTCP Path Managers

Considering the topology described in Figure 4.1, 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.

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,

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.

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.

4.2 Building the Test Environment

Mininet was chosen to orchestrate the virtual network, its switches, and its hosts. The network topology is generated programmatically, and switch behavior can be swapped through configuration files. It also ensures that all virtual hosts are running on the same configuration as the server it is running in. This ease of use allows the researchers to implement multiple changes at once.

MPTCP is available as an ns-3 model [41] or a kernel patch [42]. Juggler [1], which modifies the GRO function to fix packet ordering, is only readily available as a kernel patch. Since the experiments require hosts that require both features to be present, it was decided that the kernel patches were to be used. The MPTCP and Juggler custom kernels were merged together and applied to the server to virtualize certain hosts.

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. Since these forwarding behavior require inspection into packet headers and metadata, we utilized and modified software switches enabled by P4, a programming language used to process packets [43].

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.

4.2.1 Testbed Specifications

Experiments and tests were ran on two SuperMicro bare metal servers with the specifications listed below. One server will be patched with the MPTCP custom kernel only, and the other with the merged MPTCP and Juggler kernel.

4.2.2 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 Figure 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.

4.2.2.1 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.

4.2.2.2 Indexes and Names

The following numbering and naming conventions are used.

- The pods are numbered for 0 to $K - 1$.

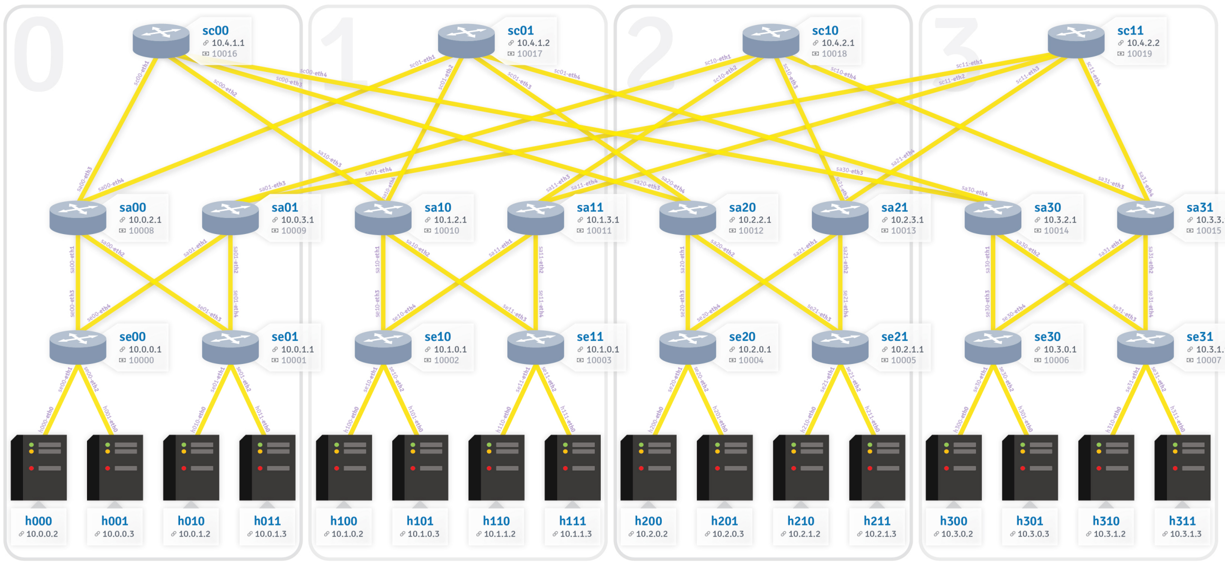

Figure 4.2: Fat tree (`k = 4`) topology in Mininet, including naming and addressing conventions.

- 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.

4.2.2.3 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>`.

4.2.2.4 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 $0 \leq pod \leq K - 1$.

4.2.2.5 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 $K - 1$.

4.2.2.6 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>`.

4.2.3 Switch Behavior

The P4 `behavioral-model` repository [44] 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 [45]. The JSON is fed to the target executable, and the tables are filled with entries during initialization.

4.2.3.1 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 switch 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 switch behaviors may differ in how packets are forwarded upstream. For this experiment, three switch 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.

4.2.3.2 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.

4.2.3.3 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.

4.2.3.4 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 [46], 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 [44] 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.

4.3 Test Design

To review, the goals of the tests are to measure the network performance given varying configurations as previously discussed. In particular, the study will test mainly throughput and flow completion time, with goodput and mean network utilization as extra data points.

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 edge and aggregate layers, and 4 switches in the core (see Figure 4.2).

With this in mind, this allows for 4 unique paths between two hosts.

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

Considering that each host uses only one interface to communicate to the entire network, preference was given to using `ndiffports` (with $n = 4$) as MPTCP's path manager in this experiment. This can be done through the `sysctl` feature. 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`. In this case, we set the path manager of MPTCP-enabled hosts to `ndiffports`.

