Single Socket Video Streaming Over Multiple Servers Using 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)
May 2022

UNDERTAKING

I declare that the work presented in this report titled "*Single Socket Video Streaming Over Multiple Servers Using 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 In case this undertaking is found incorrect, I accept that my degree may be unconditionally withdrawn.

May 2022
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 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
May 2022 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.

To combat the issue of a single point of failure, we propose the following methods -

- Using a single socket on the client side to connect to multiple streaming servers over Multipath-TCP
- Exploring adaptive streaming over multiple servers on a peer-to-peer network and using SDN to pick the best streaming server

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

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

Preface **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Abstract 1

1.2 Related Work and Literature Review 1

1.3 Importance and Relevance to CSE 2

2 Single Socket Video Streaming Over Multiple Servers Using MPTCP and SDN **3**

2.1 Technicality 3

2.2 Initial Observations 4

2.2.1 Topology 4

2.2.2 Streaming with VLC Media Player 5

2.2.3 Understanding Multipath-TCP Handshake 5

2.3 Observing Throughput 7

2.3.1 Different Link Speeds 7

2.3.2 Links Up/Down 8

2.4 Our Intuition to Use MP_JOIN 11

2.4.1 Swapping The Client and Server 11

2.4.2 Potentially Using MP_JOIN to Connect Multiple Servers . . . 12

2.5 Connecting to Multiple Servers by Redirecting MP_JOIN 13

2.5.1 Topology and Implementation on Floodlight 13

2.5.2 Floodlight Code Snippet 14
2.5.3 Observations 15
2.5.4 Issues and Limitations 15

3 Adaptive Streaming Over Multiple Servers Using SDN 17
3.1 Technicality 17
3.2 Topology 18
3.3 Working 18
3.3.1 Generating MPD File 18
3.3.2 Chord DHT 19
3.3.3 Virtual IP 19
3.3.4 Processing request to Virtual IP 20
3.3.5 Choosing the best Server 23
3.3.6 Switching Servers 24
3.4 Observations 26

4 Conclusion and Future Work 29
4.1 Conclusion 29
4.2 Future Work 29

References 31

Chapter 1

Introduction

1.1 Abstract

In the present scenario of video streaming around the world, CDN is one of the most widely used network system. But CDN is group of centralized servers this project, we aim to eliminate a single point of failure and provide a robust streaming service through the following two methods -

- Firstly, 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.
- Secondly, we explore the possibility of optimizing adaptive streaming over multiple servers on a peer-to-peer network, wherein we used Software-Defined Networking to choose the best streaming server on the basis of bandwidth, number of hops, and delay.

1.2 Related Work and Literature Review

- Hyunwoo Nam et al. [3] resolved the issue of out-of-order packet delivery in multiple paths having different delays by proposing a dynamic solution of

adding and removing MPTCP paths using Software-Defined Networking. The main concept is to monitor the available capacity of linked pathways and select the best paths based on changing network conditions.

- Savvas Zannettou et al. [4] proposed an MPTCP-aware SDN controller that facilitates an alternative routing mechanism for the MPTCP subflows by packet inspection to provide deterministic subflow assignment to paths.

- Ali Gohar et al [5] stated that MPTCP combined with evolving technologies such as Software-Defined Networking can improve the QoE of streaming multimedia based on scalable video coding (SVC). Hence, they have proposed a Dynamic Multi Path Finder (DMPF) scheduler that finds optimal techniques to improve the QoE. This scheduler accommodates numerous client requests while providing the rudimentary representation of the media requested.

- Joachim Bruneau-Queyreix et al. [6] provided a prototype for a hybrid P2P/multi-server quality-adaptive evolving HAS(HTTP Adaptive Streaming)-compliant streaming system that uses several servers and peers at the same time, trading off server infrastructure capacity and QoE gains.

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

Single Socket Video Streaming Over Multiple Servers Using MPTCP and SDN

2.1 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.

- We used the following tools and testbeds for this section of our project:
  - Network Simulation: Mininet
  - MPTCP: Linux 5.4.86 MPTCP-enabled kernel
  - SDN: Ryu, Floodlight
  - Packet Tracing: Wireshark
  - Video Streaming: VLC Media Player

2.2 Initial Observations

2.2.1 Topology

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

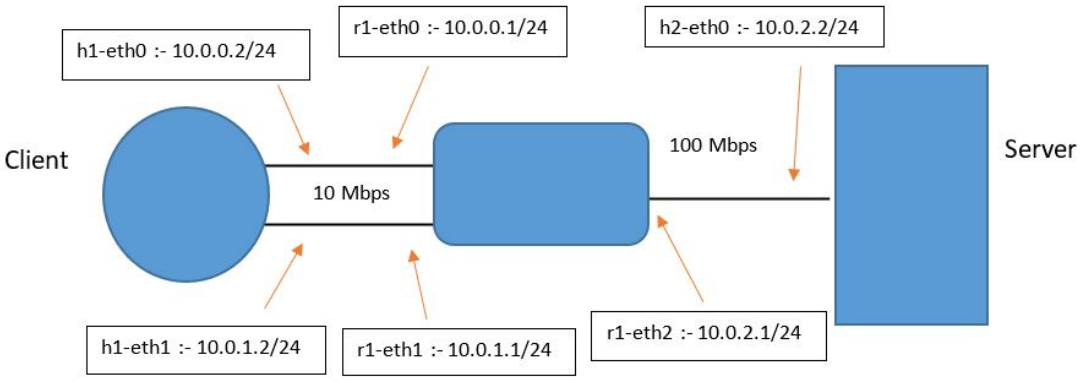


Figure 1: Initial Topology

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

2.2.2 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.

