
Contents

1	Introduction	4
2	Requirement Analysis	4
2.1	Functional Requirements	4
2.2	Non-Functional Requirements	4
3	Function Description	5
3.1	Client class	5
3.2	Server class	6
3.3	ServerWorker class	6
3.4	VideoStream class	6
3.5	RtpPacket class	6
4	Class Diagram	8
5	Implementation	10
5.1	The client side	10
5.2	RTPPacket.py	11
6	Evaluation	12
6.1	GUI Development	12
6.2	Client-server modeling	12
6.3	RTP fields and characteristics	13
6.4	Twiddling the bits	13
7	User manual	14
7.1	Running the player	14
7.2	After the GUI is run	15
8	Player extensions implementation	17
8.1	Calculate the session statistics	17
8.1.1	RTP packet lost rate	17
8.1.2	Video data rate	18
8.2	Implement the PLAY, PAUSE and STOP buttons	18
8.2.1	Basic implementation	18
8.2.2	TEARDOWN when user clicks on STOP button	19
8.3	Implement the DESCRIBE method	20
8.4	Additional functions for user interface such as: display video total time and remaining time, fast forward or backward video and a seek bar	21
8.4.1	Total time and Remaining Time	21
8.4.2	Fast forward and backward video	22
8.4.3	Seekbar	22

1 Introduction

The solution presented in this report is aimed to develop a system containing a streaming video server and client that satisfies the given specifications.

The table below presents the work assigned for each of our team members.

Name	ID	Works	Percentage
Vu Hoang Hai	1952669	- Complete Client.py - Write report	33.33%
Le Nguyen Tan Loc	1952088	- Develop the GUI	33.33%
Nguyen Le Thao Vy	1952536	- Complete RtpPacket.py - Write report	33.33%

Table 1: Individual workloads

2 Requirement Analysis

2.1 Functional Requirements

1. The system consists of a streaming video server and client that communicate using the Real-Time Streaming Protocol (RTSP)
2. Data transfer is done using the Real-time Transfer Protocol (RTP).
3. The system is able to present any MJPEG file as video.
4. The system provides an interactive GUI, with at least 4 buttons: SETUP, PLAY, PAUSE and TEARDOWN.
5. On the server side, video data must be packetized into RTP packets.

2.2 Non-Functional Requirements

1. The response time of the system must be less than or equal 0.50 seconds.
2. Server sends an RTP packet to client every 50 milliseconds.
3. Encapsulation of data must be provided for security reasons.
4. The update of Client's state must be completed immediately when needed.
5. The timeout value of datagram socket for receiving RTP data is 0.50 seconds.

3 Function Description

3.1 Client class

- *__init__(self, master, serveraddr, serverport, rtpport, filename)*: Initialize several variables and call other functions, e.g. *createWidgets()* / *connectToServer()*.
- *createWidgets(self)*: Build GUI.
- *playButtonCommand(self)*:
- *setupMovie(self)*: Setup button handler.
- *exitClient(self)*: Handle click(s) on the TEARDOWN button.
- *pauseMovie(self)*: Handle click(s) on the PAUSE button.
- *forwardMovie(self)*: Handle click(s) on the FORWARD button.
- *backwardMovie(self)*: Handle click(s) on the BACKWARD button.
- *playMovie(self)*: Handle click(s) on the PLAY button.
- *updateCountDownTimer(self)*: Update the countdown timer.
- *sliding(self, arg)*: Handle interactions with the sliding bar.
- *describeMovie(self)*: Handle click(s) on the DESCRIBE button.
- *switchWindow(self)*: Create new window for selecting video.
- *switchMovie(self)*: Handle click(s) on the SWITCH button.
- *listenRtp(self)*: Wait for RTP packets.
- *writeFrame(self, data)*: Write the received frame to a temp image file and return the image file.
- *updateMovie(self, imageFile)*: Update the image file as video frame in the GUI.
- *connectToServer(self)*: Connect to the Server and start a new RTSP/TCP session.
- *sendRtspRequest(self, requestCode)*: Send RTSP request to the server.
- *recvRtspReply(self)*: Receive RTSP reply from the server.
- *fileNameCallBack(self, *args)*:
- *parseRtspReply(self, data)*: Parse the RTSP reply from the server.
- *openRtpPort(self)*: Open RTP socket binded to a specified port.
- *handler(self)*: Handle the explicitly closing of the GUI window.
- *updateOptionsMenu(self)*: Create a drop bar.

3.2 Server class

- *main(self)*: Initialize a socket, assign SERVER_PORT to it and receive client info through RTSP/TCP session.

3.3 ServerWorker class

- *__init__(self, clientInfo)*: Assign the value of "clientInfo" to the field "self.clientInfo".
- *run(self)*: Create a new thread running concurrently with the main thread to handle the RTP/RTSP request(s) of each client.
- *recvRtspRequest(self)*: Receive RTSP request from client(s).
- *getAllMediaFiles(self)*: Return a list of name of videos type Mjpeg or mjpeg.
- *processRtspRequest(self, data)*: Process RTSP request received from client(s).
- *sendRtp(self)*: Send RTP packets over UDP.
- *makeRtp(self, payload, frameNbr)*: RTP-packetize the video data.
- *replyRtsp(self, code, seq, req = ", file = ")*: Send RTSP reply to the client.

3.4 VideoStream class

- *__init__(self, filename)*: Read the file whose name specified as the input "filename".
- *nextFrame(self, forward, backward)*: Get the next frame.
- *getWholeVideo(self)*: Append new video to the list.
- *calNumFrames(self)*: Calculate total number of frames.
- *calFps(self)*: Calculate the number of frames per second.
- *calTotalTime(self)*: Calculate the total time of the video.
- *getSize(self)*: Get the size of the video.
- *frameNbr(self)*: Get the frame number.
- *resetFrame(self)*: Reset the movie frame.

3.5 RtpPacket class

- *encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload)*: Encode the RTP packet with header fields and payload.
- *decode(self, byteStream)*: Decode the RTP packet.
- *version(self)*: Return version of RTP.
- *seqNum(self)*: Return sequence (frame) number.
- *timestamp(self)*: Return timestamp.

- *payloadType(self)*: Return type of payload.
- *getPayload(self)*: Return payload.
- *getPacket(self)*: Return RTP packet.

4 Class Diagram

The following section describes the class diagram of the media player system. It includes 7 main classes of all the associated classes inside the system modules. The classes, entities, enumerations, and subclasses are listed in the diagram.

MEDIA PLAYER CLASS

May 5th, 2022

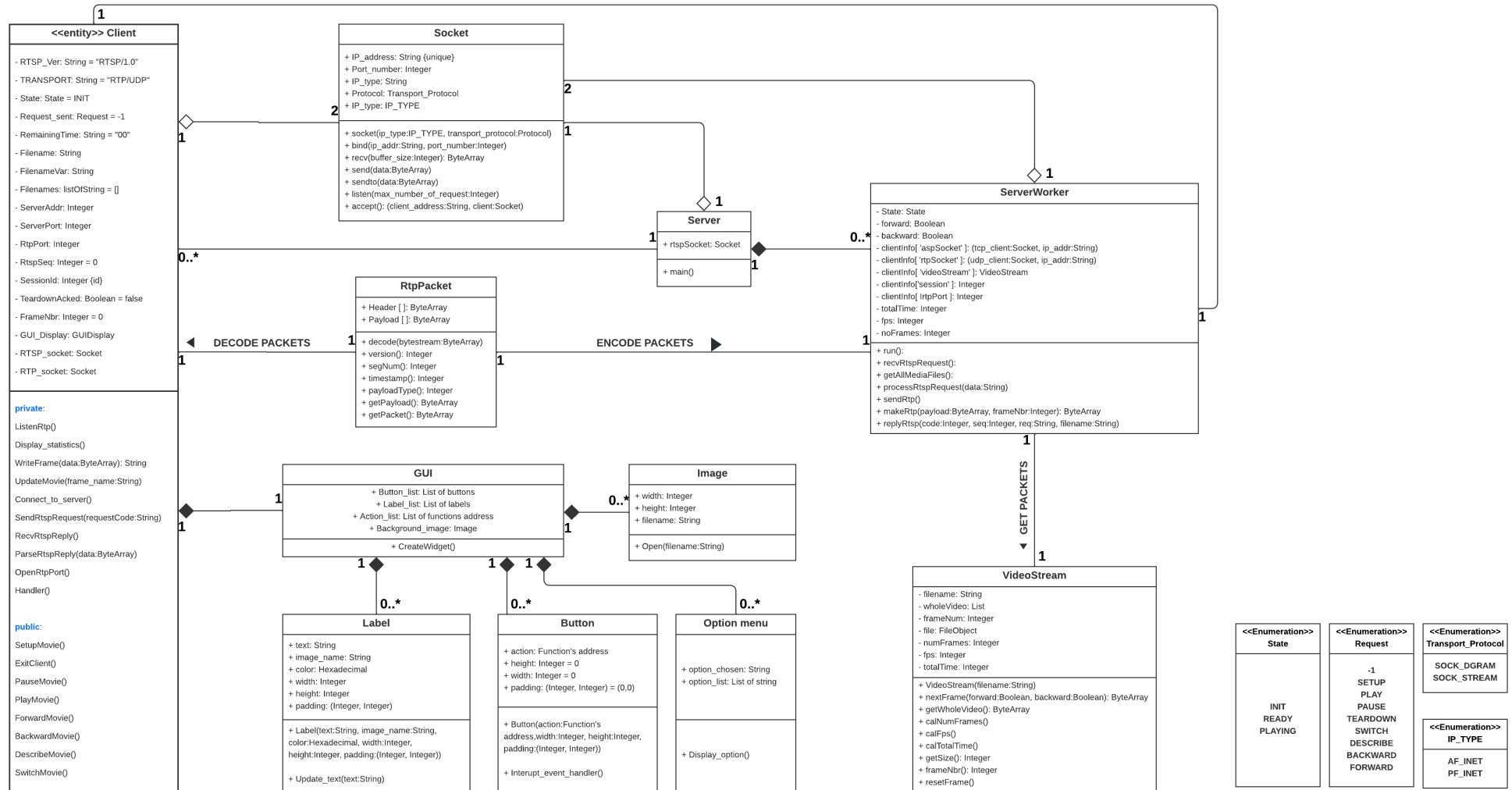


Figure 1: Class diagram of the media player system.

5 Implementation

5.1 The client side

The RTSP protocol is implemented on the client side with appropriate functions for the user interface of the media player. The follow section show describes these functions.

The methods are encoded numerically in the client class to describe the state that the player should be in for the server side.

Setting up the movie interface

To set up the `setupMovie(self)` function, we will simply initiate the setup process of the RTSP protocol via the `sendRtspRequest(self.SETUP)` method. This function will be discussed in the later implementation.

Connecting to the server

Following the set up is connecting to the RTSP server using similar socket interfaces to TCP/IP via `rtspSocket.connect((self.serverAddr, self.serverPort))`

Tearing down (exiting) the interface

Upon exiting the interface, `exitClient(self)`, the client will check whether there is a video playing that needs to be torn down. If there are, it will proceed to destroy the cached video images and signal RTSP teardown using `sendRtspRequest(self.TEARDOWN)`.

Pausing and playing video

When the user hits play, a thread is created to continuously listen for RTSP packets from the streaming server via `threading.Thread(target=self.listenRtp).start()`. This invokes the function `listenRtp(self)` that runs indefinitely. The function shall be discussed in the next part:

To pause the video, the client simply sends an RTSP pause request of `sendRtspRequest(self.PAUSE)`.

Listening for RTSP packets, writing received contents to cached memory

After securing the connection, the client starts to receive RTSP packets. An object is created via the `RtpPacket()` and starts decoding and sequencing the Mjpeg, frame by frame. If there are missing packets during the transfer (by checking whether is expected frame number is different from the one we received), the client displays the lost packets in total.

The current image frame is replaced with the newly received frame once the transfer is completed.

Updating the GUI

Once the data is present in the cached storage, the client proposes to display it on the GUI via the `updateMovie(self, imageFile)` function.

Handling RTSP requests

Each RTSP request is handled differently for each functionality. The common format will be discussed for every request type. When a client pushes an RTSP request, it is written as

```
[rqst keyword] + str(self.fileName) + "RTSP/1.0" + "CSeq:" + str(self.rtspSeq)
+ "\n" + "Session: " + str(self.sessionId)
```

This string is then transferred to the server to handle RTSP requests. The appropriate track is assigned via `self.requestSent = self.SETUP`.

5.2 RTPPacket.py

Encoding/Decoding RTSP Packets

The RTSP packet is now encoded with header fields which tell the server the proprietary fields of the video encoding. This include version, padding, extension, copyright, marker, pt for meta-datas and 16-bit sequence number and 32-bit timestamp. The encodings are illustrated in the figure below. The bits are shifted as more metadata are added.

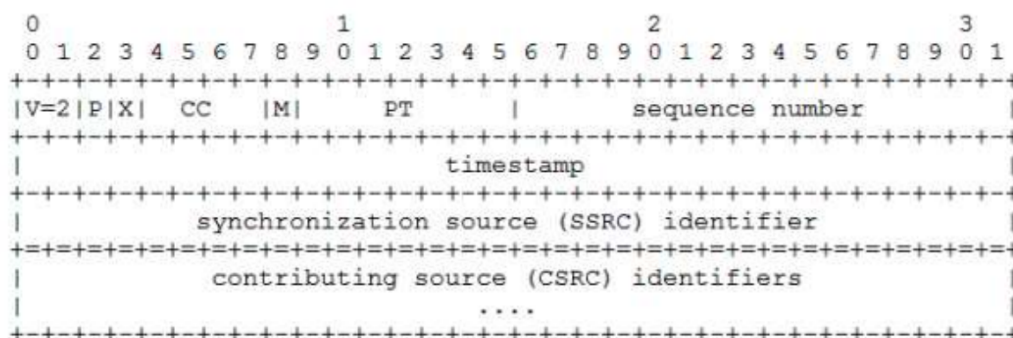


Figure 2: Bit encoding datamap.

To the decode the incoming package sent from the server, the message will be converted to a byte array and attached to the appropriate payload. The payload functions are described below.

Extracting headers from decoding process

Once the server message (preferably image data) is in a byte format in header, it is extracted for its metadata using the enlisted encoding agreement.

There are six different methods, including `version`, `seqNum`, `timestamp`, `payloadType`, `getPayload`, `getPacket`, to extract such information and it is the reserved of the current encoding method.

6 Evaluation

Throughout the procedural of this project, the overall results summarized the following achievements:

6.1 GUI Development

Using tkinter Python library, the project is able to implement a brilliant graphical user interface for the player. It is illustrated in the figure below.

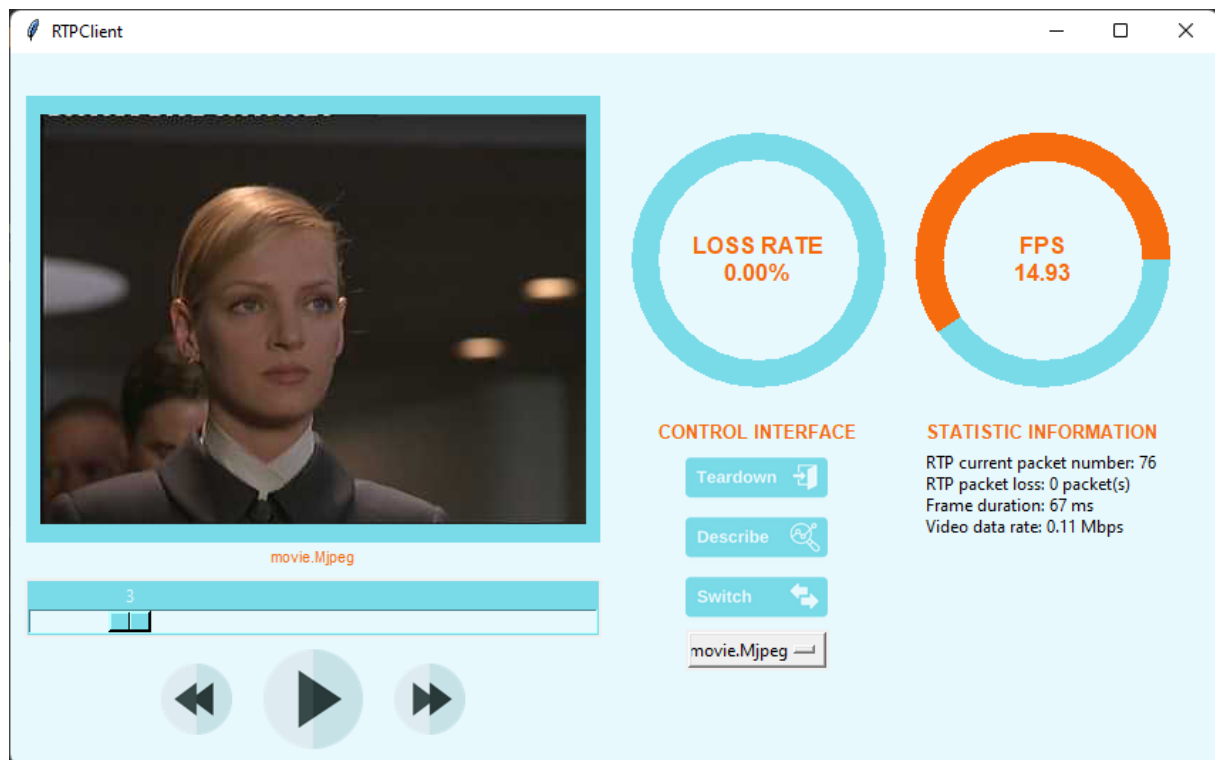


Figure 3: Media player user interface.

The basic controls for this media player includes PLAY, PAUSE, and TEARDOWN buttons. For grand extensions of the player, commonly used buttons such as FORWARD, BACKWARD, DESCRIBE, SWITCH VIDEOS are introduced into the player design.

Viewing video information, there are supportive labels that display statistical information of the currently streaming video. The displayed REMAINING time is transformed into a functional seekbar that user can slide to advance or go back to any point in the video.

RTSP transferring status, loss package percentage rate and current video frame-per-second is displayed intuitively via a guage-like meter. The color theme chosen for the player is a subtle light blue theme that makes the user-interface more attractive.

6.2 Client-server modeling

Design and implement a client-server model, with the server made online on 1 static port (larger than PORT 1024) and connected from client to watch video only any of client's port. The projects thoroughly laid out specifications of each protocol via RFC documentation, namely RFC

2326 (RTSP), RFC 1889 (RTP). These enables us to know the syntax of sending information to RTSP Socket because it needs to send correct information to the server to know what protocol to decode the video.

In this project, the decode is defined in UDP header format as 12 bytes, that gives us the payload after the 12th byte. When we extract such payload, we will be able to see the video streaming.

Due to the nature of multiple protocols in the modern world, it is essential to know about protocol type so we know where to extract our content from the header.

6.3 RTP fields and characteristics

We are able to discern each RTP field from the RTP header, the importance of sequence number to calculate packet loss rate, payload length to calculate video data rate. These are used to implement FORWARD and BACKWARD functionalities.

6.4 Twiddling the bits

We know how to shift the bits one step at a time to extract data from the header. Knowing exactly all the state of the video (INIT, READY, and PLAYING) depends on the current status of the user preference.

7 User manual

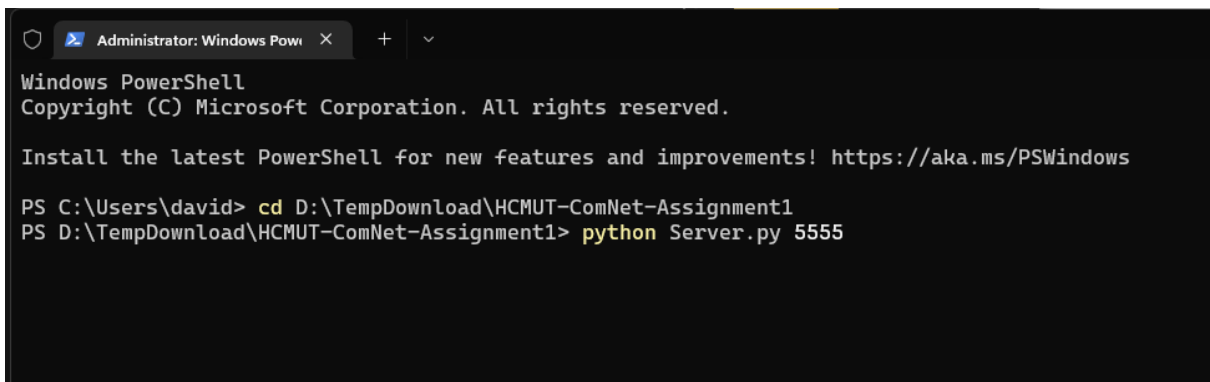
7.1 Running the player

To run the player, first go to the folder that contains the source code. Open up 2 separate terminals as follow:

- The first terminal with execute the following command:

```
python Server.py <server-port>,
```

in which <server-port> is the port that server shall listen to incoming RTSP signals. The standard RTSP port is 554, however ports larger than 1024 is preferred.



```
Administrator: Windows Powe...
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\david> cd D:\TempDownload\HCMUT-ComNet-Assignment1
PS D:\TempDownload\HCMUT-ComNet-Assignment1> python Server.py 5555
```

Figure 4: Running Server.py

- The second terminal will run the following command:

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

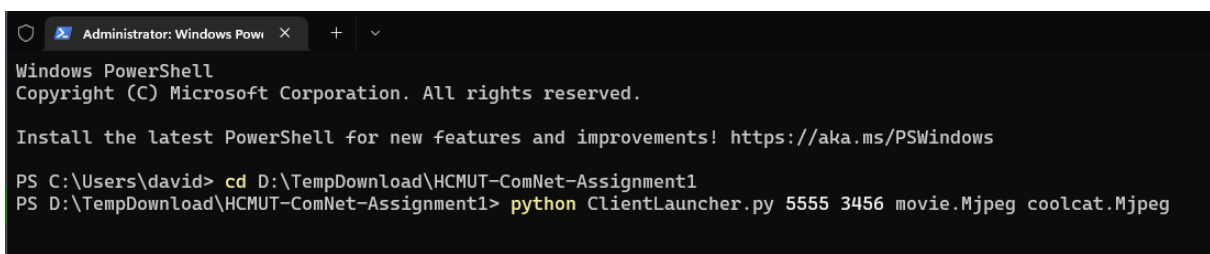
in which,

<server-host> is the IP address of the server where is it run. The localhost is 127.0.0.1

<server-port> is the listening RTSP port. If the server RTSP port is 5555, the RTP port is also 5555.

<RTP-port> port to receive incoming RTP. It is set to any positive integer number.

<video-file> is all the name of the .mjpeg files that the client would like to request to the server.



```
Administrator: Windows Powe...
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\david> cd D:\TempDownload\HCMUT-ComNet-Assignment1
PS D:\TempDownload\HCMUT-ComNet-Assignment1> python ClientLauncher.py 5555 3456 movie.Mjpeg coolcat.Mjpeg
```

Figure 5: Running ClientLauncher.py

7.2 After the GUI is run

- After successfully execute the above commands, the following GUI window will appear:

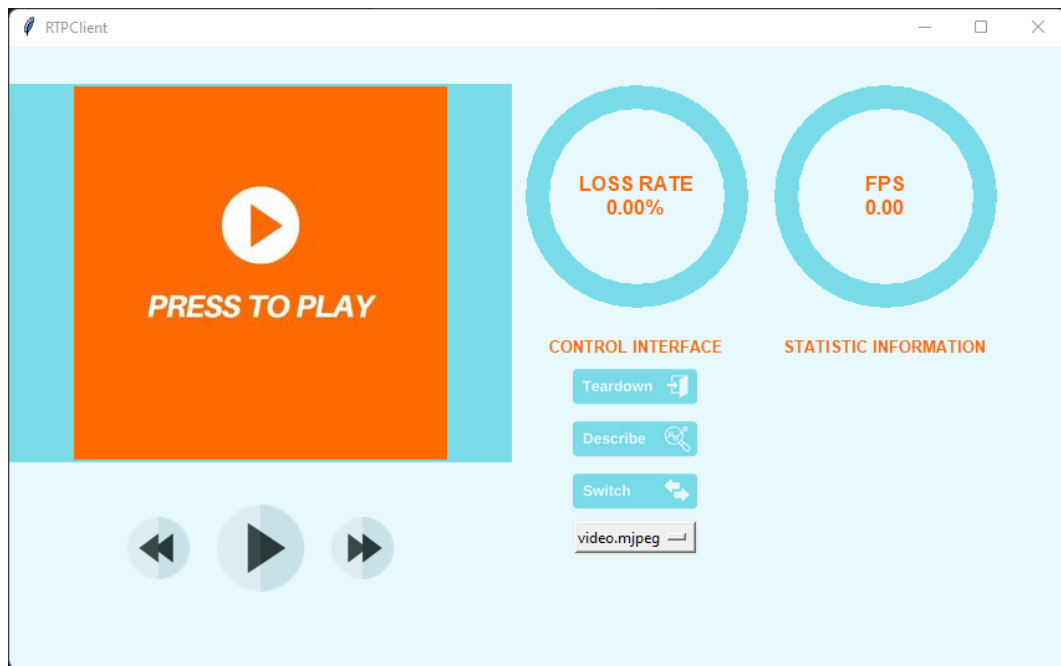


Figure 6: ClientLauncher.py user-interface

- The setup command is automatically done during startup of the player. To load in the preferred video, user click on the drop down list of videos and click SWITCH.
- To play the video, user clicks on the PLAY icon.

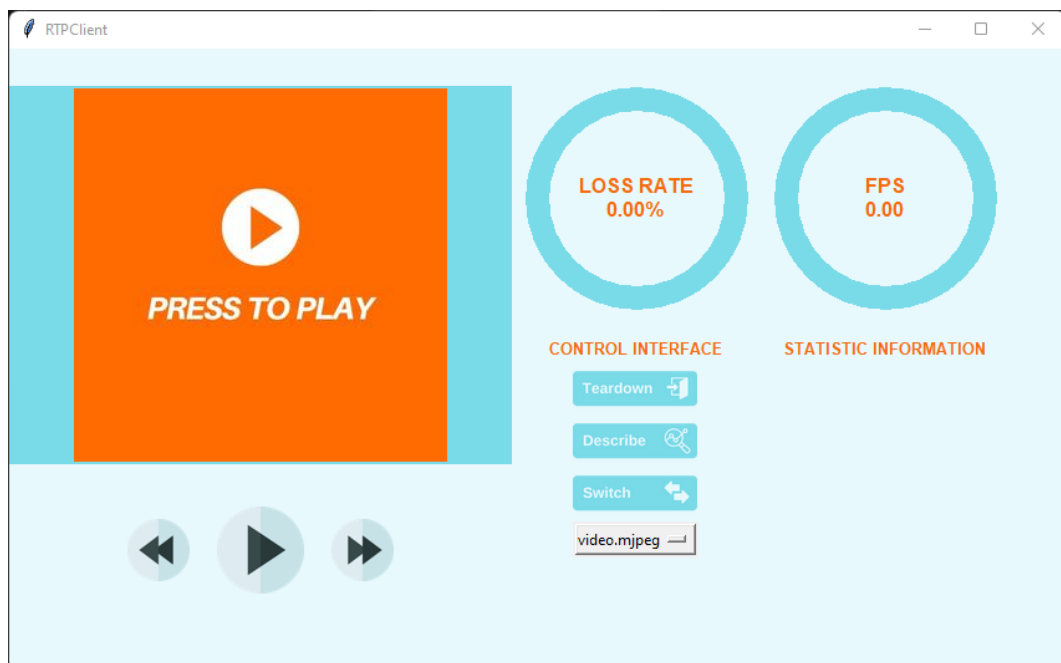


Figure 7: Playing the video

- To pause the video, user clicks on the PAUSE icon.

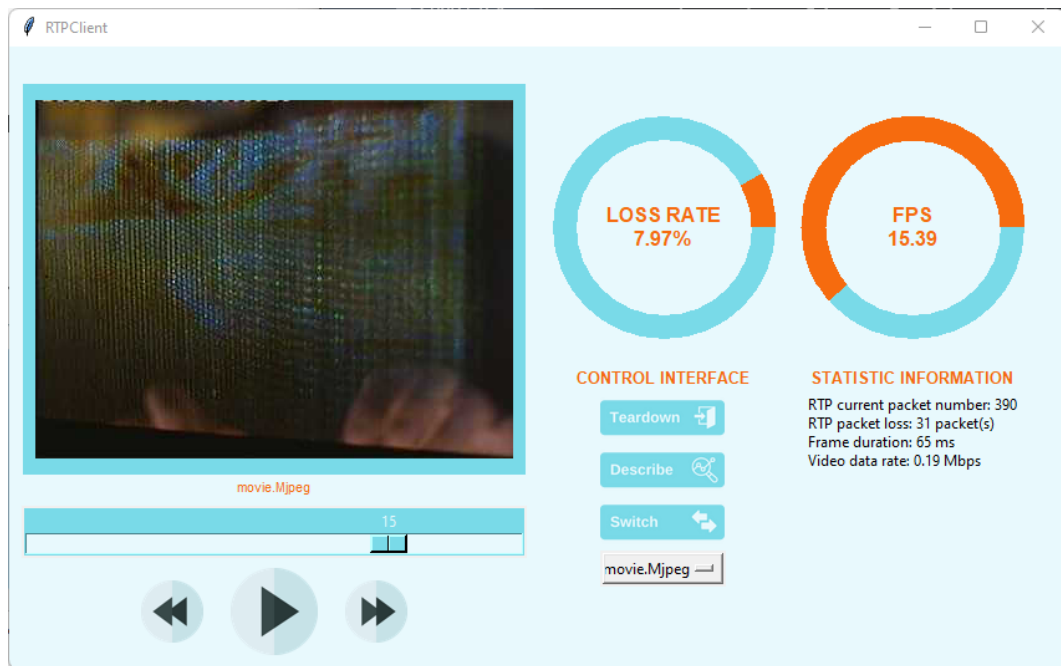


Figure 8: Pausing the video

- User can use the seekbar and drag the cursor up until the point they want to watch.
- To see current video description, user hit the DESCRIBE button.
- To fast forward or go backward, user hit the FORWARD or BACKWARD button on the sides of the PLAY/PAUSE button.
- To exit, user either press the BUTTON button to close the player or click on the X at the top right corner to exit.