4.3.1 Network Traffic Conditions

Tests were done in two simulated network conditions. The first condition assumed a network without any activity, allowing for two hosts to communicate using all of the available resources of the network (hereinafter referred to as a *silent network*). The second condition assumed some form of simulated traffic, comparable to the performance in a live network (hereinafter referred to as a *noisy network*).

These were done to observe the changes in the design metrics in different network conditions. Note that the simulation of traffic was an approximation and may not be representative of the actual datacenter network traffic.

In the *silent network*, host pairs took turns at sending data to each other, without any other activity in the network. Pairs were chosen by random, ensuring that all hosts became a server or client at some point. The clients requested a payload of certain size from the server, one after another. This request was then repeated a certain number of times. The designations were fixed for each iteration of the test. This was done through a network bandwidth measurement tool named *iperf*.

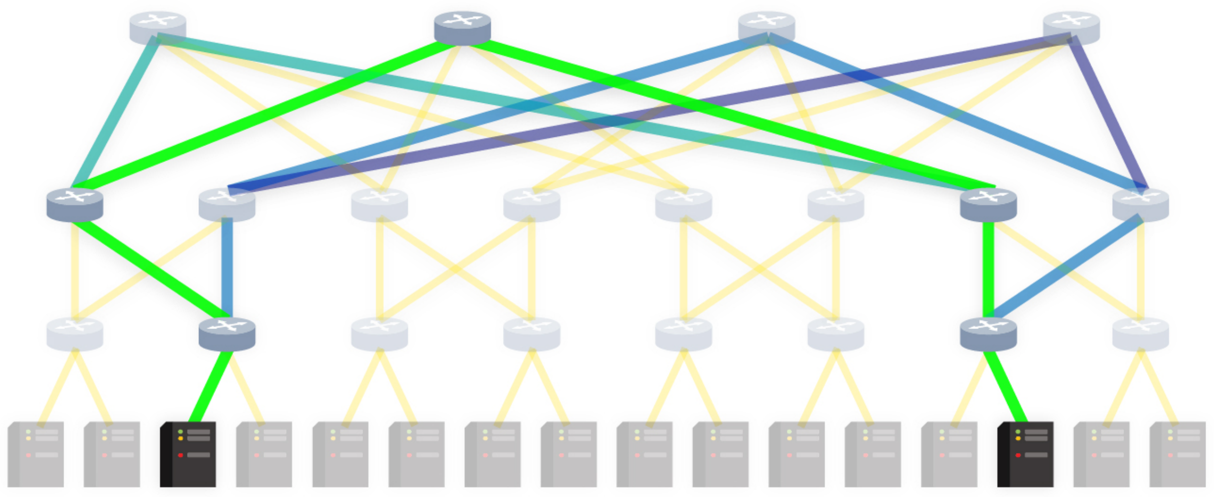

Figure 4.3: Two hosts communicating in a *silent network* setup. Note that there are no other network activity, and that there are four paths between the two hosts (indicated by the green ang bluish lines).

In the *noisy network*, two hosts requested data from multiple hosts at the same time, polluting the network with activity. The network was split between two groups with one client each, and the rest acting as servers. Considering again a fat tree topology with $K = 4$, one client was chosen at random at the start of the test and seven were chosen to be file servers. Like the *silent network*, designations were fixed for each iteration of the test. This was done through an *HTTP* server in Python, and a tool to transfer data called *curl*.

In both network conditions, multiple server-client pairs were used to obtain a thorough assessment of the different links in the network. Analysis was done through inspection of the packet capture output of Mininet. Each test is also repeated to obtain a sufficient number of data points.

4.3.2 Test Parameters

The testbed can have varying permutations of network and host configurations. To aid in this, the test can be run using different testing parameters namely, host reorder resiliency, switch behavior, transport protocol, and payload sizes.

Datacenters cater to requests of varying sizes. To have an approximation of these requests and how they affect the performance of the changes to the network stack, we included the option to test with different payload sizes. Three flow types are to be used and tested for these experiments: `query`, `short`, and `long` flows [6].

