Deliverable #C

Xinhao Xiao
Shan Qing

1 Abstract

The purpose of this project is to understand and demonstrate the use of Real-Time Streaming Protocol (RTSP) in media streaming. From what we learn, RTSP is a streaming protocol which provides streaming media server controlling. Typically, developers use Transmission Control Protocol (TCP) in data transmission and establishes connections between clients and servers. Then, servers send back data to clients through User Datagram Protocol (UDP). Actually, this transmission process is called Real-time Transport Protocol (RTP) and RTSP is used to control media transmission over RTP. In this project, we develop RTSP client and server applications, which uses RTP to transmit data and RTSP to manipulate on the transmission. We also construct our own RTP packet based on the real one. After the constructions of a server and a client, we try to figure out what parameter may cause influence on video quality and how much can they cause. In this project, we can modify packet size, frame rate which is the session between two packets, and loss rate. Moreover, in order to test original video's effect on streaming, we prepare videos of different resolutions.

2 Introduction

The Real-time Transport Protocol (RTP) is a network protocol for delivering audio and video over IP networks. It is widely used in communication and entertainment systems that involve streaming media, such as telephony, video teleconference.[1] The Real Time Streaming Protocol (RTSP) is a network control protocol designed for to control streaming media servers. The protocol is used for two end points to establish manipulation communication. In this way, clients of media servers can issue some commands, like play, pause, and set up. Mostly, RTSP servers use the RTP for media delivery.[2]

3 Class Description

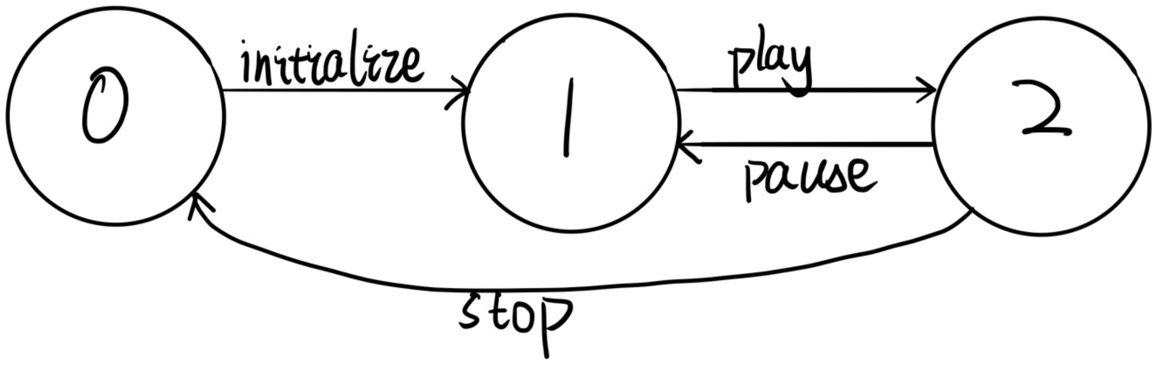
client1.py: This the client for users to watch the streaming video and give commands on server.

RTPpacket.py: This is the RTP packet that we create based on the real one.

server1.py: This is the server to open video file and send packets and manipulate on the streaming based on the commands it receive from the client.