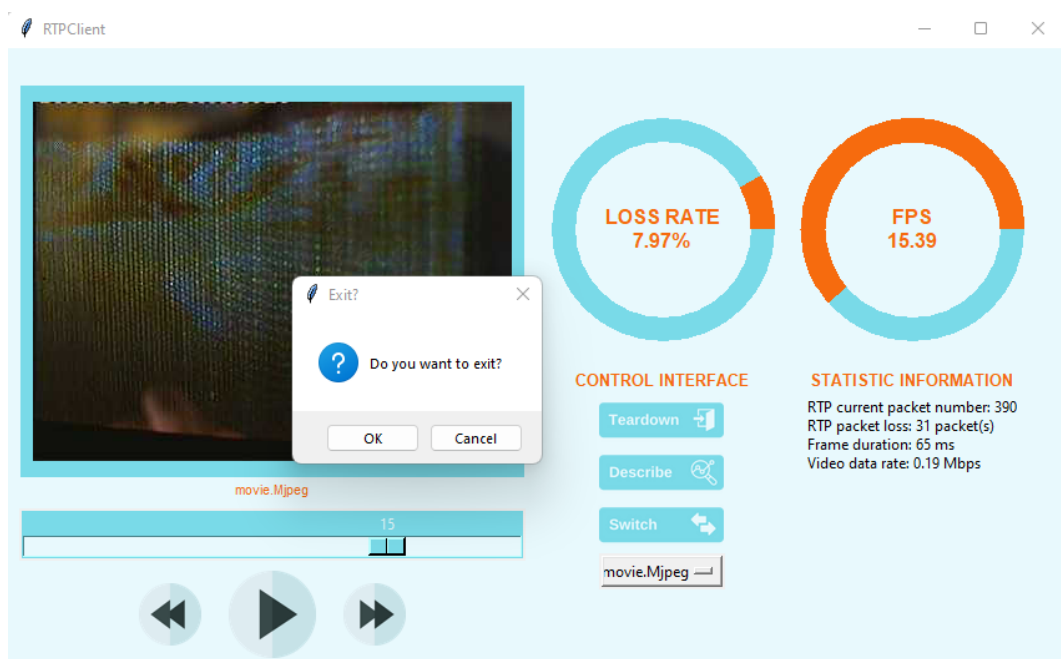


Figure 9: Exiting the player

8 Player extensions implementation

8.1 Calculate the session statistics

8.1.1 RTP packet lost rate

To identify the RTP packet loss rate, we can either use RTP Stream Analysis function from Wireshark or based on the sequence number from the client side to determine the rate - in this case, if the current sequence number (CQN) does not increase in order, this proves that a packet has been lost. In this project, we will show how to implement the second option.

In the `ServerWorker.py` file, every 0.05 seconds, the server will send exactly 1 packet. In turn, there is a total of 20 sent packets. During unstable Internet activity, only around 15 - 16 packets are sent, the rest are lost. Therefore, it cannot be determined by just counting all 20 packets in a single second.

To solve this, we assign an extra counter variable, initialize with 0 value. This variable will increase as `listenRTP` is called and caught the sent data.

The rate is calculated by the counter variable divided by the CQN (the `maxframeNbr = 500`) when the following code lines are called.

```
1 current = time.time()
2 duration = current - self.prevFrameTime
3 self.prevFrameTime = current
4 speed = len(rtpPacket.getPayload()) / duration
5 fps = 1 / duration
6 lossRate = float(self.lossCounter / self.frameNbr) * 100 if self.
    frameNbr != 0 else 0
7 self.progressbar1.step(fps / 25, 1, fps)
8 self.progressbar0.step(lossRate / 100, 0)
9
10 # Display info to the label
11 self.displayText = StringVar()
12
13 statsInfo = self.displayText.get()
14 statsInfo += 'RTP current packet number: ' + str(currFrameNbr) +
    '\n'
15 statsInfo += 'RTP packet loss: ' + str(self.lossCounter) + '
    packet(s)\n'
16 statsInfo += 'Frame duration: {:.0f} ms\n'.format(duration *
    1000)
17 statsInfo += 'Video data rate: {:.2f}'.format(speed / 1e+6) + '
    Mbps\n'
18
19 self.statsLabel.configure(textvariable=self.displayText, justify=
    LEFT)
```

8.1.2 Video data rate

Regarding video data rate, we will assert it to be as the overall storage data of *payload* (which is the data minus the bytes of the header) divided to the overall elapsed time of the video.

To do this, we'll use the `time` library from Python and initiate three variables called `timerBegin`, `timerEnd` and `timer` for the running lapse. It is calculated as follow:

- `timerBegin` is set as soon as user presses PLAY.
- `timerEnd` is set when PAUSE is pressed. The timer is then calculated as this point with `timer = timerEnd - timerBegin`.
- As one video can be paused many times, the `timer` variable is an accumulative sum. This is true for the total payload whenever `listenRtp()` function is called and caught the data.

The following shows a sample result of such calculation.

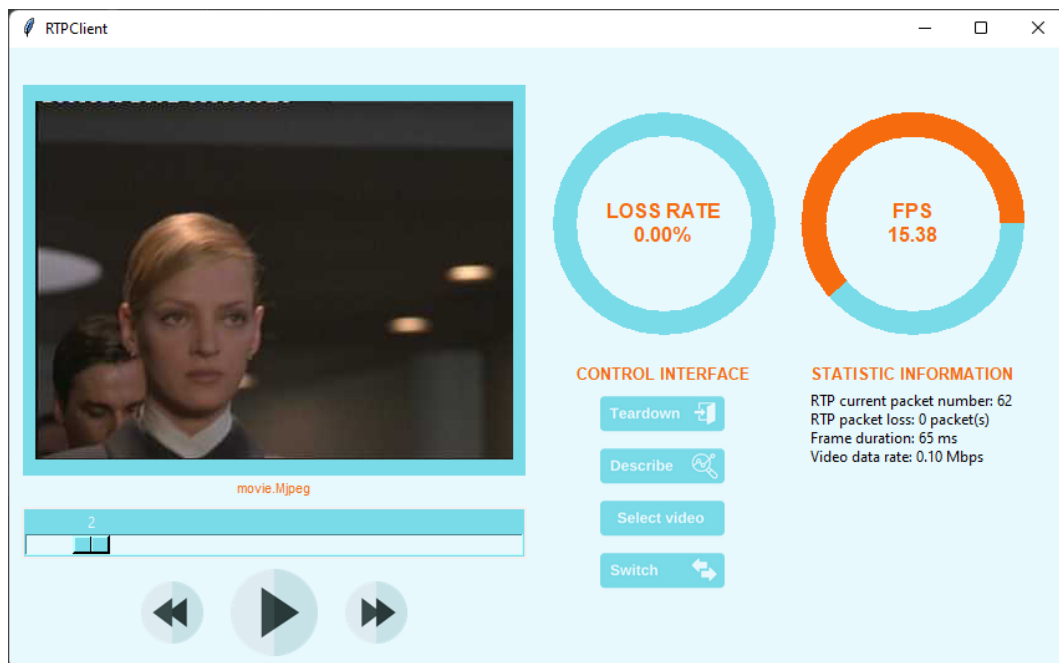


Figure 10: Player statistics

8.2 Implement the PLAY, PAUSE and STOP buttons

8.2.1 Basic implementation

There are two main ways to initialize the RTSP SETUP essential method. The first way is to trigger it as soon as the GUI is launched. The second way would be when the user presses the PLAY/PAUSE button for the first time or after the final STOP. This project will implement the second method.

The entire code is based on this state machine diagram.

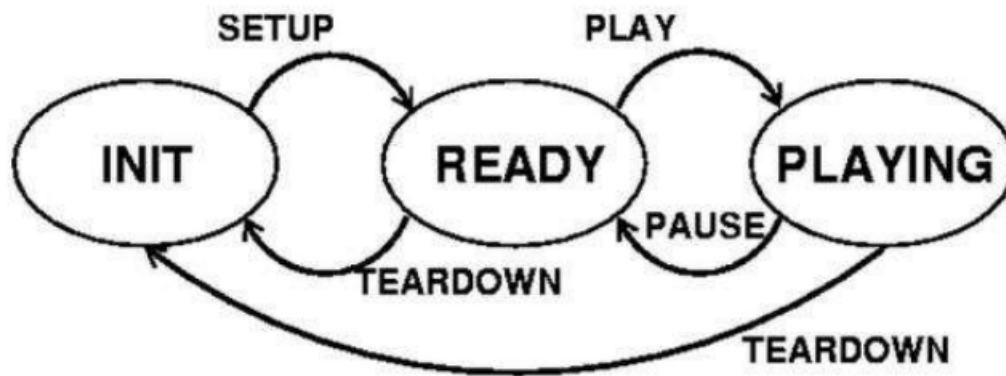


Figure 11: Player state diagram

For the STOP method, based on the listed diagram, the player will move back to the INIT state and reset all the statistical variables. After that, when we press PLAY, the program SETUPS and streams the video. The ways we can exit is to TEARDOWN or click the X at the top right corner of the window.

The following summarizes the basic functionalities:

- The program runs on command execution.
- User presses PLAY, the program SETUPS and streams the video.
- User presses PAUSE, video pauses, all related statistical information from the PLAY onward is displayed.
- User presses STOP, video stops, player reverts back to INITIALIZATION. TEARDOWN is called and the player is ready for SETUP.
- User presses PLAY, player re-SETUPS and play as normal.
- Finally, user presses the X at the top right corner of the window, program asks for confirmation and close down if user says yes.

8.2.2 TEARDOWN when user clicks on STOP button

It is not entirely correct to call TEARDOWN after stopping, as viewers have the tendency to re-watch the video after they presses STOP. This, in turn, makes the disconnection of the socket to the server a boggle as the player have to resetup the entire thing whenever the user chooses to PLAY. However, statistically speaking, the tendency of users actually do the actions above are not high, so the implementation is plausible but not entirely stable.

In this project, in order to remain highest stability and smooth operation of the player, the implemented source code includes the socket disconnection as user privileged method. This is incorporated inside the STOP and PLAY methods. Because of this, the action above is considered as a connection maintaining procedure.

8.3 Implement the DESCRIBE method

The DESCRIBE event is sent once the user has successfully SETUP the player or the program is in the READY state or PLAYING state. Once the server successfully received the DESCRIBE request via RTSP, the following code fragment in `sendRtspRequest(self, requestCode)` function in `ServerWorker.py` is executed to handle such requests.

```
1 # Code omitted...
2
3 elif requestCode == self.DESCRIBE and not self.state == self.INIT
4 :
5     self.rtspSeq += 1
6
7     # RTSP request and send to the server.
8     request = "DESCRIBE " + str(self.fileName) + " RTSP/1.0\n" +
9     "CSeq: " + str(self.rtspSeq) + "\n" + "Session: " + str(self.
10     sessionId)
11
12     self.requestSent = self.DESCRIBE
```

The description is print out on the *STATISTICS INFORMATION* section of the player via the following code fragment

```
1 # Create Describe button
2 describe = Image.open("2.png")
3 self.describeImage = ImageTk.PhotoImage(describe)
4 self.describeButton = Button(self.controlFrame,
5     image=self.describeImage,
6     command=self.describeMovie,
7     borderwidth=0,
8     border=0,
9     bg=LIGHT_BLUE)
10 self.describeButton.grid(row=2, column=0, padx=10)
```

The default port for RTSP communication is 554 and this is used for both UDP and TCP transmission as well. The receiving message to the client is in a Session Description Protocol format (SDP), which is used for describing transmissions protocol of multimedia players. The client handles this message and display relevant information.

The describing information is shown in the following graphic and contains the following information:

- version = 2, this is the 2nd version of the communication protocol.
- type = Video movie.Mjpeg RTP/Mjpeg, this is type of the media protocol and the transport address. The format of SDP type includes the video, the RTPport, the protocol RTP/AVP and the file format mjpeg.

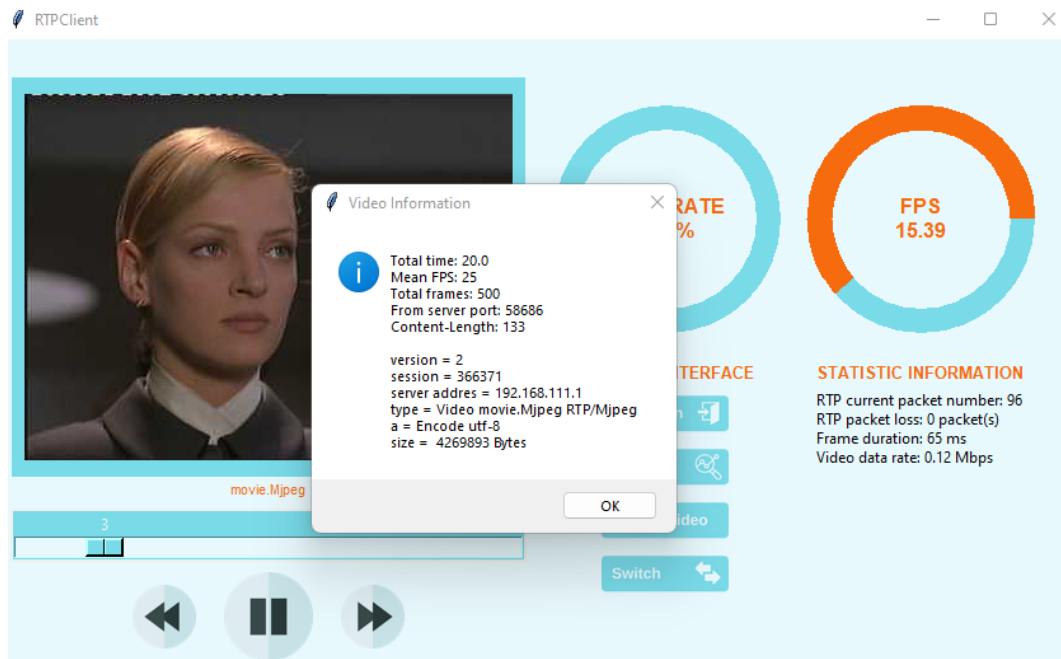


Figure 12: Video information

- a = Encode utf-8 the encoding format of the video
- session = 366371 is the session id of the video stream.
- size = 4269893 Bytes is the size of the video stream.

8.4 Additional functions for user interface such as: display video total time and remaining time, fast forward or backward video and a seek bar

In this final extension phrase, the player is implemented with an elapsed time and total time label for the video, the forward and backward with 1 second advance and a seekbar.