2.2.3 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 endpoints 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

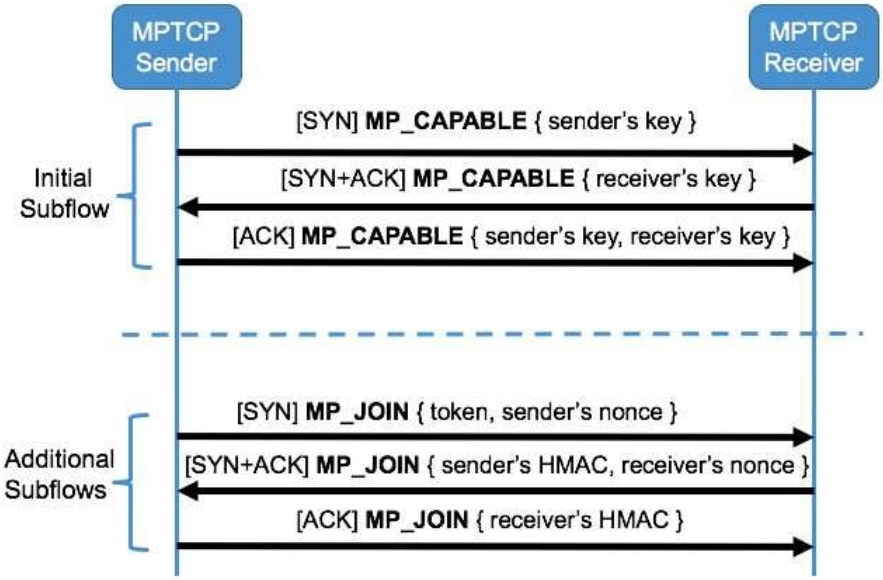


Figure 2: Multipath-TCP Handshake Process

2.3 Observing Throughput

2.3.1 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:

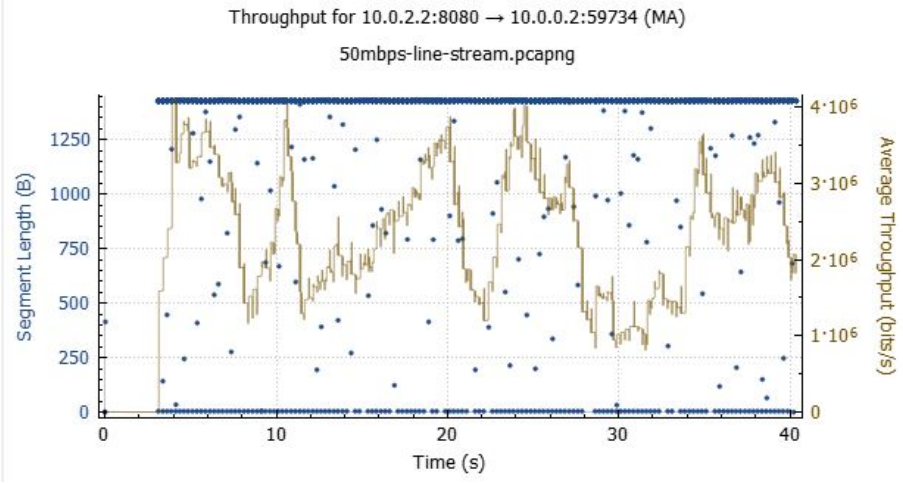


Figure 3: 10.0.0.2 has a link speed of 50 Mbps

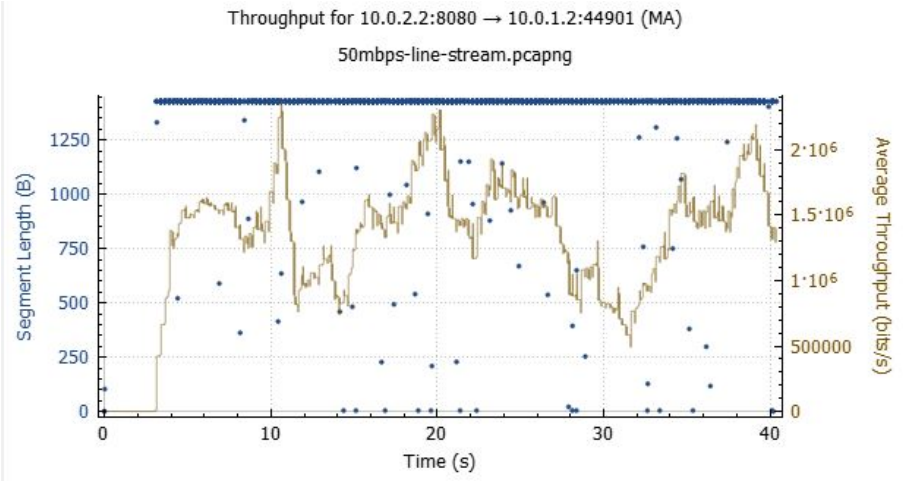


Figure 4: 10.0.1.2 has a link speed of 50 Mbps

2.3.2 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.