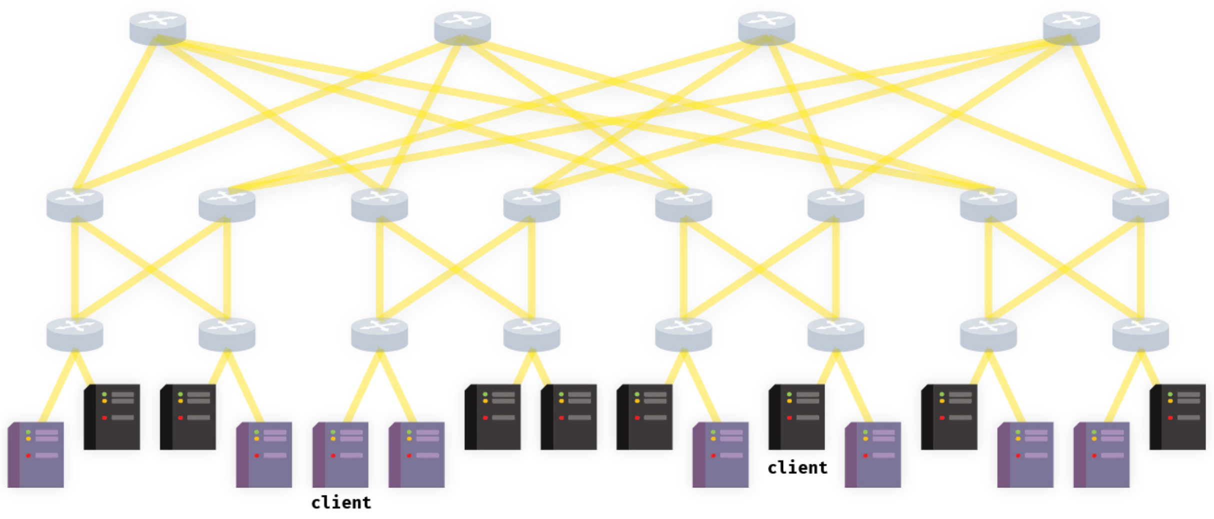

Figure 4.4: Two host groups in a *noisy network* setup, colored in black and purple. Note that one host in each group is designated as the client, to communicate with the rest of the group.

Flow	Examples	Silent Network	Noisy Network
query	Quick, bursty connections	128 kiB	10 kiB
short	Web page requests	500 kiB	500 kiB
long	Streaming, VoIP, file hosting and transfer	25 MiB	25 MiB

Table 4.5: Flow definitions, examples, and their corresponding sizes on both simulated network traffic conditions.

Note that for the silent network, a query flow is defined with a flow size of 128KB as the tool used did not allow for the payload size to go below 128KB.

As a baseline comparison for the improvements of MPTCP, a TCP option was included in the testing environment. These results served as a form of control for the performance of MPTCP as a protocol. In addition, the performance of the changes in the network stack with TCP as its transport protocol was also measured and analyzed.

4.3.3 Performance Metrics

To review, throughput, goodput, flow completion time, and mean network utilization between end-hosts were measured to assess the effects of reorder resiliency in end-hosts and packet spraying in routers.

			Test Clusters					
			Silent Network (1-1)			Noisy Network (1-many)		
Independent Variables	`ecmp-mptcp`	`vanilla`	query	short	long	query	short	long
		`juggler`	query	short	long	query	short	long
	`ps-mptcp`	`vanilla`	query	short	long	query	short	long
		`juggler`	query	short	long	query	short	long

Table 4.6: Test clusters and independent variables used in this experiment.

4.3.3.1 Throughput and Goodput

We define throughput to be the rate of the bytes transferred between two hosts in the network (i.e., total bytes transferred / flow completion time). In contrast, goodput is the rate of the actual data (only the size of the payload) transferred over a period of time (i.e., payload size in bytes / flow completion time).

A comparison between goodput and throughput can be used to indicate how much overhead affects the rate of transmission of actual data. If throughput is higher than goodput, this can be an indication to the presence of added MPTCP headers or packet retransmissions. This translates to no perceptible improvement from the perspective of the hosts.

4.3.3.2 Flow Completion Time

We define flow completion time to be the amount of time that it takes for a request between two hosts to complete. This includes the first request packet sent from the client to the server and the last `FIN` packet sent from server to the client.

An increase in flow completion time can be attributed to the overhead of connection establishment (for MPTCP), packet retransmissions, and/or `ACK` timeouts.

4.3.3.3 Mean Network Utilization

For this paper, we consider mean network utilization to be the ratio of used links between two hosts. For a fat tree network topology with $K = 4$, there are at most 4 paths that connect two hosts. Ideally, mean network utilization is maximized if all of paths have the near-equal (if not equal) minimum amount of bytes transferred in them (i.e., all paths have a balanced load with regards to the flow).

4.3.4 Statistical Tests

All in all, the test clusters and independent variables can be summarized with the following table.

There are two independent variables: the switch behavior combined with the underlying transport protocol (`ecmp-mptcp`, `ps-mptcp`), as well as the inclusion/non-inclusion of Juggler (`juggler`, `vanilla` respectively). There are six test clusters, grouped by network traffic conditions (*silent network*, *noisy network*), permuted with the three payload sizes (`query`, `short`, `long`).

There are four dependent variables pertaining to the performance metrics discussed previously: throughput, goodput, flow completion time, and mean network utilization.

Each test cluster was subjected to a two-way MANOVA to verify the significance of the differences, and ranked via comparison of means. In addition, a comparison of percent differences of goodput and throughput was executed to inspect the effects of hotspots in the network.

Another experiment was also executed that includes TCP alongside MPTCP (the addition of `static-tcp` and `ps-tcp`), to serve as control. For the majority of the next chapter, we will be discussing the experiment concerning MPTCP tests only.

Chapter 5

Results and Analysis

A proper multivariate analysis of variance (MANOVA) shall be executed by meeting the following assumptions [47]: dependent variables should be continuous, independent variables should consist of two or more categorical groups, there should be independence between observations, and there should be a sufficient number of data points. These four assumptions were met by virtue of the experimental setup.

Other assumptions for MANOVA, if unmet, reduce the validity of the results. Some tests show that our data set does not meet some of these assumptions, which means that the following results should be interpreted with reservations.