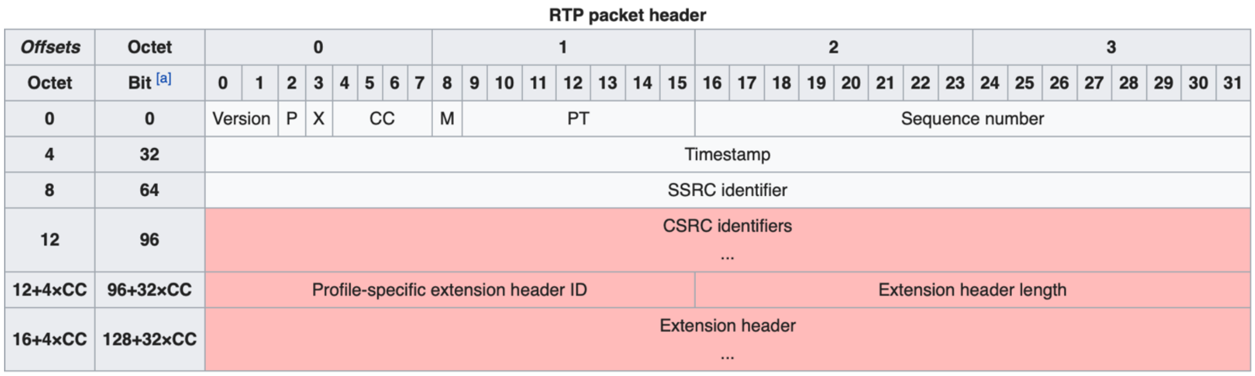
4 Implementation

This project includes only two machines, one server and one client. To understand how the application is implemented, we need to fist understand status of client server. Here, we assign three status to both client and server and the status is the same in the server and the client at all time. We call these three statuses 0, 1, and 2. These statuses are switched based on the command they receive. From S0 to S1, we need to hit the initialize button and so on. S0 means that the connection is established. S1 means that the streaming is ready to start and S2 means that the video is being streamed. The following graph shows the statuses and how they are switched.



Now we learn about statuses and how they switch, we can start our process of streaming. At first, we start server and wait for one client to connect to it. Then we start the client and connect it to the server. Once the connection is established, status will be assigned to 0 and the user interface will show up on the user's machine. Next, we need to hit the initialize button, which means that the client will send a initialize message in a TCP packet to the server and this message contains the packet size and frame rate information. In our project, frame rate is the session between each frame being sent. After the server receive the initialize information, it will open the video file and decode it into bytes through cv2 library in Python and set all parameters based on the initialize information. At this time, the status can be switched to 1 now.

After we finish the initialize step, we are ready to start streaming. Thus, we click the play button. Since we have other commands like pause and stop, we cannot use one packet to handle both data and commands. Therefore, we come up with idea of using thread, with one specifically sending and receiving commands and another for data. Moreover, all commands are contained in a TCP packet. After the server receives the TCP packet, it can send video file int UDP packets back to the client. Here, we create our own RTP packet with 12bytes header. The following is the RTP packet that our packet is based on:



RTP packet header

Offsets	Octet	0								1								2								3							
Octet	**Bit [a]**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
0	**0**	Version		P	X	CC				M	PT							Sequence number															
4	**32**	Timestamp																															
8	**64**	SSRC identifier																															
12	**96**	CSRC identifiers ...																															
12+4×CC	**96+32×CC**	Profile-specific extension header ID																Extension header length															
16+4×CC	**128+32×CC**	Extension header ...																															

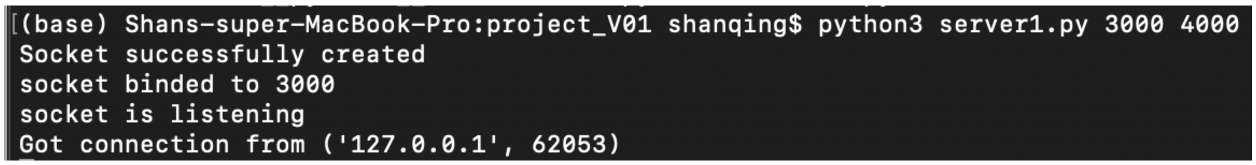
[1]

A video file is compromised of many frames with each one followed by the next one. We planned to put one frame in one packet, but we find that many frames are too big for a single packet and this may cause loss in frame information. Therefore, we decided to divide each frame in some packets with the first one having a header and rest ones with no header. To deal with packet loss in UDP, we make use of sequence number field in the header. When the sequence number that the client receive does not match the expected sequence number(when a packet loss is expected), we only play it if it is greater than the expected number. Also, we use the cv2 library to encode and play the video. The status is switched to 2 after the streaming begins.

If the user wants to pause the video, he or she can hit the pause button and the client will send a pause command to the server. The sequence number will be recorded and the thread that receives video data is stopped. At the server side, its thread that sends video file is also deleted and the last sent packet's sequence number is recorded. Here, the status is switched back to 1. Once the user hit the play button, new threads will be created so that the client and server can send and receive data from the last packet and status is 2 now. The stop command is similar to pause command and the only difference is that once the streaming is stopped, all other commands are unavailable. And the status is switched to 0. The last minor function that we implement is the quit button on the in the top bar of GUI. Only this button can totally close the connection so that we can reuse the port numbers.

