VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE & ENGINEERING

COMPUTER NETWORKING

Assignment 1

Video Streaming with RTSP & RTP

Instructor: Hoang Nguyen Minh Duc
Student: Le Khac Minh Dang - 1810109

HO CHI MINH CITY, NOVEMBER 2020

Contents

1 Requirements Analysis

In this section, I will analyse the requirements, explain more about what I need to do to satisfy the given requirements in the assignment description. The detailed implementation on what I did in the code to achieve that will be explained in the *Function Description* section.

1.1 The RTSP protocol in the client

Our first task is to implement how the client handle the RTSP protocol. We need to implement the actions that are taken by the client when each button, namely Setup, Play, Pause and Teardown, is pressed. In the given `Client.py` file, the functions that handle each of the mentioned actions are `setupMovie`, `playMovie`, `pauseMovie` and `exitClient`, they are already completed, what we actually need to do is implement the `sendRtspRequest` function that is called in each of the functions above. In that function, our job is to make sure the correct RTSP request packet is sent for each type of request.

We also have to implement the part where the client receive an RTSP reply and how it would react to each replies after they are parsed. Explicitly, we need to update the client state machine for each corresponding received reply, implemented in the function `parseRtspReply`.

Finally, we have to write a little bit of code to open the RTP port in the `openRtpPort` function.

1.2 The RTP packetization in the server

Our next task is to complete the `encode` function in `RtpPacket.py`. All this function does is literally packetize all the information of a RTP packet into its header and payload, corresponding to the diagram given in the description. What we need to do is manipulating all the bits and bytes and putting them in their correct place in the header.

1.3 The customized interactive player

The GUI for the interactive player is already implemented in the `createWidgets` and several other functions. But it is just a very simple and minimal one for us to work with. Our job is to customize it and make it better in our own way.

2 Function Description

In this section I will explain in details all the functions in the `Client.py`, `ServerWorker.py` and `RtpPacket.py` files. The other files are just simple helper or wrapper classes to take in the arguments and initialize the aforementioned classes.

2.1 Client

The constructor __init__

The `Client.py` file implement the `Client` class. First, let's talk about its constructor. The constructor is already given, I just added some lines to help us implement some more functionalities in the Extend section.

The constructor initializes the following properties of the class:

- `master`: contains Tk root widget, it's also set to call the `handler` method when the close button is clicked.
- `serverAddr` and `serverPort`: contain the RTSP server and port.
- `rtpPort`: contains the RTP port.
- `fileName`: contains the file name of the desired video to stream.
- `rtsqSeq`: contains the sequence number of the RTSP packets.
- `sessionId`: contains the RTSP session ID.
- `requestSent`: contains the information about the last sent request, the purpose of this property will be made clear when I explain later methods.
- `teardownAcked`: this value will be set to 1 when an ACK for a TEARDOWN request is received from the server.
- `frameNbr`: contains the current frame number of the streaming video.

Also, the constructor will make a call to `createWidgets` to create the GUI, and `connectToServer` to connect to the RTSP socket opened at the server.

connectToServer

This is a short method to open a TCP socket on the client and connect to the TCP socket opened at the server. The reason this socket is a TCP socket is because RTSP uses TCP to maintain connection. Also, if the connection fails, it will throw a tkinter error box.

createWidgets

This method creates a GUI with 4 buttons and a label. The 4 buttons are organized on the same row, with the text Setup, Play, Pause and Teardown inside. The label is placed above the buttons and is where the video will be shown. What I did to customize the GUI is that I resized the buttons and the label a bit to make them more fitting, I also added a black background to the label. Moreover, I initially disabled all buttons other than Setup, this is because clicking other buttons doesn't make sense if the connection hasn't been setup yet.

handler

This is a small handler method that gets invoked when we click the close button. What it does is it will send a PAUSE request to the server, then open up a message box to ask if the user really wants to quit. If that is the case, it will send a TEARDOWN request to the server, otherwise, a PLAY request will be sent.

openRtpPort

This is a method that we need to complete. It is quite simple though, all it does is open a UDP port, set a timeout for it, and bind it with the RTP port on the server. The reason it is a UDP port is because RTP is a streaming protocol built on top of UDP. It will also throw an error box if the binding process is unsuccessful.

setupMove, playMovie, pauseMovie and exitClient

These 4 methods correspond to the 4 actions when the user presses the buttons. All of them will send a corresponding RTSP request to the server. Furthermore, `playMovie` will create a thread to listen to incoming RTP, and `exitClient` will destroy the widgets and cleanup the cache files.

listenRtp

This method is run on the separate thread created when the user clicks the Play button. It continuously listen to incoming RTP packets, decode them, then display its payload as a frame on the video label through the `updateMovie` method. It also takes the sequence number of the RTP packet and compares it to the current video frame number. If the RTP sequence number if equal or less than the current frame, the packet will be treated as a late packet and will be discarded.

