VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF TECHNOLOGY

Computer Network 211

ASSIGNMENT 1: VIDEO STREAMING

Lecturer:	Pham Tran Vu	
	Vu Van Tien	
Class:	L03	
Student:	Vo Hong Phuc	1911881
	Vo Anh Nguyen	1914405
	Nguyen Trung Kien	1911441
	Nguyen Truong Hai Dang	1911044

Ho Chi Minh, October 2021

Contents

1 Problems analysis:

Since the network connected the world, information and media have started spreading the news for human. Video streaming is the demand of people to transfer video from computer to computer. To reduce the file stored in the destination computer, developer uses the Real-time Streaming Protocol and Real-time Transfer Protocol to stream the video. By that method, video is send/received part by part and allow users to watch once the mini-video is received.

1.1 Fundamental Knowledge:

1.1.1 Client-server model

The Client-server model is a distributed application structure that partitions task or workload between the providers of a resource or service, called servers, and service requesters called clients.
In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and deliver the data packets requested back to the client.
Example: Email, World Wide Web, etc.

1.1.2 Real Time Streaming Protocol

The Real Time Streaming Protocol (RTSP) is a network control protocol designed for use in entertainment and communications systems to control streaming media servers.
The protocol is used for establishing and controlling media sessions between endpoints (play, record, pause, teardown, ...).

1.1.3 Real-time Transfer Protocol/Real-time Transport Protocol

The Real-time Transport Protocol (RTP) is a network protocol for delivering audio and video over IP networks.
RTP is used in communication and entertainment systems that involve streaming media, such as telephony, video teleconference applications including WebRTC, television services and web-based push-to-talk features.

1.1.4 Transmission Control Protocol:

Transmission Control Protocol (TCP) is a standard that defines how to establish and maintain a network conversation by which applications can exchange data.

Once the TCP connection is established, it maintained until the applications at each end have finished exchanging messages.

Action:

- Break the data into packets to deliver
- Send and accept packets
- Check for error in transimision

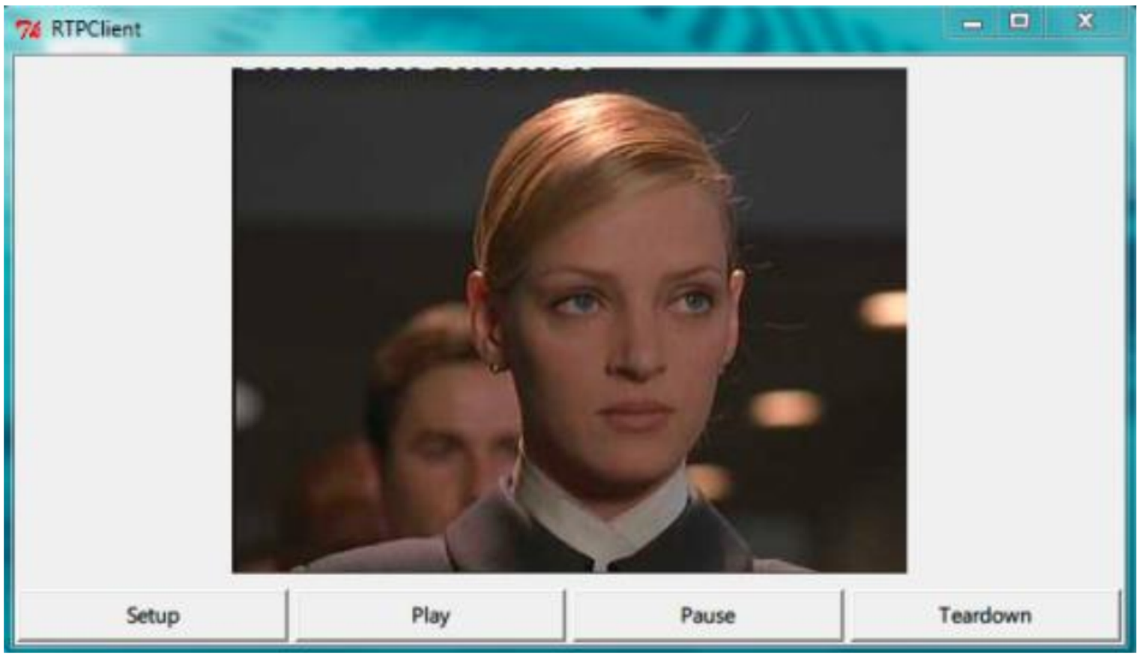
1.1.5 User Datagram Protocol

The User Datagram Protocol (UDP) is a lightweight data transport protocol that works on top of IP.
UDP provides a mechanism to detect corrupt data in packets, but it does not attempt to solve other problems that arise with packets, such as lost or out of order packets. That's why UDP is sometimes known as the Unreliable Data Protocol.

1.2 Requirement:

2 Design system:

2.1 Target design:



- Button Setup
- Button Play
- Button Pause
- Button Teardown
- Button Exit

2.2 List of function:

- ConnectToServer
- sendRtspRequest
- openPtpPort
- recvRtspReply
- parseRtspReply
- openRtpPort

2.3 Component List:

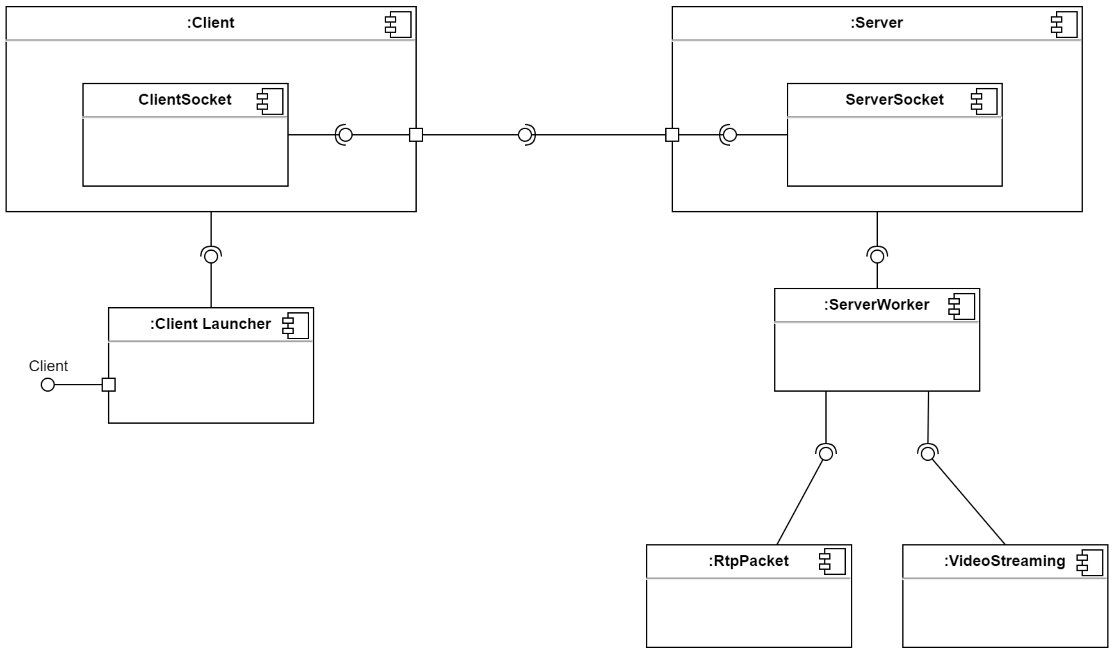
Component server includes:

- Server
- Server Worker
- Video Streaming
- Server Socket

Component client includes:

- Client
- Client Socket
- Client Launcher

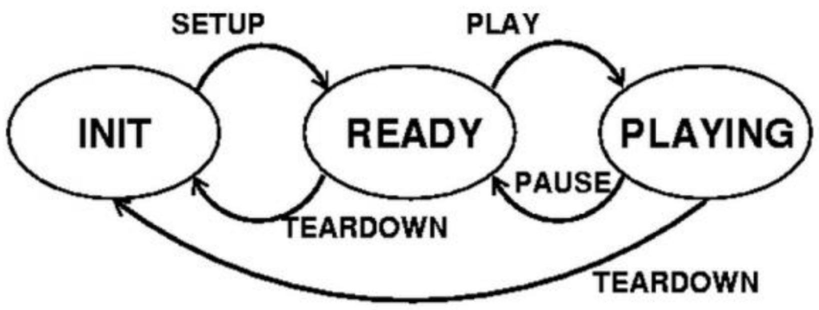
Component RtpPacket



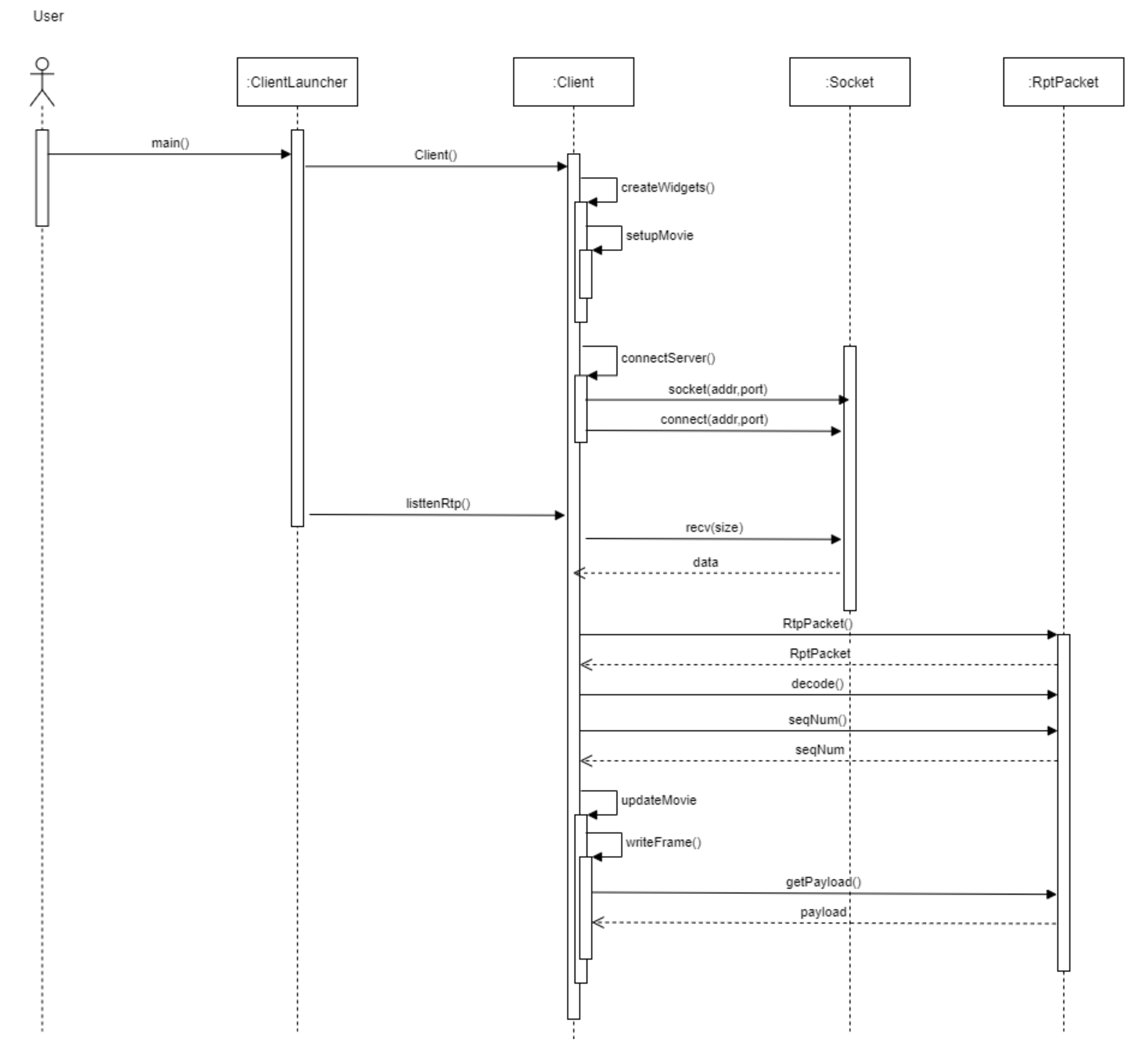
Component diagram

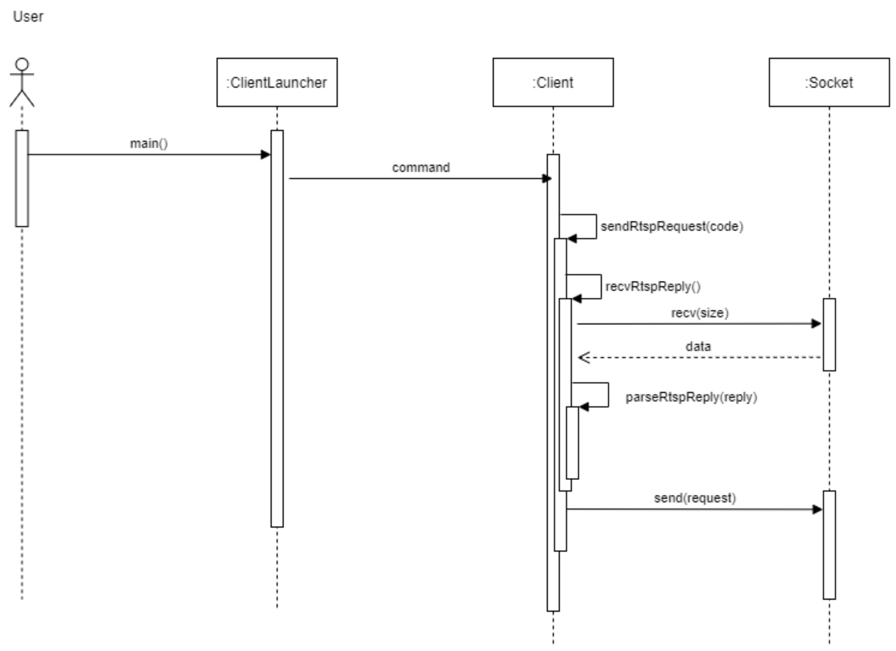
2.4 Model and data flow:

