VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Computer Networks (CO3093)

Assignment 1

Video Streamming Application

CC02 Group 3

Professor:	Nguyễn Mạnh Thìn	
Group Members:	Nguyễn Đình Khương Duy	1952207
	Nguyễn Lê Nhật Dương	1952638
	Nguyễn Chính Khôi	1952793

HO CHI MINH CITY, November 2021

Contents

Member list & Workload

No.	Fullname	Student ID	Workload
1	Nguyễn Đình Khương Duy	1952207	33.33%
2	Nguyễn Lê Nhật Dương	1952638	33.33%
3	Nguyễn Chính Khôi	1952793	33.34%

1 Overview of Problem

1.1 Analysis of problem requirements

Our goal is to implement a streaming video server and client that communicate using the Real-Time Streaming Protocol (RTSP) and send data using the Real-time Transfer Protocol (RTP); and in order to achieve this, we will have to fullfill these requirements.

1.1.1 Funtional requirement

- Implement a video streaming server.
- Implement RTSP protocol in the client.
- Implement RTP packetization in the server.
- Have a interactive player that has the function of SETUP, PLAY, PAUSE, TEARDOWN, DESCRIBE.

1.1.2 Non-functional requirement

- The maximum waiting time of the datagram socket to receive RTP data from the server is 0.5s
- The server port number must be greater than 1024.
- The time difference between each frame sent to the client is 50 milliseconds.

1.2 Description of functions

In our application, there are several key features

- Create a video streaming server and allow client to connect and communicate through the RTSP protocol: This function will initialize a server and create RTSP socket to listen and allow clients to connect
- Initialize the client and a graphical user interface(GUI) to send the RTSP requests: In the GUI there are multiple buttons such as: PLAY, SETUP, TEARDOWN, PAUSE, each button has a different function depending on the state of the client and sent different requests to the server to tell the server what it should do to satisfy the client.
- Both the client and the server will use RTP protocol to process information and communicate with one another.

1.3 List of components

We have 4 main components in our application

- Client: this component will represent the client, initialize the client launcher, handling the interaction between the user and our application and communicate with the server
- Server:this will act as a server, create a RTSP socket and listen to the clients request.
- RtpPacket: this component will define the format (including header and payload) and the actions that the client and the server can act on the RTP packet that they use to communicate.
- VideoStream: this will define the functions that the server can use to process the video file and send the necessary information to the client.

1.4 Model and data flow

When we run the command `python Server.py server_port` on the `terminal` in the folder that contains the files, the server will create the socket `RTSP/TCP` and assigns the `IP` address and `port` that we provided. After that, it will constantly listen for the `client` requests.

```python
class Server:

 def main(self):
         try:
                 SERVER_PORT = int(sys.argv[1])
         except:
                 print("[Usage: Server.py Server_port]\n")
         rtspSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
         rtspSocket.bind(('', SERVER_PORT))
         rtspSocket.listen(5)

         # Receive client info (address,port) through RTSP/TCP session
         while True:
                 clientInfo = {}
                 clientInfo['rtspSocket'] = rtspSocket.accept()
                 ServerWorker(clientInfo).run()
```

Listing 1: The server is created and constantly listen for requests

When we run the command `python ClientLauncher.py server_host server_port RTP_port Video-file`, `ClientLauncher.py` it will receive all the parameters that we provided and call `Client` in `Client.py`, at `Client`, it will initialize the status, parameters, and it will create a socket to connect to the `Server`.

```
if __name__ == "__main__":
 try:
         serverAddr = sys.argv[1]
         serverPort = sys.argv[2]
         rtpPort = sys.argv[3]
         fileName = sys.argv[4]
 except:
         print("[Usage: ClientLauncher.py Server_name Server_port RTP_port
         Video_file]\n")

 root = Tk()

 # Create a new client
 app = Client(root, serverAddr, serverPort, rtpPort, fileName)
 app.master.title("RTPClient")
 root.mainloop()
```

Listing 2: The `ClientLauncher` create new client

The `server` will listen and create connection when it receives requests from the `client`. At `ServerWorker`, the states, the codes are initialized to be ready for call from the `client`.At the same time, `client` setup the UI of the program.

```python
def createWidgets(self):
        """Build GUI."""
        # Create Setup button
        self.setup = Button(self.master, width=20, padx=3, pady=3)
    self.setup["text"] = "Setup"
        self.setup["command"] = self.setupMovie
        self.setup.grid(row=1, column=0, padx=2, pady=2)

        # Create Play button
        self.start = Button(self.master, width=20, padx=3, pady=3)
        self.start["text"] = "Play"
        self.start["command"] = self.playMovie
        self.start.grid(row=1, column=1, padx=2, pady=2)

        # Create Pause button
        self.pause = Button(self.master, width=20, padx=3, pady=3)
        self.pause["text"] = "Pause"
        self.pause["command"] = self.pauseMovie
        self.pause.grid(row=1, column=2, padx=2, pady=2)

        # Create Teardown button
        self.teardown = Button(self.master, width=20, padx=3, pady=3)
        self.teardown["text"] = "Teardown"
        self.teardown["command"] =  self.exitClient
        self.teardown.grid(row=1, column=3, padx=2, pady=2)

        # Create a label to display the movie
        self.label = Label(self.master, height=19)
        self.label.grid(row=0, column=0, columnspan=4,
        sticky=W+E+N+S, padx=5, pady=5)
```

Listing 3: The `UI widget` is created

After having created `server` and `client` connected to each other by `RTCP/TCP`. `Client` will sent to the server commands include:

- SETUP.
- PLAY.
- PAUSE.
- TEARDOWN.

