Comparing Flavors of TCP & UDP

Louie DiBernardo[*]

Johns Hopkins University
ldibern1@jhu.edu

May 9, 2022

Abstract

This paper aims to compare and contrast multiple flavors of TCP various topologies. This is an important topic to research as there is no unified way to look at all TCP flavors and compare their performance on many different network environments. We have several thousand papers that aim to explain and test one TCP flavor on one specific network topology. However this means that we do not and cannot know how these flavors perform on other topologies.

I. Introduction

This project has taken several twists and turns throughout its journey. Originally the scope of this project was to compare several flavors of TCP across a variety of different topologies to see which performed best in certain circumstances. This was going to be an exciting project that would allow us to really understand the different TCP flavors and learn their inner workings. Not only would we learn about TCP, but also about a few different topologies and how they would perform in different scenarios. However, due to several issues in during the development phase, the project had to become drastically reduced. The following sections will explain the original project, followed by the new reduced project and its ending.

II. Original Project

i. Topologies

There are a few different topologies I had planned to run performance experiments on.

First was Fat Tree, also known as a Clos Network. This was going to be a tree topology with a $k = 4$. Figure **1a** shows an example topology.

Next was a Leaf Spine topology. Figure **1b** shows an example topology.

Third was a new topology I had not seen before called Super Spine, also called Super Spine Mesh. This topology was interesting to me since it almost combined the leaf spine topology with a fat tree topology. Figure **2a** shows an example topology.

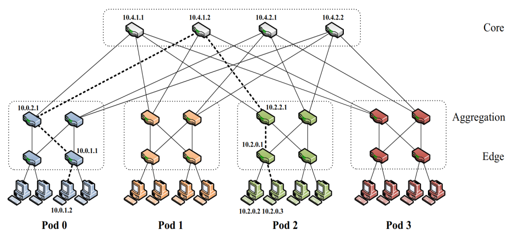
Last was a new topology I also had not seen before called Loopless. This topology was designed to not have any loops between hosts and end networks. This topology was dropped since upon linking multiple of these topologies together, we were given what resembled a leaf spine topology. Figure **2b** shows an example topology.

ii. Problems

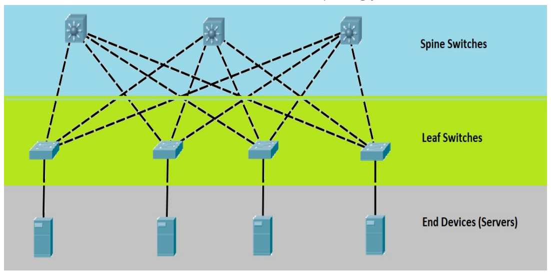
ii.1 OpenDayLight

As time progressed, several issues were met and were unable to be overcome. For example, one of the first issues I encountered was getting my controller to run on the same machine as my Mininet installation. I wanted to use OpenDayLight as my controller, mainly due to its visualization packages, but was forced to run it on a separate machine adding some

[*]Special thanks to David Mahler for his tutorial videos from 2013.