2.4.1 User flow:

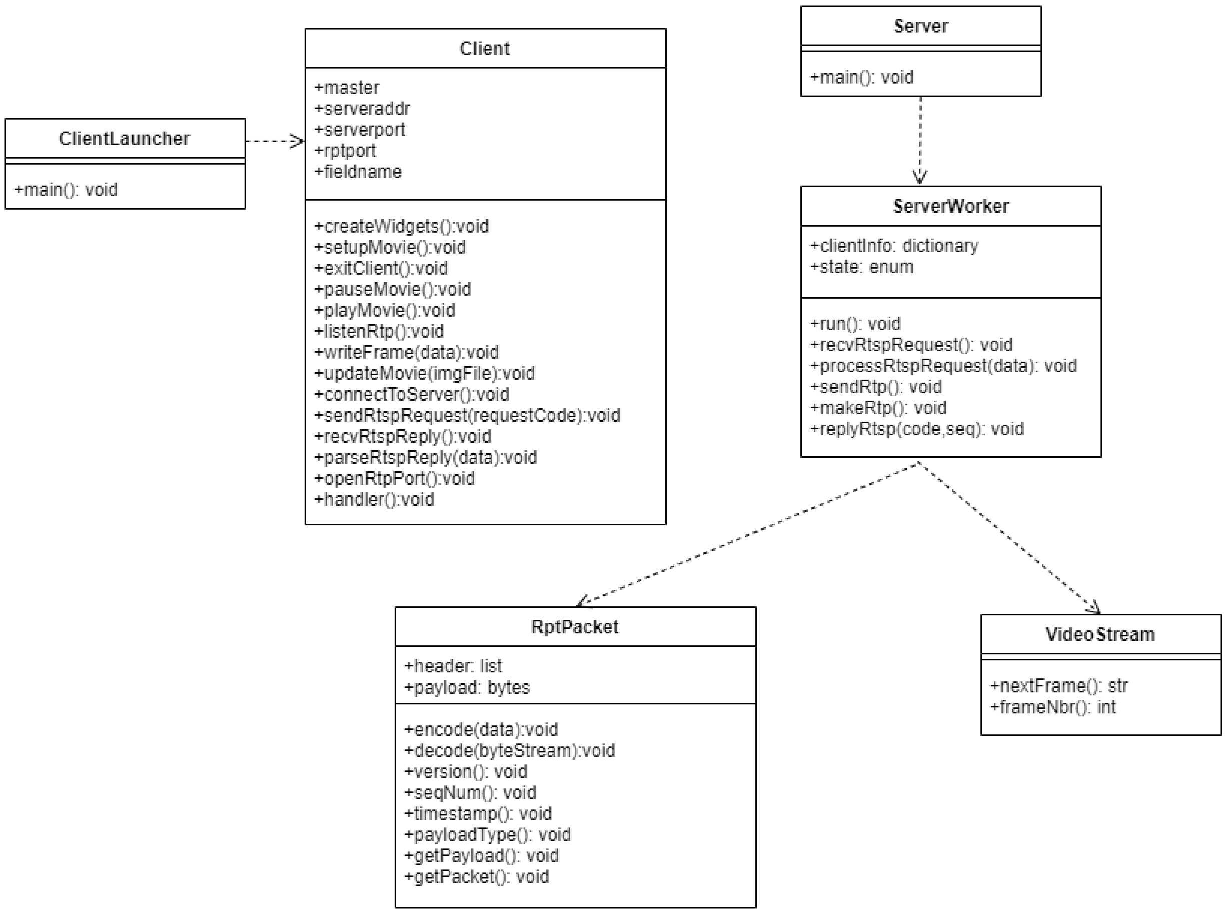


2.4.2 Sequence diagram:





2.4.3 Class diagram:



3 Function description

In this section, we will describe the major component implementation we have completed, including RTSP client side and RTP encoding function.

3.1 RTP/RTSP Client

`connectToServer()`: This function creates a RTSP socket so that both client and server can communicate to each other. This socket is a TCP socket, which is used to generate RTSP requests to control the video streaming session.

```python
def connectToServer(self):
    """Connect to the Server. Start a new RTSP/TCP session."""
    self.rtspSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #https://www.youtube.com/watch?v=7-O7yeO3hNQ
    #this is client socket in youtube

    try:
        #this is ip and port for connect, input a tuple
        self.rtspSocket.connect((self.serverAddr, self.serverPort))
    except:
        tkinter.messagebox.showwarning('Connection Failed',
        'Connection to \'%s\' failed.' %self.serverAddr)
```

`setUpMovie()`: This function will set the initial state of client to INIT and also set up the ”setup” button handler.

```python
def setupMovie(self):
    """Setup button handler."""
    if self.state == self.INIT:
        self.sendRtspRequest(self.SETUP)
```

The group of functions `exitClient()`, `pauseMovie()` and `playMovie()` handles three other buttons: Exit, pause and play respectively. They send appropriate request to the server. In addition:

- `exitClient()`: The client destroys the cache image for the video.
- `playMovie()`: It creates a new thread to listen for RTP packets from the server.

```python
def exitClient(self):
    """Teardown button handler."""
    self.sendRtspRequest(self.TEARDOWN)
    self.master.destroy() # Close the gui window
    os.remove(CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT)
    # Delete the cache image from video

def pauseMovie(self):
    """Pause button handler."""
    if self.state == self.PLAYING:
        self.sendRtspRequest(self.PAUSE)

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)
```

`sendRtspRequest()`: This function will send the RTSP request to the server. It receives a *requestCode* parameter, which identifies the event of the users. It saves message with a fixed format so that the server can access and response accordingly. In case of SETUP request, the function opens a new thread to receive replies from the server.

```python
def sendRtspRequest(self, requestCode):
"""Send RTSP request to the server."""
#-------------
# TO COMPLETE
#-------------
# TODO TO UNDERSTAND
# Setup request
if requestCode == self.SETUP and self.state == self.INIT:
    threading.Thread(target=self.recvRtspReply).start()
    # Update RTSP sequence number.
    self.rtspSeq += 1

    # Write the RTSP request to be sent.
    request = 'SETUP ' + self.fileName
    + ' RTSP/1.0\nCSeq: ' + str(self.rtspSeq) + '\nTransport: RTP/UDP; client_port= '
    + str(self.rtpPort)

    # Keep track of the sent request.
    self.requestSent = self.SETUP

# Play request
#save the request so the server can access and know what to do
elif requestCode == self.PLAY and self.state == self.READY:
    self.rtspSeq += 1
    request = 'PLAY ' + self.fileName + ' RTSP/1.0\nCSeq: ' + str(self.rtspSeq)
    + '\nSession: ' + str(self.sessionId)
    self.requestSent = self.PLAY

# Pause request
elif requestCode == self.PAUSE and self.state == self.PLAYING:
    self.rtspSeq += 1
    request = 'PAUSE ' + self.fileName + ' RTSP/1.0\nCSeq: ' + str(self.rtspSeq)
    + '\nSession: ' + str(self.sessionId)
    self.requestSent = self.PAUSE

# Teardown request
elif requestCode == self.TEARDOWN and not self.state == self.INIT:
    self.rtspSeq += 1
    request = 'TEARDOWN ' + self.fileName + ' RTSP/1.0\nCSeq: ' + str(self.rtspSeq)
    + '\nSession: ' + str(self.sessionId)
    self.requestSent = self.TEARDOWN
else:
    return
```

`OpenRtpPort()`: This function creates a RTP socket, which is a UDP socket, to receive RTP packets from the server that holds the media content.

```
def openRtpPort(self):
    """Open RTP socket binded to a specified port."""
    #-------------
    # TO COMPLETE
    #-------------
    # Create a new datagram socket to receive RTP packets from the server
    self.rtpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Set the timeout value of the socket to 0.5sec
    self.rtpSocket.settimeout(0.5)

    try:
        # Bind the socket to the address using the RTP port given by the client user
        self.rtpSocket.bind(("", self.rtpPort))
    except:
        tkinter.messagebox.showwarning
        ('Unable to Bind', 'Unable to bind PORT=%d' % self.rtpPort)
```

`recvRtspReply()`: This functions receives RTPS reply message from the server. The message is then decoded and parsed to process. There are several remarkable points here:

- If the requestSent is SETUP then a RTP open is created to receive RTP packet from the server.
- In the case user presses the TEARDOWN button, the teardownAcked flag is set to 1 to order to indicate the client to close the socket

```python
def recvRtspReply(self):
    """Receive RTSP reply from the server."""
    while True:
        reply = self.rtspSocket.recv(1024)

        if reply: #RTSPSOCKET keep adding the new data
            self.parseRtspReply(reply.decode("utf-8"))

        # Close the RTSP socket upon requesting Teardown
        if self.requestSent == self.TEARDOWN:
            self.rtspSocket.shutdown(socket.SHUT_RDWR)
            self.rtspSocket.close()
            break
```

`parseRtspReply()`: Parse the RTSP reply from the server.

```python
def parseRtspReply(self, data):
    """Parse the RTSP reply from the server."""
    lines = data.split('\n')
    seqNum = int(lines[1].split(' ')[1])

    # Process only if the server reply's sequence number is the same as the request's
    # match the request and reponse seq like lab wireshark
    if seqNum == self.rtspSeq:
        session = int(lines[2].split(' ')[1])
        # New RTSP session ID can assign to any other session
        if self.sessionId == 0:
            self.sessionId = session

        # Process only if the session ID is the same
        if self.sessionId == session:
            if int(lines[0].split(' ')[1]) == 200:
                #OK CODE
                if self.requestSent == self.SETUP:
                    #-------------
                    # TO COMPLETE
                    #-------------
                    # Update RTSP state.
                    self.state = self.READY
                    # Open RTP port.
                    self.openRtpPort()
                    #set time out for session
                elif self.requestSent == self.PLAY:
                    self.state = self.PLAYING
                elif self.requestSent == self.PAUSE:
                    self.state = self.READY
                    # The play thread exits.
                    # A new thread is created on resume.
                    self.playEvent.set()
                elif self.requestSent == self.TEARDOWN:
                    self.state = self.INIT
                    # Flag the teardownAcked to close the socket.
                    self.teardownAcked = 1
            #else: pass
```