Theses commands indicate the user's actions in the future. The `server` is supposed to response `client` via `RTP`. The responses include:

- OK 200.
- FILE_NOT_FOUND 404.
- CONNECTION_ERROR 500.

After received the responses from the `server`, the `client` will change its stat corresponding to the response.

1.5 Class diagram

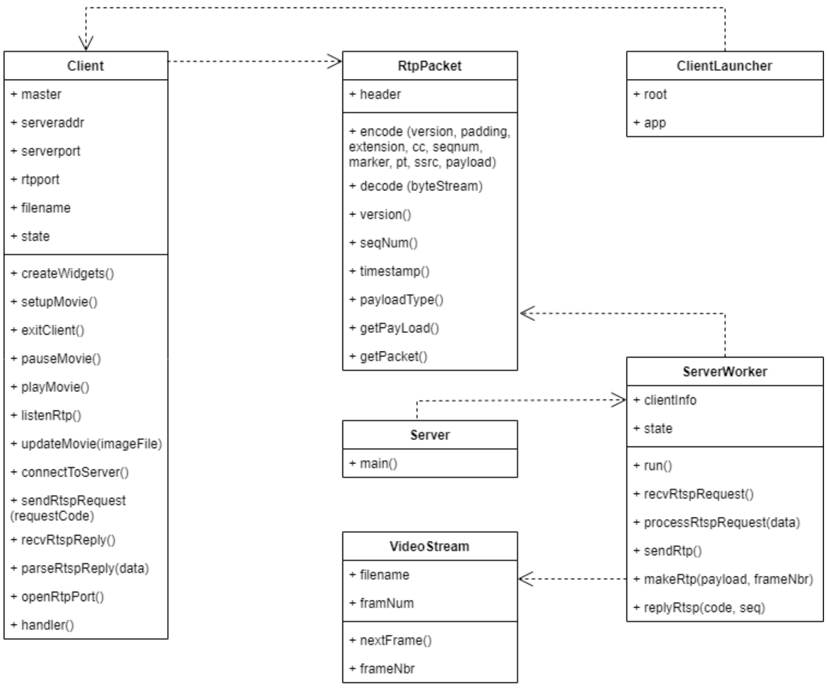

Figure 1: The class diagram of the program

2 Solution Implementation

2.1 Code implementation

2.1.1 SET UP

When the `client` send the `SETUP` request to the `server`, it will receive the response of the `RTSP` from the `ServerWorker`.The `SETUP` request contain:

- SETUP command.
- The name of the video.
- The sequence of the RTSP packet start from 1.

- Type of protocol: RTSP/1.0.
- Transport protocol: RTP/UDP.
- The client port.

```
request = "SETUP " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(
    self.rtspSeq) + "\nTransport: RTP/UDP; client_port= " + str(self.rtpPort)
self.rtspSocket.send(request.encode())
```

Listing 4: The structure of the SETUP command

When the server receives the SETUP request from the client, it will:

1. Generate a randomized RTSP session ID.
2. If there was error, it will reply with the code 404 FILE NOT FOUND, otherwise, it will reply with the code 200 OK.
3. Open the video corresponding to the filename and initialize the frame number to 0.

```
if requestType == self.SETUP:
        if self.state == self.INIT:
                # Update state
                print("processing SETUP\n")

                try:
                        self.clientInfo['videoStream'] = VideoStream(filename)
                        self.state = self.READY
                except IOError:
                        self.replyRtsp(self.FILE_NOT_FOUND_404, seq[1])

                # Generate a randomized RTSP session ID
                self.clientInfo['session'] = randint(100000, 999999)

                # Send RTSP reply
                self.replyRtsp(self.OK_200, seq[1])

                # Get the RTP/UDP port from the last line
                self.clientInfo['rtpPort'] = request[2].split(' ')[3]
```

Listing 5: The job of server when receive the command SETUP

2.1.2 PLAY

When we press the `PLAY` button on the UI, the `client` will call the `playMovie` function, this function will start a new thread to listen for the `RTP` reply from the `server` and call the function `sendRtspRequest` to send the `RTSP` request to the `server`.

```python
def playMovie(self):
        """Play button handler."""
        if self.state == self.READY:
                # Create a new thread to listen for RTP packets
                threading.Thread(target=self.listenRtp).start()
                self.playEvent = threading.Event()
                self.playEvent.clear()
                self.sendRtspRequest(self.PLAY)
```

Listing 6: The job of the `playMovie` function

The request send to the `server` is formatted as follow:

- The file name
- The sequence number
- The session

```python
elif requestCode == self.PLAY and self.state == self.READY:
    # Update RTSP sequence number.
    # ...
    self.rtspSeq += 1

    # Write the RTSP request to be sent.
    # request = ...
    request = "PLAY " + str(self.fileName) + " RTSP/1.0\nCSeq: " + \
        str(self.rtspSeq) + "\nSession: " + str(self.sessionId)
    self.rtspSocket.send(request.encode())
    # Keep track of the sent request.
    # self.requestSent = ...
    self.requestSent = self.PLAY
```