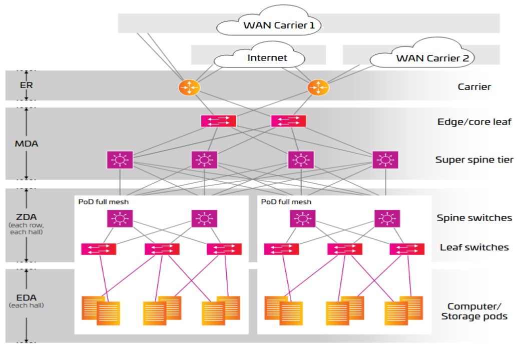


(a) *Fat Tree Topology*

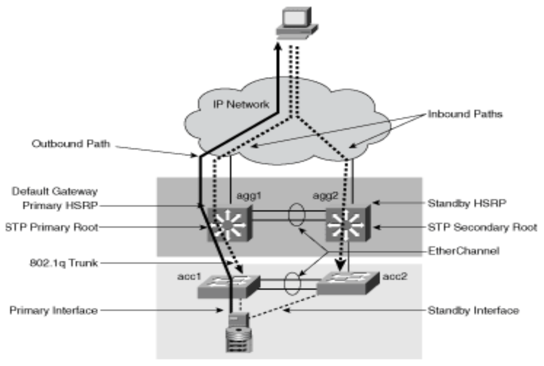


(b) *Leaf Spine Topology*

Figure 1: *Fat Tree and Leaf Spine Topology Examples*



(a) *Super Spine Topology*



(b) *Loopless Topology*

Figure 2: *Super Spine and Loopless Topology Examples*

complexity. This was not a large issue, but it meant I could only work on the project when I had two machines present. Related to ODL, I found out that the visualization I was hoping to use had many limitations as my topology got larger. ODL would refuse to display the hosts at all once I surpassed 4 hosts, and the switches appeared to have random links in the graph rather than their proper structured form in mininet.

Despite the issues with ODL, I was able to find and use another visualizer, Spear Narmox (linked directly in mininet's github) to visualize the topologies I created to verify that they were correct.

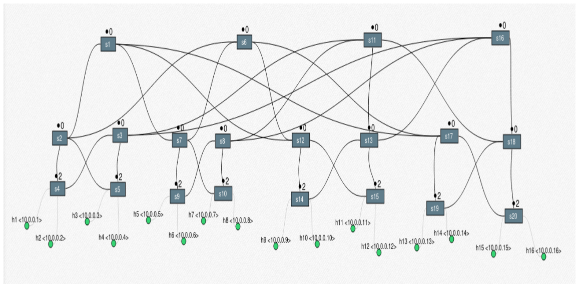
I continued to essentially fight with ODL on different issues I would have over the course of the project. Whether it was the controller not connecting randomly or disconnecting while mininet was active led to a lot of debugging and overall wasted time. I made the decision near the last couple weeks of the project to switch to a different controller, one that was far simpler and easier to use.

ii.2 DCTCP

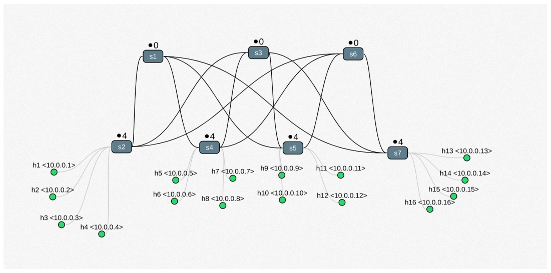
When the time came to install DCTCP, I thought it would be largely the same as MPTCP: follow mininet documentation and install the kernel. However, when I attempted to do that I got several errors and it would refuse to install. I tried to find documentation online for how to do this but everything I could find was about 10 years old or even older! At that point I essentially gave up on DCTCP since the kernel was not able to be installed.

iii. Topology Results

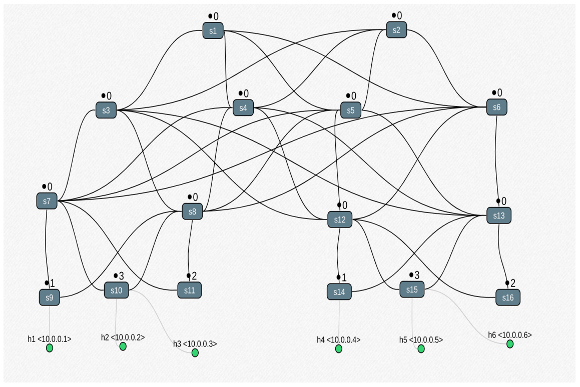
This section will introduce the final topology graphs that were generated in the original project. I aimed to replicate the example graphs that I showed earlier so they will be very simi-



(a) *Fat Tree Topology*



(b) *Leaf Spine Topology*



(c) *Super Spine Topology*

Figure 3: *Original Project Final Topologies*

lar. Note that the loopless topology mentioned earlier was dropped. This is because when I attempted to add duplicates of that topology and link them together, the topology I would create essentially mimicked the Super-Spine topology very closely. Figure **3** shows the final topologies.

III. Reducing the Scope

After discussing my project with the professor, I was told to dramatically reduce my project scope to just a singular linear topology with two hosts and to just try to run a flavor of TCP to get any results possible. I followed along with what the professor outlined so I could have some results by the time this report was due.

IV. New Reduced Project

i. Topology

This new reduced project will have merely one topology. A simple linear topology with two hosts connected to it. Figure **4** shows the final topology generated in mininet.

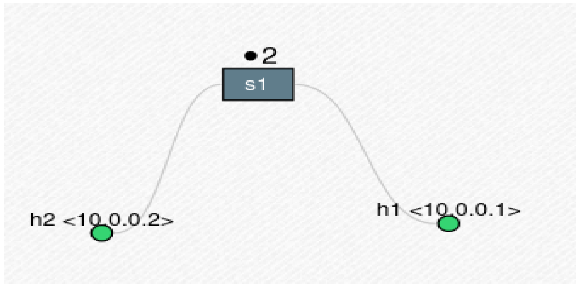


Figure 4: *Linear Topology*

ii. Problems

ii.1 IPERF

I found out during my testing phase that the new iperf versions included in Ubuntu LTS cause issues with mininet. Specifically when running TCP tests, iperf will hang and produce no output. This is an issue documented in Mininet's github issues tracker[1]. There are a few workarounds but I was unable to get it to work so I was forced to then move my project to a mininet VM. While not difficult, I had to make a several changes to my code to get things to work properly since I was originally working in python 3.6 but the mininet VM used python 2.7.

The reason for using a newer version of iperf is to more easily test STCP on this topology. Iperf3 contained bandwidth testing for

[1]Mininet Github Issue #1060: iperf fails (hangs) in TCP tests

TCP, STCP, and UDP. However, any public version above 2.0.5, would hang when run with mininet.

ii.2 Link Parameters

Since I now had to move to a different python version, I found that now all of the link parameters I had set originally now did not work. I was not sure why, but when I tried to compile my code, it would fail saying each parameter I gave was an unexpected argument.

Below are the parameters I was attempting to set:

Link Parameters

```
bw=10, delay='5ms', loss=2,
    max_queue_size=50
```

Due to this constraint of not being able to set link limits, the following was the result of a successful iperf test:

Output from running linear_test.py

```
Pox running separately!
Testing network connectivity
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
Testing TCP Bandwidth!
testing
*** Iperf: testing TCP bandwidth between
    h1 and h2
*** Results: ['59.1 Gbits/sec', '59.3
    Gbits/sec']
Bandwidth:59.1 Gbits/sec
```

As you can see the bandwidth is ridiculously high. However, unable to properly set link parameters makes it impossible to properly limit the links. Iperf does contain a way to limit bandwidth, but only for UDP.

ii.3 MPTCP Kernel

I originally had the MPTCP kernel installed on hardware with the ubuntu environment I was using to test mininet. Upon the move to the mininet VM I had setup from an assignment, I made changes to my code as mentioned before so it would run, and I also had to re-install the MPTCP kernel.

I followed the same directions as I did the first time when it was successful, however this time I had to upgrade my VM to be able to install the kernel. I did so, and proceeded to install the kernel.

I then tried to boot into the MPTCP kernel, but was immediately met with a kernel panic. I tried to then boot into the generic linux kernel, and was met once again by a kernel panic. I shutdown the vm, and started it up again and was able to successfully boot into the generic linux kernel, but when I tried the same process to boot into the MPTCP kernel, I was met with a kernel panic.

I wondered if perhaps the ubuntu version was just too old for this kernel version, so I upgraded ubuntu since a newer 16 version was available.

Unfortunately, after going through the upgrade process, the kernel was still unable to be booted. Due to this, MPTCP was unable to be successfully tested.

This has caused a fundamental change in this project. Instead of comparing flavors of TCP now, this project will aim to compare TCP to UDP in a linear topology. This is not what the project had originally planned for, but will still prove to give interesting results.

iii. Methods

All tests were performed using iperf. The following section outlines the results from each test.

The first test run was a simple 60 second bandwidth test between h1 and h2, h2 acting as the server.

The second test run was the same as the first but over 500 seconds to get a more real-life situation example.

The third test is was a dual test in which both h1 and h2 would send and receive from each other for 60 seconds.

The forth and final test was the same as the third, but over 500 seconds to get a more real-life situation example.

Commands for tests

```
500 Second Unidirectional TCP Test:
iperf -c 10.0.0.2 -t 500 -i 0.5 >
    h1toh2-500sec.txt

500 Second Dual-directional UDP Test:
iperf -c 10.0.0.2 -d -i 1 -t 500 -u >
    dual500sec-udp.txt
```

iv. Results

iv.1 Bandwidth

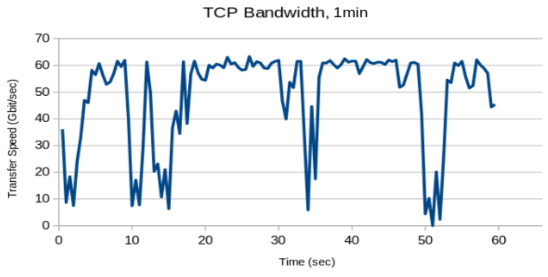
Bandwidth testing on this topology was interesting given that I was unable to limit the links with mininet's built in parameters. However, I was able to produce a few graphs over a few different periods of times. These first tests are unidirectional from h1 to h2.

Figure **5** shows the unidirectional graphs I created running iperf at 60 and 500 seconds for TCP. These graphs at first seem rather unexpected since I did not program any loss into the links. However, I am using pox as a remote controller, and using the default forwarding hub, so we can chalk up these packet losses to flow hard timeouts within the pox controller. We see the same time of packet loss in both the 60 second test, and the 500 second tests.

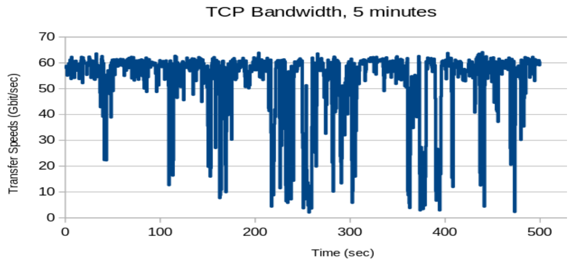
I decided to also run dual tests where the server and host would be sending data to each other constantly. The dual test graphs are very akin to a video call between these two hosts on the same network. They are both sending data to each other until the call ends.

All of these bandwidth tests were run on both TCP and UDP. You can see them and how they compare in figures **5**, **6**, **7**, and **8**.

I found it interesting how UDP performed compared to TCP. Of course UDP has no congestion control like TCP does, but comparing their graphs is quite intriguing. On the one hand, TCP offer vast amounts of bandwidth compared to UDP. Of course this comes with the trade off of TCP's slow start, something UDP does not have to worry about. Without the slow start, UDP is clearly more favorable to smaller flows since it can keep and maintain a slow rate rather than fluctuating as much as TCP.

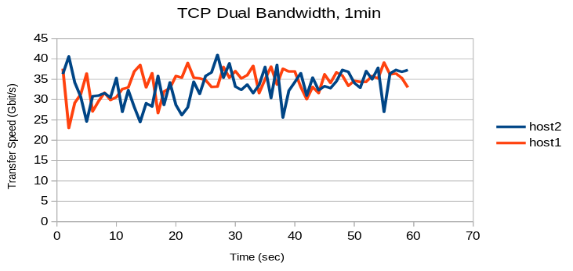


(a) *TCP Bandwidth over 60 Seconds*

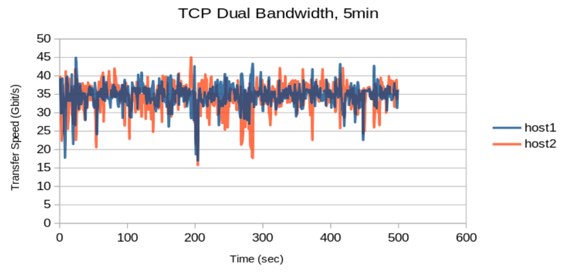


(b) *TCP Bandwidth over 500 Seconds*

Figure 5: *TCP Bandwidth Graphs for 60 and 500 seconds*

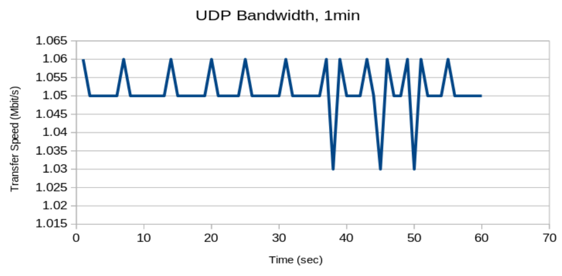


(a) *TCP Dual Bandwidth over 60 Seconds*

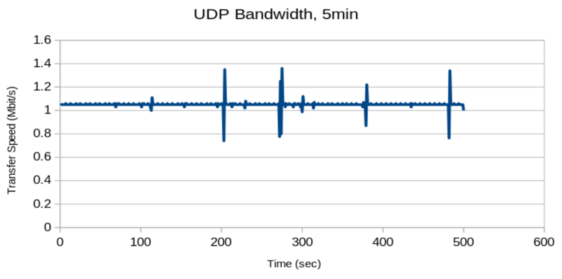


(b) *TCP Dual Bandwidth over 500 Seconds*

Figure 6: *TCP Dual Bandwidth Graphs for 60 and 500 seconds*

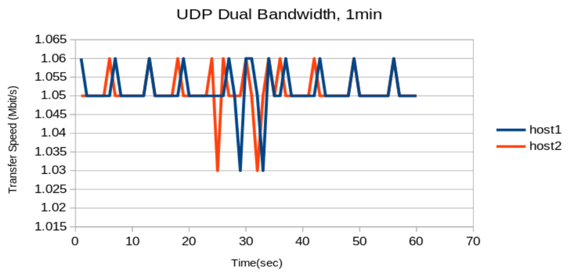


(a) *UDP Bandwidth over 60 Seconds*

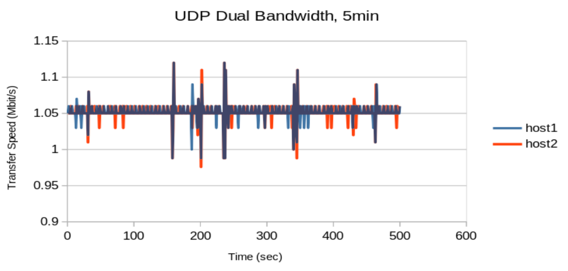


(b) *UDP Bandwidth over 500 Seconds*

Figure 7: *UDP Bandwidth Graphs for 60 and 500 seconds*



(a) *TCP Dual Bandwidth over 60 Seconds*



(b) *TCP Dual Bandwidth over 500 Seconds*

Figure 8: *UDP Dual Bandwidth Graphs for 60 and 500 seconds*

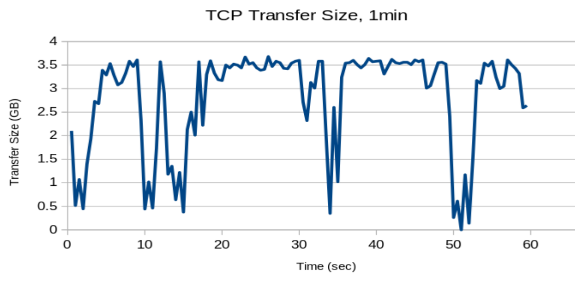
All of these tests were run using iperf, and also graphed using Excel.

v. Transfer Sizes

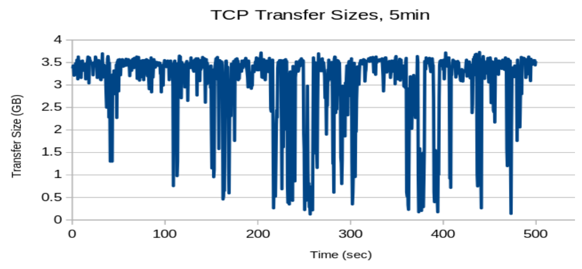
After the bandwidth tests I performed earlier completed, from iperf I was also given transfer sizes. I graphed these results as well and expected similar graphs to the bandwidth graphs in terms of overall shape. As you can see in figure **9**, both the 60 second and 500 second tests are graphed. Their overall shape matches the bandwidth tests.

Similar observations can be made regarding the TCP Dual, UDP, and UDP dual graphs. All their peaks line up as expected. This is more important for TCP than UDP, as UDP does not have any congestion control. TCP, however, must balance throughput and reliability. It cannot favor one over the other, which is why we get very jagged graphs and it attempts to find a balance between the two.

The following figures are relevant to this section: **9**, **10**, **11**, and **12**.

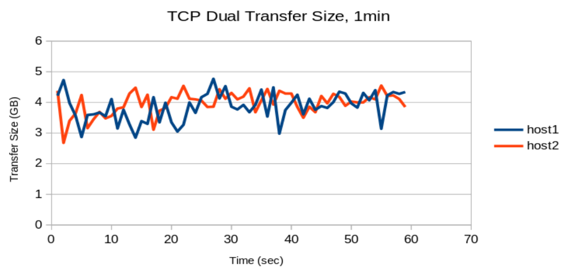


(a) *TCP Transfer Sizes over 60 Seconds*

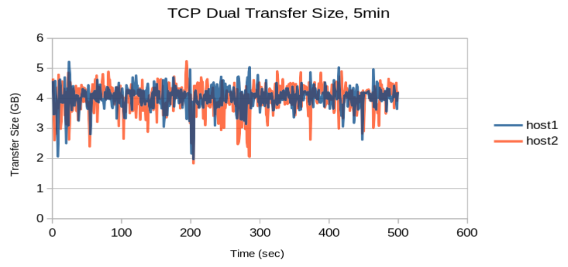


(b) *TCP Transfer Sizes over 500 Seconds*

Figure 9: *TCP Transfer Size Graphs for 60 and 500 seconds*

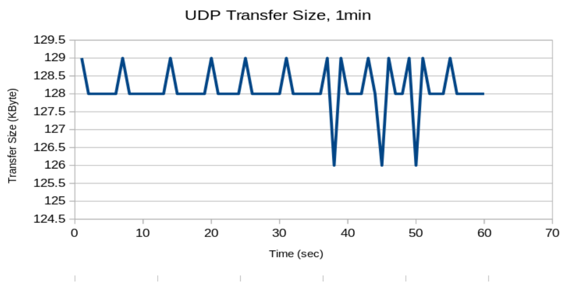


(a) *TCP Dual Transfer Sizes over 60 Seconds*

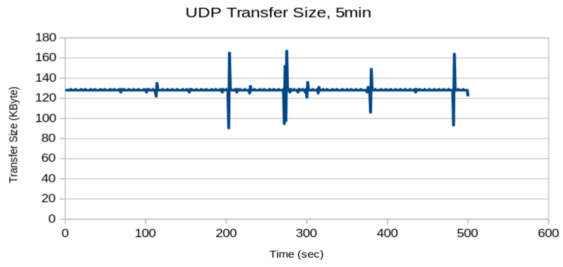


(b) *TCP Dual Transfer Sizes over 500 Seconds*

Figure 10: *TCP Dual Transfer Size Graphs for 60 and 500 seconds*



(a) *UDP Transfer Sizes over 60 Seconds*



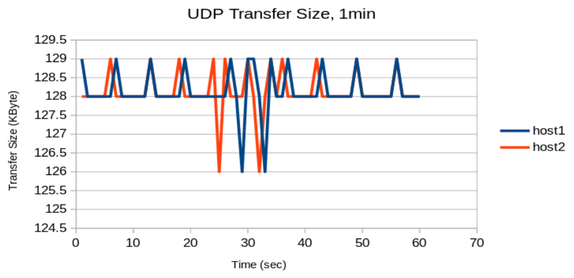
(b) *UDP Transfer Sizes over 500 Seconds*

Figure 11: *UDP Transfer Size Graphs for 60 and 500 seconds*

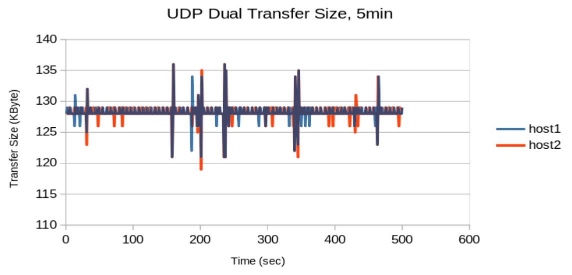
V. Discussion

i. Retrospective

This project overall has taught me a lot of very valuable lessons. Not only did I get to explore several different tools I did not think I would ever use, but I also got the valuable experience of simply learning how to develop virtual networks and how to perform tests on them. Of course, this did not go as smoothly as I would have liked. I found myself often searching the internet for tutorials on the tools I was trying to use but not finding many. ODL specifically gave me a lot of trouble. Even though they had decent documentation, I would often run into problems that I would try and find the solution for, only to find a forum post or email thread from 11 years ago from someone with the same problem but with no solution given. This made the process very arduous, time consuming, and often just frustrating. There are very few knowledge bases or people on the internet with helpful tutorials that would successfully teach a concept. Mininet for example, has a great setup guide, but once you are done with that, you are on your own. A while later I



(a) *UDP Dual Transfer Sizes over 60 Seconds*



(b) *UDP Dual Transfer Sizes over 500 Seconds*

Figure 12: *UDP Dual Transfer Size Graphs for 60 and 500 seconds*

found out that deep in their github wiki they had helpful tutorials and videos. This is exactly the problem. These tools have documentation spread out across several places it can make it impossible to find what you are looking for. I think I could spend a year trying to learn mininet and other tools, and still only scratch the surface.

However through trial and error, many errors, I found myself slowly learning about the difficulty in creating and testing a massive project like this. If I could go back in time and tell my past self what to do, I would tell myself to consider the amount of time that will be spent doing research and to reconsider the scale of my first idea.

ii. Future Work & Improvement

I think I could later revisit the original project and successfully run experiments. One of the reasons it was abandoned was because routing it a tedious and necessary process to ensure that packets don't get lost in loops. Towards the end of the project I learned about a controller that is "topology" aware. This means it is able to know the network graph and remove paths that would be loops. This is the floodlight controller. This eliminates the need for any routing to be done by the programmer and should allow for tests to run without issue.

Another improvement I would make is from the beginning to define a smaller project. I got so caught up in the idea that I would be able create a project that encompassed all TCP variants and many topologies that I did not stop to think about how much work and research would have to be done to accomplish this. This led to lots of time being spent building and debugging topologies, debugging controller issues, and not enough time spent developing and running the actual tests.

I do plan to come back to this project and perform the original project experiments. However I first need to sit down and just read and compile information and documentation for the tools I plan to use rather than just jumping in, not knowing anything like I did this first time. I hope returning to this project at a later time will enable me to come back humbled at my failures here, but also to come back re-energized and ready to tackle the experiment.

I do believe that this project is important. Not only to me, but to others like me who want to understand these transfer protocols in a more intimate manner, and who otherwise would not be able to compare them directly against each other. While I do understand that this project and paper currently fails to provide that, I plan to return to it soon with a fresh mind, and more help to get this project off the ground, and to give a more comprehensive look at these systems in different scenarios.

VI. Reproduce The Project

You can find the current version of the project at:

https://github.com/JustATechie/CC-Project-Public.

Please note that two branches exist. The main branch is the newest reduced version of the project, in branch V1, resides the original project in its broken glory.

References

[David Mahler's Youtube Channel.] David Mahler's Youtube Channel. Retrieved May 9th, 2022, from https://www.youtube.com/c/DavidMahler. His videos on mininet and controllers, made this project possible.

[Mininet Github.] Mininet's main repository with its vast knowledge base: https://github.com/mininet/mininet.

[Mininet Github Issue Tracker.] Mininet's issue tracker where the iperf error was found: https://github.com/mininet/mininet/issues/1060.