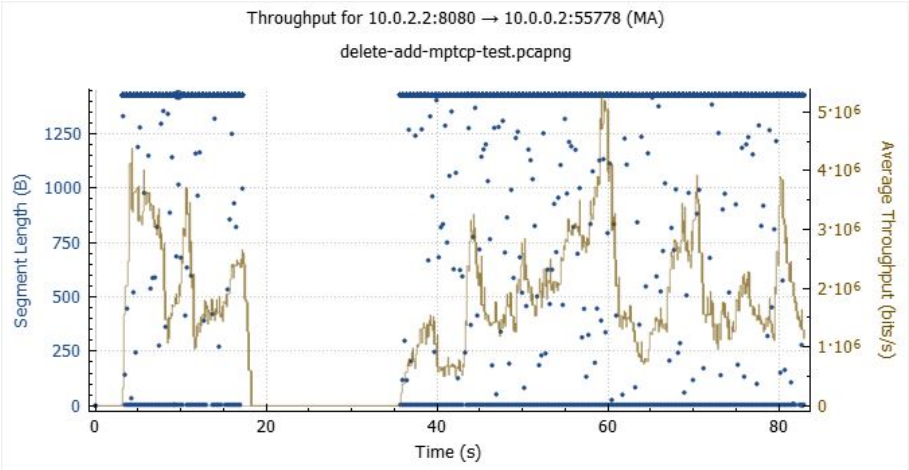


Figure 5: Throughput of Link 1 when Link 2 is down from 45-60s

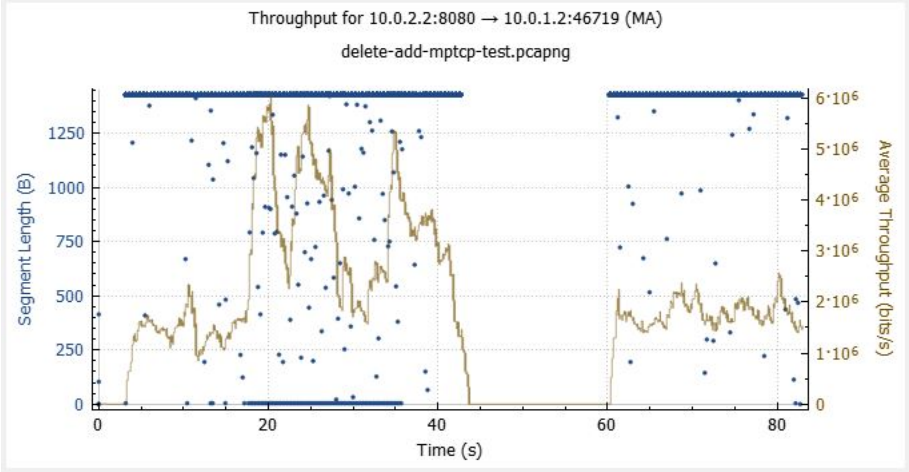


Figure 6: Throughput of Link 2 when Link 1 is down from 19-35s

2.4 Our Intuition to Use MP_JOIN

2.4.1 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.