Also if there is a PAUSE or a TEARDOWN RTSP packet received, there will be no more RTP packets to receive, when that happens, it will handle the exception by stop listening and close the RTP port on TEARDOWN.

updateMovie and writeFrame

These are 2 small helper methods to show the payload of an RTP packet as an image on the video label. What they do is to save the payload to a cache file, then display the content of that file on the GUI.

sendRtspRequest

This is one of the most important methods. It takes in one parameter, which is the request to be sent. First and foremost, before building the request packet and send it, it must check if that request can be sent in the current state of the client or not: the SETUP request can only be sent in the INIT state, the PLAY request can only be sent in the READY state, the PAUSE request can only be sent in the PLAYING state, and the TEARDOWN request can be sent in both READY and PLAYING states.

In case of a SETUP request, the client will start a thread to listen to incoming RTSP replies. Because SETUP indicates the start of an RTSP session, the RTSP sequence number is set to 1. The SETUP request packet is built as follow:

```
request = "SETUP " + str(self.fileName) + " RTSP/1.0\n"
request += "CSeq: " + str(self.rtspSeq) + "\n"
request += "Transport: RTP/UDP; client_port= " + str(self.rtpPort)
```

For the other types of request, the sequence number is incremented by 1. Also the content of the packets are identical, except for the request code, here is the packet for the PLAY request:

```
request = "PLAY " + str(self.fileName) + " RTSP/1.0\n"
request += "CSeq: " + str(self.rtspSeq) + "\n"
request += "Session: " + str(self.sessionId)
```

One more thing, because I am programming with python 3, the data to be sent and received over a socket must be of type `bytes`, not `strings`, therefore, the request (currently a string) must be encoded before sending by the method `encode()`.

recvRtspReply

This method is ran on another thread to receive incoming RTSP replies. It simply continuously listens for replies, decodes it in `utf-8` encoding if there is one, and then passes if to the `parseRtspReply` method. Also it will close the RTSP socket on the client if a TEARDOWN reply is received.

parseRtspReply

This is another important method. Its purpose is to parse the RTSP reply received from the server, perform several checks, and then update the state of the client corresponding to each replies. The parsing part is extremely simple by just using the `split` method of python strings.

The checks are as followed: first, it checks if the RTSP sequence number of the reply is the same as the sent request, only continue if it's the same. Then if the current session ID saved on the client is 0, that means this is the first setup reply, it then will set the session ID to the received session ID, otherwise it will check if the session ID is still the same as the saved one, and only proceed if it is.

After that is the important part: the client will update its state based on the request. Because the reply from the server doesn't carry the info about what type of request it is, I have to keep track of it by the property `requestSent` I mentioned earlier. If the request was SETUP, the state will be updated to READY and the RTP port will be opened by calling `openRtpPort`, I also disable the SETUP and the PAUSE buttons in this state, because they don't have any purpose in it. If the request was PLAY, the state will be updated to PLAYING, the PAUSE button will be enabled and the PLAY button will be disabled. If the request was PAUSE, the state will be updated to READY, the thread that is listening to RTP and play the video will exit, and the PAUSE and PLAY buttons will change their states. If the request was TEARDOWN, the state will be back to INIT, and the property `teardownAcked` will be set to 1, telling other functions to close the sockets and exit.

2.2 ServerWorker

The constructor __init__

In the `ServerWorker` class, the properties are not stored separately like in `Client`. Instead, they are stored in one single dictionary called `clientInfo`, which gets initilized in this constructor.

recvRtspRequest

This method is pretty much the same as the `recvRtspReply` in the client. It is run in another thread to listen to incoming RTSP request packets. When there is a packet, it will proceed to decode it in `utf-8` encoding and pass it to `processRtspRequest`.

processRtspRequest

This is the most important method on the server. It is almost identical to `parseRtspReply` of the client in the sense that it will parse the RTSP request packet by `split()`, and then process it and update its state differently based on the request type, and finally send a RTSP reply using `replyRtsp`. If the request is SETUP, the state will be set to READY, a random session ID will be generated. If the request is PLAY, the state will be set to PLAYING, an UDP port will be opened for streaming RTP packet, and a thread will be open to do just that. If the request is PAUSE, the state is set to READY, the streaming thread will be stopped. If case of TEARDOWN, the streaming thread will be stopped and the RTP socket will also be closed.

replyRtsp

This method is used to form the reply packet and send it to the client. There are 3 RTSP status code implemented in this system: 200 for OK, 404 when the requested file isn't found, and 500 when there is connection error.

sendRtp

This method runs in another thread to continuously stream video frames as RTP packets to the client every 0.05 seconds. It first gets the next video frame using the helper methods in the `VideoStream` class, then puts it in a valid RTP packet and finally sends it to the client.

makeRtp

This is where an RTP packet is encoded. This method passes several hard-coded header fields into the `encode` method of the `RtpPacket` class (more on this later).

2.3 RtpPacket

encode

This is the most important method in the class and we are required to complete it. Basically, all this does is to put the bits and bytes of the values in the header fields and puts them in their correct place, as shown in the diagram in the assignmet description. A brief summary of the diagram:

- The first byte of the header contains: 2 bits for version, 1 padding bit, 1 extension bit, 4 bits for contribution source bit.
- The second byte of the header contains: 1 marker bit, 7 bits for payload type.
- The next 2 bytes are for sequence number (big-endian).
- The next 4 bytes are for timestamp (big-endian).
- The next 4 bytes are for SSRC (big-endian).
- The diagram shows 4 more bytes for CSRC, but I simply don't include it in this assignment, as suggested in the description.

Finally I add the payload to the packet.

decode

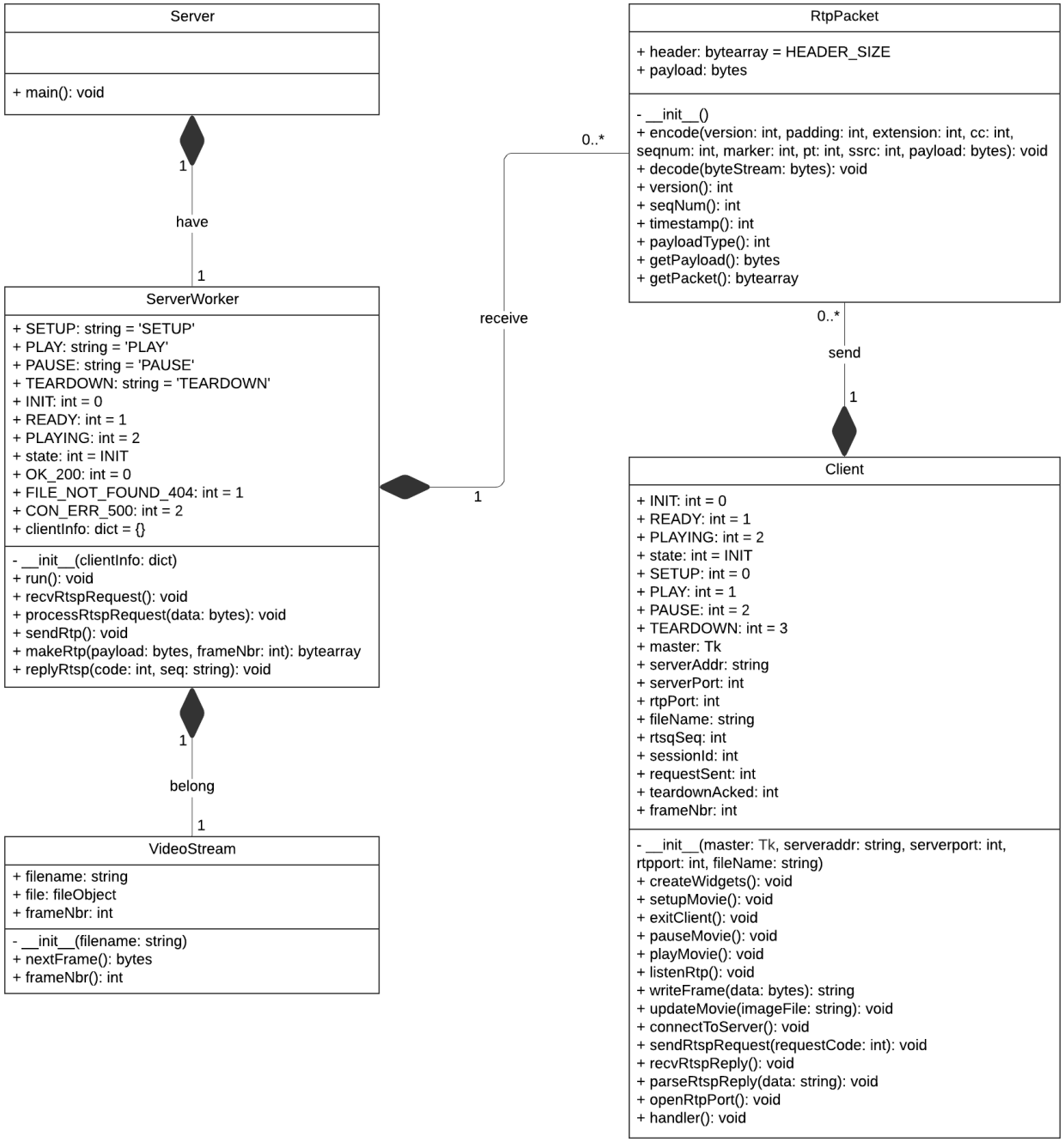
This method contains only 2 lines to separate the header and the payload from a whole RTP packet.