5.1 Tests of Between-Subjects Effects and Best Performing Independent Variables

We chose a confidence interval of 95% ($\alpha = 0.05$) to determine the significance of the various test groups. For this chapter, we will be focusing solely on the tests of between-subjects effects. Table 5.1 shows the values for each test group. We then listed the independent variables (Table 5.2) with the highest mean for outcomes (Appendix __) that have a significant result.

For a given link in the network, bandwidth is limited to 20 Mbps. Tests within a silent network, specifically for throughput, goodput, and flow completion time are inconclusive as they can maximize the bandwidth regardless of switch behavior and transport protocol. The throughput and goodput for a given flow naturally reach the bandwidth cap in a silent network as there are no other flows competing for resources.

Test Clusters		Independent Variables	Dependent Variables			
Network Traffic Conditions	Flow Type		Goodput	Throughput	Flow Completion Time	Mean Network Utilization
Silent Network (1-1)	query	Switch Behavior + Transport Protocol	0.640	0.888	0.310	**0.000**
		Inclusion of Juggler	**0.000**	**0.000**	**0.000**	0.675
		Combined Interaction	**0.010**	0.110	**0.009**	0.230
	short	Switch Behavior + Transport Protocol	**0.007**	0.571	0.057	**0.000**
		Inclusion of Juggler	**0.000**	**0.040**	**0.000**	0.133
		Combined Interaction	0.259	0.623	0.748	0.600
	long	Switch Behavior + Transport Protocol	**0.015**	0.895	0.479	**0.000**
		Inclusion of Juggler	**0.000**	0.238	0.171	**0.001**
		Combined Interaction	**0.000**	**0.000**	0.541	**0.001**
Noisy Network (1-many)	query	Switch Behavior + Transport Protocol	**0.023**	**0.008**	0.127	**0.000**
		Inclusion of Juggler	**0.019**	0.139	0.821	0.805
		Combined Interaction	0.061	0.249	0.063	0.843
	short	Switch Behavior + Transport Protocol	**0.000**	**0.000**	**0.000**	**0.000**
		Inclusion of Juggler	**0.001**	**0.000**	**0.000**	0.126
		Combined Interaction	**0.005**	**0.000**	**0.000**	0.150
	long	Switch Behavior + Transport Protocol	0.113	**0.000**	**0.008**	**0.000**
		Inclusion of Juggler	0.374	**0.014**	0.716	0.137
		Combined Interaction	0.545	0.309	0.589	0.232

Table 5.1: Significance values of each test group in the two-way MANOVA. Significant tests are marked in bold.

Test Clusters		Independent Variables	Dependent Variables			
Network Traffic Conditions	Flow Type		Goodput	Throughput	Flow Completion Time	Mean Network Utilization
Silent Network (1-1)	query	Switch Behavior + Transport Protocol				ps-mptcp
		Inclusion of Juggler	vanilla	vanilla	vanilla	
		Combined Interaction	ps-mptcp-vanilla		ps-mptcp-vanilla	
	short	Switch Behavior + Transport Protocol	ps-mptcp			ps-mptcp
		Inclusion of Juggler	vanilla	juggler	vanilla	
		Combined Interaction				
	long	Switch Behavior + Transport Protocol	ps-mptcp			ps-mptcp
		Inclusion of Juggler	juggler			juggler
		Combined Interaction	ecmp-mptcp-juggler	ps-mptcp-juggler		ps-mptcp
Noisy Network (1-many)	query	Switch Behavior + Transport Protocol	ps-mptcp	ps-mptcp		ps-mptcp
		Inclusion of Juggler	juggler			
		Combined Interaction				
	short	Switch Behavior + Transport Protocol	ps-mptcp	ps-mptcp	ps-mptcp	ps-mptcp
		Inclusion of Juggler	vanilla	juggler	vanilla	
		Combined Interaction	ps-mptcp-vanilla	ps-mptcp-juggler	ps-mptcp-vanilla	
	long	Switch Behavior + Transport Protocol		ps-mptcp	ps-mptcp	ps-mptcp
		Inclusion of Juggler	vanilla		vanilla	
		Combined Interaction				

Table 5.2: Best performing configurations, with non-significant results filtered out.

Network Traffic Conditions	Flow Type	Switch Behavior and Transport Protocol			
		Statically-configured switches (TCP)	Packet Spraying-based switches (TCP)	ECMP-based switches (MPTCP)	Packet Spraying-based switches (MPTCP)
Silent Network (1-1)	query	8.99%	10.54%	12.96%	13.39%
	short	9.68%	11.41%	18.62%	17.50%
	long	31.69%	30.18%	35.44%	34.47%
Noisy Network (1-many)	query	16.29%	23.61%	31.25%	31.65%
	short	32.05%	43.71%	47.69%	47.71%
	long	51.16%	54.37%	57.80%	58.10%

Table 5.3: Percentage of traffic in excess of ”good” traffic (i.e., actual payload size).

5.2 Gap Between Goodput and Throughput as an Indicator of Network Hotspots

Network hotspots, defined as switches that have links that are near to exceed their capacity, result in an increase of dropped packets and increased packet retransmissions. This results in an increase in flow completion time, as well as the total amount of bytes transferred. Therefore, a significant gap between throughput and goodput may be noticed if network hotspots are in the network. In this experiment, we took the observed means of goodput and throughput for statically-configured switches (with hosts running TCP), ECMP-based switches (with hosts running MPTCP), and PS-based switches (with hosts running TCP and hosts running MPTCP) and calculated the percentage of traffic in excess of the actual flow size.

If we assume that the percentage of excess traffic is caused by retransmission, then we can see that statically-configured switches has significantly less retransmissions over both ECMP-based switches and PS-based switches for query and short flows. However, during long flows, the difference between all the switch behaviors seem insignificant. In all cases, ECMP-based switches and PS-based switches (with hosts running MPTCP) performed comparably equally, implying that packet spraying did not mitigate the hotspot problem.

5.3 Flow Completion Time Performed Worse with Reorder-Resilient Hosts; Inconclusive Effects on Throughput

Since Juggler, the tool to enable reorder-resiliency in the hosts, was created to increase tolerance between TCP packet arrival times, it might not perform as well with MPTCP. Based on our results, Juggler had a negative effect on flow completion time for a silent network with query and short flows, as well as for a noisy network for short flows.

This can also explain the inconclusive results for throughput. Results are mixed, with Juggler increasing throughput for short flows (on both silent and noisy networks), but failing to increase throughput for query flows on a silent network and long flows on a noisy network.

This leads us to believe that Juggler cannot be simply placed in a network without negatively affecting its performance.

5.4 Packet Spraying Generally Performed Better than ECMP

As for the switch behavior, packet spraying increased mean network utilization for all test groups compared to ECMP. In addition, packet spraying performed better than ECMP for goodput and throughput tests in a noisy environment.

This behavior may be preferred as most datacenter networks are high in network activity.

5.5 Effects of Combining Packet Spraying-based Switches with Reorder-Resilient Hosts

A combination of packet spraying and reorder resiliency improves throughput for both silent and noisy networks. However flow completion time increases (due to the presence of Juggler as seen in the Juggler-only analysis).

Goodput results turn out to be inconclusive as tests performed better with PS-based switches, but some preferred the inclusion of Juggler while others don't. Mean network utilization remains significantly higher when compared to the rest which can be attributed to the packet spraying behavior.

Chapter 6

Conclusion and Recommendations

Based on preliminary tests, we found that MPTCP increases throughput significantly compared to TCP, especially with using multiple paths. However the connection establishment overhead needed to create multiple subflows penalizes the speed at which flows are created and consequently completed.

In addition, the paper promised to experimentally prove that ECMP-based switches causes hotspots in the network. By comparing the throughput and goodput of ECMP-based switches without reorder resiliency, we saw that there are was no improvements to the hotspots in the network. However hotspots in the network may be a consequence of the experiment setup. This could either be attributed to the amount of traffic in the noisy network or due to the limited bandwidth at the single interface of the host for the silent network.

It was also found that packet spraying improves throughput, goodput, and mean network utilization for a noisy environment. Considering that most datacenters will, more often than not, operate with a lot of concurrent flows in the network, packet spraying may be beneficial.

Packet spraying and reorder resiliency show a significant difference between groups (except for a noisy network with long and query-length flows and silent network with short flows). Looking at a noisy network with short flows, throughput is improved and flow completion time worsens.

From this, we can say that packet spraying is better in terms of throughput, goodput, and mean network utilization. Flow completion time for packet spraying turned out inconclusive data.

6.1 Recommendations and Future Work

Further analysis may be done on TCP combined with packet spraying and reorder resilient hosts. According to initial testing, TCP with reorder resilient hosts and packet spraying showed an increase in throughput and goodput. This may be promising as there would be no need to implement a new transport protocol.

Network hotspots may also be studied further by creating a special testbed specifically built to measure the amount of traffic going through specific nodes in the network. We hypothesize that packet spraying may still improve network hotspots caused by ECMP.

Data collected in an datacenter network with real traffic would be beneficial for the validity of the tests since the simulated traffic in this experiment is merely an approximation.

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

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

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

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

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

[43] 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.

[44] 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.

[45] 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.

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

[47] "How to perform a two-way MANOVA in SPSS Statistics | Laerd Statistics."