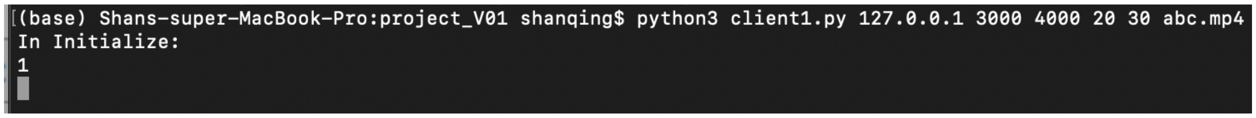
5 Running

To run this project, first of all, we need to make sure that our computer has a python3 compiler. Next, we need to have a cv2 library to process video file. To install this library, we can use the following commands:
pip install opencv-contrib-python
pip install opencv-python
Next, to run the server, we use command: python3 server1.py tcp_port udp_port.
To run the client, we use command: python3 client1.py ip_address tcp_port udp_port packetsize framerate filename lossrate
In these two commands, TCP port number and UDP port number can be assigned by users but must be matched for server and client. Also, the IP address must be the address of the server. Loss rate must be in rang of 0 to 100. Packet size must be between 1 and 1000 and frame rate should be between 0 and 100. At last, we need a video file on server, called abc.mp4.
Here is an example of commands running the program:
Server:



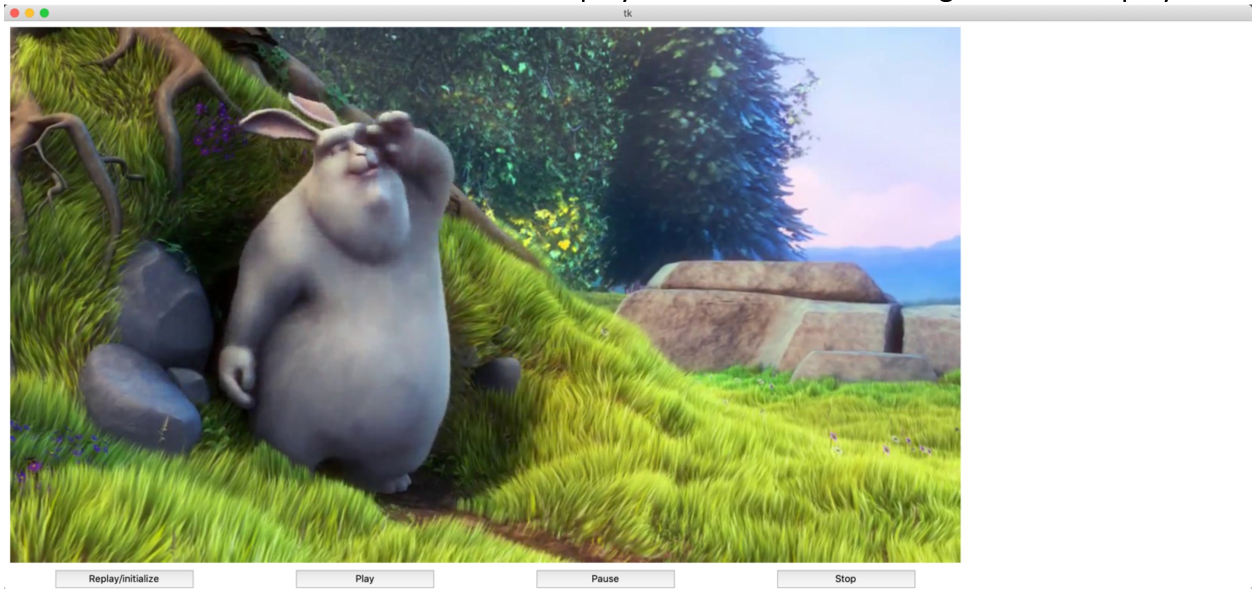
```
[(base) Shans-super-MacBook-Pro:project_V01 shanqing$ python3 server1.py 3000 4000
Socket successfully created
socket binded to 3000
socket is listening
Got connection from ('127.0.0.1', 62053)
```

Client:



```
[(base) Shans-super-MacBook-Pro:project_V01 shanqing$ python3 client1.py 127.0.0.1 3000 4000 20 30 abc.mp4
In Initialize:
1
```

Once the connection is established, a graphical user interface(GUI) will show up, with initialize, play, pause, and stop on it. Currently, the center of the interface is blank. To start streaming, we need to initialize it first and then we can play the video. The following is a video in play:



6 Performance Analysis and Discussion

For this project, we plan to do performance analysis based on three factors: packet size which is the size of each packet in bytes, frame rate which is the session between two frames, and loss rate which means we intentionally choose not to transmit some packets to simulate packet loss. To judge the application's performance, we have both subjective and objective metrics: Mean Opinion Score (MOS) and transmission time. We choose packet size and loss rate as factors because UDP is not reliable and some packets many be lost during transmission so. Therefore, if packet size and loss rate are small enough, human can hardly detect minor errors in the video. And we choose frame rate since some packets may be received earlier than those that have smaller sequence number and we always play packet that has greater sequence number. Thus, this can also cause packet loss due to internet condition so an appropriate frame rate can actually greatly improve the video quality.