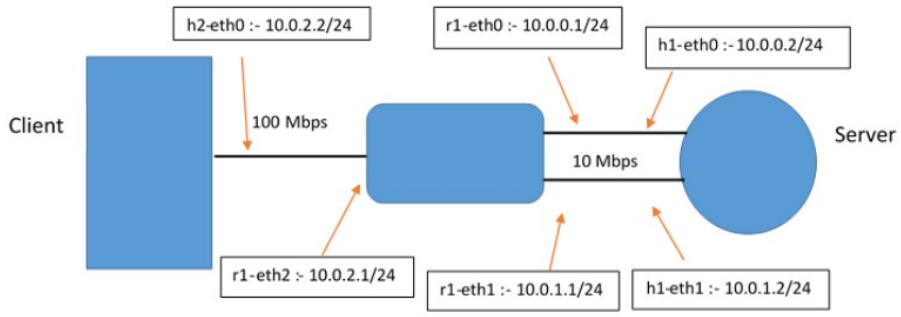


Figure 7: Swapped Topology: Server has two links

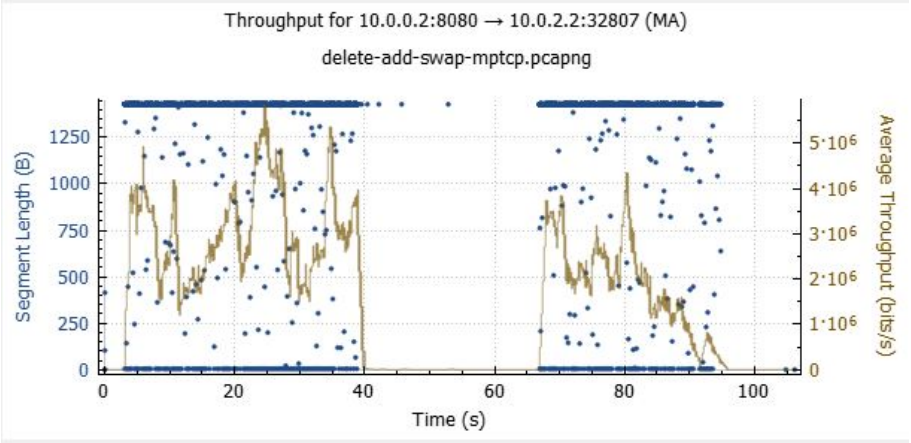


Figure 8: Throughput of Link 2 when Link 1 is down from 40-65s

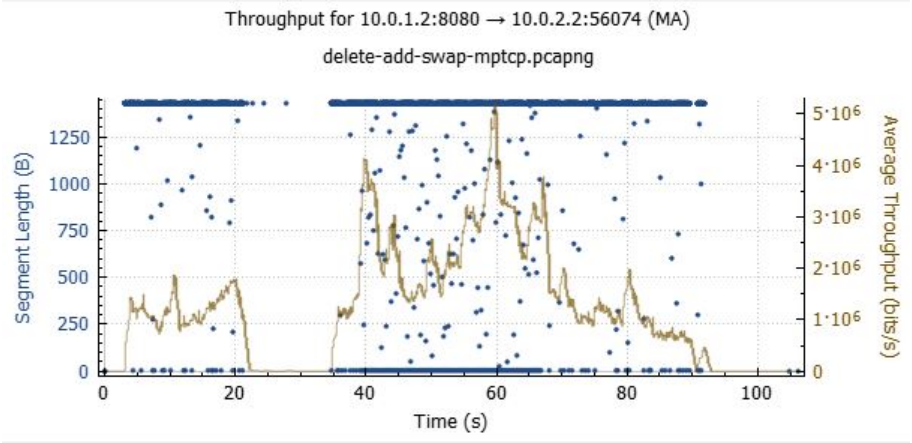


Figure 9: Throughput of Link 1 when Link 2 is down from 23-35s

2.4.2 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.

2.5 Connecting to Multiple Servers by Redirecting MP_JOIN

2.5.1 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).

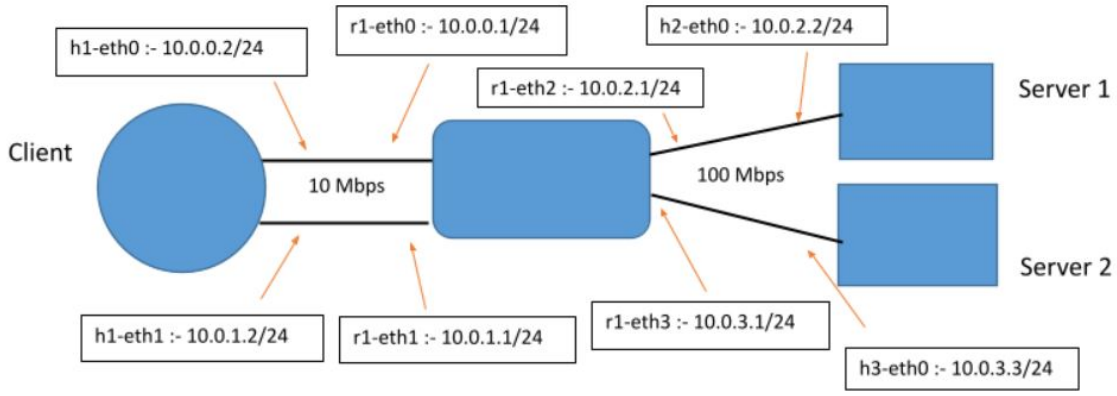


Figure 10: Topology: Connecting to Multiple Servers

2.5.2 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("----------------------------");
  }
}
```

2.5.3 Observations

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

2.5.4 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

Adaptive Streaming Over Multiple Servers Using SDN

3.1 Technicality

- MPD file - A media presentation description (MPD) file is used to store information about the different streams and the bandwidths with which they are related. The MPD file tells the browser where the various bits of media are placed; it also provides metadata such as mimeType and codecs and other details such as byte-ranges - it is an XML document that will be created in many circumstances.
- Chord DHT - Chord is a protocol and algorithm for peer-to-peer distributed hash table. A distributed hash table saves key-value pairs by allocating keys to various computers (referred to as "nodes"); each node keeps the values for all the keys for which it is responsible. Chord defines how keys are assigned to nodes and how a node may determine the value of a particular key by first identifying the node responsible for that key.
- We used the following tools and testbeds for this section of our project:
  - Network Simulation: Mininet
  - MPD file generator: python-ffmpeg-video-streaming

– SDN: Ryu

– Packet Tracing: Wireshark

– Video Streaming: VLC Media Player

3.2 Topology

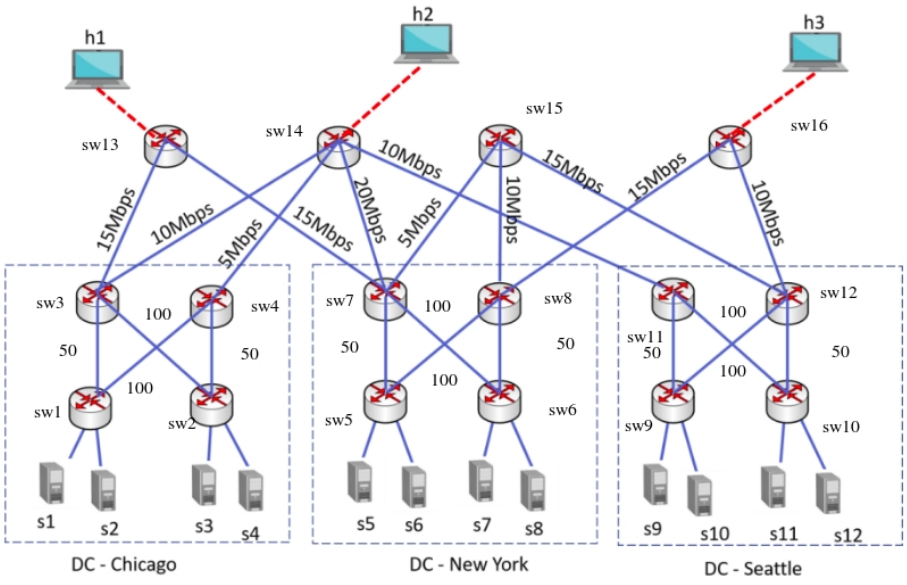


Figure 11: Topology: Using SDN for Adaptive Streaming over Multiple Servers

Here we have a topology of 16 switches, 3 clients, and 12 servers. There are multiple paths from one node to another, with different bandwidths.

3.3 Working

3.3.1 Generating MPD File

First of all, a MPD file for the video was generated using the FFMPEG python library. Along with the MPD file, the video chunks are also generated for different quality of the video(240p, 480p, 720p etc.). This is important as these separated

chunks allow us to fetch only the required portion (the chunks which are not yet fetched) when switching to a different server. Also the different bitrate/quality chunks helps to switch the video quality when needed like in case of bandwidth dropping.

3.3.2 Chord DHT

Next a peer-to-peer network is created using Chord DHT. The main purpose of chord for now is to provide the list of nodes which contain the MPD file for the video which the client wants to stream. It also keeps tracks of new nodes joining and the existing nodes leaving the network.

3.3.3 Virtual IP

Switching servers mid-stream would also require to create a new TCP session as the IP address for the new server would be different. But this would be quite problematic as the application (i.e VLC Media Player) doesn't keep track of the information about the chunks that were already fetched and the chunk that was being fetched when switching to a new TCP session. It would start fetching the chunks from the beginning.
To solve these issue, we are instead using a virtual (dummy) IP address to represent the servers. The application connects to one virtual IP while the SDN, behind the scenes, maps this virtual IP address to an available server. This way even when server is switched mid-way, the application has no idea about this as it still makes requests to the the virtual IP. This way the information about the chunks are maintained. The virtual IP used during the tests is '10.0.0.100'.
Also one scenario could be that the host/client is behind NAT but this won't affect our mapping of Virtual IP because NAT only masks the IP addresses and port number of the host. If we consider that the border router or in this case the border switch that has the NAT table, it converts the local (private) IP address to a global (public) IP address. But no change is done to the destination IP address. It would be handled as in the flow rules installed in the border switch.

3.3.4 Processing request to Virtual IP