8.4.1 Total time and Remaining Time

Whenever the user initialize the player, RTSP Request will be sent to the server. After that, while calling `replyRtsp()`, the `ServerWorker.py` also calls `replySetup()` in which the total time of the video is received in the `updateCountDownTimer(self)` below.

```

1 def updateCountDownTimer(self):
2     remainingTime = (self.noFrames - self.frameNbr) / (self.
3         fps)
4     print(remainingTime)
5     self.slideVideoTime.set(self.videoTime - remainingTime)
6     self.currentVideoTime = self.videoTime - remainingTime
7     self.statsTimeLabel.config(text=str(remainingTime))

```

This function calculates the relative timing by using the total number of frame of the video and multiply with the time between the server sent the pack to the client two times. This way of calculating the total time shall have slight inaccuracy with the real total elapsed time of the video, due to Internet connection issues, delays. This, in turn, makes the time the packet reaches

the client have delays from when it leaves the server. There is certainly no way to predict such situation when you stream a video, so this is the most viable way.

8.4.2 Fast forward and backward video

In this project the buttons are implemented besides the pause button. When the user presses the forward button, it will send the an RTSP request with a parameter `self.FORWARD`. When the user presses the backward button, it will send the an RTSP request with a parameter `self.BACKWARD`

```
1 def forwardMovie(self):
2     """Forward button handler."""
3     if self.state == self.PLAYING:
4         self.sendRtspRequest(self.FORWARD) #send forward request
5
6 def backwardMovie(self):
7     """Backward button handler."""
8     if self.state == self.PLAYING:
9         self.sendRtspRequest(self.BACKWARD) #send backward
        request
```

For the server part, once the request is received, the data pointer of the nextFrame will return the next frame of the video in the order of 1 second later frame of the video, if the next 1 second duration of frame is not possible to advance, the pointer will go to the last frame segment.

For the prevFrame, the function will bring the data pointer back to the beginning of the video and perform advances until it reaches the 1 second previous frame of the current frame. And when it cannot reach the first 1 second, the pointer will go to the beginning of the video.

8.4.3 Seekbar

A seekbar is implemented as a type of track bar in the media player so that the user can use it can scroll to any point of the video. The following code shows the implemenation of the seekbar.

```
1 self.videoTime = (self.noFrames) / (self.fps)
2 self.sli1 = Scale(self.sliderFrame, from_=0, to=self.videoTime,
3     orient=HORIZONTAL,
4     length=400,
5     command=self.sliding,
6     variable=self.slideVideoTime,
7     bg = LESS_LI_BLUE,
8     activebackground=ORANGE,
9     troughcolor=LIGHT_BLUE,
10    bd=1,
11    fg=LIGHT_BLUE)
12 if self.slideFlag == False:
13     self.sli1.pack()
14     self.slideFlag = True
```

Whenever a user slides the seekbar up, the implementation is continuously calling the FORWARD method until it reaches the point on the seekbar, only then the desired frame shall be displayed on the player. The same implementation is correct for sliding back by continuously calling BACKWARD method. The following code shows how it is done.

```
1 def sliding(self, arg):
2     if (self.currentVideoTime < self.slideVideoTime.get()):
3         self.forwardMovie()
4     if (self.currentVideoTime > self.slideVideoTime.get()):
5         self.backwardMovie()
```

It should be noted that when advancing using the slider for the first time, the frame loss rate dramatically increases due to the current video frames not yet fetched from the server. This is normal behavior and whenever the server completes its transaction, the frame loss rate is reduced.

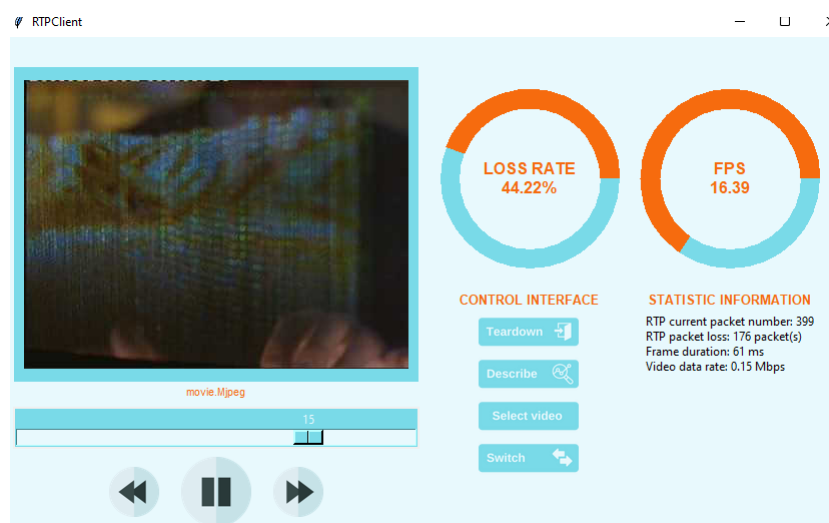


Figure 13: High frame rate loss due to insufficient data from seeking.

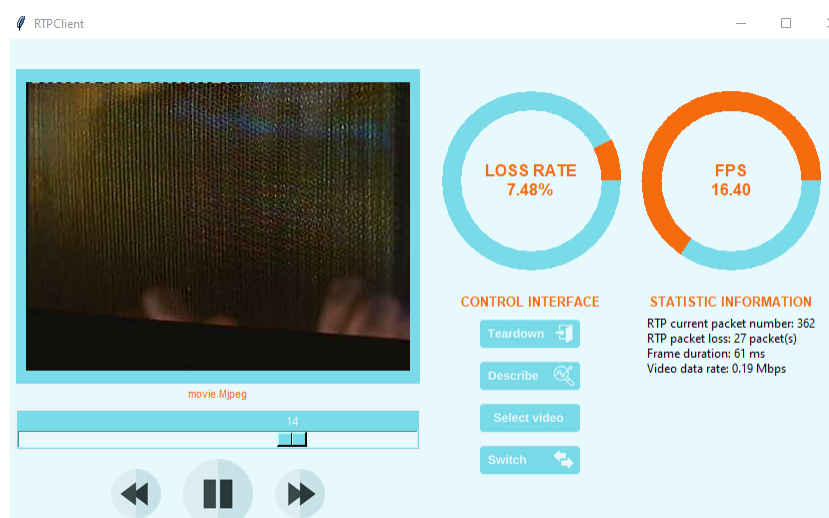


Figure 14: Frame rate recovered afterwards.

- End of report -