Mean Opinion Score is a human-judged assessment of the overall quality of the audio or video. We develop the assessment system based on our own enjoyableness. The enjoyableness of the video is affected by the image quality, screen tearing, and fluency, etc. Based on these factors, we make the following scoring standard:

Assessment Factor	Rating (1-5)	Criteria
Intelligible	5	Frames are perfectly clear and fluent, approximately the same as the original
	4	Some frames have minor jitters and not very fluent
	3	Lots of blurry images and pauses and
	2	Many teared images and images may freeze for seconds and people cannot bear with it
	1	Video cannot be played

As for the objective metric, we choose the time difference between the packet is sent and the packet is received and calculate the average of all packets. We accomplish this by recording the time a packet is sent in this packet's timestamp field so that the client can know the time it is sent by decoding the headers and then substitute it from current time to get the time difference. The unit is milliseconds.

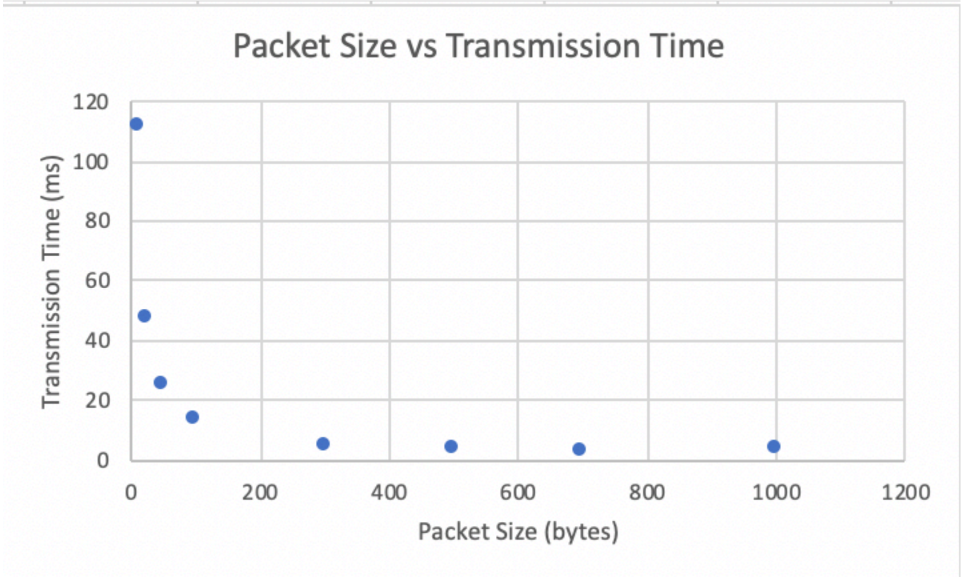
As out parameter indicates, we can directly modify packet size, frame rate and loss rate when we run it from command line. So how we test these factors by increasing one parameter a specific value each time, while keeping other parameter unchanged. By the way, in all of the following tests, we use abc.mp4 except for original video resolution in which we prepare three videos of same content but different resolution, called 240p.mp4, 360p.mp4, and 720p.mp4 respectively.

Packet Size

Here is the test result for packet size:
Loss rate is set to 0% and frame rate is set to 30 ms

Packet Size (bytes)	**MOS Rating**	**Transmission Time (ms)**
10	3 (lots of pauses)	111
25	2	47
50	4 (more fluent than 25 bytes)	25
100	4 (slightly more pauses compared with original)	13
300	5	4
500	5 (almost original)	3
700	5	2.5
1000	5	3

When we are testing, we find that the transmission time is subject to network fluctuations, so we test each packet size for over several times and remove some impurity data. The following chart shows the relationship between packet size and transmission time.



As the chart shows, transmission time decline dramatically before packet size increases to around 100 and after that it remains fairly low. As for the enjoyableness, it is almost like the original video as the packet size is 300 bytes or above. We hypothesize that the reason for declining transmission time is that there are fewer packets to be delivered so the total delay will reduce. The reason for enjoyableness is that many packets are not played since they experience delay, thus too many pauses when packet is small.