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}]
#               [−−pcap]
#               [−−K K {4}]
#                                   [−−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('--K', default='4', type=int, metavar='num_ports', help='number of
    ports per switch. defaults to 4.')
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 = True)
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 = True)
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 = True)
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': 20}

    #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], cls=TCLink,**linkopt)

    #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], cls=TCLink,**linkopt)

```

```
#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], cls=TCLink,**linkopt)


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:
#          sudo ./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}]
#              [--juggler]
#              [--payloadsize query|long|short  {short}]
#              [--runcount num_counts {10}]
#              [--mode onetoone|onetomany {onetoone}]
#      Make sure to set env.sh  first  before  proceeding.
#
  #################################################################


# *** Check arguments before script execution ***
# Quit the program if it  wasn't executed  as  root.
if  [[  $EUID -ne 0 ]]; then
```

```
    echo "[Error] run.sh must be executed as root. Quitting."
    exit 1
fi

# Check the curl version.
# TODO: Add argument skipping this test.

latestVer=$(curl -s 'https://curl.haxx.se/download/' | grep -oP 'href="curl-\K
    [0-9]+\.[0-9]+\.[0-9]+' | sort -t. -rn -k1,1 -k2,2 -k3,3 | head -1)
installedVer=$(curl -V | grep -oP 'curl \K[0-9]+\.[0-9]+\.[0-9]+')

if [[ $latestVer != $installedVer ]]; then
    echo "Installing curl $latestVer"
    sudo apt-get build-dep curl
    mkdir ~/curl
    pushd ~/curl
    wget http://curl.haxx.se/download/curl-$latestVer.tar.bz2
    tar -xvjf curl-$latestVer.tar.bz2
    cd curl-$latestVer

    ./configure
    make
    sudo make install

    sudo ldconfig
    popd
fi


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

```

# 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;;
    --juggler) juggler="true";;
    --payloadsize) payloadsize="$2"; shift;;
    --runcount) runcount="$2"; shift;;
    --mode) mode="$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 "$runcount" ]]; then runcount="10"; fi
if [[ -z "$mode" ]]; then mode="onetoone"; fi
```

```

# Quit the script if it is run with K < 4 and mode of 'onetomany'.
if (( K < 4 )) && [[ "$mode" == "onetomany" ]]; then
    echo "run.sh: onetomany does not work on K < 4"
    exit 1
fi

# Set the TOPO_JSON and TOPO_TABLEGEN paths accordingly, quit if incorrect value.
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

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

# Clean old .pcap traces in case of errors.
rm s*.pcap &> /dev/null

# 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

# 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

if [[ ! -z "$juggler" ]]; then
    echo "\"juggler\": \"true\"," >> ../network-tests/logs/args.txt
else
    echo "\"juggler\": \"false\"," >> ../network-tests/logs/args.txt
fi

if [[ ! -z "$pcap" ]]; then
    echo "\"pcap\": \"true\"," >> ../network-tests/logs/args.txt
else
    echo "\"pcap\": \"false\"," >> ../network-tests/logs/args.txt
fi

echo "\"payloadsize\": \"$payloadsize\"," >> ../network-tests/logs/args.txt
echo "\"runcount\": \"$runcount\"," >> ../network-tests/logs/args.txt
echo "\"mode\": \"$mode\"" >> ../network-tests/logs/args.txt
echo "}" >> ../network-tests/logs/args.txt

# Prepare topogen.py arguments
TOPOGEN_ARGS=(--exec_path $TOPO_EXEC_PATH --json_path $TOPO_JSON_PATH
    --cli_path $TOPO_CLI_PATH --tablegen_path $TOPO_TABLEGEN_PATH --K $K)
if [[ ! -z "$test" ]]; then TOPOGEN_ARGS+=(--test $test); fi

# Remove previous mid.json
rm ../network-tests/logs/mid.json

# Finally, run topogen.py with the arguments specified above.
./topogen.py ${TOPOGEN_ARGS[*]} || exit 1

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

# If mid.json wasn't written, assume the test failed.
if [ ! -f ../network-tests/logs/mid.json ] && [[ ! -z "$post" ]]; then
    echo "Network testing failed."
    exit 1
fi

# Remove the /tmp/ trash from onetomany. Uncomment this if necessary.
if [[ "$mode" == "onetomany" ]]; then
    rm /tmp/mmfsdt-*
fi

# Fix the file ownership of the *.pcap at log files.
chown $SUDO_USER:$SUDO_USER ../network-tests/logs/args.txt
chown $SUDO_USER:$SUDO_USER ../network-tests/logs/aggregate.db
chown $SUDO_USER:$SUDO_USER ../network-tests/logs/mid.json
chown $SUDO_USER:$SUDO_USER s*.pcap
```

```

# Run the postprocessing file, then delete all traces.
if [[ ! -z "$post" ]]; then
    sudo -u $SUDO_USER ./$post "$@" || exit 1
    rm s*.pcap &> /dev/null
fi
```

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

```
# from os import path, makedirs, environ
# from random import sample, choice, uniform
from random import sample, choice
from subprocess import Popen, PIPE, STDOUT
from time import sleep, time
# from mininet.util import pmonitor
import shlex
import json

# Configuration
# 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 not stderr:
    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 not stderr:
        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 not stderr:
            print stderr

    print ""

# Payload Size --payloadsize [(query),short,long]
# Addendum: Quarter size lonf due to test time
# Set maximum time for test
# Note that these payloads are in MiB, KiB
if args['payloadsize'] == "query":
    payloadSize = "10K"
elif args['payloadsize'] == "short":
    payloadSize = "500K"
elif args['payloadsize'] == "long":
    payloadSize = "25M"

```

```
# Extract args
mode = args['mode']
runCount = int(args['runcount'])
K = int(args['K'])

print "*** running test mode {}".format(mode)


def getPort(currentRun):
    startRangePort = 50000
    return startRangePort + currentRun


length = len(net.hosts)

# We might have to log these down into another log file later on for parsing.
# Indicate the pairing, the time executed, and other pertinent details.

if mode == "onetomany":
    # Generate randomized sender/receiver arrays.
    # two hosts will be isolated in its own array, the rest will be in another
    if K < 4:
        raise ValueError('### ERROR: onetomany does not work on K < 4.')

    print "*** changing host directories to ../network-tests/files\n"
    for host in range(0, length):
        if net.hosts[host].cmd("pwd")[-5:] != "files":
            cmd = "cd ../network-tests/files"
            net.hosts[host].cmd(cmd)

    pickList = []
    restList = []

```

```
for x in range(0, runCount):
    pick = sample(range(0, length), 2)
    pickList.append(pick)

    temp = sample([x for x in range(0, length)
                   if x not in pick], (length - 2) / 2)
    restList.append(
        [temp, [x for x in range(0, length) if x not in pick + temp]])

for run, (pick, rest) in enumerate(zip(pickList, restList)):
    print "*** run {}".format(run)
    for i, each in enumerate(pick):
        print str(net.hosts[each]) + " : " + \
            str([str(net.hosts[x]) for x in rest[i]])
    print ""

entries = []
timestamp = int(time())

for run, (pick, rest) in enumerate(zip(pickList, restList)):
    print "*** Run #{}: starting HTTP Python servers\n".format(run)

    # Prepare the command to run within the nodes.
    # The first few spaces in the beginning is necessary as either Bash or Python
removes the first character sometimes.
    cmd = "  python -m SimpleHTTPServer {} &".format(getPort(run))
    # sendCmd() and waitOutput() runs and prints the script output.
    expectedOutput = "Serving HTTP on 0.0.0.0 port {} ...".format(getPort(run))

    for servers in rest:
        for host in servers:
            # We'll be using a simple backoff algorithm to ensure that our servers are
running.
```

```
            backoff = 0.1
            retries = 0
            maxRetries = 3

            while retries < maxRetries:
                net.hosts[host].sendCmd(cmd)
                sleep(backoff * (retries + 1))

                output = net.hosts[host].waitOutput()

                # Uncomment these lines for further debugging.
                # print output
                # print expectedOutput in output

                if expectedOutput in output:
                    print "Host {} is up.".format(net.hosts[host])
                    break
                else:
                    # Print the error message if it fails.
                    print output
                    retries = retries + 1

            # If the error persists, stop testing, and mark the test as a failure.
            if retries == maxRetries:
                raise ValueError("Can't start server within node.")

    print "\n*** Run #{}: sending requests".format(run)

    client_popens = []
    for host1, host2 in zip(rest[0], rest[1]):
        # I'm sorry Kyle. I switched to curl because I thought wget wasn't working
properly.
        cmd1 = "curl -v http://{}:{}/{}.out --retry 3 --retry-connrefused -o /tmp/
```

```
mmfsdt-{}-run{}-{}-{}".format(net.hosts[host1].IP(), getPort(run), args['payloadsize'], timestamp, run, args['payloadsize'], net.hosts[host1])
        cmd2 = "curl -v http://{}:{}/{}.out --retry 3 --retry-connrefused -o /tmp/mmfsdt-{}-run{}-{}-{}".format(net.hosts[host2].IP(), getPort(run), args['payloadsize'], timestamp, run, args['payloadsize'], net.hosts[host2])

        client_popens.append({"from": net.hosts[pick [0]], "to": net.hosts[host1], "command": cmd1, "process": net.hosts[pick[0]].popen(shlex.split (cmd1), stderr=STDOUT)})
        client_popens.append({"from": net.hosts[pick [1]], "to": net.hosts[host2], "command": cmd2, "process": net.hosts[pick[1]].popen(shlex.split (cmd2), stderr=STDOUT)})

        print "sent request to {} and {}".format(net.hosts[host1], net.hosts[host2])

    print "\n*** Run #{}: waiting for clients to finish request".format(run)

    for p in client_popens:
        (out, err) = p["process"].communicate()
        for item in out.split ("\n"):
            if "curl: " in item:
                print " !!! Download error."
                print item.strip ()
                print ""
                print "*** *** Output"
                print out.strip ()
                print ""
                print ">>> >>> from: {}; to: {}; command: {}".format(p["from"], p["to"], p["command"])
                raise ValueError("Download error. Retry test.")

    print "*** Run #{}: terminating simple python servers\n\n".format(run)
    for servers in rest:
            for host in servers:
                net.hosts[host].sendInt()
                net.hosts[host].monitor()

        # Format the results into a json format
        # Load the first pick and its paired rests
        for server in rest[0]:
            # print server
            entry = {'serverName': str(net.hosts[server]), 'serverIP': str(net.hosts[server].
    IP()),
                     'clientName': str(net.hosts[pick[0]]), 'clientIP': str(net.hosts[pick
    [0]].IP()),
                     'run': str(getPort(run))
                    }
            # print entry
            entries.append(entry)

        for server in rest[1]:
            entry = {'serverName': str(net.hosts[server]), 'serverIP': str(net.hosts[server].
    IP()),
                     'clientName': str(net.hosts[pick[1]]), 'clientIP': str(net.hosts[pick
    [1]].IP()),
                     'run': str(getPort(run))
                    }
            entries.append(entry)

    # print entries

elif mode == "onetoone":
    # Generate randomized sender/receiver pairs.
    clients = sample(xrange(length), length)

    servers = []
```

```
for each in range(0, length):
    servers.append(choice([x for x in clients if x not in servers]))

    while servers[-1] == clients[len(servers) - 1]:
        # Fix list index error.
        choices = [x for x in clients if x not in servers]
        if not choices:
            servers[-2], servers[-1] = servers[-1], servers[-2]
        else:
            servers[-1] = choice(choices)

print "***␣Servers:␣" + str(servers)
print "***␣Clients:␣" + str(clients)
print ""

# Iterate through the previously generated server/client pairs.
entries = []

for run in range(0, runCount):
    print "***␣Run␣#{}:␣starting␣iperf␣servers.".format(run)

    # Prepare the command to run within the nodes.
    # The first few spaces in the beginning is necessary as either Bash or Python removes the first character sometimes.
    serverCmd = "␣␣iperf␣-s␣-p␣{}␣&".format(getPort(run))
    # sendCmd() and waitOutput() runs and prints the script output.
    expectedOutput = "Server␣listening␣on␣TCP␣port␣{}".format(getPort(run))

    for server in servers:
        # We'll be using a simple backoff algorithm to ensure that our servers are running.
        backoff = 0.1
        retries = 0
```

```
            maxRetries = 3

            while retries < maxRetries:
                net.hosts[server].sendCmd(serverCmd)
                sleep(backoff * (retries + 1))

                output = net.hosts[server].waitOutput()

                # Uncomment these lines for further debugging.
                # print output
                # print expectedOutput in output

                if expectedOutput in output:
                    print "Host {} is up.".format(net.hosts[server])
                    break
                else:
                    # Print the error message if it fails.
                    print output
                    retries = retries + 1

            # If the error persists, stop testing, and mark the test as a failure.
            if retries == maxRetries:
                raise ValueError("Can't start server within node.")

        # # Start iperf on server on port <run>
        # for server in servers:
        #     serverCmd = "iperf -s -p {} &".format(getPort(run))
        #     # serverCmd = "iperf -s -p {} &> /dev/null".format(getPort(run))
        #     net.hosts[server].sendCmd(serverCmd)
        #     print net.hosts[server].waitOutput()
        #     sleep(0.1)

        print "*** Run #{}: sending requests".format(run)
```

```

    for server, client in zip(servers, clients):

        # Prepare the command to run within the nodes.
        # The first few spaces in the beginning is necessary as either Bash or Python
  removes the first character sometimes.
        clientCmd = "iperf -c {} -n {} -p {} -y c -x CSMV".format(net.hosts[server
  ].IP(), payloadSize, getPort(run))
        # sendCmd() and waitOutput() runs and prints the script output.
        expectedOutput = "Server listening on TCP port {}".format(getPort(run))

        # We'll be using a simple backoff algorithm to ensure that our servers are running
  .
        backoff = 0.1
        retries = 0
        maxRetries = 3

        while retries < maxRetries:
            net.hosts[client].sendCmd(clientCmd)
            sleep(backoff * (retries + 1))

            output = net.hosts[client].waitOutput()

            # Uncomment these lines for further debugging.
            # print output
            # print expectedOutput in output

            if len([i for i, letter in enumerate(output) if letter == ","]) == 8:
                print "{}-{} succeeeded.".format(net.hosts[server], net.hosts[client])
                break
            else:
                # Print the error message if it fails.
                print output
```

```
                    retries = retries + 1

            # If the error persists, stop testing, and mark the test as a failure.
            if retries == maxRetries:
                raise ValueError("Failed iperf with {}-{}".format(net.hosts[server], net.hosts
[client]))

            entry = {'serverName': str(net.hosts[server]), 'serverIP': str(net.hosts[server].
IP()),
                     'clientName': str(net.hosts[client]), 'clientIP': str(net.hosts[client].
IP()),
                     'run': str(getPort(run))
                    }
            entries.append(entry)

        print "*** Run #{}: quitting iperf servers".format(run)
        for server in servers:
            net.hosts[server].sendInt()
            net.hosts[server].monitor()

# 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)

    #       # Format the results into a json format
    #       entry = {'serverName': str(net.hosts[server]), 'serverIP': str(net.hosts[server].IP
    ()),
    #               'clientName': str(net.hosts[client]), 'clientIP': str(net.hosts[client].IP
    ()),
    #               'results': []}
    #       for each in results:
    #           entry['results'].append({'throughput': 0, '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 "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

The test sets were defined at 4.6. All results (including those that included TCP end-hosts), SPSS outputs, and SPSS-ready `.csv` files is available at `https://github.com/MMfSDT/thesis-results`.