```
if dst_ip == self.virtual_ip:
  self.logger.info("---Processing␣request␣to␣virtual␣IP---")
  ...
  ...
  if result:
    src_sw, dst_sw = result[0], result[1]
    if dst_sw:
      path = self.get_path(src_sw, dst_sw, weight=self.weight)
      self.logger.info("---PATH␣TO␣SERVER---")
      self.logger.info("[PATH]%s<-->%s:␣%s"
              % (src_ip, server_ip, path))
      port_pair = self.get_port_pair_from_link
      (self.awareness.link_to_port, path[0], path[1])

      self.add_flow_for_virtualIP(datapath, in_port,
                src_ip, server_ip, port_pair)

      self.logger.info("***Preparing␣ARP␣REPLY***")
      reply_pkt = packet.Packet()
      e = ethernet.ethernet(src, server, ether_types.ETH_TYPE_ARP)
      a = arp.arp(1, 0x0800, 6, 4, 2,
            server, self.virtual_ip, src, src_ip)
      reply_pkt.add_protocol(e)
      reply_pkt.add_protocol(a)
      reply_pkt.serialize()
      actions = [parser.OFPActionOutput(ofproto.OFPP_IN_PORT)]
      out = parser.OFPPacketOut(
                  datapath=datapath,
                  buffer_id=ofproto.OFP_NO_BUFFER,
                  in_port=in_port,
```

```
                actions=actions,
                data=reply_pkt.data)
        datapath.send_msg(out)
        self.logger.info("***ARP␣REPLY␣SENT***")
        self.logger.info("---Saving␣Node's␣Data---")

        node = client.Client(datapath = datapath, in_port = in_port,
                mac_address = src, ip_address = src_ip,
                server_mac_address = server, server_ip = server_ip)

        if not self.client_list:
          self.client_list.append(node)
        else:
          self.client_list[0] = node

        self.logger.info("---Node's␣Data␣Saved---")

    self.logger.info("---Finished␣processing␣request␣to␣virtual␣IP---")
    return
```

When a request to virtual IP is made, SDN does two important steps to map it a real server's IP. First of all, the optimal path is calculated to the chosen server and flow rules are added to the client's gateway switch to make sure that the packets sent to the virtual IP is instead sent to the server. Next step is to prepare an ARP reply containing the MAC address of the server in response to the ARP request for the MAC address of the virtual IP.