Loss Rate

Here is the test result for loss rate:
Packet size is set to 200 and frame rate is set to 30

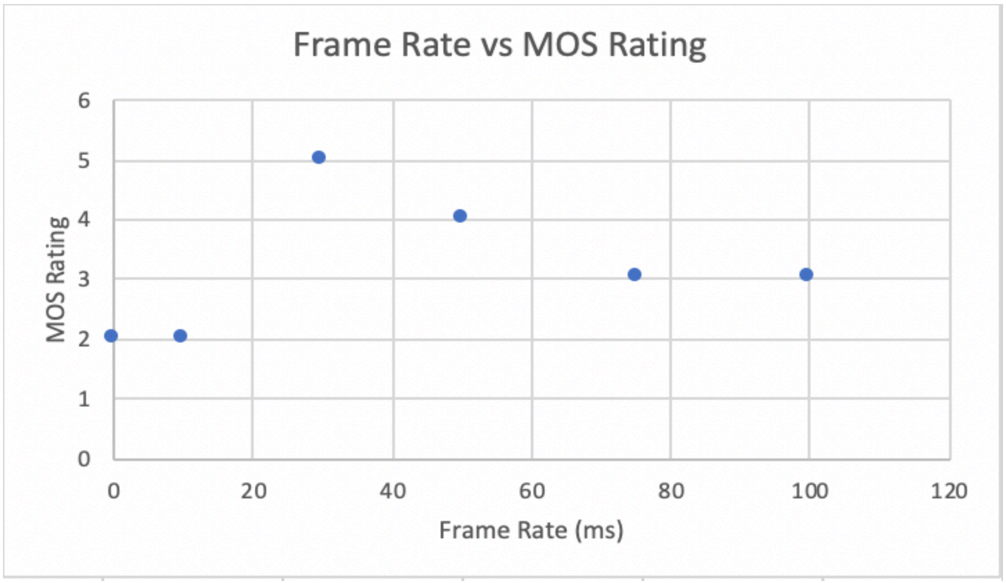
Loss Rate (%)	**MOS rating**	**Transmission Time (ms)**
0	5	6
10	5	8
30	4 (a few pauses)	7
50	3 (some pauses)	7
75	2 (too many pauses)	7
90	1 (some images freeze for seconds)	7

From the table above, we can learn that loss rate has no influence on transmission time which makes sense since we just intentionally choose not to send some packets. Thus, there is no influence on the total number of packets sent. However, loss rate does cause some effect on enjoyableness. When the loss rate is too high, the video freezes on some frames. We find that loss rate under 50 per cent is acceptable and under 10 per cent, the application can provide best video quality.

Frame Rate

Here is the test result for frame rate:
Packet Size is set to 200

Frame Rate (ms)	**MOS Rating**	**Transmission Time**
0	2 (some images are teared)	70
10	2 (just like when frame rate is 0)	56
30	5	6
50	4	6
75	3 (play speed slower than original)	6
100	3 (play speed greatly slower)	7



As the chart above shows, the most optimal frame rate for enjoyableness is around 30 ms. When frame rate is smaller than 30 ms, the frames the client receives is too fast to handle so they are not played. We believe that this phenomenon can be mitigated by creating a buffer. However, when frame rate is greater than 30 ms, frames are played too slow compared to the original one, resulting in users' bad experience. As for transmission time, it declines from 70 to 6 at 30 ms and after that remains almost constant. Therefore, we agree that it is best to set framerate to 30 ms.

Original Video Resolution

To test original video resolution's impact, we prepare
Here is the test result for original video resolution:
Packet size is set to 200 bytes, frame rate is set to 30 ms and loss rate is set to 0 percent.

Original Video Resolution	**MOS Rating**	**Transmission Time (ms)**
240p	5	1.5
360p	4 (some minor pauses)	2.5
720p	3 (more pauses and some images teared)	30

By streaming video files of different resolutions, we find that video of lower resolution tends to have enjoy better quality than higher ones. Also, lower resolution videos have less transmission time. In this case, we think that in our application, each frame is actually transmitted in several UDP packets and frames of a higher resolution video are larger in size than those of lower resolution one. Therefore, more packets have to be transmitted for a single frame, so more packets are likely to be lost.

7 Conclusion

This project has helped us understand how RTP and RTSP works together to accomplish video streaming and controlling over the streaming. Moreover, by testing a few factors, we explore what factors can influence streaming quality and transmission time. Actually, we find that a good-quality streaming requires large packet size, lower loss rate, and an optimal frame rate. Also, under the same network condition, video of low resolution tends to enjoy better streaming quality. If we have enough time, we may further develop the application to design a buffer for the client so that we can mitigate effect of frame and network fluctuations. We may also finish implementing the replay button and improve the server to stream to a bunch of clients. For performance analysis, we plan to test how Internet speed can affect streaming quality.

8 Reference

[1] https://en.wikipedia.org/wiki/Real-time_Transport_Protocol

[2] https://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol