ECE 4400/6400 Final Report: Video Streaming

Team Members: Andrew Eubanks, Eric Mitchell, Joshua Silva, Sam Quan

Motivation

Video streaming is one of the most popular internet data requests today. Through various platforms such as YouTube, Netflix, and Hulu, people stream movies, TV shows, or videos they wish to see. Our group members enjoy watching videos online, so we chose this topic.

When people stream video, there is always a wait before the video loads. In addition, the video can stop playing and start buffering. We wanted to explore the networking concepts that cause these types of video delays or automatic quality adjustments in the middle of playback.

Background and Project Description

Streaming services use a transport layer protocol to send content to user devices. As an example of a streaming service, we found that YouTube uses TCP to establish a connection between the server and a device before sending data. This means there will be some overhead before the device can start streaming. In our project, we wish to measure how delay is affected by various network conditions such as user connection speed, server queue size, and video quality (higher packet output rate). We would also like to observe packet loss under varying home network capabilities.

Network Architecture

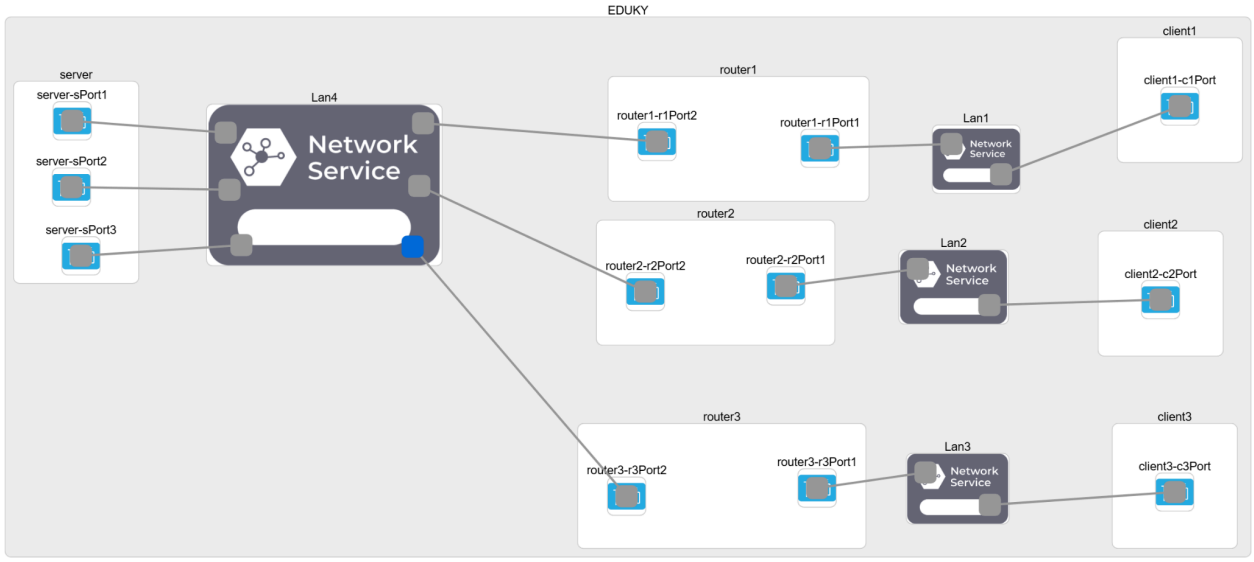


Figure 1: Network architecture

The network architecture is a star network with the highest network node representing the internet service provider (ISP). There are three router nodes that connect to the streaming

platform's three server ports via a L3 network. Each router has one other port which connects to the corresponding client's port. See Figure 1 for our network setup.

IP Routing is also set on every client node and server node, to allow each node to communicate with each other through the router. Furthermore, to allow video streaming, IPv4 forwarding is set on the router to let it know to accept and send packets from the server to its respective client. Afterwards, IPTables sets the rules for packets by telling the router to accept incoming packets from its input port and send them to the output port. All of these steps allow videos to be sent through, while testing the effects of limit rates, latency, etc. on the router.