Listing 7: The request structure to the `server`

In the function `listenRTP`, we keep listening until there is the command `PAUSE` or `TEARDOWN`, if we receive the data, we will try to decode it to get the `seqNum`, and `frameNbr` to the `seqNum`, this would mean that we will discard any packets that arrived late.

```python
def listenRtp(self):
    """Listen for RTP packets."""
    while True:
        try:
            data = self.rtpSocket.recv(20480)
            if data:
                rtpPacket = RtpPacket()
                rtpPacket.decode(data)

                curFrameNbr = rtpPacket.seqNum()
                print("Current Seq Num: " + str(curFrameNbr))

                if curFrameNbr > self.frameNbr:  # Discard the late packet
                    self.frameNbr = curFrameNbr
                    self.updateMovie(rtpPacket.getPayload())
        except:
            # Stop listening upon requesting PAUSE or TEARDOWN
            if self.playEvent.isSet():
                break

            # Upon receiving ACK for TEARDOWN request,
            # close the RTP socket
            if self.teardownAcked == 1:
                self.rtpSocket.shutdown(socket.SHUT_RDWR)
                self.rtpSocket.close()
                break
```

Listing 8: The `listenRtp` function

When we set the `frameNbr` to the `currFramNbr`, we then call the function `updateMovie` to write the frame directly to the movie. We did not use the function `writeFrame` as in the source code to write the frame to a temporary file.

```python
def updateMovie(self, image_data):
    """Update the image file as video frame in the GUI."""
    image = Image.open(io.BytesIO(image_data))
    photo = ImageTk.PhotoImage(image)
    self.label.configure(image=photo, height=288)
    self.label.image = photo
```

Listing 9: The `updateMovie` function

On the `server` side, when receive the request from the `client`, it will call the function `replyRtsp` to reply the request and at the mean time, it must create the `RTP` packet to send to the `client`.

```python
elif requestType == self.PLAY:
 if self.state == self.READY:
   print("processing PLAY\n")
   self.state = self.PLAYING

   # Create a new socket for RTP/UDP
   self.clientInfo["rtpSocket"] = socket.socket(socket.AF_INET,
   socket.SOCK_DGRAM)

   self.replyRtsp(self.OK_200, seq[1])

   # Create a new thread and start sending RTP packets
   self.clientInfo['event'] = threading.Event()
   self.clientInfo['worker']= threading.Thread(target=self.sendRtp)
   self.clientInfo['worker'].start()
```

Listing 10: The job of the `server`

In the `replyRtsp` function, we will print out the status according to the code.

```python
def replyRtsp(self, code, seq):
        """Send RTSP reply to the client."""
        if code == self.OK_200:
                #print("200 OK")
                reply = 'RTSP/1.0 200 OK\nCSeq: ' + seq +
                '\nSession: ' + str(self.clientInfo['session'])
                connSocket = self.clientInfo['rtspSocket'][0]
                connSocket.send(reply.encode())

        # Error messages
        elif code == self.FILE_NOT_FOUND_404:
                print("404 NOT FOUND")
        elif code == self.CON_ERR_500:
                print("500 CONNECTION ERROR")
```

Listing 11: The `replyRtsp` function

Now we come to the part that we must make the `RTP` packet and send it back to the `client`, the 2 illustrates the `RTP` format that we need to send.

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

Figure 2: The `RTP` header

First, we make the packet in the `makeRtp` function, here we must pass in `payload` and `frameNbr` as parameters.

```python
def makeRtp(self, payload, frameNbr):
        """RTP-packetize the video data."""
        version = 2
        padding = 0
        extension = 0
        cc = 0
        marker = 0
        pt = 26 # MJPEG type
        seqnum = frameNbr
        ssrc = 0
        rtpPacket = RtpPacket()
        rtpPacket.encode(version, padding, extension, cc,
        seqnum, marker, pt, ssrc, payload)
        return rtpPacket.getPacket()
```

Listing 12: The `makeRtp` function

After having make the `RTP` packet, we must now encode it before sending it to the `client`, we can accomplish that using the `encode` function from the module `RtpPacket`. Here we did somw bitwise operation to make the header of the `RTP` packet.

```python
def encode(self, version, padding, extension,
cc, seqnum, marker, pt, ssrc, payload):
    """Encode the RTP packet with header fields and payload."""
    timestamp = int(time())
    header = bytearray(HEADER_SIZE)
    # --------------
    # TO COMPLETE
    # --------------
    # Fill the header bytearray with RTP header fields

    self.header[0] = (version << 6
                      | padding << 5
                      | extension << 4
                      | cc)

    self.header[1] = (marker << 7
                      | pt)

# 2 bytes of sequence number
    self.header[2] = (seqnum >> 8) & 0xFF
    self.header[3] = seqnum & 0xFF

# 4 bytes of timestamp
    self.header[4] = (timestamp >> 24) & 0xFF
    self.header[5] = (timestamp >> 16) & 0xFF
    self.header[6] = (timestamp >> 8) & 0xFF
    self.header[7] = timestamp & 0xFF

# 4 byte of SSRC
    self.header[8] = ssrc >> 24
    self.header[9] = ssrc >> 16
    self.header[10] = ssrc >> 8
    self.header[11] = ssrc

    # Get the payload from the argument
    self.payload = payload
```

Listing 13: Bitwise operation to encode the message

Finally, when everything is done, we could now send the `RTP` packet via the `sendRtp` function.

```
def sendRtp(self):
        """Send RTP packets over UDP."""
 while True:
        self.clientInfo['event'].wait(0.05)

        # Stop sending if request is PAUSE or TEARDOWN
        if self.clientInfo['event'].isSet():
                break

        data = self.clientInfo['videoStream'].nextFrame()
        if data:
         frameNumber = self.clientInfo['videoStream'].frameNbr()
         try:
                address = self.clientInfo['rtspSocket'][1][0]
                port = int(self.clientInfo['rtpPort'])
                self.clientInfo['rtpSocket'].sendto(self.makeRtp(data,
                frameNumber),(address,port))
         except:
                print("Connection Error")
```

Listing 14: The `sendRtp` function

2.1.3 PAUSE

When the command `PAUSE` is pressed, the `client` send `RTSP` request to the `SERVER` to stop the `server` from sending more frames to the `client` and also change its state into `READY`

```
elif requestCode == self.PAUSE and self.state == self.PLAYING:
    self.rtspSeq += 1

    request = "PAUSE " + str(self.fileName) + " RTSP/1.0\nCSeq: " + \
        str(self.rtspSeq) + "\nSession: " + str(self.sessionId)
    self.rtspSocket.send(request.encode())

    self.requestSent = self.PAUSE
```

Listing 15: The `RTSP` request when we press the `PAUSE` command

Next, when the `server` receives this command it will stop sending frames and wait for the next request from the `client`.

```
elif requestType == self.PAUSE:
        if self.state == self.PLAYING:
                print("processing PAUSE\n")
                self.state = self.READY

                self.clientInfo['event'].set()

                self.replyRtsp(self.OK_200, seq[1])
```

Listing 16: The `server` set it state to `READY` and wait

2.1.4 Teardown

When the `TEARDOWN` command is sent from `client` to `server`, it will stop the `server` from sending more frames to the `client` and also close the socket that connects both side. The `client` also sets its state into `INIT`

```
elif requestCode == self.TEARDOWN and not self.state == self.INIT:
    # Update RTSP sequence number.
    # ...
    self.rtspSeq += 1
    # Write the RTSP request to be sent.
    # request = ...
    request = "TEARDOWN " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(
        self.rtspSeq) + "\nSession: " + str(self.sessionId)
    self.rtspSocket.send(request.encode())
    # Keep track of the sent request.
    # self.requestSent = ...
    self.requestSent = self.TEARDOWN
```

Listing 17: The `TEARDOWN` command

```
elif requestType == self.TEARDOWN:
        print("processing TEARDOWN\n")

        self.clientInfo['event'].set()

        self.replyRtsp(self.OK_200, seq[1])

        # Close the RTP socket
        self.clientInfo['rtpSocket'].close()
```

Listing 18: The `server` closes the socket and stop sending frames

2.2 User manual

2.2.1 Run the program

At the folder containing the source code, open two terminals.
In the first terminal, run the command:

`py Server.py <server-port>`

`<server-port>` is the port that you want to listen all the `RTSP` requests at. Standard `RTSP` port is 554, but we have to choose > 1024. For example:

`py Server.py 2000`

At the second terminal, run the command:

`py ClientLauncher.py <server-host> <server-port> <RTP-port> <video-file>`

In this command:

- `server-host` is the IP address of the `server`, in my case it is `192.168.0.102`, it will be different in your case.
- `server-post` is the same at the port you created in the first terminal (2000).
- `RTP-port` is the port that you want to receive the `RTP` packets at, you can choose a random positive integer.
- `video-file` the video name you want to be played, in this case it is `movie.Mjpeg`

For example:

`py ClientLauncher.py 192.168.0.102  2000 100 movie.Mjpeg`

2.2.2 Basic GUI Manual

In the basic UI of our application, there are four buttons representing the four basic functions such as SETUP, PLAY, PAUSE and TEARDOWN.

Figure 3: The UI of the basic version

When in the UI, the first thing you need to do is to press the Setup button, this is the most important step in order to run our application successfully. The Setup button's function is as the name suggested, to set up the connection between the client and the server and change the client state from INIT to READY so that the client can play the video.

After that, press the PLAY button and the video file will be played in the UI. While the video is playing, you can interact with the UI by pressing Pause, which will pause the video, or Teardown, which will kill the connection, close the socket and terminate the UI.

One important notice is that the response of the action when pressing each button is different depending on the state of the client at the time of pressing. For example, when you first launch the UI but have not press the Setup button (the client is still in INIT state), pressing the other three buttons will have no effect; similarly, while the video file is being played (the client's current state is PLAY), pressing the Play or Setup button will do nothing.

2.3 Source Control Version

Our source code are hosted at GitHub.

3 Extended Problems

For this section, the whole User Interface is reconsidered and designed differently from the fore-mentioned because of the requirements. Moreover, files that were not supposed to be modified now have some changes to adapt to the new functionalities, those are: `ServerWorker.py`, `VideoStream.py`, and in addition, the file `Client.py` also has a major update.

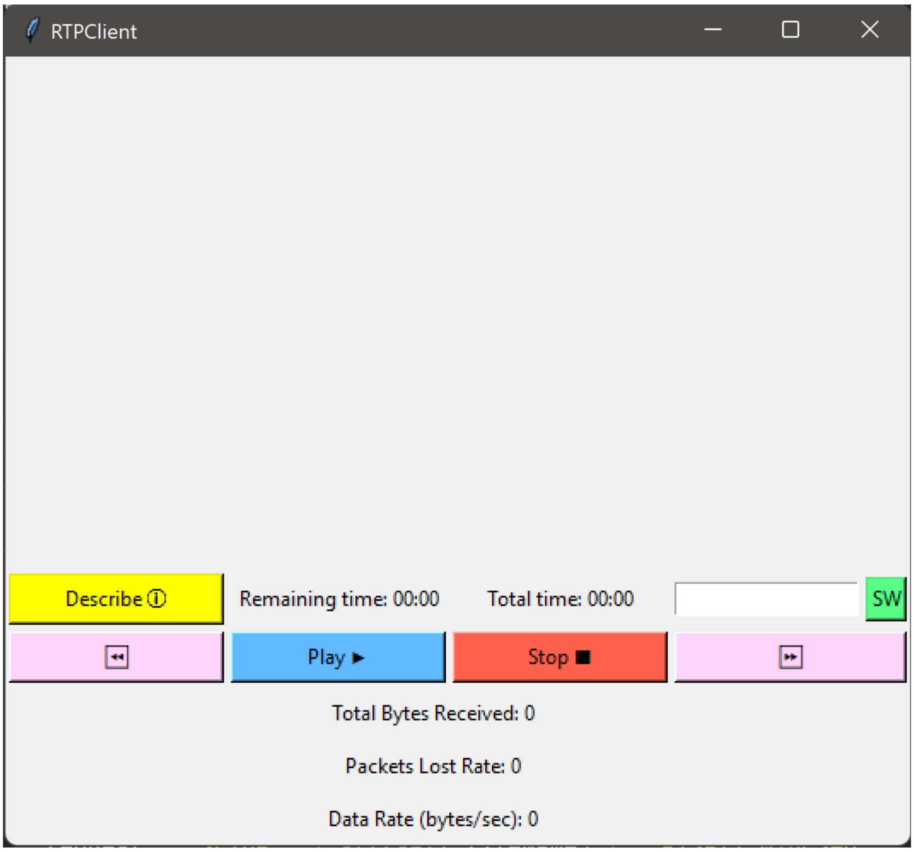

Figure 4: New UI of the extend version

In `ServerWorker.py`, there is a delay of 0.05s between every frame being sent to the client. Although this is fine, the true frame per second of the file `movie.mjpeg` is 25 fps. This means a delay of 0.05s is not appropriate for calculating the duration of the video, so in order to measure precisely, the delay is changed to 0.04.

```
TIME_PER_FRAME = 0.04

...
    self.clientInfo['event'].wait(TIME_PER_FRAME)
...
```

3.1 Extend 1

This problem involves visualizing the statistics of the current packets transportation, which includes total bytes received, lost rate and data rate.

In the figure above we see that the last three rows describe the statistics we need. The statistical number will begin to fluctuate when the client initialize the connection and then start requesting the videos:

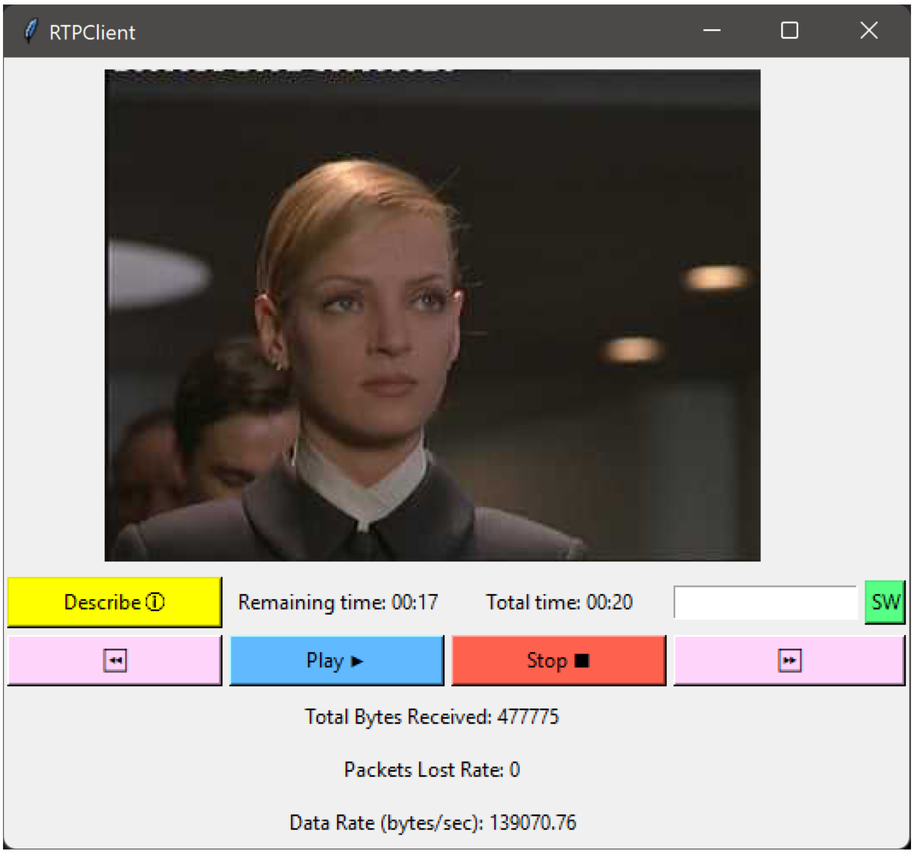

Figure 5: Statistics of the RTP protocol transmission

We observed that the lost rate is non-existent. In reality, this is unlikely, but due to server and the client is conducted in the same local computer, which leads to the same router so it is very efficient to transmit data without any lost packets.

3.2 Extend 2

As illustrated in the first figure of this extended section, there is no `SETUP` button. We also combine two distinct functionality (`PLAY` and `PAUSE`) into a single button because of their nature being only executed after the other is done. The `TEARDOWN` button is also renamed to `STOP`, which is the equivalent to the button that are widely used in many media players.

The removal of the `SETUP` button is achieved by measuring the first play button of the client when the application is first opened. If the client has just opened, the state is in initialized state but not ready state. When the PLAY button is pressed, the application automatically sends SETUP request, only after receiving a successful SETUP reply then it sends a PLAY request.

```
if self.requestSent == self.SETUP:
    ...

    self.playMovie() # Send PLAY request to server
```

The STOP button is another function that modern media players use. Its function is to revert the video to the beginning state. This is quite different from tearing down a whole connection, but statistically users when stopping a video rarely play again. This allows the STOP button to behave similarly to the TEARDOWN button.

3.3 Extend 3

DESCRIBE requests are used to get the current information of the video being streamed to the client. There are many fields are included in a single reply, but here the problem only states that the reply only includes what kinds of streams are in the session and what encodings are used.

Here we included a yellow button on the top-left corner and labeled it *Describe*. When the user press the button, the client application sends a DESCRIBE request to the server. After receiving the reply from the server, it parses the information as text in the console:

```
Data sent:
DESCRIBE movie.mjpeg RTSP/1.0
CSeq: 4
Session: 452359
----------------------------------------
Data received:
RTSP/1.0 200 OK
CSeq: 4
Session: 452359
Content-Base: movie.mjpeg
Content-Length: 86
v=0
m=video 5008 RTP/AVP 26
a=control:streamid=452359
a=mimetype:string;"video/Mjpeg"
```

3.4 Extend 4

As mentioned before, in order to observe the exact time of the video, the frame rate has to match the delay, which is why the frame rate is 25 fps and the delay is $1/25 = 0.04s$.

This allows the application to calculate accordingly to the frame number received. To visualize the current time we only need to divide the current frame number by the frame rate. But what we actually need is the remaining time and a display of total time. For that the server must return a total duration of the video, specifically from the `SETUP`, so that the client can retrieve the data and then calculate the remaining time.

```
def replySetup(self, code, seq):
    if code == self.OK_200:
        ...
        with self.lock:
            totalFrame = self.clientInfo['videoStream'].get_total_frame()
        reply = 'RTSP/1.0 200 OK\nCSeq: ' + seq + '\nSession: ' + \
            str(self.clientInfo['session']) + \
            '\nTotalFrame: ' + str(totalFrame)
        ...
```

Fast forward and go backward are also implemented here. This is achieved by creating whole new methods to specify the request and reply messages. These are not include in standard RTSP methods, they are `FORWARD` and `BACKWARD`. Here the steps leaping from the current time is 5 seconds, which is translated to 125 frames. If the client sends a request the server will skip to the required frame to start to send the video from there. The buttons are colored pink and labeled the respective direction.

In the solving of this problem arise a new issue, that is the race conditions. Altering the current frame involves reading to a specific point of the video file. This makes the server executing the procedure inside of the receiving request thread, which is running concurrently to the sending packets thread. Both of the two threads have a functionality to read the video file, which causes race condition. We resolve this by introducing thread locks to every reading procedure.

3.5 Extend 5

Here we provide a method for the user to insert a new video file name and then start receiving that corresponding movie. In the extended UI, there is a text box that the user can insert text into, and a green button labeled *SW* to submit the text.

Similar to the forth problem, we invent a new method called `SWITCH`. When the server receives this request, it renders the new file name and start sending that video from the beginning (Note that the file name that the client sent must exist in the server side).

Here the server currently has two files: `movie.mjpeg` and `movie2.mjpeg`. The first video file is used in every problem above, the second one is double in length of the first and second half of it is just the rewinding of the first video file.

Opening the application first with `movie.mjpeg` file:

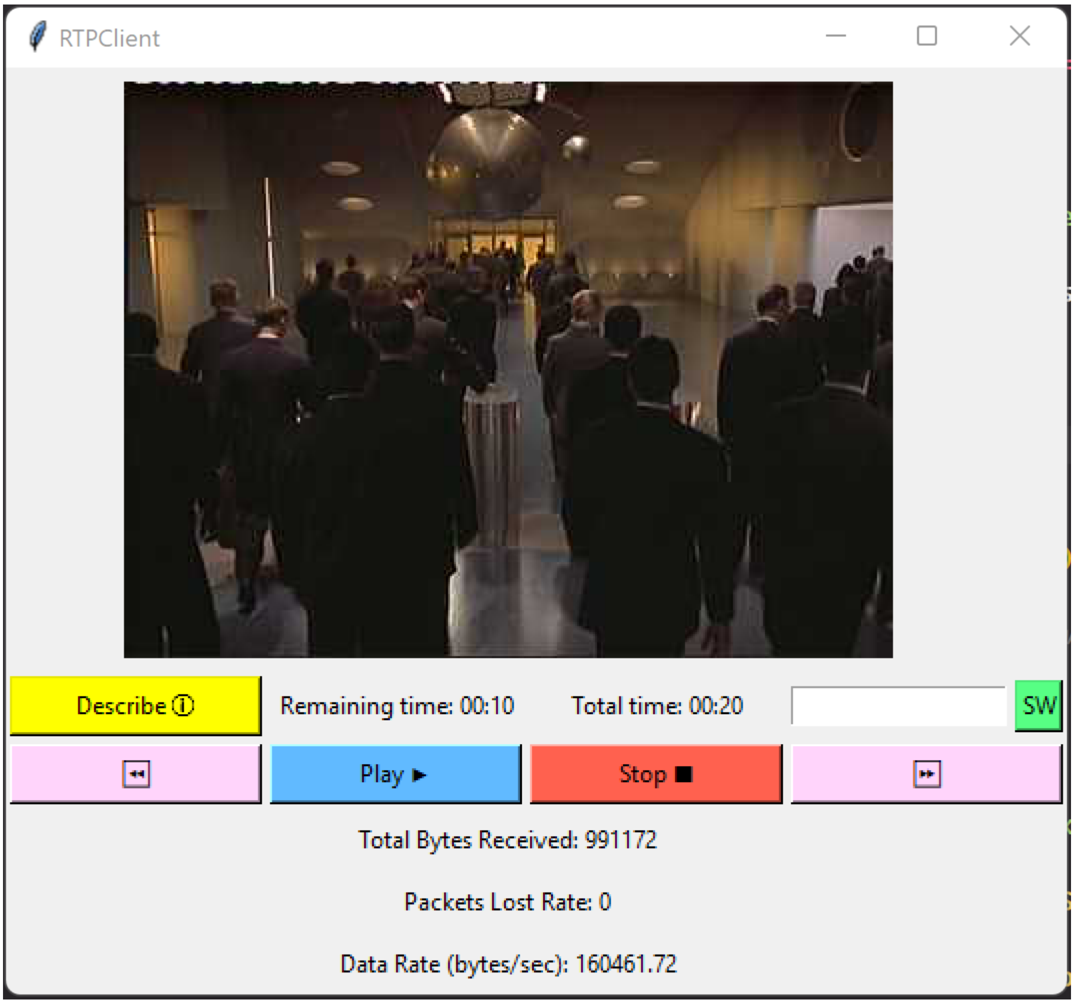

Figure 6: Streaming movie.mjpeg.

Inserting `movie2.mjpeg` into the text box:

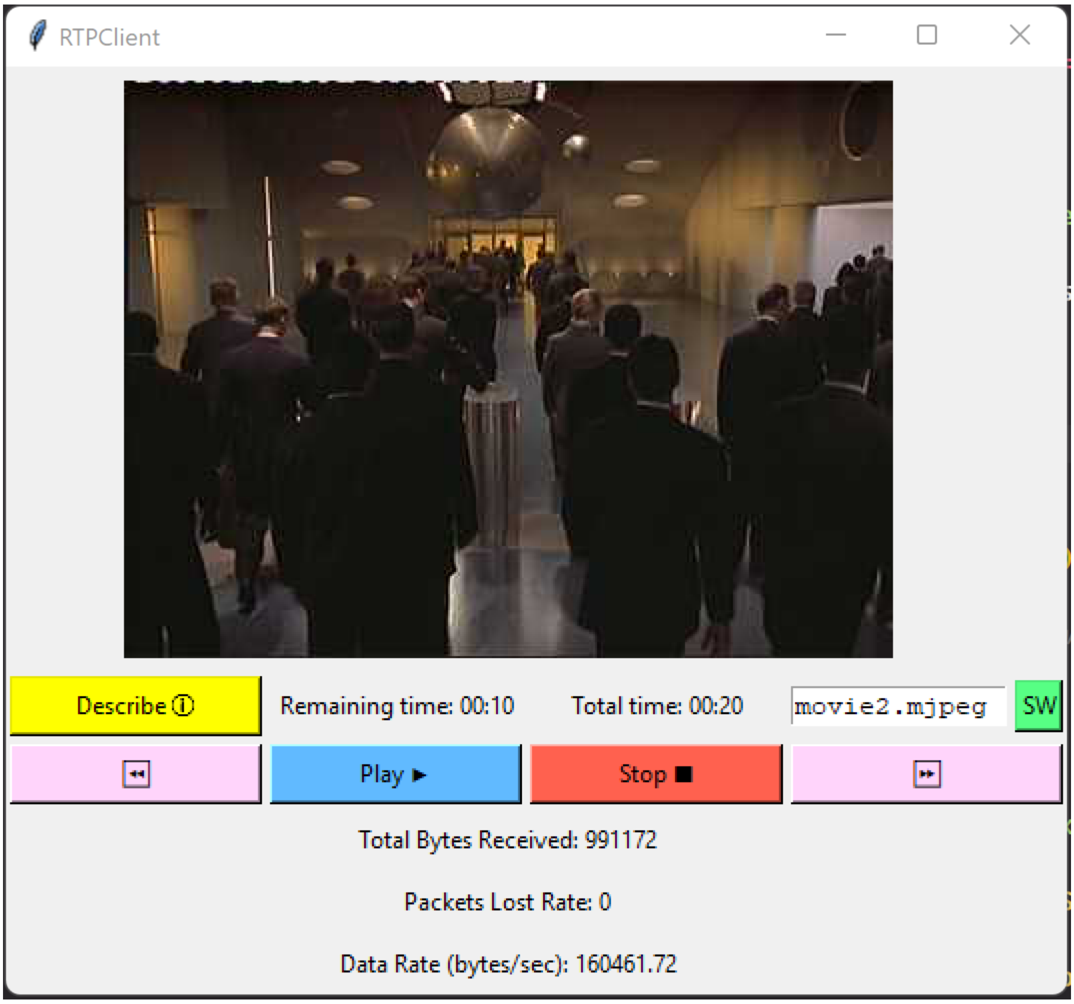

Figure 7: Text box with movie2.mjpeg.

Press the `SW` button and then the new video is being streamed:

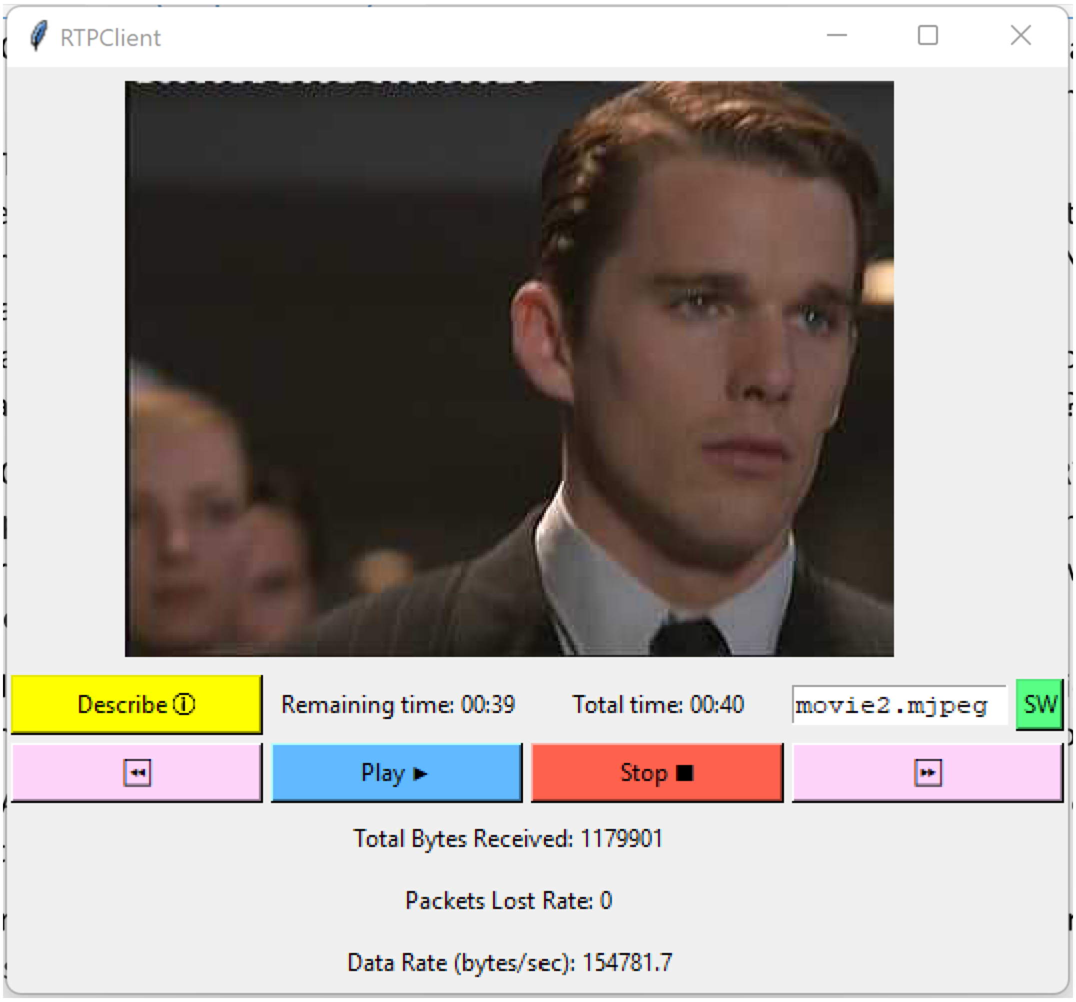

Figure 8: Streaming movie2.mjpeg, the total length now has been doubled.

References

[1] Audio-Video Transport Working Group et al. *RFC1889: RTP: A transport protocol for real-time applications*. 1996.

[2] James F Kurose. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[3] Colin Perkins. *RTP: Audio and Video for the Internet*. Addison-Wesley Professional, 2003.

[4] Henning Schulzrinne, Anup Rao, and Robert Lanphier. "Real time streaming protocol (RTSP)". In: (1998).

[5] Henning Schulzrinne et al. *RFC3550: RTP: A transport protocol for real-time applications*. 2003.

[6] Henning Schulzrinne et al. *RTP: A transport protocol for real-time applications*. 1996.