Reverse-engineering HTTP Video Streaming

Kendrick Nguyen
PID: A16889378

March 7, 2026

Abstract

Video streaming services use TCP connections to deliver videos to end hosts. Each platform uses different strategies to deliver packets to users. The purpose of this project is to identify the strategies used by Youtube, Vimeo, Dropbox, and Google Drive for HTTP video streaming. Wireshark is used to analyze qualities like persistent/non-persistent TCP connection, constant/variable bit-rate, and parallel/single TCP connections during a video streaming session. Each platform is tested two times: one next to a WiFi access point and another in an enclosed closet away from a WiFi access point.

1 Introduction

The TCP protocols of each streaming service will be investigated for their payload sizes and inter-arrival times. Each video streaming service will be playing the same video per platform when testing the distance and accessibility to a WiFi access point.

For determining persistent/non-persistent and parallel/serial TCP connections, both the router and closet packets will have the same conclusions. The TCP connection type does not change via distance from an access point.

Determining constant and variable bitrate is done by using the Wireshark I/O graphs by plotting bps over time from the `tcp.stream` display filter.

2 Determining Persistent/Non-Persistent TCP Connection

Persistent TCP Connection is defined by a server leaving a connection open after sending a response. This way, request/responses can be made over a single connection rather than multiple connections. It can be easily identified with a `TCP Keep-Alive header` in Wireshark.

Youtube[1] and Vimeo[2] have these TCP Keep-Alive headers that keep the connection open for a short period of time. However, Google Drive[4] and Dropbox[3] do not use the Keep-Alive headers. This could be from the file already being downloaded onto the file hosting server, so there is no need to provide a persistent TCP connection from the server to the end user.

3 Determining Parallel/Serial TCP Connection

A parallel TCP connection will open multiple connections to increase performance and maximize throughput between the server and end user. Using the Wireshark `tcp.stream` display filter, we can determine if multiple TCP streams are opened over a short period of time.

Youtube[1], Vimeo[2], and Google Drive[4] have different stream indices over a short period of time during video play, suggesting that they use parallel TCP connections to stream video to the user. Dropbox[3] is the exception, with TCP protocols only happening over a single TCP stream index.

3.1 Youtube

No.	Time	Source	Destination	Protocol	Length	Info	stream-idx
27…	64.336309013	2603:8001:32f0:1820…	2a04:4e42:600::347	TLSv1.2	125	Application Data	1
27…	64.336428515	2603:8001:32f0:1820…	2a04:4e42:200::347	TLSv1.2	125	Application Data	5
27…	64.354621798	2a04:4e42:200::347	2603:8001:32f0:1820…	TLSv1.2	125	Application Data	5
27…	64.354622423	2a04:4e42:200::347	2603:8001:32f0:1820…	TCP	86	443 → 44120 [ACK] Seq=137 Ack=815 W…	5
27…	64.354668105	2603:8001:32f0:1820…	2a04:4e42:200::347	TCP	86	44120 → 443 [ACK] Seq=815 Ack=176 W…	5
27…	64.354713071	2603:8001:32f0:1820…	2a04:4e42:200::347	TCP	86	[TCP Dup ACK 2752#1] 44120 → 443 [A…	5
27…	64.362534078	2a04:4e42:600::347	2603:8001:32f0:1820…	TCP	86	443 → 44696 [ACK] Seq=237 Ack=1258 …	1
27…	64.362534731	2a04:4e42:600::347	2603:8001:32f0:1820…	TLSv1.2	125	Application Data	1
27…	64.362603018	2603:8001:32f0:1820…	2a04:4e42:600::347	TCP	86	44696 → 443 [ACK] Seq=1258 Ack=276 …	1
28…	68.097039189	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	86	[TCP Keep-Alive] 50582 → 443 [ACK] …	2
28…	69.120052691	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	86	[TCP Keep-Alive] 50582 → 443 [ACK] …	2
28…	69.141873089	2607:f8b0:4007:80e:…	2603:8001:32f0:1820…	TCP	86	[TCP Keep-Alive ACK] 443 → 50582 [A…	2
28…	69.273909265	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	1514	50582 → 443 [ACK] Seq=65390 Ack=270…	2
28…	69.273999791	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	1514	50582 → 443 [ACK] Seq=66818 Ack=270…	2
28…	69.274025250	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	1514	50582 → 443 [ACK] Seq=68246 Ack=270…	2
28…	69.274048200	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TLSv1.2	396	Application Data	2
28…	69.298062194	2607:f8b0:4007:80e:…	2603:8001:32f0:1820…	TCP	86	443 → 50582 [ACK] Seq=270387 Ack=66…	2
28…	69.298062461	2607:f8b0:4007:80e:…	2603:8001:32f0:1820…	TCP	86	443 → 50582 [ACK] Seq=270387 Ack=68…	2
28…	69.298062695	2607:f8b0:4007:80e:…	2603:8001:32f0:1820…	TCP	86	443 → 50582 [ACK] Seq=270387 Ack=69…	2
29…	69.335778852	2607:f8b0:4007:80e:…	2603:8001:32f0:1820…	TLSv1.2	499	Application Data	2
29…	69.335875746	2603:8001:32f0:1820…	2607:f8b0:4007:80e:…	TCP	86	50582 → 443 [ACK] Seq=69984 Ack=270…	2
30…	73.216006567	2603:8001:32f0:1820…	2600:1901:0:5e8a::	TCP	86	[TCP Dup ACK 77#7] 39174 → 443 [ACK…	3
30…	73.231991860	2600:1901:0:5e8a::	2603:8001:32f0:1820…	TCP	86	[TCP Dup ACK 78#7] 443 → 39174 [ACK…	3
31…	76.313779835	2603:8001:32f0:1820…	2a04:4e42::347	TLSv1.2	125	Application Data	21
31…	76.314468821	2603:8001:32f0:1820…	2a04:4e42::347	TLSv1.2	110	Application Data	21
31…	76.333267721	2a04:4e42::347	2603:8001:32f0:1820…	TCP	86	443 → 54342 [ACK] Seq=40 Ack=79 Win…	21

Figure 1: A sample of Youtube TCP video packets. Notice that the Keep-Alive header happens on the same stream index. Multiple TCP connections happen over a short period of time as evidenced by the different stream index numbers.

3.2 Vimeo

No.	Time	Source	Destination	Protocol	Length	Info	stream-idx
31…	73.729026069	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 49830 → 443 [ACK] …	0
31…	73.749701593	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 49830 [A…	0
31…	74.241036115	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 50576 → 443 [ACK] …	8
31…	74.258119079	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 50576 [A…	8
31…	74.752023151	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 49822 → 443 [ACK] …	10
31…	74.752075517	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 50602 → 443 [ACK] …	7
31…	74.752133902	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 50586 → 443 [ACK] …	9
31…	74.770529817	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 49822 [A…	10
31…	74.774995890	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 50602 [A…	7
31…	74.776252031	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 50586 [A…	9
31…	74.809960151	192.168.1.222	34.120.208.123	TLSv1.2	105	Application Data	32
31…	74.826482585	34.120.208.123	192.168.1.222	TLSv1.2	105	Application Data	32
31…	74.826554828	192.168.1.222	34.120.208.123	TCP	66	45414 → 443 [ACK] Seq=79 Ack=79 Win…	32
31…	75.264028873	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 42094 → 443 [ACK] …	30
31…	75.282272328	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 42094 [A…	30
31…	75.776024271	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 42520 → 443 [ACK] …	11
31…	75.776085728	192.168.1.222	162.159.128.61	TCP	66	[TCP Dup ACK 471#7] 42492 → 443 [AC…	14
31…	75.776107323	192.168.1.222	162.159.128.61	TCP	66	[TCP Dup ACK 472#7] 42496 → 443 [AC…	15
31…	75.776122982	192.168.1.222	162.159.128.61	TCP	66	[TCP Dup ACK 473#7] 42516 → 443 [AC…	16
31…	75.776138130	192.168.1.222	162.159.128.61	TCP	66	[TCP Dup ACK 474#7] 42510 → 443 [AC…	17
31…	75.792925064	162.159.128.61	192.168.1.222	TCP	66	[TCP Keep-Alive ACK] 443 → 42520 [A…	11
31…	75.796952801	162.159.128.61	192.168.1.222	TCP	66	[TCP Dup ACK 475#7] 443 → 42516 [AC…	16
31…	75.796953461	162.159.128.61	192.168.1.222	TCP	66	[TCP Dup ACK 477#7] 443 → 42510 [AC…	17
31…	75.796953630	162.159.128.61	192.168.1.222	TCP	66	[TCP Dup ACK 476#7] 443 → 42496 [AC…	15
31…	75.796953805	162.159.128.61	192.168.1.222	TCP	66	[TCP Dup ACK 478#7] 443 → 42492 [AC…	14
31…	76.288041691	192.168.1.222	162.159.128.61	TCP	66	[TCP Keep-Alive] 50592 → 443 [ACK] …	6

Figure 2: Vimeo video streaming packets. Keep-Alive headers happen over different TCP stream indices, which suggest both persistent and parallel TCP connections.

3.3 Dropbox

No.	Time	Source	Destination	Protocol	Length	Info	stream-idx
479	8.509203629	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=5750 Ack=3887…	1
480	8.509276069	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	40070	Application Data, Application Data,…	1
481	8.510296313	2620:100:6017:16::a…	2603:8001:32f0:1820…	TCP	1514	443 → 34594 [PSH, ACK] Seq=3927256 …	1
482	8.510296769	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	28259	Application Data, Application Data,…	1
483	8.510496838	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=5750 Ack=3956…	1
484	8.648174664	2603:8001:32f0:1820…	2620:100:6017:16::a…	TLSv1.2	584	Application Data	1
485	8.667471461	2620:100:6017:16::a…	2603:8001:32f0:1820…	TCP	86	443 → 34594 [ACK] Seq=3956857 Ack=6…	1
486	9.022407575	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	16492	Application Data	1
487	9.022587424	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	16492	Application Data	1
488	9.022652154	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	4370	Application Data	1
489	9.023440148	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=3993…	1
490	9.024023314	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	12337	Application Data	1
491	9.024074371	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4006…	1
492	9.024384092	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	41498	Application Data, Application Data	1
493	9.024645968	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	34358	Application Data, Application Data	1
494	9.024895533	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4081…	1
495	9.025567295	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	7226	Application Data, Application Data	1
496	9.025567616	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	41498	Application Data, Application Data	1
497	9.025679974	2620:100:6017:16::a…	2603:8001:32f0:1820…	TCP	5798	443 → 34594 [ACK] Seq=4130440 Ack=6…	1
498	9.025997297	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4136…	1
499	9.034141025	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	35786	Application Data, Application Data,…	1
500	9.034213828	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4171…	1
501	9.034293772	2620:100:6017:16::a…	2603:8001:32f0:1820…	TCP	7226	443 → 34594 [ACK] Seq=4171852 Ack=6…	1
502	9.034483601	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4178…	1
503	9.035701519	2620:100:6017:16::a…	2603:8001:32f0:1820…	TLSv1.2	34358	Application Data, Application Data,…	1
504	9.035782482	2603:8001:32f0:1820…	2620:100:6017:16::a…	TCP	86	34594 → 443 [ACK] Seq=6248 Ack=4213…	1