Software Architecture

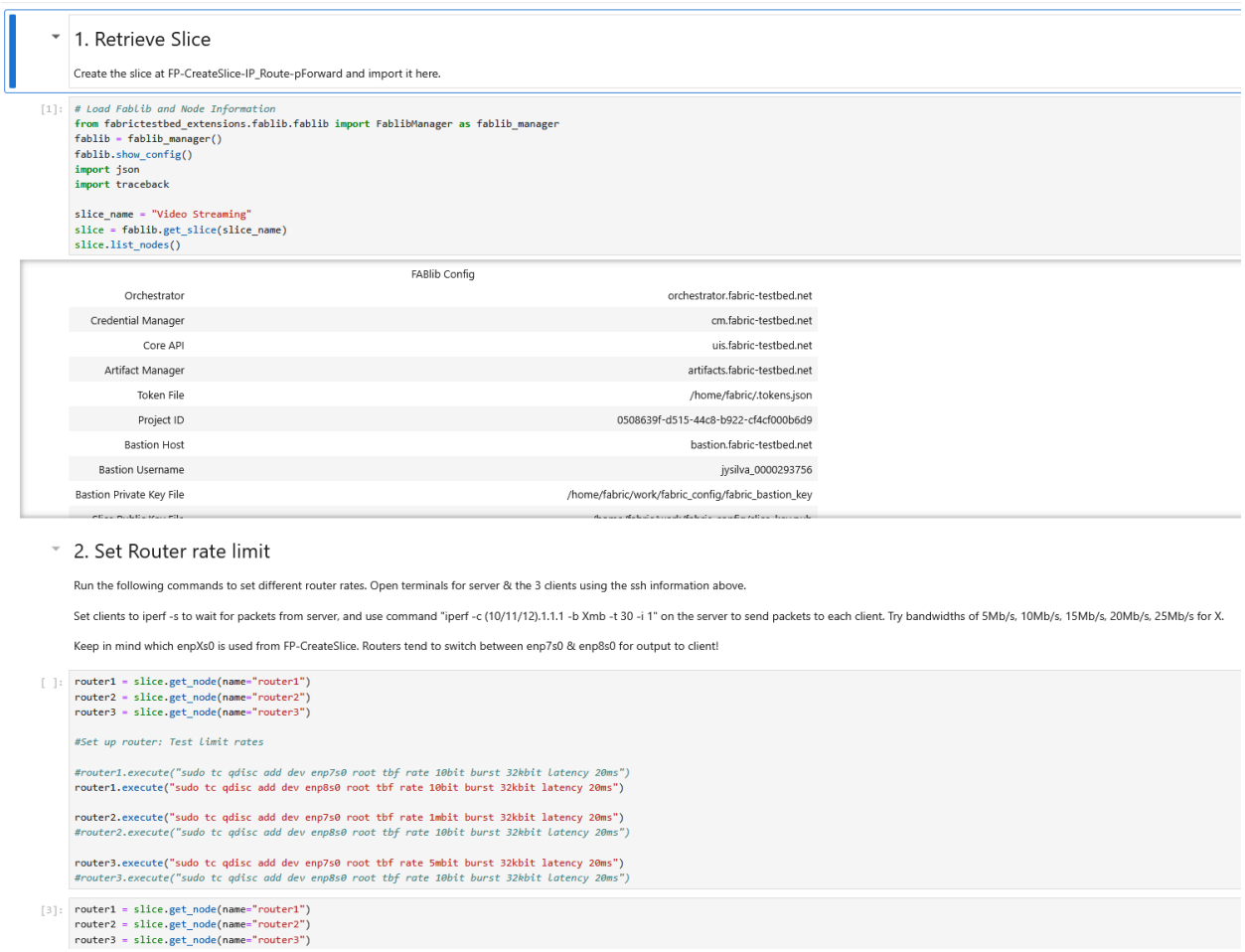
We used a few different software tools to test our theories. We used ping to make sure connections were working as expected. We used iPerf to generate continuous streams of traffic. The tc command helped us control network speeds. Lastly, we used ffmpeg to stream video through fabric to show what we discovered.

To test out the network & streaming services recommendations about network requirement, several UDP & TCP experiments are conducted. Both use iPerf for the server & client to communicate with each other, and use tc to set router parameters such as limit rate, latency/delay, queue size etc. The experiments are implemented using FABRIC Portal, and Jupyter Notebook for easy access to terminals & Python script to setup the network & install necessary libraries.

Taking inspiration from previous experiments during this course, each experiment is intended to have a "CreateSlice" and a "Notebook" .ipynb file. The CreateSlice allocates resources for the network, and clarifies the connections & naming of each node in the slice. The "Notebook" files differ depending on the experiment, but are designed to simplify the experiment by running scripts that assign differing router values & input bandwidths to the client from the server. See below in Figure 2 and Figure 3.

[19]:
```
# Define and Submit Slice
slice_name = "Video Streaming"
site = "EDUKY"

image = "default_ubuntu_20"
nicmodel = "NIC_Basic"

cores = 1
ram = 2
disk = 10

try:
    # Create Slice
    slice = fablib.new_slice(name=slice_name)
    # Server
    server = slice.add_node(name="server", site=site)
    server.set_capacities(cores=cores, ram=ram, disk=disk)
    server.set_image(image)
    sPort1 = server.add_component(model=nicmodel, name="sPort1").get_interfaces()[0]
    sPort2 = server.add_component(model=nicmodel, name="sPort2").get_interfaces()[0]
    sPort3 = server.add_component(model=nicmodel, name="sPort3").get_interfaces()[0]

    # client 1
    client1 = slice.add_node(name="client1", site=site)
    client1.set_capacities(cores=cores, ram=ram, disk=disk)
    client1.set_image(image)
    c1Port = client1.add_component(model=nicmodel, name="c1Port").get_interfaces()[0]

    # client 2
    client2 = slice.add_node(name="client2", site=site)
    client2.set_capacities(cores=cores, ram=ram, disk=disk)
    client2.set_image(image)
    c2Port = client2.add_component(model=nicmodel, name="c2Port").get_interfaces()[0]

    # client 3
    client3 = slice.add_node(name="client3", site=site)
    client3.set_capacities(cores=cores, ram=ram, disk=disk)
    client3.set_image(image)
    c3Port = client3.add_component(model=nicmodel, name="c3Port").get_interfaces()[0]

    # router 1
    router1 = slice.add_node(name="router1", site=site)
    router1.set_capacities(cores=cores, ram=ram, disk=disk)
    router1.set_image(image)
    r1Port1 = router1.add_component(model=nicmodel, name="r1Port1").get_interfaces()[0]
    r1Port2 = router1.add_component(model=nicmodel, name="r1Port2").get_interfaces()[0]

    # router 2
    router2 = slice.add_node(name="router2", site=site)
    router2.set_capacities(cores=cores, ram=ram, disk=disk)
    router2.set_image(image)
    r2Port1 = router2.add_component(model=nicmodel, name="r2Port1").get_interfaces()[0]
    r2Port2 = router2.add_component(model=nicmodel, name="r2Port2").get_interfaces()[0]

    # router 3
    router3 = slice.add_node(name="router3", site=site)
    router3.set_capacities(cores=cores, ram=ram, disk=disk)
    router3.set_image(image)
    r3Port1 = router3.add_component(model=nicmodel, name="r3Port1").get_interfaces()[0]
    r3Port2 = router3.add_component(model=nicmodel, name="r3Port2").get_interfaces()[0]
```

Figure 2: Example of CreateSlice file



1. Retrieve Slice

Create the slice at FP-CreateSlice-IP_Route-pForward and import it here.

```
[1]: # Load Fablib and Node Information
from fabrictestbed_extensions.fablib.fablib import FablibManager as fablib_manager
fablib = fablib_manager()
fablib.show_config()
import json
import traceback

slice_name = "Video Streaming"
slice = fablib.get_slice(slice_name)
slice.list_nodes()
```

FABlib Config

Orchestrator	orchestrator.fabric-testbed.net
Credential Manager	cm.fabric-testbed.net
Core API	uis.fabric-testbed.net
Artifact Manager	artifacts.fabric-testbed.net
Token File	/home/fabric/.tokens.json
Project ID	0508639f-d515-44c8-b922-cf4cf000b6d9
Bastion Host	bastion.fabric-testbed.net
Bastion Username	jysilva_0000293756
Bastion Private Key File	/home/fabric/work/fabric_config/fabric_bastion_key

2. Set Router rate limit

Run the following commands to set different router rates. Open terminals for server & the 3 clients using the ssh information above.

Set clients to iperf -s to wait for packets from server, and use command "iperf -c (10/11/12).1.1.1 -b Xmb -t 30 -i 1" on the server to send packets to each client. Try bandwidths of 5Mb/s, 10Mb/s, 15Mb/s, 20Mb/s, 25Mb/s for X.

Keep in mind which enpXs0 is used from FP-CreateSlice. Routers tend to switch between enp7s0 & enp8s0 for output to client!

```
[ ]: router1 = slice.get_node(name="router1")
router2 = slice.get_node(name="router2")
router3 = slice.get_node(name="router3")

#Set up router: Test limit rates

#router1.execute("sudo tc qdisc add dev enp7s0 root tbf rate 10bit burst 32kbit latency 20ms")
router1.execute("sudo tc qdisc add dev enp8s0 root tbf rate 10bit burst 32kbit latency 20ms")

router2.execute("sudo tc qdisc add dev enp7s0 root tbf rate 1mbit burst 32kbit latency 20ms")
#router2.execute("sudo tc qdisc add dev enp8s0 root tbf rate 10bit burst 32kbit latency 20ms")

router3.execute("sudo tc qdisc add dev enp7s0 root tbf rate 5mbit burst 32kbit latency 20ms")
#router3.execute("sudo tc qdisc add dev enp8s0 root tbf rate 10bit burst 32kbit latency 20ms")
```

```
[3]: router1 = slice.get_node(name="router1")
router2 = slice.get_node(name="router2")
router3 = slice.get_node(name="router3")
```

Figure 3: Example of Notebook file

Designed Experiments - UDP

For the UDP experiments, the main goal was to test the performance of different network configurations by adjusting rate limits and video stream bandwidth. Where TCP provides retransmission for loss packets, UDP does not since it prioritizes speed rather than perfect transmission. This is used for streaming services such as Zoom and Twitch. The goal was to stream at varying bandwidth for different clients with different network setups to relate levels of bandwidths to streaming video quality. Our hypothesis was that the higher the rates (client with strong connection), the better it could handle high bandwidth levels. Generally, when you attempt to watch streams, the quality will be much better if your internet can handle it.

To perform this experiment, we set up three different clients with different rates and equal latency amounts. Since we wanted to test extremely poor connections, we tested levels of 10, 50, and 200 Mbps. Other variables such as latency were kept constant at 20 ms. To start, the server was set up to send a constant stream of UDP traffic using iperf. Then, each client was set up to

listen for traffic from the server. The server was configured to send five different levels of UDP bandwidth: 2, 4, 6, 8, and 10 Mbps. These represent the following streaming qualities: 480p, 720p, 1080p, 1440p, and 2160p. After each test, we collected the Iperf output from each client to view packet loss and other metrics. Code was written to automate this process.

Designed Experiments - TCP

For the TCP experiments, two main variables are tested to determine the impact they have on the network: input bandwidth and limit rates on the router. All other variables are kept constant for both experiments, those being latency and queue size. The variables for latency and queue size are determined from sources that state recommended queue sizes for stability, and latency typically seen in home networks. The intent of the TCP experiment, alongside the UDP experiments, is to prove that the simulated network is functional and to test out video streaming services claims about recommended download speeds. With the hypothesis that higher limit rates mean less buffering, while higher bandwidths lead to more buffering.

To do so, we set rate limits on the routers using tc, effectively being our “download speed” for the client. There are 3 routers, so each receives a different limit rate to test the effects of download speed on the client side. We tested rate limits of 100 Mb/s, 300 Mb/s, and 1 Gb/s because they are the most common network speeds offered by today’s ISPs. All other variables are kept constant, such as latency (20 ms), burst/queue (32 Kbit), and bandwidth coming from the server is kept to a constant rate (20 Mbps). After the initial router parameters are set, the clients are told to “listen” through “iperf -s”, acting as a “server” waiting for input from the “streaming service,” or server. Afterwards, the server is told to send a “video” through the use of “iperf -c”, acting as a typical “client” in a normal server-client relationship. iPerf is used instead of ffmpeg for this experiment due to iPerf providing more relevant data for our network, but the values chosen are meant to simulate a video coming from a streaming service to a client.

Transfer Rate is the data the experiment is interested in, taking into account how much data the client is able to receive. If the transfer rate does not match its expected value, (bandwidth * time) / (8 bits), then the interpretation is that there is buffering but no data loss. Due to how TCP operates on these streaming sites, the video is expected to pause or decrease in quality to allow the client to catch up, but it does not skip ahead in the video like it does with UDP.

After the initial limit rates are measured and altered to display more accurate trends for video streaming, different bandwidths are sent to the same limit rates to measure the impact of higher quality videos being streamed to the client. This is again simulated through iPerf to give more relevant data for our network. Different bandwidths are selected based on sources such as Vodlix and Epiphan, as seen below in Figure 4. The bandwidths are meant to represent different video quality, ranging from “720p” to “2160p”, to test out each limit rate’s effectiveness.

Bandwidth Requirements for Different Video Resolutions

Here are general bandwidth guidelines for various video resolutions and bitrates:

- **480p (SD):** 1.5 - 2.5 Mbps
- **720p (HD):** 3 - 5 Mbps
- **1080p (Full HD):** 5 - 8 Mbps
- **1440p (2K):** 10 - 16 Mbps
- **2160p (4K):** 20 - 35 Mbps

Figure 4: Bandwidths to simulate different Video Resolutions from VODLIX

Demonstration Description

For our demonstration, we will use the FFmpeg libraries and programs to perform real-world video streaming experiments by throttling the transmission router (simulating network speed) on a live video stream, observing the effect of lower network speeds. Since designating a proper display capture is challenging on FABRIC, we will use libcaca to view the video output as ASCII on the command line.

We will use a slice with a router between two nodes as a base for our project. Next, we install ffmpeg, yt-dlp, and libcaca. To stream across a router, we will need to configure IP table forwarding for the video streaming port, which is shown in the code snippet below.

```
#Handle IP routes & IP forwarding
client1.execute("sudo ip route add 13.1.1.0/24 via 10.1.1.2")
client2.execute("sudo ip route add 13.1.1.0/24 via 11.1.1.2")
client3.execute("sudo ip route add 13.1.1.0/24 via 12.1.1.2")

#server.execute("sudo sysctl -w net.ipv4.ip_forward=1")
server.execute("sudo ip route add 10.1.1.0/24 via 13.1.1.2")
server.execute("sudo ip route add 11.1.1.0/24 via 13.1.1.4")
server.execute("sudo ip route add 12.1.1.0/24 via 13.1.1.6")

router1.execute("sudo sysctl -w net.ipv4.ip_forward=1")
router2.execute("sudo sysctl -w net.ipv4.ip_forward=1")
router3.execute("sudo sysctl -w net.ipv4.ip_forward=1")

#Allow forwarding through routers
router1.execute("sudo iptables -A FORWARD -i enp7s0 -o enp8s0 -j ACCEPT")
router2.execute("sudo iptables -A FORWARD -i enp8s0 -o enp7s0 -j ACCEPT")
router3.execute("sudo iptables -A FORWARD -i enp7s0 -o enp8s0 -j ACCEPT")

#Install caca on clients
client1.execute("sudo apt install ffmpeg libcaca0")
client2.execute("sudo apt install ffmpeg libcaca0")
client3.execute("sudo apt install ffmpeg libcaca0")
```

Figure 5: Code to Handle IP Routing

From here, we will run a server streaming command on the server to the specified port, and make the client listen to that port to view the video. Once this is established, we can then throttle the router using a tc command to simulate a weaker network and observe the effects on the video quality. To stream, we will use an ffmpeg command on the server to a port on the client IP, and another FFmpeg on the client to listen for input from the server IP on the same port. The FFmpeg commands are listed below.

```
ubuntu@server:~$ ffmpeg -re -i video.mp4 -c:v libx264 -f mpegts udp://10.1.1.1:1234
ubuntu@client1:~$ ffmpeg -i udp://13.1.1.1:1234 -pix_fmt rgb24 -f caca -
```

Figure 6: Commands to Stream

We will run a video streaming test with UDP and TCP. The type of errors we expect to see are different for each type of stream due to the characteristics of both. For UDP, we expect to see dropped frames and loss of data as we throttle the network speed. For TCP, we expect to not drop any frames or data, but we do expect to see buffering and much slower “playback” speed.

Significance/Relation to Networking

Our project is related to networking because it explores queuing and it explores the difference between bandwidth and throughput. The effects of having different queue sizes will be examined by limiting the queue size at the server side. This will enhance our understanding of how output queue size affects delay and packet loss in different transmission scenarios. Also, we will be able to observe the server while it operates at different bandwidths, which will eventually limit throughput as the limit is decreased. We will also have a better understanding of how output rate affects packet loss and delay with different receiving bandwidths on the other end.

Outcomes and Explanation

From our experiments, we can theorize what effects we would see in actual video streaming. Using UDP, we can see that there was significantly more packet loss for the client with a poor connection than the client with a great connection. We can also see that as the bandwidth for streaming was increased, packet loss increased overall. Using TCP, we see that the transfer rate drops significantly when the router can no longer handle the amount of data being sent through it. The queue fills up, and the network becomes unstable. A similar phenomenon occurs when using UDP to send data. When the network becomes overloaded, the router begins to drop many packets. Since no data can be sent in either case, we would expect to have the video begin buffering or stop streaming completely. We tested this theory using FFMPEG. When we recreated the network conditions while streaming, we got our expected result in part. However, the video began to reduce in quality instead of stopping completely. This is part of Youtube's adaptive streaming which is also present on other platforms.

For the UDP experiments, the dependent variables were the bandwidth that was tested for each client and the latency. The independent variables were the limit rates that were present on each network. The results of the experiment showed that the client with a great connection was able to stream content at all bandwidth rates, meaning that they would be able to view streams at all quality levels. The client with the moderate connection was only able to stream up to 720p before seeing packet loss, and anything beyond that led to significant packet loss. Lastly, the client with a very poor connection received packet loss at all levels of bandwidth, so they might be able to stream at low quality levels of 420p with some buffering. Our results from this experiment can be seen in Table 1 and Figure 7.

For the TCP experiments, the dependent variables are bandwidth and limit rates, while the independent variables are burst/queue and latency. The results are intended to prove network functionality and to test popular video streaming services recommendations. The results of the experiments can be seen below in Figure 8.

From the data collected, it is evident that higher limit rates minimize transfer rate loss, or buffering for our purposes. Meanwhile, higher bandwidths lead to higher transfer rate loss, again buffering in our case. This is expected behavior, as more traffic leads to higher chances of "traffic jams" if our limit rates are not able to prevent clogging in our queue.

With the data collected, Netflix's recommendations for streaming tend to match the results, but seem to allow for buffering. This is probably due to the experiment choosing the higher end of bandwidth rates from Vodlix to simulate each resolution, due to the resilience of TCP when it comes to preventing packet loss.

Otherwise, the data concludes that a minimum download speed of 5 Mb/s is needed to watch HD (720p) videos if the user is fine with a small chance of buffering (<5%). From there, each download speed increases by about 5 Mb/s to 10 Mb/s in order to handle the next highest resolution with minimum or no chance of buffering.

		Packet Loss (%)		
Bandwidth (Mbps)	Associated Quality	Client 1 (Poor Connection)	Client 2 (Moderate Connection)	Client 3 (Great Connection)
2	480p	55	0	0
4	720p	75	0	0
6	1080p	85	9.3	0
8	1440p	87	33	0
10	2160p	91	47	0

Table 1: Results From UDP Experiment

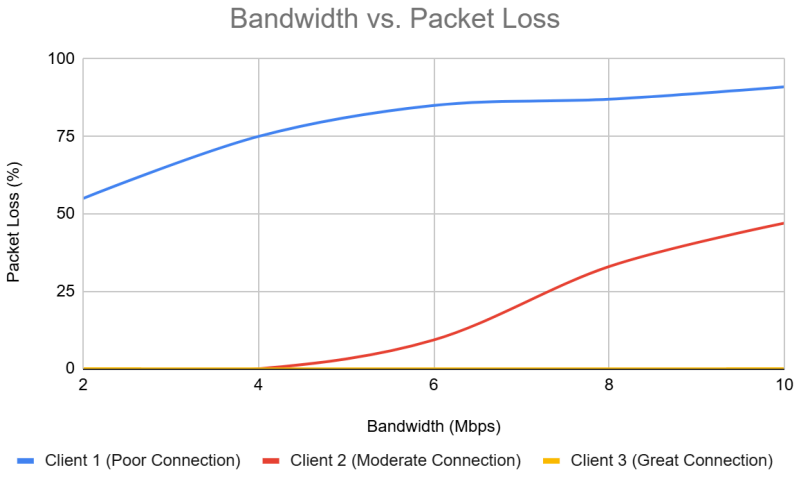


Figure 7: Graph of Results From UDP Experiment

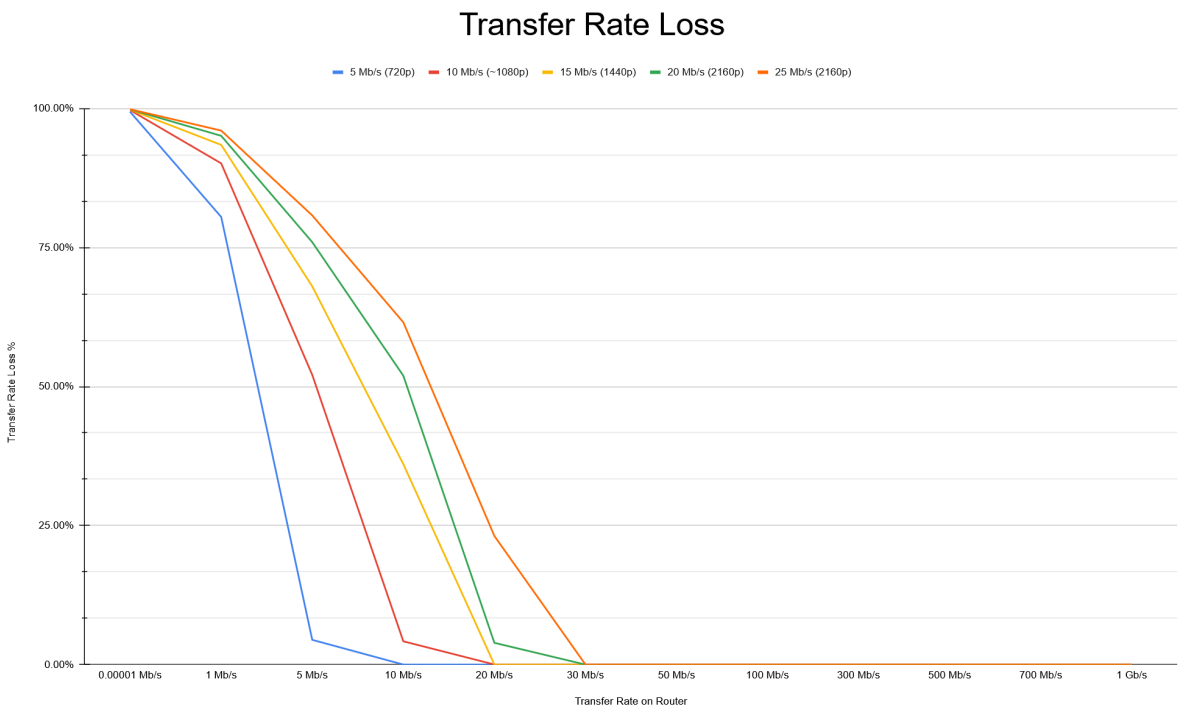


Figure 8: Effects of differing bandwidth & limit rates on TCP

Internet connection speed recommendations

To watch TV shows and movies on Netflix, we recommended having a stable internet connection with a download speed shown below in megabits per second (Mbps).

Video quality	Resolution	Recommended speed
High definition (HD)	720p	3 Mbps or higher
Full high definition (FHD)	1080p	5 Mbps or higher
Ultra high definition (UHD)	4K	15 Mbps or higher

