Single Socket Video Streaming Over Multiple Servers Using MPTCP and SDN

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

in

Information Technology

by

Saket Mahto (20188108)
Ruthvik Josh Pentareddy (20188112)
Rajan Jayswal (20185117)
Nikhil Ranjan Jha (20188080)

under the guidance of

Dr. Mayank Pandey
(Associate Professor)

to the
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
MOTILAL NEHRU NATIONAL INSTITUTE OF TECHNOLOGY, ALLAHABAD
PRAYAGRAJ, UTTAR PRADESH (INDIA)
November 2021

UNDERTAKING

I declare that the work presented in this report titled "*Single Socket Video Streaming Over Multiple Servers Using MPTCP and SDN***", submitted to the Computer Science and Engineering Department, Motilal Nehru National Institute of Technology, Allahabad, Prayagraj, Uttar Pradesh (India) for the award of the *Bachelor of Technology* degree in *Information Technology*, is my original work. I have not plagiarized or submitted the same work for the award of any other degree. In case this undertaking is found incorrect, I accept that my degree may be unconditionally withdrawn.**

November 2021
Prayagraj
U.P. (India)

(Saket Mahto, 20188108)

(Ruthvik Josh Pentareddy, 20188112)

(Rajan Jayswal, 20185117)

(Nikhil Ranjan Jha, 20188080)

'

CERTIFICATE

Certified that the work contained in the report titled "*Single Socket Video Streaming Over Multiple Servers Using MPTCP and SDN***", by** *Saket Mahto (20188108), Ruthvik Josh Pentareddy (20188112), Rajan Jayswal (20185117), Nikhil Ranjan Jha (20188080)***, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.**

Dr. Mayank Pandey
(Associate Professor)
Computer Science and Engineering Dept.
M.N.N.I.T, Allahabad
November 2021 **Prayagraj, Uttar Pradesh (INDIA)**

Preface

As there is an increasing demand for video streaming services such as Netflix, Youtube, among others, users prefer to have a seamless service even at an event of failure or when the main connection gets congested. With a single connection, this poses an issue where the connection is lost or that there is a significant lag at the user's end with such scenarios of link failure or high-traffic between the client and the server.

With the advent of MPTCP and it's capabilities, we could mimic an opposite scenario where a client is connected to multiple servers on just one socket, i.e., if one server goes down or the line gets congested, the other server must be able to know how much of the video segments were already sent and what needs to be sent after that.

The above proposal could be highly robust and provide a seamless streaming experience for the user even at an event of failure or when there's high traffic.

Acknowledgements

The completion of this project required a lot of effort, guidance and support from many people. We feel privileged and honoured to have got this all along the development of the project. We would like to express our gratitude to our mentor, Mayank Pandey (Associate Professor) for his perennial support, guidance and monitoring throughout the making of this project. We would also like to acknowledge and thank our professors, colleagues and seniors for supporting us and enabling us to successfully complete our project.

Contents

Chapter 1

Origin of the Project

1.1 Abstract

Multi-path TCP or MPTCP is an internet protocol-based and extended on the traditional TCP. It allows for the simultaneous use of several IP addresses/interfaces by a modification of TCP that presents a regular TCP interface to applications while spreading data across several subflows. Benefits of this include better resource utilization, better throughput, and smoother reaction to failures. In this project, we will be trying to use MPTCP to video stream a video from multiple servers (all containing the same instance of the video file) in a way that if one of the server/link is broken/down, the connection can still remain stable via the other servers. Our main goal is to establish a single socket connection from the client to multiple servers such that the packets are synchronized across the different servers, i.e., if one server goes down, the other server must be able to know how much of the video segments were already sent and what needs to be sent after that.

1.2 Technicality

- To stream a video file over multiple servers, we speculate that the following requirements need to be met:
  - Multiple connections from a single client to multiple servers
  - A mechanism to order/distinguish the packets from multiple servers

- The above requirements can be met using various ways, but the first approach we are taking is through the use of MPTCP protocol.
- Since MPTCP already allows for simultaneous use of several IP addresses/interfaces by a modification of TCP that presents a regular TCP interface to applications while spreading data across several subflows, we hope to be able to modify it so that the subflows can connect to multiple servers instead of a lone server. Furthermore, we speculate that the second requirement can be met by modifying the Data Sequence Number that MPTCP uses to maintain track of packets that flow through different subflows.
- The next approach could be through the use of a middle-man application that manages connections to multiple servers and also acts as a scheduler for the incoming packets from multiple servers.
- We used the following tools and testbeds for our project:
  - Network Simulation: Mininet
  - MPTCP: Linux 5.4.86 MPTCP-enabled kernel
  - SDN: Ryu, Floodlight
  - Packet Tracing: Wireshark
  - Video Streaming: VLC Media Player
  - Proxy: LittleProxy

1.3 Importance and Relevance to CSE

The current scenario for the video streaming services are mostly limited to single servers. While there are cases of multiple servers, backup servers that are running so that we can connect to the nearest/fastest one, they are still limited to a single point of failure or a single connection for data transfer. What we are attempting to do in this project is expand on the single connection by simultaneously having multiple connections to different servers. With this we can not only eliminate the single point failure, but we can also do things like pick/change the optimal paths for data transfer without breaking the connection, pre fetch the additional chunks for the video file over different connections etc. These changes would make video streaming much faster and would be a big change on how the current video streaming services work.

Chapter 2

Proposed Work

2.1 Initial Observations

2.1.1 Topology

For our initial testbed, we used Dr. Chih-Heng Ke's mininet script for MPTCP. It had the following topology -

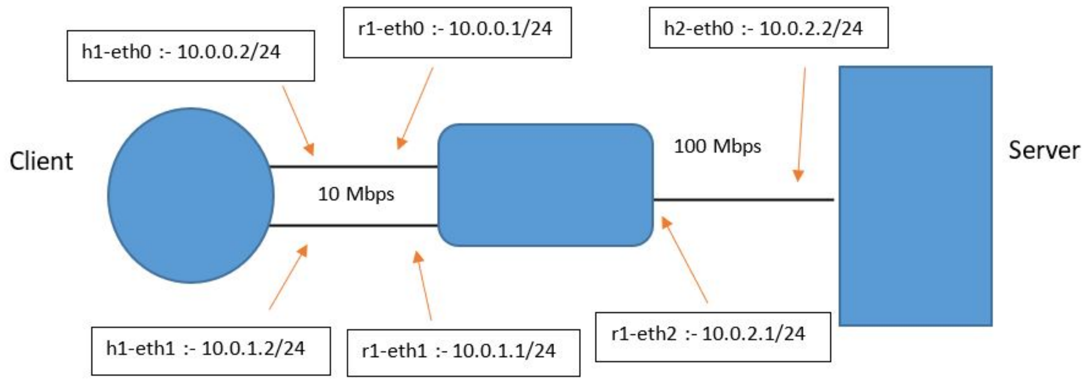

Figure 1: Initial Topology

The topology contains:

- **Client (h1), Router (r1), Server (h2)**
- **Two lines between the client and the router, where each line has a link speed of 10 Mbps**

- Host h1 in this scenario acts as the master link and host h2 acts as the slave link
- A single line from the server to the router with 100 Mbps

2.1.2 Streaming with VLC Media Player

- VLC media player contained an option to stream video over RTP, RTSP, HTTP among other protocols.
- We initially tested the stream in our topology using RTP/RTSP (over TCP) options in VLC and observed that most packets transferred were using UDP and not TCP/MPTCP. However, MPTCP was fully functional over HTTP.
- In our testbed, the server streamed the video to the client and traced the packets on Wireshark. The following observations were made:
  - Unlike TCP, MPTCP has additional options along with the packets such as MP_CAPABLE, MP_JOIN and DSS.
  - Both lines on the client side were equally used as they had the same link speeds.

2.1.3 Understanding Multipath-TCP Handshake

▮ *MP_CAPABLE Option*

- The initial connection is very similar to TCP's handshake, but SYN, SYN + ACK and ACK additionally carry the MP_CAPABLE option in MPTCP.
- The option has a variable length and servers multiple purposes
  - It verifies whether the remote host supports MPTCP
  - It allows the hosts to exchange some information to authenticate the establishment of additional subflows

▮ *MP_JOIN Option*

- The exchange of keys carried out in the initial handshake process with MP_CAPABLE generates a token to authenticate the end-points when new subflows will be set up.

- **Just like in the initial connection, the subflows carry an MP_JOIN option with SYN, SYN + ACK and ACK.**
- **The token generated from the key is used to identify which MPTCP connection is joining, and HMAC (Hash-based Authentication Code) is used for authentication.**

▮ *Data Sequence Number (DSN)*

- **MPTCP uses two levels of sequence numbers**
  - **Subflow Sequence Number, the same as the one that appears in a TCP Header**
  - **Data Sequence Number (DSN), used inside MPTCP options**
- **DSN enable the receiver to reorder the data received over different subflows**
  - **It is incremented every time data is sent**
  - **An Initial Data Sequence Number is negotiated during the three-way handshake of the first subflow**

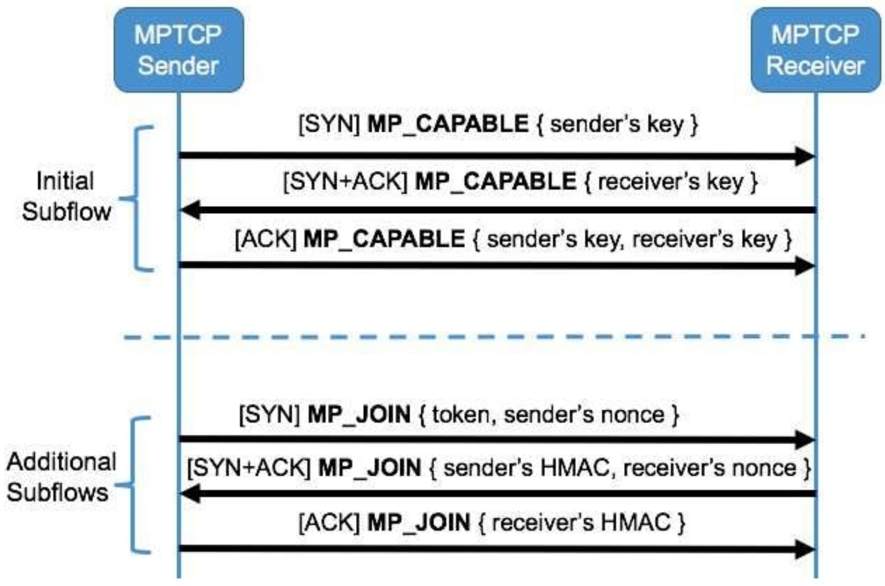
MPTCP Sender
MPTCP Receiver
Initial Subflow
[SYN] MP_CAPABLE { sender's key }
[SYN+ACK] MP_CAPABLE { receiver's key }
[ACK] MP_CAPABLE { sender's key, receiver's key }
Additional Subflows
[SYN] MP_JOIN { token, sender's nonce }
[SYN+ACK] MP_JOIN { sender's HMAC, receiver's nonce }
[ACK] MP_JOIN { receiver's HMAC }

Figure 2: Multipath-TCP Handshake Process

2.2 Observing Throughput

2.2.1 Different Link Speeds

- **Unlike the previous scenario, we decided to observe the throughput on the client side using a link speed of 50 Mbps on one line and continue using 10 Mbps on the other line.**
- **We observed that the link with a higher speed was utilised more while transferring data as shown in the following throughput graphs:**

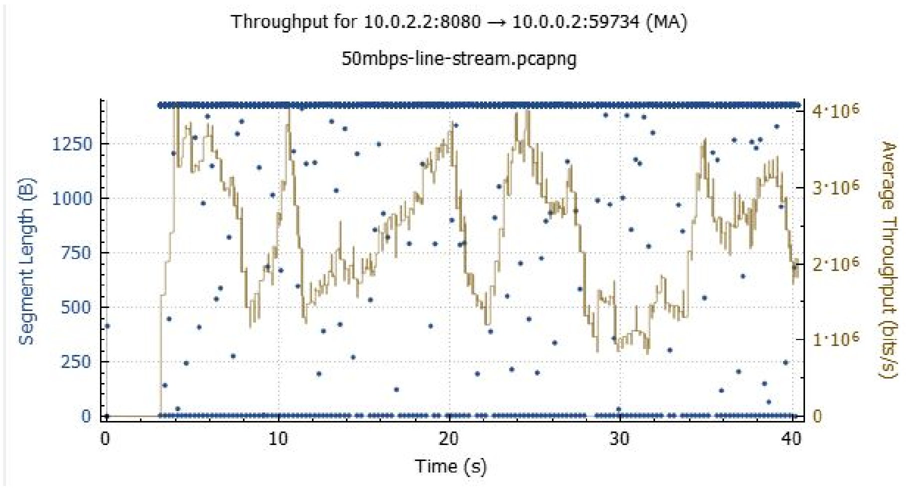

Figure 3: 10.0.0.2 has a link speed of 50 Mbps

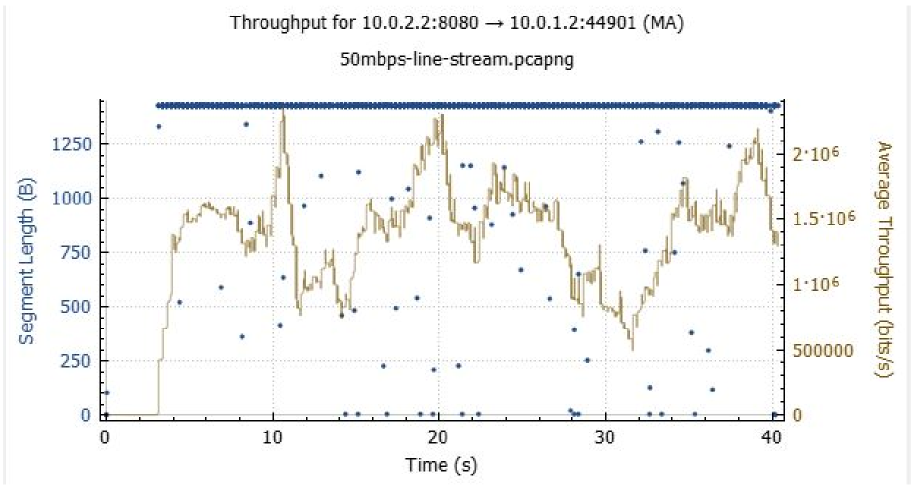

Figure 4: 10.0.1.2 has a link speed of 50 Mbps

2.2.2 Links Up/Down

- **Next we observed the behaviour of MPTCP when one or more of the links/subflows were up/down.**
- **In our scenario, we had h1-eth0 (10.0.0.1/24, 10 Mbps) as the master flow and h1-eth1 (10.0.1.2/24, 50 Mbps) as the TCP subflow.**
- **We observed that when the sub-flow interface was brought up/down the MPTCP connection remained stable and simply switched to the master flow as it was initially using the sub-flow because of its higher bandwidth.**
- **Even after bringing the interface back up, the already established MPTCP connection doesn't seem to pick it back up and continues using the single master flow.**
- **Similarly, bringing the master flow interface down also seems to have the same effect i.e. it switches to the sub-flow to transfer data and is unaware even if the master interface is brought back up.**
- **Upon further observation, it was found that this was a limitation of flow/ip rules that are added on the end nodes based on the script. As when the interface is brought down the ip rules corresponding to the interface are deleted.**
- **Switching to a network aware SDN seems to solve this issue as it can dynamically add flow rules based on the working interfaces. So with a SDN, as long at least one flow is present, the connection is stable and upon bringing the interfaces back up, the MPTCP starts using the most suitable flow. The RYU SDN controller was used to verify this.**

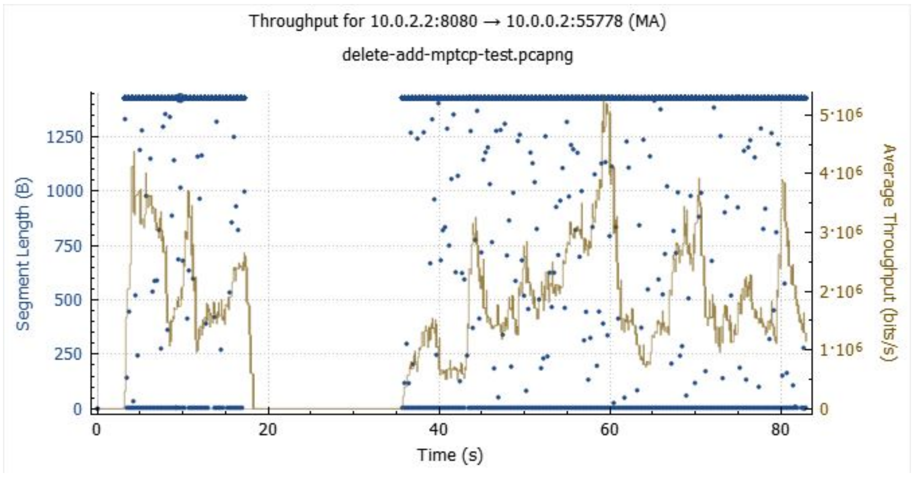

Figure 5: Throughput of Link 1 when Link 2 is down from 45-60s

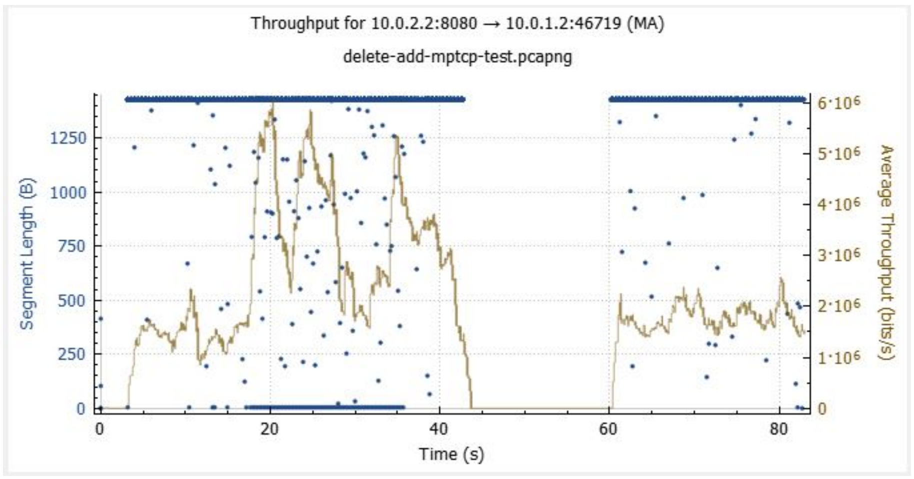

Figure 6: Throughput of Link 2 when Link 1 is down from 19-35s

2.3 Our Intuition to Use MP_JOIN

2.3.1 Swapping The Client and Server

- **In our next testbed, we decided to swap the scenario where the server had two links instead of the client and performed the tasks in Section 2.2.**
- **We observed that when links were up/down i.e. on an event of failure or congestion at one of the links, the connection remained stable and packets were transferred regardless.**

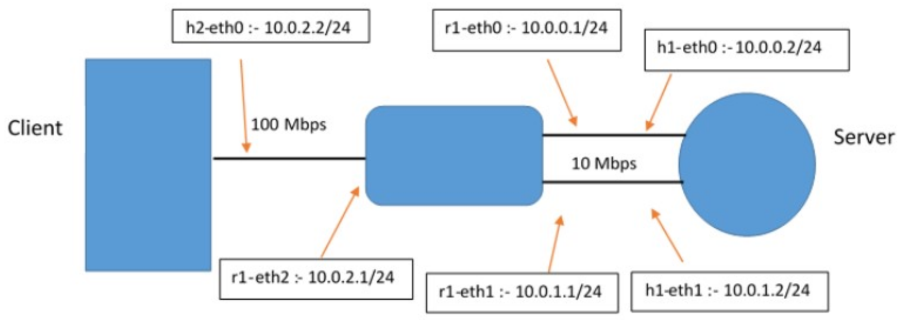

Figure 7: Swapped Topology: Server has two links

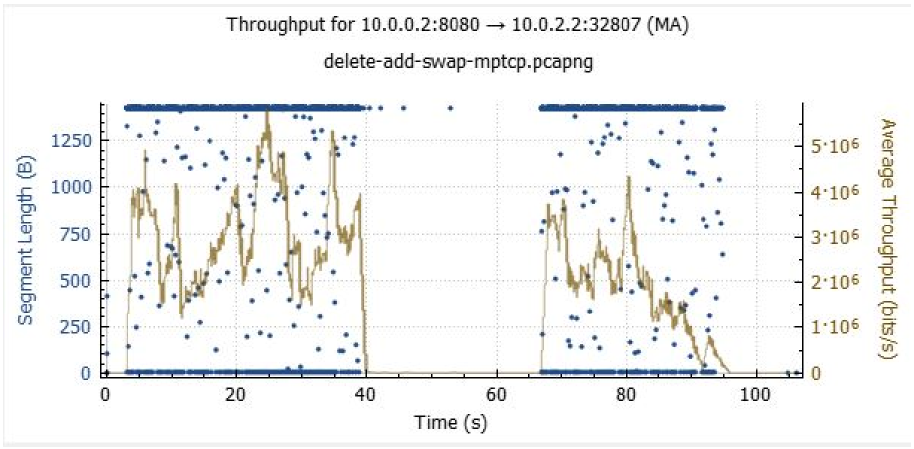

Figure 8: Throughput of Link 2 when Link 1 is down from 40-65s

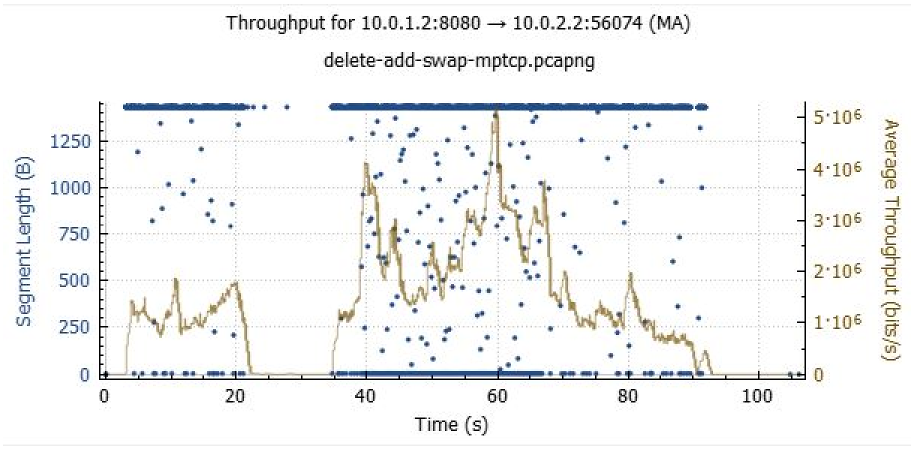

Figure 9: Throughput of Link 1 when Link 2 is down from 23-35s

2.3.2 Potentially Using MP_JOIN to Connect Multiple Servers

- **Based on the above scenario, we had a notion that we could send an MP_JOIN option to a second server to join the connection. In such a scenario, MPTCP would handle all the incoming packets and make sure they are in-order.**
- **Furthermore, our investigation in Section 2.3.1 gives us further evidence that the video stream would remain stable even on an event of failure.**
- **Finally, our main goal of using a single socket to stream video from multiple servers could be made possible by using an SDN to redirect the MP_JOIN packet to other servers.**

2.4 Connecting to Multiple Servers by Redirecting MP_JOIN

2.4.1 Topology and Implementation on Floodlight

- For this we switched the router with a L2 ovs-switch and connected a SDN controller to the topology.
- Since RYU didn't have a MPTCP aware module, we used a MPTCP aware Floodlight controller.
- The following functions were mainly used for the redirection of the MPTCP packets:
  - isMPTCPEnabled() - to check whether the hosts are MPTCP enabled or not
  - getMptcpSubtype() - to get the subtype of the MPTCP packet i.e. MPTCP_CAPABLE or MPTPC_JOIN
  - getMptcpSenderKey() - to get sender key that is exchanged during the MPTCP handshake process.
  - getMptcpToken() - to get the token that was generated during the MPTCP handshake process.
- Firstly, we started the video streaming of the same file on both the servers using VLC
- We filtered out all the MPTCP packets using the isMPTCPEnabaled()
- Then we checked whether the packet was MP_JOIN or not using getMptcpSubtype()
- Since we had only 2 interfaces, we would have only one subflow handshake using the MPTCP_JOIN signal.
- So we modified the said packet using the controller. We changed its destination IP address to the IP address of the 2nd server (from the IP address of the 1st server).

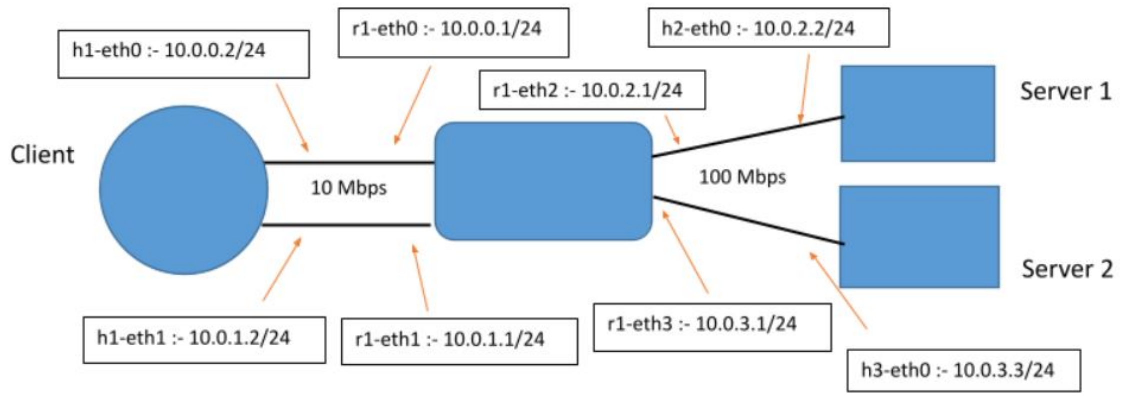

Figure 10: Topology: Connecting to Multiple Servers

2.4.2 Floodlight Code Snippet

```
if(ipv4.getProtocol() == IpProtocol.TCP) {
TCP tcp = (TCP) ipv4.getPayload();

int subType = tcp.getMptcpSubtype();
if(subType == 0) {
log.info("It is a MPTCP_CAPABLE packet");
}
else if(subType == 1) {
log.info("It is a MPTCP_JOIN signal");

log.info("--------------------------");
log.info("Src add : " + ipv4.getSourceAddress());
log.info("Dest add : " + ipv4.getDestinationAddress());

String destAddr = ipv4.getDestinationAddress().toString();
if(destAddr.equals("10.0.0.2")) {
ipv4 = ipv4.setDestinationAddress("10.0.0.3");
eth = eth.setDestinationMACAddress("82:84:45:c0:5a:78");
log.info("Changed dest addr to : " + ipv4.getDestinationAddress());
log.info("Change dest MAC to : " + eth.getDestinationMACAddress());
}
log.info("--------------------------");
}
}
```

2.4.3 Observations

- **The packet was simply dropped by the 2nd server.**
- **The client advertises the interface to the server using the MPTCP_ADD_ADDR signal.**

2.4.4 Issues and Limitations

- **Though our intuition seemed applicable and possible with the use of SDN, the glaring drawback was key and token exchange as part of MPTCP's security protocol.**
- **By observing Forwarding.java, we noticed that unlike a regular 4-tuple used to identify a TCP connection, an MP_JOIN option had a 5-tuple system, where the last parameter is the token generated during the MPTCP handshake process.**
- **Although we attempted to redirect this token generated from the handshake to the second server, based on our observations the packet was dropped completely.**
- **Another important observation that can be made is that the MP_JOIN signal alone cannot start a new MPTCP connection with a server. It must be followed by a MP_CAPABLE signal. This can further be supported by the following lines mentioned in the rfc 8684 section 3.2:**
  - **It uses keying material that was exchanged in the initial MP_CAPABLE handshake (Section 3.1), and that handshake also negotiates the cryptography algorithm in use for the MP_JOIN handshake.**

- **Thus, while the MP_JOIN signal is established like a normal TCP connection, it cannot start a connection without a prior successful MP_CAPABLE signal.**

Chapter 3

Conclusion and Future Work

3.1 Conclusion

As discussed in the previous section, with the current implementation of MPTCP and the MPTCP handshake process, it is not quite possible to connect to multiple servers using the MPTCP_JOIN connection. Mostly because of how the keying materials are used in the initial connection setup. It may be possible in the future with some changes to the how the initial connection is established but currently it doesn't seem possible. Therefore, we would be moving to the second approach.

3.2 Future Work

As mentioned in the Technicality section, MPTCP is only one of the ways to achieve this. Since, MPTCP has certain limitations, our future work will be trying to achieve the same goal using the middle-man application approach. In this appraoch, instead of using MPTCP to connect to multiple servers, we would be forwarding our request to a middle-man application which would then duplicate and forward our requests to multiple servers.

For our experiments, we first decided to use Little Proxy as the middle-man application. LittleProxy is a high performance HTTP proxy written in Java atop Trustin Lee's excellent Netty event-based networking library. It's quite stable, performs well, and is easy to integrate into your projects.

3.2.1 Proposed Topology

First of all, the VLC will be configured to use the Little Proxy's server. Since our goal is to use a single socket, the Proxy server will be using virtual interfaces to connect to different servers. The proxy server will be responsible for both redirecting the request to multiple servers as well as scheduling the different packets that arrive from different servers before forwarding the data to VLC.

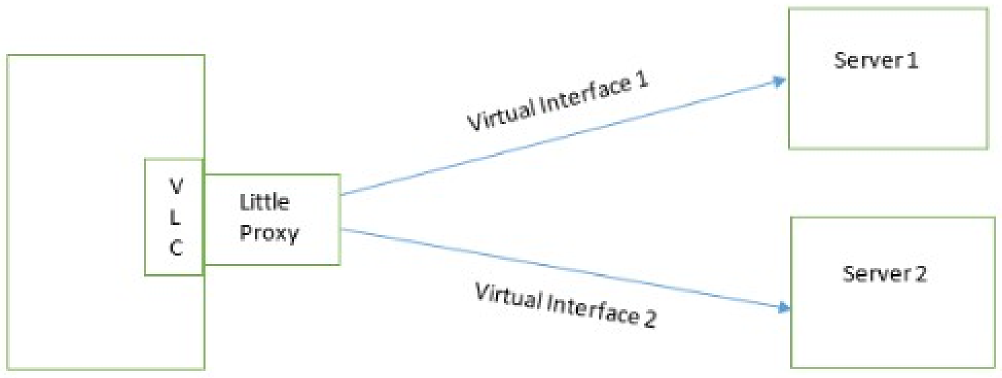

Figure 11: Topology: Connecting to Multiple Servers

3.2.2 Challenges

- **Using the middle-man application to duplicate the requests to multiple servers.**
- **Being able to fetch different chunks of video from different servers.**
- **Scheduling the chunks at the middle-man application before delivering it back to the VLC.**

References

[1] **MPTCP Aware SDN [Online] :**
`https://github.com/zsavvas/MPTCP-aware-SDN`

[2] **MPTCP Enabled Topology [Online] :**
`http://csie.nqu.edu.tw/smallko/sdn/mptcp-test.htm`

[3] **Towards Dynamic MPTCP Path Control Using SDN [Online/PDF] :**
`http://www.cs.columbia.edu/ hn2203/papers/12`$_{n}etsoft2016.pdf$

[4] **Exploiting Path Diversity in Datacenters Using MPTCP-aware SDN [Online/PDF] :**
`https://arxiv.org/pdf/1511.09295.pdf`

[5] **TCP Extensions for Multipath Operation with Multiple Addresses [Online] :**
`https://datatracker.ietf.org/doc/html/rfc8684`

[6] **Little Proxy [Online] :**
`https://github.com/adamfisk/LittleProxy`