Figure 3: Dropbox shows a TCP connection over the same port over a short period of time, suggesting non-persistent TCP connection. A single stream index over time suggests a single TCP connection for video streaming.

3.4 Google Drive

No.	Time	Source	Destination	Protocol	Length	Info	stream-idx
697	40.510466199	162.125.40.1	192.168.1.222	TLSv1.2	177	Application Data	1
698	40.510525920	192.168.1.222	162.125.40.1	TCP	66	43100 → 443 [ACK] Seq=421 Ack=224 W…	1
699	40.694301146	192.168.1.222	162.125.40.1	TLSv1.2	168	Application Data	1
700	40.694365618	192.168.1.222	162.125.40.1	TLSv1.2	384	Application Data	1
701	40.743741315	162.125.40.1	192.168.1.222	TCP	66	443 → 43100 [ACK] Seq=224 Ack=523 W…	1
702	40.743741751	162.125.40.1	192.168.1.222	TCP	66	443 → 43100 [ACK] Seq=224 Ack=841 W…	1
707	41.328048552	162.125.40.1	192.168.1.222	TLSv1.2	177	Application Data	1
708	41.328088445	192.168.1.222	162.125.40.1	TCP	66	43100 → 443 [ACK] Seq=841 Ack=335 W…	1
709	41.688321340	192.168.1.222	162.125.40.1	TLSv1.2	168	Application Data	1
710	41.688395351	192.168.1.222	162.125.40.1	TLSv1.2	396	Application Data	1
711	41.690176281	192.168.1.222	162.125.40.1	TLSv1.2	111	Application Data	1
712	41.730848563	162.125.40.1	192.168.1.222	TCP	66	443 → 43100 [ACK] Seq=335 Ack=943 W…	1
713	41.738378457	162.125.40.1	192.168.1.222	TCP	66	443 → 43100 [ACK] Seq=335 Ack=1273 …	1
714	41.738378783	162.125.40.1	192.168.1.222	TCP	66	443 → 43100 [ACK] Seq=335 Ack=1318 …	1
893	44.155769800	192.168.1.222	34.107.243.93	TLSv1.2	105	Application Data	6
894	44.155878308	2603:8001:32f0:1820…	2a04:4e42::347	TLSv1.2	132	Application Data	7
895	44.184503855	34.107.243.93	192.168.1.222	TLSv1.2	105	Application Data	6
896	44.184504139	34.107.243.93	192.168.1.222	TCP	66	443 → 33038 [ACK] Seq=1 Ack=40 Win=…	6
897	44.184830728	2a04:4e42::347	2603:8001:32f0:1820…	TCP	86	443 → 32998 [ACK] Seq=1 Ack=47 Win=…	7
898	44.184830912	2a04:4e42::347	2603:8001:32f0:1820…	TLSv1.2	132	Application Data	7
899	44.224449686	2a04:4e42::347	2603:8001:32f0:1820…	TCP	132	[TCP Retransmission] 443 → 32998 [P…	7
900	44.224477844	2603:8001:32f0:1820…	2a04:4e42::347	TCP	98	32998 → 443 [ACK] Seq=47 Ack=47 Win…	7
901	44.225456280	192.168.1.222	34.107.243.93	TCP	66	33038 → 443 [ACK] Seq=40 Ack=40 Win…	6
902	44.410495503	162.125.40.1	192.168.1.222	TLSv1.2	178	Application Data	1
903	44.410541176	192.168.1.222	162.125.40.1	TCP	66	43100 → 443 [ACK] Seq=1318 Ack=447 …	1
905	45.855949631	192.168.1.222	162.125.40.1	TLSv1.2	168	Application Data	1

Figure 4: Google Drive has a more complex system between its non-persistent and parallel TCP connection. Looking at the port numbers, we see that only one connection is made at a stream index, but multiple stream indices are recorded over time. So Google Drive has a non-persistent connections per port, but runs all ports in parallel.

4 Determining Constant or Variable Bitrate Streaming

Using the I/O graphs on Wireshark, we can determine if video streaming on these different platforms has constant or variable bitrate. After plotting both the router and closet packets, we can determine that all video streaming services use variable bitrate streaming based on each spike of data being different over time. The rate data is sent over time is correlates to how far and accessible the device is from the WiFi access point. The closer the end user's device is to an Internet Router, the higher the bitrate over time. The closet with the door closed impedes the bitrate over time. More detail about stream adaptation is described in each video streaming services' subsection.

4.1 Youtube

By comparing the two plots below, we can see that being close to a WiFi access point will send the most packets at the beginning. The next spike of data is sent near the end of the video, at a smaller size compared to the initial bitrate but significantly more than the subsequent packets after the initial.

Being farther and less accessible from a WiFi access point will change the bitrate over time. Youtube will not have a high initial bitrate, only increasing over its bitrate over time until it reaches its maximum near the end of the video streaming. The period where not many packets are sent from around 25 seconds to 310 seconds is either (or some combination of) the stream sending lower quality video or playback still being sent from the video server.

4.1.1 Router

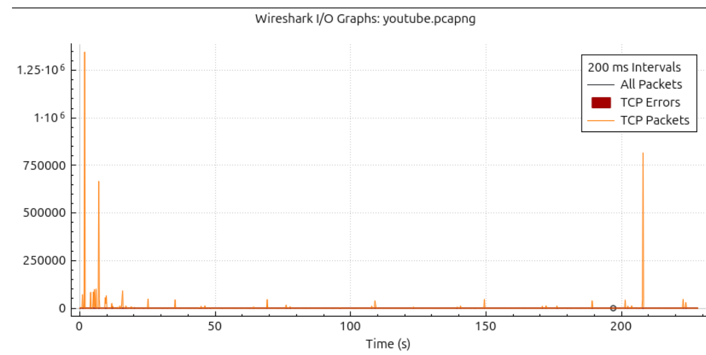


Figure 5: Youtube Video Streaming TCP Packets Next to Router

4.1.2 Closet

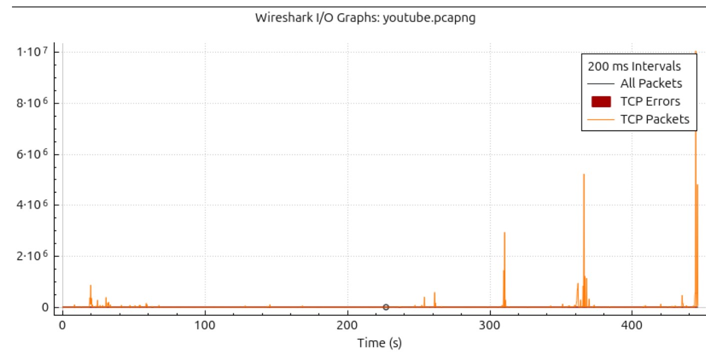


Figure 6: Youtube Video Streaming TCP Packets in Closet

4.2 Vimeo

Vimeo has variable bitrate streaming, but sends packets at a consistent time interval. The Router packets have a higher bitrate initially, as seen by the 8.2×10^6 bps around 9 seconds, but gradually oscillates bitrates until the end of the video. The closet packets have similar time intervals between bitrate peaks and a similar oscillation pattern over the video playtime. The maximum peaks of the closet packets are actually similar to the maximum peaks of the router packets after the absolute maximum spike in the beginning.

4.2.1 Router

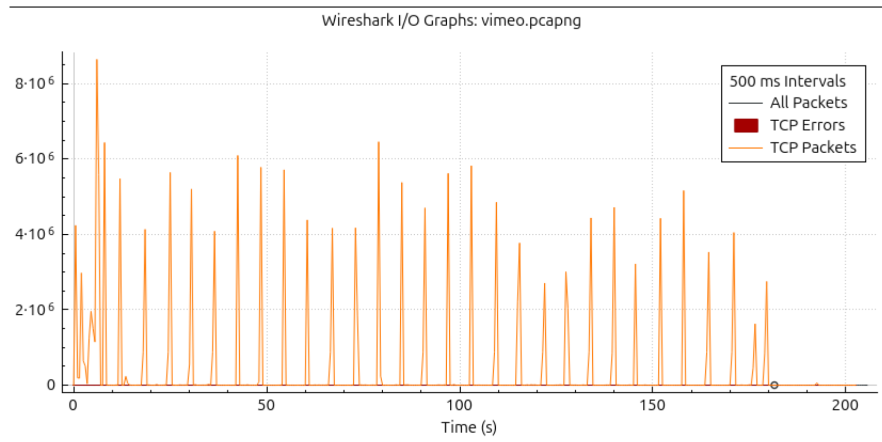


Figure 7: Vimeo Video Streaming TCP Packets Next to Router

4.2.2 Closet

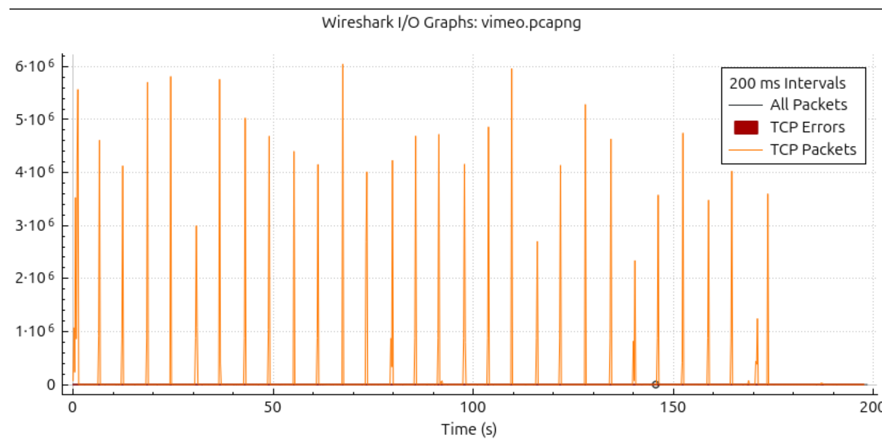


Figure 8: Vimeo Video Streaming TCP Packets in Closet

4.3 Dropbox

When the end user's device is close to a WiFi access point, Dropbox has a high bitrate in the beginning, sending all video streaming packets even before the client reaches the end of play out. However, the bitrate pattern changes significantly with less access to the router. The bitrate per spike is decreased significantly by three orders of magnitude, and is spread out over time until the video finally finishes.

4.3.1 Router

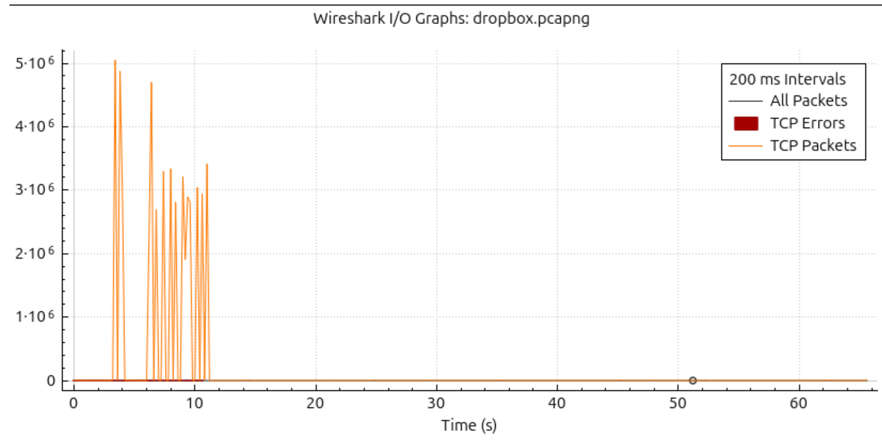


Figure 9: Dropbox Video Streaming TCP Packets Next to Router

4.3.2 Closet

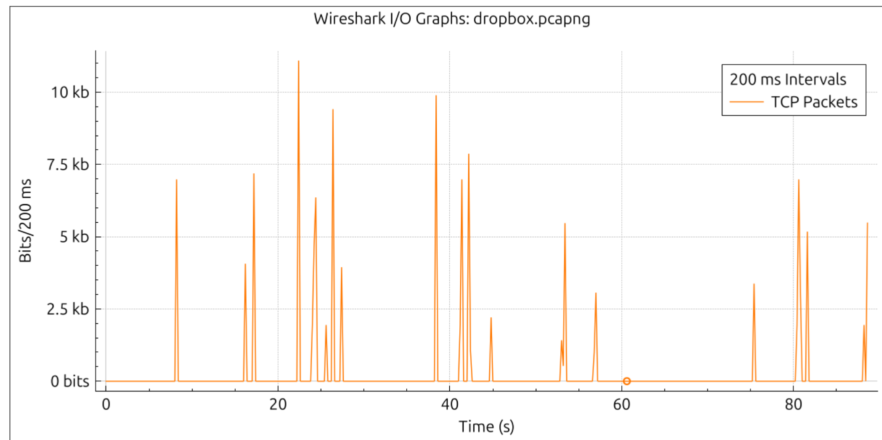


Figure 10: Dropbox Video Streaming TCP Packets in Closet

4.4 Google Drive

Google Drive is fairly consistent in bitrate sizes over video streaming. The only difference between video streaming based on accessibility to a WiFi access point is the density of bitrate spikes during a period of time. For example, the largest group of bitrate spikes over 10 seconds (40 seconds to 50 seconds) has a higher density in the router group than the closet group (25 seconds to 35 seconds). This could suggest that the video may be playing in lower quality and/or the playback speed is not loading as fast on the client side.

4.4.1 Router

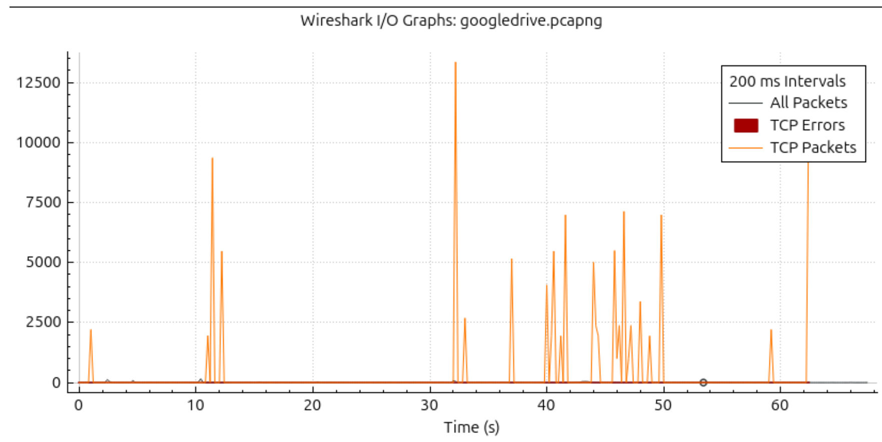


Figure 11: Google Drive Video Streaming TCP Packets Next to Router

4.4.2 Closet

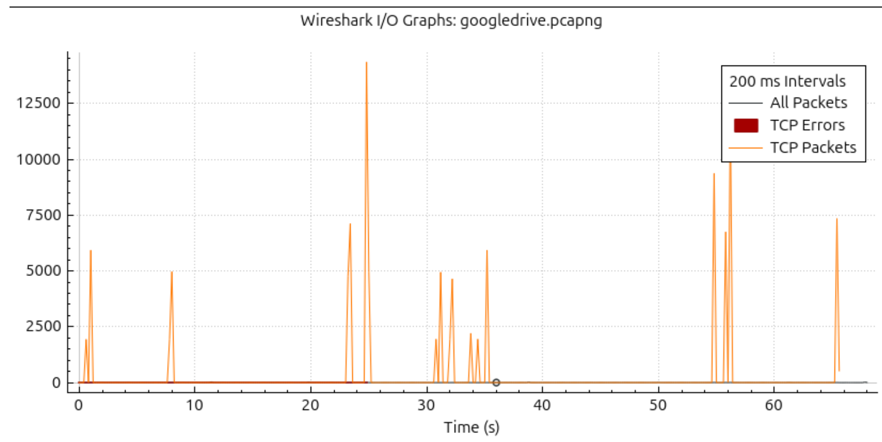


Figure 12: Google Drive Video Streaming TCP Packets in Closet

5 Comparative Statistics

The kernel density estimation (KDE) and cumulative distribution function (CDF) was plotted for both the router and closet packets for all video streaming services. These additional statistics can help analyze the probability a certain packet size or inter-arrival time may occur. Both the closet and router packets are plotted on the same KDE and CDF graphs, respectively. The source code and plots can be found in the appendix.

5.1 Payload Size

13 For Youtube, the estimation density of the packets is highest when the client is close to the router, but observes low density when the router isn't as accessible. The opposite can be said of Vimeo. The closet KDE is much denser than the router KDE density, which is surprising as Youtube and Vimeo share the strategy of persistent, parallel TCP connections.

Google Drive and Dropbox have a similar density but for the opposite locations of router and closet. The density has more variety compared to Youtube and Vimeo.

5.2 Inter-Arrival Times

14 The inter-arrival times of Youtube have a higher density in the closet than near the router. This could be related to the inverted relationship of bitrate and accessibility.

Vimeo has a higher density next to the router compared to the closet. Even if the variable bitrate streaming looks similar for both the router and closet, the estimation densities of each inter-arrival interval are still distinguishable.

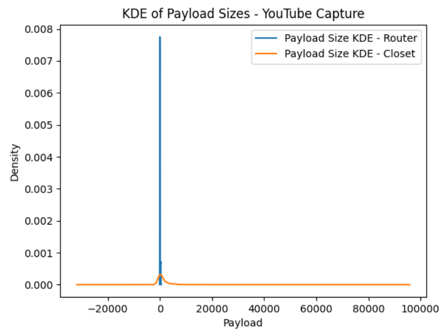
Google Drive has the least difference between the height of its density in the closet and its density next to the router. Comparing it along with the payload size KDE means the TCP connection strategy is consistent no matter the accessibility to the WiFi access point.

Dropbox has a fairly large difference between the router and closet density estimations. This could be attributed to the difference in bitrate depending on the accessibility to a WiFi access point.

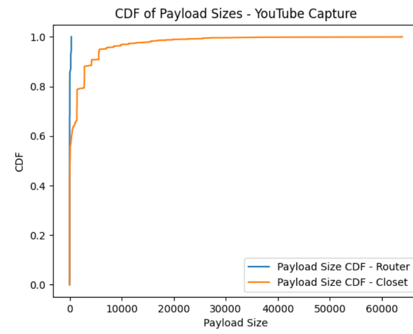
6 Conclusion

Comparing different video streaming services reveal the different TCP connection strategies based on purpose, with primarily video hosting platforms like Youtube and Vimeo vs. general file hosting cloud services like Dropbox and Google Drive. These statistics and comparisons also reveal the differences between each website, even if they have the same purpose. Persistent and parallel TCP connections are the usual strategies for most video streaming services, while all services use variable bitrate streaming.

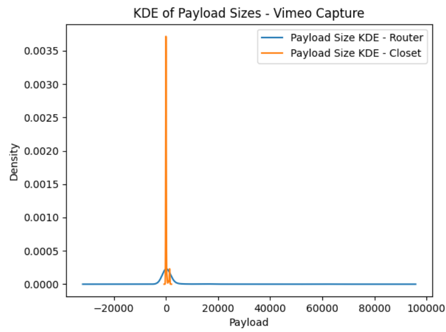
A Appendix



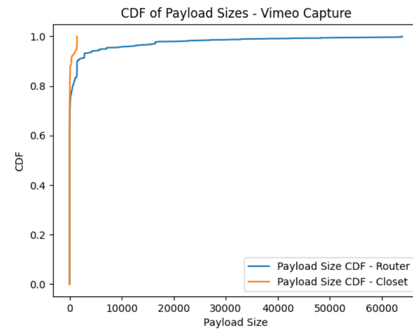
(a)



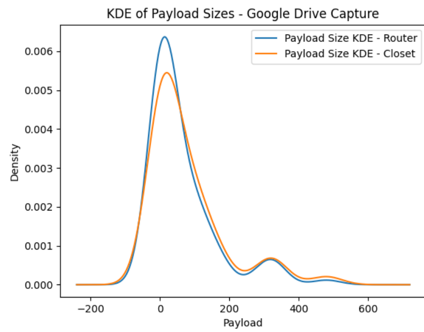
(b)



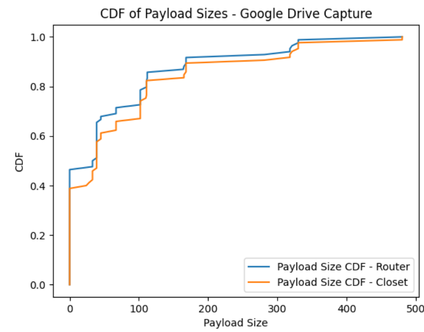
(c)



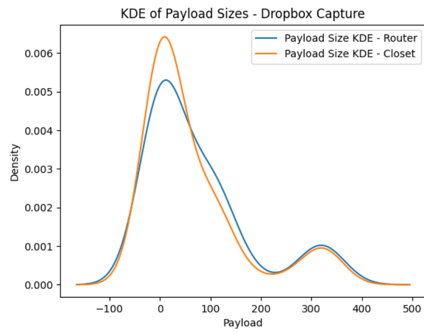
(d)



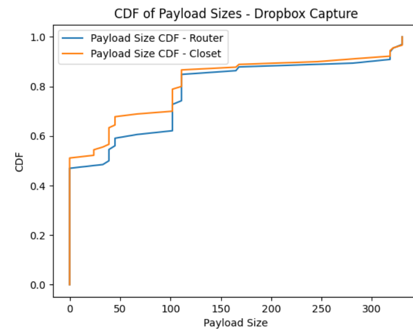
(e)



(f)

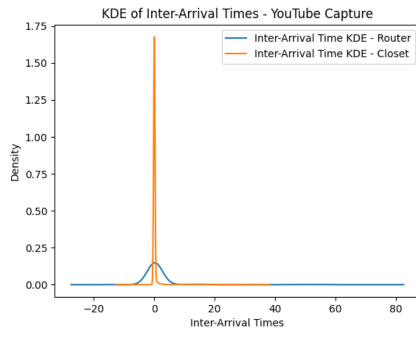


(g)

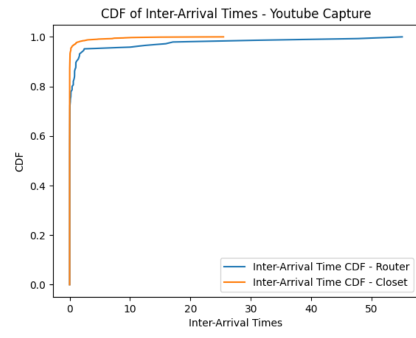


(h)

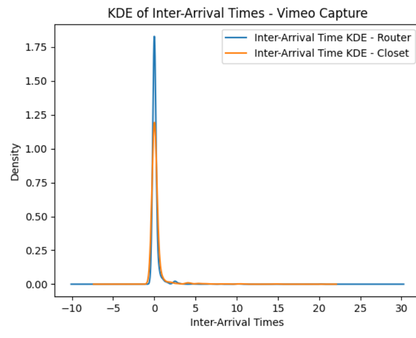
Figure 13: KDE and CDF of payload sizes for all tested video streaming services



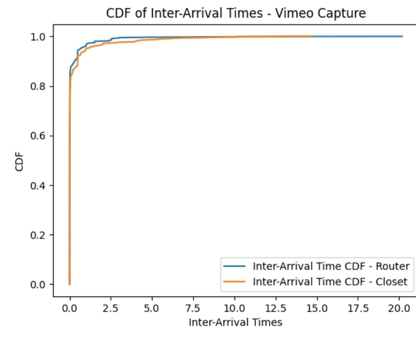
(a)



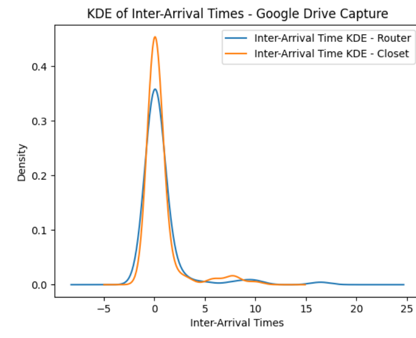
(b)



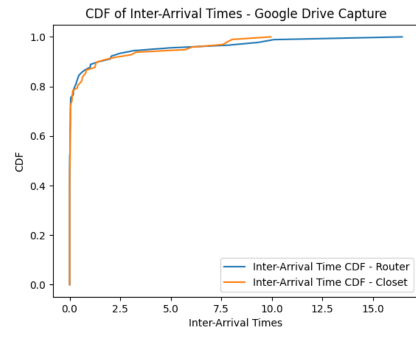
(c)



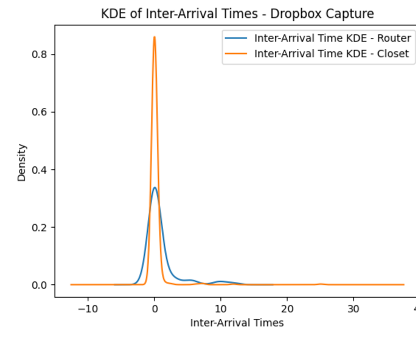
(d)



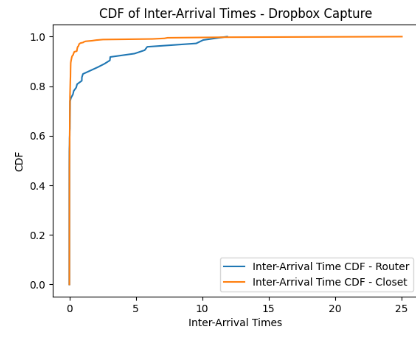
(e)



(f)



(g)



(h)

Figure 14: KDE and CDF of inter-arrival times for all tested video streaming services

```
import dpkt
from matplotlib.axes import Axes
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import pandas as pd

###Packet Parsing from pcap file###

#Packet parsing for Router packets
tcp_payload1 = []
tcp_traffic1 = []
inter_arrival_times1 = []
prev_timestamp1 = None
timestamp1 = []

with open('/home/kb/ECE 158B/Project 2/Router/dropbox.pcap', 'rb') as file: #opening
    and reading .pcap file
    pcap_reader1 = dpkt.pcap.Reader(file)
    for timestamp, packet_data in pcap_reader1: #iterating through the timestamps of
    the pcap file
        eth1 = dpkt.ethernet.Ethernet(packet_data)
        if isinstance(eth1.data, dpkt.ip.IP):
            ip1 = eth1.data
            if isinstance(ip1.data, dpkt.tcp.TCP): #for UDP capture, dpkt changed to
    dpkt.udp.UDP
                tcp1 = ip1.data
                tcp_payload_size1 = len(tcp1.data)
                tcp_payload1.append(tcp_payload_size1)
            if prev_timestamp1 is not None:
                inter_arrival_time = timestamp - prev_timestamp1
                inter_arrival_times1.append(inter_arrival_time)
            prev_timestamp1 = timestamp



#packet parsing for Closet packets
tcp_traffic2 = []
tcp_payload2 = []
inter_arrival_times2 = []
prev_timestamp2 = None

with open('/home/kb/ECE 158B/Project 2/closet/dropbox.pcap', 'rb') as file: #opening
    and reading .pcap file
    pcap_reader2 = dpkt.pcap.Reader(file)
    for timestamp, packet_data in pcap_reader2: #iterating through the timestamps of
    the pcap file
        eth2 = dpkt.ethernet.Ethernet(packet_data)
        if isinstance(eth2.data, dpkt.ip.IP):
            ip2 = eth2.data
            if isinstance(ip2.data, dpkt.tcp.TCP): #for UDP capture, dpkt changed to
    dpkt.udp.UDP
                tcp2 = ip2.data
                tcp_traffic2.append(len(packet_data))
                tcp_payload_size2 = len(tcp2.data)
                tcp_payload2.append(tcp_payload_size2)
            if prev_timestamp2 is not None:
                inter_arrival_time = timestamp - prev_timestamp2
                inter_arrival_times2.append(inter_arrival_time)
            prev_timestamp2 = timestamp



###CDF and PDF Calculation###

#For Router packets
payload_sizes_sorted1 = np.sort(tcp_payload1)
payload_sizes_cdf1 = 1.0 * np.arange(len(payload_sizes_sorted1)) / float(len(
    payload_sizes_sorted1)-1)

#For Closet packets
```

```python
payload_sizes_sorted2 = np.sort(tcp_payload2)
payload_sizes_cdf2 = 1.0 * np.arange(len(payload_sizes_sorted2)) / float(len(
    payload_sizes_sorted2)-1)

#Plot KDE
plt.figure(1)
s1 = pd.Series(payload_sizes_sorted1)
s2 = pd.Series(payload_sizes_sorted2)
s1.plot.kde(label='Payload Size KDE - Router')
s2.plot.kde(label='Payload Size KDE - Closet')
plt.xlabel('Payload')
plt.ylabel('Density')
plt.title('KDE of Payload Sizes - Dropbox Capture')
plt.legend()
plt.show()


#Plot CDF
plt.figure(2)
plt.plot(payload_sizes_sorted1, payload_sizes_cdf1, label='Payload Size CDF - Router')
plt.plot(payload_sizes_sorted2, payload_sizes_cdf2, label='Payload Size CDF - Closet')

plt.xlabel('Payload Size')
plt.ylabel('CDF')
plt.title('CDF of Payload Sizes - Dropbox Capture')
plt.legend()
plt.show()




###Inter-arrival times###

#For Router packets
inter_arrival_times_sorted1 = np.sort(inter_arrival_times1)
inter_arrival_time_cdf1 = np.arange(len(inter_arrival_times_sorted1)) / float(len(
    inter_arrival_times_sorted1)-1)


#For Closet packets
inter_arrival_times_sorted2 = np.sort(inter_arrival_times2)
inter_arrival_time_cdf2 = np.arange(len(inter_arrival_times_sorted2)) / float(len(
    inter_arrival_times_sorted2)-1)

#Plot KDE
plt.figure(3)
s1 = pd.Series(inter_arrival_times_sorted1)
s2 = pd.Series(inter_arrival_times_sorted2)
s1.plot.kde(label='Inter-Arrival Time KDE - Router')
s2.plot.kde(label='Inter-Arrival Time KDE - Closet')
plt.xlabel('Inter-Arrival Times')
plt.ylabel('Density')
plt.title('KDE of Inter-Arrival Times - Dropbox Capture')
plt.legend()
plt.show()


#Plot CDF
plt.figure(4)
plt.plot(inter_arrival_times_sorted1, inter_arrival_time_cdf1, label='Inter-Arrival
    Time CDF - Router')
plt.plot(inter_arrival_times_sorted2, inter_arrival_time_cdf2, label='Inter-Arrival
    Time CDF - Closet')
plt.xlabel('Inter-Arrival Times')
plt.ylabel('CDF')
plt.title('CDF of Inter-Arrival Times - Dropbox Capture')
plt.legend()
plt.show()
```

Listing 1: KDE and CDF Python Code