`handler()`: Handler will be invoked when user chooses to exit. First it have to be paused according to the state diagram.

```python
def handler(self):
        """Handler on explicitly closing the GUI window."""
        self.pauseMovie()
        if tkinter.messagebox.askokcancel("Quit?", "Are you sure you want to quit?"):
                self.exitClient()
        else: # When the user presses cancel, resume playing.
                self.playMovie()
```

3.2 RtpPacket

`encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload)`: Encode is the processing function to get the information. Based on the agreement between client-server, the assigned bits of information are concatenate as 1 RtpPacket.

Both the client and server has the ability to access the encode function to undestand the content inside.

In the requirement, we set these parameter as follow:

- Set the RTP-version field (V) to 2.
- Set padding (P), extension (X), number of contributing sources (CC), and marker (M) fields to 0.
- Set payload type field (PT) to 26.
- Set the sequence number as the frameNbr argument
- Set the timestamp
- Set the source identifier (SSRC) randomly.
- Set field CC = 0, CSRC = 0
- The length of the packet header is 12 bytes, equals the first three lines.

In addition, we use the technique ”Twiddling the Bits” to create field by field in the RtpPacket. Finaalt, we got the header as a list of bits that every bits carry a their special meanings.

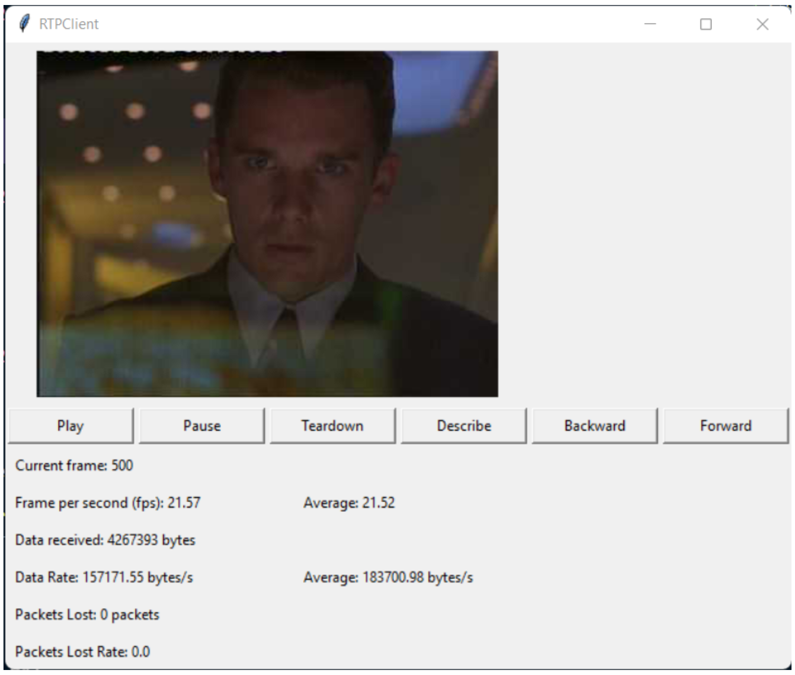
```python
def encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload):
                """Encode the RTP packet with header fields and payload."""
                timestamp = int(time())
                header = bytearray(HEADER_SIZE)
                # Fill the header bytearray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

```
            header[0] = header[0] | version << 6;
            header[0] = header[0] | padding << 5;
            header[0] = header[0] | extension << 4;
            header[0] = header[0] | cc;
            header[1] = header[1] | marker << 7;
            header[1] = header[1] | pt;
            header[2] = (seqnum >> 8) & 0xFF;
            header[3] = seqnum & 0xFF;
            header[4] = (timestamp >> 24) & 0xFF;
            header[5] = (timestamp >> 16) & 0xFF;
            header[6] = (timestamp >> 8) & 0xFF;
            header[7] = timestamp & 0xFF;
            header[8] = (ssrc >> 24) & 0xFF;
            header[9] = (ssrc >> 16) & 0xFF;
            header[10] = (ssrc >> 8) & 0xFF;
            header[11] = ssrc & 0xFF

            self.header = header

            # Get the payload
            # ...
            self.payload = payload

```

4 Extension

4.1 Calculate RTP packet loss rate, video data rate, frame rate

My team calculated some values: RTP packet loss rate, Frame rate, Data rate



Current frame: 500

Frame per second (fps): 21.57 Average: 21.52

Data received: 4267393 bytes

Data Rate: 157171.55 bytes/s Average: 183700.98 bytes/s

Packets Lost: 0 packets

Packets Lost Rate: 0.0

To find the above values, we edited the `Client.py` file.
First import ”time” module to handle various operations regarding time:

```
import time
```

In `__init__()`, we initiate some values: `expFrameNbr`, `statPacketsLost`, `statTotalBytes`, `statTotalPlayTime`, `startTime`, `totalFrames`.

```python
self.expFrameNbr = 0
self.statPacketsLost = 0
self.statTotalBytes = 0
self.statTotalPlayTime = 0
self.startTime = 0
self.totalFrames = 0
```

Create labels to display statistics about the session by adding this code to `createWidgets()`

```python
self.displays = []
for i in range(6):
    DLabel = Label(self.master, height=1)
    DLabel.grid(row=2 + i, column=0, columnspan=4,sticky=W,padx=5, pady=5)
    self.displays.append(DLabel)
```

In `listenRTP()`, by comparing expected frame number and gotten frame number, we can find the number of packets lost and Packets Lost Rate. Frame per second, Data received, Data Rate can be calculated via current time, start time and payload's length.

```python
def listenRtp(self):
    """Listen for RTP packets."""
    while True:
        try:
            data = self.rtpSocket.recv(20480)
            # this is packet recieve. 20480 is buffer size
            if data:
                rtpPacket = RtpPacket()
                rtpPacket.decode(data)

                # make the packet receive has the class type of Packet
                self.expFrameNbr += 1

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

                if currFrameNbr > self.frameNbr: # Discard the late packet
                    self.frameNbr = currFrameNbr
                    payload = rtpPacket.getPayload()
                    self.updateMovie(self.writeFrame(payload))
                    #this packet cop to cache
                    #ready to fast present
                    self.totalFrames += 1

                # Compare expected frame number and gotten frame number
                if self.expFrameNbr != currFrameNbr:
                    self.statPacketsLost += 1

                # Calculate total data received
                payload_length = len(payload)
                self.statTotalBytes += payload_length

                # Calculate total play time of the session
                curTime = time.time()
                self.statTotalPlayTime += curTime - self.startTime
                self.operationTime = curTime - self.startTime
                self.startTime = curTime

                # Display the statistics about the session
                self.displays[0]["text"] = 'Current frame: '
                  + str(currFrameNbr)
                self.displays[1]["text"] = 'Frame per second (fps): '
                  + str(format(1/self.operationTime,".2f")) + '\t\tAverage: '
                  + str(format(self.totalFrames/self.statTotalPlayTime,".2f"))
                self.displays[2]["text"] = 'Data received: '
                  + str(self.statTotalBytes) + ' bytes'
                self.displays[3]["text"] = 'Data Rate: '
                  + str(format(payload_length / self.operationTime,".2f"))
                  + ' bytes/s' + '\t\tAverage: '
                  + str(format(self.statTotalBytes / self.statTotalPlayTime,".2f"))
                  + ' bytes/s'
                self.displays[4]["text"] = 'Packets Lost: '
                  + str(self.statPacketsLost) + ' packets'
                self.displays[5]["text"] = 'Packets Lost Rate: '
                  + str(float(self.statPacketsLost / currFrameNbr))
                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
```

4.2 Update video player interface

First, we update media player interface by removing the SETUP button.

When a PLAY request is called at client side, the client's state is checked. If its value is INIT (0), a SETUP request is sent, after that a PLAY request is sent to the server.

```
def playMovie(self):
    """Play button handler."""
    wait = True
    if self.state == self.INIT:
        self.setupEvent = threading.Event()
        self.sendRtspRequest(self.SETUP)
        wait = self.setupEvent.wait(timeout=0.5)
    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)
```

4.3 Add DESCRIBE request

In `Client.py` we initiate the value for `DESCRIBE`. Also in this file, we add the function `describeSession()` to send `DESCRIBE` request to the server.

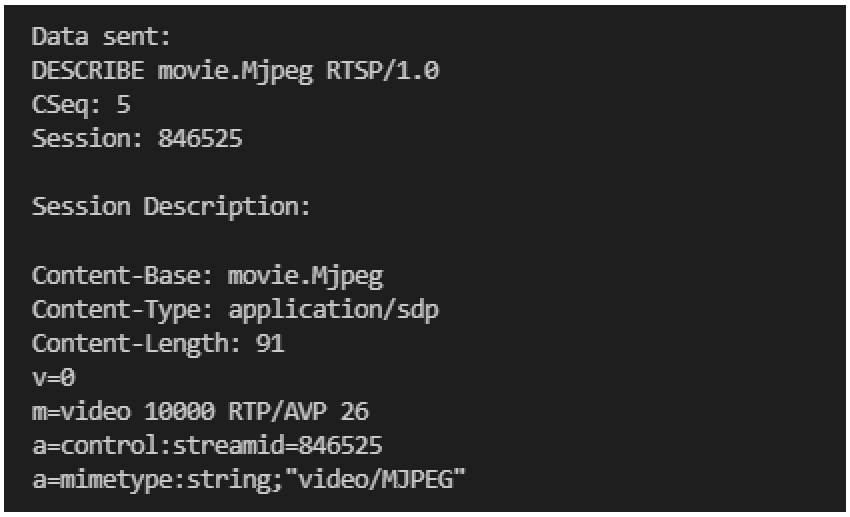
```
def describeSession(self):
    """Describe button handler."""
    self.sendRtspRequest(self.DESCRIBE)
```

In `ServerWorker.py` we add `sendDescription()` to print in the terminal what kinds of streams are in the session and what encodings are used.

The server's reply ”content” contains informations about our streaming ”method” such as Content Base (File name), Content Type (), Content Length, video type, streaming type, server port