Figure 9: Netflix Recommendations for Download Speed

```
ubuntu@server:~$ ffmpeg -re -i rick.mp4 -c:v libx264 -f mpegts udp://10.1.1.1:1234
ffmpeg version 4.2.7-0ubuntu0.1 Copyright (c) 2000-2022 the FFmpeg developers
  built with gcc 9 (Ubuntu 9.4.0-1ubuntu1~20.04.1)
  configuration: --prefix=/usr --extra-version=0ubuntu0.1 --toolchain=hardened --libdir=/usr/lib/x86_64-linux-gnu --incdir=/usr/include/x86_64-linux-gnu --arch=amd64 --enable-gpl --disable-stripping --enable-avresample --d
isable-filter=resample --enable-avisynth --enable-gnutls --enable-ladspa --enable-libaom --enable-libass --enable-libbluray --enable-libbs2b --enable-libcaca --enable-libcdio --enable-libcodec2 --enable-libflite --enable-l
ibfontconfig --enable-libfreetype --enable-libfribidi --enable-libgme --enable-libgsm --enable-libjack --enable-libmp3lame --enable-libmysofa --enable-libopenjpeg --enable-libopenmpt --enable-libopus --enable-libpulse --en
able-librsvg --enable-librubberband --enable-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh --enable-libtheora --enable-libtwolame --enable-libvidstab --enable-libvorbis --enable-libvpx --en
able-libwavpack --enable-libwebp --enable-libx265 --enable-libxml2 --enable-libxvid --enable-libzmq --enable-libzvbi --enable-lv2 --enable-omx --enable-openal --enable-opencl --enable-opengl --enable-sdl2 --enable-libdc139
4 --enable-libdrm --enable-libiec61883 --enable-nvenc --enable-chromaprint --enable-frei0r --enable-libx264 --enable-shared
  libavutil      56. 31.100 / 56. 31.100
  libavcodec     58. 54.100 / 58. 54.100
  libavformat    58. 29.100 / 58. 29.100
  libavdevice    58.  8.100 / 58.  8.100
  libavfilter     7. 57.100 /  7. 57.100
  libavresample   4.  0.  0 /  4.  0.  0
  libswscale      5.  5.100 /  5.  5.100
  libswresample   3.  5.100 /  3.  5.100
  libpostproc    55.  5.100 / 55.  5.100
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'rick.mp4':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2mp41
    encoder         : Lavf58.76.100
  Duration: 00:03:32.05, start: 0.000000, bitrate: 3884 kb/s
    Stream #0:0(und): Video: vp9 (Profile 0) (vp09 / 0x39307076), yuv420p(tv, bt709), 1920x1080, 3751 kb/s, 25 fps, 25 tbr, 16k tbn, 16k tbc (default)
    Metadata:
      handler_name    : ISO Media file produced by Google Inc. Created on: 04/29/2025.
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 44100 Hz, stereo, fltp, 128 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
Stream mapping:
  Stream #0:0 -> #0:0 (vp9 (native) -> h264 (libx264))
  Stream #0:1 -> #0:1 (aac (native) -> mp2 (native))
Press [q] to stop, [?] for help
[libx264 @ 0x562973f2af00] using cpu capabilities: MMX2 SSE2Fast SSSE3 SSE4.2 AVX FMA3 BMI2 AVX2
[libx264 @ 0x562973f2af00] profile High, level 4.0
Output #0, mpegts, to 'udp://10.1.1.1:1234':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2mp41
    encoder         : Lavf58.29.100
    Stream #0:0(und): Video: h264 (libx264), yuv420p(progressive), 1920x1080, q=-1--1, 25 fps, 90k tbn, 25 tbc (default)
    Metadata:
      handler_name    : ISO Media file produced by Google Inc. Created on: 04/29/2025.
      encoder         : Lavc58.54.100 libx264
    Side data:
      cpb: bitrate max/min/avg: 0/0/0 buffer size: 0 vbv_delay: -1
    Stream #0:1(und): Audio: mp2, 44100 Hz, stereo, s16, 384 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
      encoder         : Lavc58.54.100 mp2
frame=  199 fps= 12 q=28.0 size=    2893kB time=00:00:07.93 bitrate=2988.3kbits/s speed=0.484x
```

Figure 10: Command Line Output from Server for Streaming

Figure 11: Client View from Viewing Stream

Conclusion

From our experiments, we got the results we expected. As the network begins to become unstable, the streaming quality and smoothness go down. This is because the client can no longer receive the video packets to show on-screen. This is present in real-world applications as well, but with slight differences. Streaming platforms such as YouTube use adaptive streaming. As network conditions begin to degrade, YouTube begins to reduce the quality of the video to conserve network resources. When nothing can be sent, the video begins to buffer. In the case of platforms such as Zoom, which use UDP, the stream freezes temporarily when the connection is lost.

Theoretically, network delay or other devices on a network may prevent streaming devices from receiving packets for longer than expected. While we did not experiment with this aspect, our research did show that streaming platforms that use TCP have built-in methods to prevent loss of video. Some operate a few seconds in advance, storing extra video to wait out periods where no packets are received. It is only when the connection is lost for multiple seconds that the platform must stop and buffer.

Since video streaming transfer rates are fairly low, a home network at 100 Mb/s should be able to support 2 devices streaming 4K video. If devices stream 1080p instead, the network should be able to support at least 10 streaming devices simultaneously. Our standard home networks today can support many streaming devices simultaneously.

Team Member Responsibilities

- AJ: UDP Experiments, wrote code for automating experiments and making requests to different interfaces
- Eric: Set up network environment, wrote proposal and proposal presentation, TCP experiments (latency (not implemented in final tests))
- Josh: Assisted with network environment (set up IP Routing, IPv4 Forwarding, & IPTables), TCP experiments (limit rates, bandwidth)
- Sam: Researched and ran real video-streaming with FFMPEG for TCP/UDP experiment and demonstration, using libcaca to view video as ASCII

Timeline

- February 24 - 28: Set up experiment network notebook on FABRIC
- March 3 - 7: Complete first experiment with the increasing outgoing traffic
- March 10 - 14: Complete experiment and create figures
- March 17 - 21: Spring Break
- March 24 - 28: Perform second experiment
- March 31 - April 4: Midpoint Presentation
- April 7-11: Complete second experiment

- April 14 - 18: Final Adjustments and Experiments
- April 21 - 25: Final Presentation Report Writing & Presentation Preparation
- April 28 - May 2: Final Presentation Week