```
def add_flow_for_virtualIP(self, datapath, in_port, src_ip,
                server_ip, port_pair):
    self.logger.info("---ADDING FLOW RULES FOR VIRTUAL IP---")
```

```
#Flow Rules for Client ---> Server
match = parser.OFPMatch(in_port=in_port,
        ipv4_dst = self.virtual_ip, eth_type=0x0800)

actions = [parser.OFPActionSetField(ipv4_dst=server_ip),
      parser.OFPActionOutput(port_pair[0])]

self.add_flow(datapath, 100, match, actions, cookie = 1)

#Flow Rules for Server ---> Client
match = parser.OFPMatch(in_port = port_pair[0],
      ipv4_src = server_ip, ipv4_dst = src_ip, eth_type=0x0800)
actions = [parser.OFPActionSetField(ipv4_src=self.virtual_ip),
      parser.OFPActionOutput(in_port)]

self.add_flow(datapath, 100, match, actions, cookie = 1)

self.logger.info("---FLOW RULES FOR VIRTUAL IP ADDED---")
```

A typical packet that is sent from the client to the gateway switch has the following information:

- SOURCE MAC (Client's MAC Address)
- SOURCE IP (Client's IP)
- SOURCE PORT
- DESTINATION MAC (The MAC address of the server assigned to the client)
- DESTINATION IP (Virtual IP - 10.0.0.100)
- DESTINATION PORT

Once the SDN installs the flow rules in the gateway switch, every time a packet matching the destination IP of the virtual IP, it modifies the packet and changes the Destination IP to the real IP of the server. So once the packet is forwarded from the gateway switch, the DESTINATION IP is the real IP of the server. Similarly, every time a packet with Source IP as server's IP arrives at the gateway switch, the switch modifies the source IP to Virtual IP.

3.3.5 Choosing the best Server

The RYU network module used in this project allows for collecting different useful information about the topology such as the bandwidth, delays, number of hops etc. Using these statistics, the list of available servers are sorted from best to worst. Initially, when a client makes a request to the virtual IP, the SDN maps it to the IP address of the any of the available server at that time. Then, at regular intervals, the servers are checked and sorted and then switched to best server available.

```
def sort_best_server(self):
 self.logger.info('---Sorting Servers---')
  for client in self.client_list:
   cost = {}
    for server in self.servers:
     ...
     ...
     if result:
      src_sw, dst_sw = result[0], result[1]
       if dst_sw:
        path = self.get_path(src_sw, dst_sw, weight=self.weight)
        self.logger.info("---PATH TO SERVER---")
        self.logger.info("[PATH]%s<-->%s: %s"
            %(client.ip_address, server_ip, path))

        hops = len(path)
        for i in range(0, len(path)-1):
```

```
            self.logger.info('---Path[i] = %s, Path[i+1] = %s'
              %(path[i], path[i+1]))
            total_delay =
              self.awareness.graph[path[i]][path[i+1]]['delay']

          cost[server]['hops'] = hops
          cost[server]['delay'] = total_delay

      n = len(self.servers)

      for i in range(n-1):
       for j in range(0, n-i-1):
        diff = cost[self.servers[j]]['delay'] -
            cost[self.servers[j+1]]['delay']
        if abs(diff) < 0.000001:
         if cost[self.servers[j][hops]] >
          cost[self.servers[j+1]]['hops']:
           self.servers[j], self.servers[j+1] =
              self.servers[j+1], self.servers[j]
        elif diff > 0:
          self.servers[j], self.servers[j+1] =
              self.servers[j+1], self.servers[j]
    self.logger.info('---Servers Sorted---')
```

3.3.6 Switching Servers

The switching of servers is done using a separate thread, which every few seconds, checks and sorts the list of available servers (given by the chord) from best to worse. Suppose a new server comes having better connection than the one that the client is currently connected to, the thread switches the server to the new one. Or if the current connection becomes worse, maybe due to change in bandwidth, delays etc, it would change the server to the next best one. Hence, the IP address of the server

is being changed dynamically.

There are two main steps performed while switching the servers. First is rewriting the existing flows, which maps the virtual IP to the old server. This is done by removing the old flow rules and adding new ones. Next step is to generate an ARP request from the client side so that an ARP reply can be generated to provide the client with the MAC address of the new server.

```
def _server_manager(self):
  while True:
    self.sort_best_server()
    hub.sleep(setting.CHECK_SERVER_PERIOD)
    if len(self.client_list) > 0:
      for client in self.client_list:
        ...
        if result:
          src_sw, dst_sw = result[0], result[1]

          if dst_sw:
            path = self.get_path(src_sw, dst_sw, weight=self.weight)
            ...
            self.remove_flow_for_virtualIP(client.datapath,
              client.in_port, client.ip_address,
              server_ip, port_pair)

            client.server_ip = server_ip
            client.server_mac_address = self.servers[0]
        self.logger.info("---SERVER CHANGED---")
        self.logger.info("---GENERATE ARP REQUEST FOR VIRUTAL IP---")

        ...
        pkt = packet.Packet()
```

```
e = ethernet.ethernet('00:00:00:00:00:00',
    '00:00:00:00:00:01', ether_types.ETH_TYPE_ARP)
a = arp.arp(1, 0x0800, 6, 4, 1,
    '00:00:00:00:00:01', client.ip_address,
    '00:00:00:00:00:00', self.virtual_ip)
...
actions = [parser.OFPActionOutput(ofproto.OFPP_FLOOD)]

out = parser.OFPPacketOut(datapath = client.datapath,
            buffer_id = ofproto.OFP_NO_BUFFER,
            actions = actions,
            in_port = client.in_port,
            data = pkt.data
            )
client.datapath.send_msg(out)

self.logger.info("---ARP REQUEST FOR VIRUTAL IP SENT---")
```

3.4 Observations

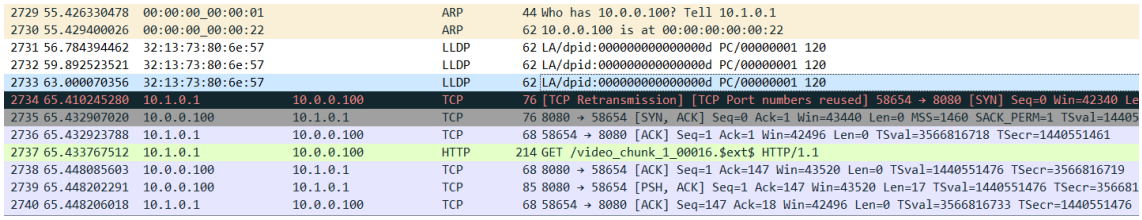
We observed the Wireshark Packet Trace to check whether the server switching was working properly or not.

4	7.515582336	00:00:00_00:00:01		ARP	44	Who has 10.0.0.100? Tell 10.1.0.1
5	7.532765714	00:00:00_00:00:34		ARP	62	10.0.0.100 is at 00:00:00:00:00:34
6	7.532772365	10.1.0.1	10.0.0.100	TCP	80	58588 → 8080 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 TSval=3566758800 TSecr=0 WS=512
7	7.542070581	10.0.0.100	10.1.0.1	TCP	76	8080 → 58588 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=773860021 TSec
8	7.542084215	10.1.0.1	10.0.0.100	TCP	68	58588 → 8080 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=3566758827 TSecr=773860021
9	7.542107531	10.1.0.1	10.0.0.100	HTTP	208	GET /video.mpd HTTP/1.1
10	7.556980001	10.0.0.100	10.1.0.1	TCP	68	8080 → 58588 [ACK] Seq=1 Ack=141 Win=43520 Len=0 TSval=773860036 TSecr=3566758827
11	7.557090032	10.0.0.100	10.1.0.1	TCP	98	8080 → 58588 [PSH, ACK] Seq=1 Ack=141 Win=43520 Len=30 TSval=773860036 TSecr=3566758827 [TCP
12	7.557093484	10.1.0.1	10.0.0.100	TCP	68	58588 → 8080 [ACK] Seq=141 Ack=31 Win=42496 Len=0 TSval=3566758842 TSecr=773860036
13	7.557205043	10.0.0.100	10.1.0.1	TCP	328	8080 → 58588 [PSH, ACK] Seq=31 Ack=141 Win=43520 Len=260 TSval=773860036 TSecr=3566758842 [TC
14	7.557207271	10.1.0.1	10.0.0.100	TCP	68	58588 → 8080 [ACK] Seq=141 Ack=291 Win=42496 Len=0 TSval=3566758842 TSecr=773860036
15	7.563730860	10.0.0.100	10.1.0.1	TCP	2964	8080 → 58588 [PSH, ACK] Seq=291 Ack=141 Win=43520 Len=2896 TSval=773860043 TSecr=3566758842 [

Figure 12: Initial request to Virtual IP

When the initial ARP request is sent out by the application (client) for the virtual IP, the server having the MAC Address 00:00:00:00:00:34 (IP - '10.0.2.4') is

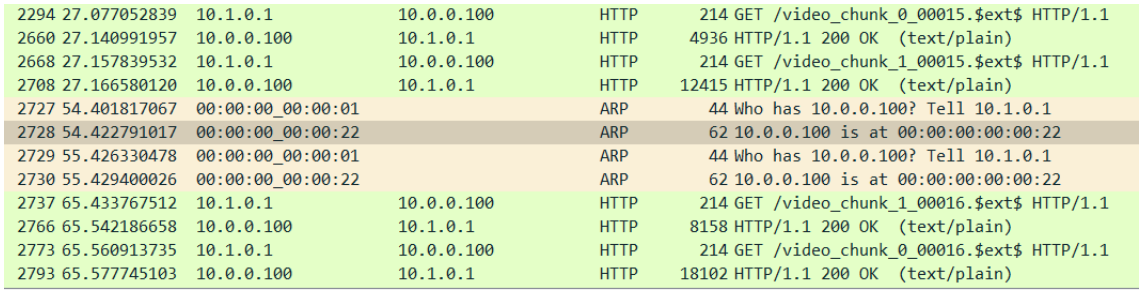
assigned to the client by the SDN. The application doesn't have any idea about the real IP and communicates with the virtual IP.



2729	55.426330478	00:00:00_00:00:01		ARP	44	Who has 10.0.0.100? Tell 10.1.0.1
2730	55.429400026	00:00:00_00:00:22		ARP	62	10.0.0.100 is at 00:00:00:00:00:22
2731	56.784394462	32:13:73:80:6e:57		LLDP	62	LA/dpid:000000000000000d PC/00000001 120
2732	59.892523521	32:13:73:80:6e:57		LLDP	62	LA/dpid:000000000000000d PC/00000001 120
2733	63.000070356	32:13:73:80:6e:57		LLDP	62	LA/dpid:000000000000000d PC/00000001 120
2734	65.410245280	10.1.0.1	10.0.0.100	TCP	76	[TCP Retransmission] [TCP Port numbers reused] 58654 → 8080 [SYN] Seq=0 Win=42340 Le
2735	65.432907020	10.0.0.100	10.1.0.1	TCP	76	8080 → 58654 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SACK_PERM=1 TSval=14405
2736	65.432923788	10.1.0.1	10.0.0.100	TCP	68	58654 → 8080 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=3566816718 TSecr=1440551461
2737	65.433767512	10.1.0.1	10.0.0.100	HTTP	214	GET /video_chunk_1_00016.ext HTTP/1.1
2738	65.448085603	10.0.0.100	10.1.0.1	TCP	68	8080 → 58654 [ACK] Seq=1 Ack=147 Win=43520 Len=0 TSval=1440551476 TSecr=3566816719
2739	65.448202291	10.0.0.100	10.1.0.1	TCP	85	8080 → 58654 [PSH, ACK] Seq=1 Ack=147 Win=43520 Len=17 TSval=1440551476 TSecr=356681
2740	65.448206018	10.1.0.1	10.0.0.100	TCP	68	58654 → 8080 [ACK] Seq=147 Ack=18 Win=42496 Len=0 TSval=3566816733 TSecr=1440551476

Figure 13: Switching to a different server

Later on, when the SDN sorts the servers, it finds that a different server is best for the client so it tries to switch the server. In the above packet trace, we can see that application once again broadcasts an ARP request for the virtual IP as the SDN deleted the flow rules to switch servers. This time, the SDN maps the server having the MAC Address 00:00:00:00:00:22 (IP - '10.0.1.2') to the client. Thus, successfully switching servers.



2294	27.077052839	10.1.0.1	10.0.0.100	HTTP	214	GET /video_chunk_0_00015.ext HTTP/1.1
2660	27.140991957	10.0.0.100	10.1.0.1	HTTP	4936	HTTP/1.1 200 OK (text/plain)
2668	27.157839532	10.1.0.1	10.0.0.100	HTTP	214	GET /video_chunk_1_00015.ext HTTP/1.1
2708	27.166580120	10.0.0.100	10.1.0.1	HTTP	12415	HTTP/1.1 200 OK (text/plain)
2727	54.401817067	00:00:00_00:00:01		ARP	44	Who has 10.0.0.100? Tell 10.1.0.1
2728	54.422791017	00:00:00_00:00:22		ARP	62	10.0.0.100 is at 00:00:00:00:00:22
2729	55.426330478	00:00:00_00:00:01		ARP	44	Who has 10.0.0.100? Tell 10.1.0.1
2730	55.429400026	00:00:00_00:00:22		ARP	62	10.0.0.100 is at 00:00:00:00:00:22
2737	65.433767512	10.1.0.1	10.0.0.100	HTTP	214	GET /video_chunk_1_00016.ext HTTP/1.1
2766	65.542186658	10.0.0.100	10.1.0.1	HTTP	8158	HTTP/1.1 200 OK (text/plain)
2773	65.560913735	10.1.0.1	10.0.0.100	HTTP	214	GET /video_chunk_0_00016.ext HTTP/1.1
2793	65.577745103	10.0.0.100	10.1.0.1	HTTP	18102	HTTP/1.1 200 OK (text/plain)

Figure 14: The information about chunks is maintained

As mentioned in the Virtual IP section, we don't have to worry about keeping track of the request that were already sent and those that need to be sent after changing servers as evident from the above packet trace. Since the client has no idea that the servers have changed, as it still is communicating to the same Virtual IP, it will continue with the next request that needs to be sent.

In the above packet trace, the server initially was 00:00:00:00:00:34, the last chunk that was requested was 'video_chunk_1_00015' and after switching to 00:00:00:00:00:22, the first chunk that was requested was 'video_chunk_1_00016'.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

Hence, we used SDN and Chord DHT to provide adaptive streaming over multiple servers and thus eliminating a single point of failure. This also shows that protocols like DASH, which is used for adaptive streaming can also be achieved using SDN only.

4.2 Future Work

- As of now, we can only connect to one server at a time using the virtual IP. But like we tried in our MPTCP approach to connect to multiple servers simultaneously using sub-flows, we can also explore the possibility of simultaneously connecting to multiple servers using regular TCP connections by maintaining one active (main) and several passive (sub-flows) TCP connections.
- Also one of our main goals is to decentralize video streaming i.e. each peer that streams a video should also be able to relay the video stream to other peers as well. In order to make this work, our peers should be able to buffer/store chunks of the video stream (for some duration) and the SDN should be aware of various content across peers in order to deliver a smooth streaming experience to other peers.

- Currently, one of the popular protocols for adaptive streaming being used is DASH (Dynamic Adaptive Streaming over HTTP). But DASH has an overhead of using Application Layer Ping to determine the network conditions of the client/server. But our approach using SDN does not have such overhead. Therefore, we can also evaluate the performance of Adaptive Streaming using SDN in comparison to DASH.

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] :
https://ieeexplore.ieee.org/document/7502424

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

[5] Multipath Dynamic Adaptive Streaming over HTTP Using Scalable Video Coding in Software Defined Networking
https://www.mdpi.com/2076-3417/10/21/7691

[6] Increasing End-User's QoE with a Hybrid P2P/Multi-Server streaming solution based on dash.js and webRTC
https://hal.archives-ouvertes.fr/hal-01585219/document

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

[8] Network Awareness Module for RYU Controller
https://github.com/muzixing/ryu/

[9] Chord Protocol - A Distributed Hash Table
https://github.com/bhavinkotak07/chord$_protocol$

[10] Topology Used for testing Adaptive Streaming using SDN
https://github.com/amitsk1994/mininet-RYU-ShortestPath

20188108

ORIGINALITY REPORT

6%	4%	0%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.coursehero.com Internet Source	4%
2	Submitted to Engineers Australia Student Paper	1%

Exclude quotes	On	Exclude matches	< 50 words
Exclude bibliography	On		