```
def sendDescription(self):
    description = '\nSession Description: \n'

    body = 'v=0' + CRLF
    body += 'm=video ' + str(self.clientInfo['server_port'])
    + ' RTP/AVP ' + MJPEG_TYPE + CRLF
    body += 'a=control:streamid=' + str(self.clientInfo['session']) + CRLF
    body += 'a=mimetype:string;\"video/MJPEG\"' + CRLF

    description += "Content-Base: " + self.clientInfo['videoFileName'] + CRLF
    description += "Content-Type: " + 'application/sdp' + CRLF
    description += 'Content-Length: ' + str(len(body)) + CRLF
    description += body

    connSocket = self.clientInfo['rtspSocket'][0]
    connSocket.send(description.encode())
```

When we click on the ”DESCRIBE” button, we will get the result that is similar to this picture:



```
Data sent:
DESCRIBE movie.Mjpeg RTSP/1.0
CSeq: 5
Session: 846525

Session Description:

Content-Base: movie.Mjpeg
Content-Type: application/sdp
Content-Length: 91
v=0
m=video 10000 RTP/AVP 26
a=control:streamid=846525
a=mimetype:string;"video/MJPEG"
```

4.4 Fast forward or backward video

In `Client.py`:

```python
def backWardSession(self):
    """Forward button handler."""
    self.sendRtspRequest(self.BACKWARD)

def forwardSession(self):
    """Forward button handler."""
    self.sendRtspRequest(self.FORWARD)
```

When the client click the BACKWARD button, a `BACKWARD` request is sent to the server. In `ServerWorker.py`, the function `sendRTP()` will call the function `backward()` to get the backward frame. When the client click the FORWARD button, a `FORWARD` request is sent to the server, the function `sendRTP()` in `ServerWorker.py` will call the function `forward()` to get the fast forward frame.
In `ServerWorker.py`:

```python
if self.clientInfo['forward'].isSet():
    data = self.clientInfo['videoStream'].forward()
    self.clientInfo['forward'].clear()
elif self.clientInfo['backward'].isSet():
    data = self.clientInfo['videoStream'].backward()
    self.clientInfo['backward'].clear()
```

In order to implement the backward feature, we initiate the list `storeFilePos` in VideoStream's `__init__()` so that we can trace back to the frame that we want.

In `VideoStream.py` file, in the class `VideoStream` that lies on Server, we have the functions `backward()` and `forward()` to return the frame when the server receive the FORWARD/BACKWARD request.

In the `backward()` function, we return the frame which is 50 less in frameNum than the current frame when it is played or the first frame in case the current frame has `frameNum` less than 50. In the `forward()` function, we return the frame which has `frameNum` is 50 more than the current frame when it is played.
In `VideoStream.py`:

```
def backward(self):
    backwardFrame = 50
    numberOfFrame = self.frameNum
    if (numberOfFrame > backwardFrame):
        self.file.seek(self.storedFilePos[sel.frameNum-backwardFrame],0)
        self.frameNum -= backwardFrame
        if (self.frameNum < 0):
            self.frameNum = 0
    else:
        self.file.seek(self.storedFilePos[0],)
        self.frameNum = 0

    return self.nextFrame()

def forward(self):
    """Get next frame."""
    ForwardFrame = 50
    data = bytes()
    while (ForwardFrame > 0):
        self.nextFrame()
        ForwardFrame -= 1

    return self.nextFrame()
```

5 Reference

- Socket programming and OpenCv in Python — webcam video transmit and receive over wifi in Python
- Socket programming with multiple clients and OpenCV in Python
- How To Code A Video Streaming Server in NodeJS
- Computer Networking A Top-Down Approach, 7th Edition by James Kurose, Keith Ross (z-lib.org)
- Understanding Application Layer Protocols
- RTSP: The Real-Time Streaming Protocol Explained (Update)
- Real Time Streaming Protocol