The other methods

All the other methods are helper methods to get the value of a specific field of the header, or to get the payload.

3 Class diagram

This is the class diagram for the basic, un-extended implementation of the system.



4 Extended Functionalities

4.1 Performance Evaluation

For the first problem in the Extend section, I added some hooks in the client to calculate 2 performance metrics: RTP packet loss rate and video data rate.

RTP packet loss rate

To calculate the packet loss rate, the idea is to count how many packets were lost, and how many packets were sent. For the amount of packets that were lost, I count them by comparing the sequence number of the received RTP packet to the frame number that it should be, which is the current frame number plus 1. If it is not equal, we have a packet loss. For the total packet count, that is just the current frame number at the time the packet loss rate is calculated. The packet loss rate will be printed to stdout when the stream exit.

The first hook is added to `listenRtp`:

```
if self.frameNbr + 1 != rtpPacket.seqNum():
    self.lossCounter += (rtpPacket.seqNum() - (self.frameNbr + 1))
    print("[*]Packet loss!")
```

The second hook is added to `exitClient` (when TEARDOWN is sent, we calculate the packet loss):

```
if self.frameNbr != 0:
    lossRate = self.lossCounter / self.frameNbr
    print("[*]RTP Packet Loss Rate: " + str(lossRate) +"\n")
```

Video data rate

To calculate the video data rate, I start a timer when the Play button is pressed, and end it when the Pause button is pressed, so that I can calculate the streaming time. The number of video data received is the total length of all the RTP packets' payloads. The data rate will be printed out to stdout when the stream is paused.

The first hook is added to `listenRtp`:

```
self.bytesReceived += len(rtpPacket.getPayload())
```

The second hook is added to `parseRtspReply`, for the PAUSE case (I calculate the video rate at this action):

```
dataRate = int(self.bytesReceived / (time.time() - self.startTime))
print("[*]Video data rate: " + str(dataRate) + " bytes/sec\n")
```

4.2 Three-button client

For the second Extended problem, I implemented a client with 3 buttons: Play, Pause and Stop. To answer the question of how I handle the SETUP request, it is as simple as sending the SETUP request once when the Play button is pressed for the first time. For the Stop action, I think that sending TEARDOWN is an appropriate action, and I did just that.

This is how I implement the Play action:

```
def playMovie(self):
    """Play button handler."""
    # If it's the first time PLAY is clicked, send SETUP
    if self.state == self.INIT and self.firstPlay:
        self.sendRtspRequest(self.SETUP)
        self.firstPlay = False
        # Wait until ready
        while self.state != self.READY:
            pass
        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 The DESCRIBE request

A Describe button is added to the Client. Each time the button is pressed, the client will send a RTSP DESCRIBE request to the server, and the server will respond with a reply that contains information about the current session (the information is only replied as several hard-coded fields for demonstration purpose), the cilent will save that information as a file called

`session.txt`. The snippets of code that are added to the client is pretty simple: create one more button, one more `sendRtspRequest` case, and one more `parseRtspReply` case. The `ServerWorker` is also modified to process DESCRIBE request.

4.4 Streaming time

The client GUI is also updated to show how long the video has been played. To calculate the played time, I simply multiply the frame number of the current frame by 0.05 and take the integer part, this is because the server sends a frame to the client every 0.05 seconds. The following code is appended to `listenRtp`:

```
currentTime = int(currFrameNbr * 0.05)
self.timeBox.configure(text="%02d:%02d" % (currentTime // 60, currentTime % 60))
```

5 Result Evaluation

5.1 Testing environment

All the testing was done in 2 different experiments:

1. locally on 1 machine, which is a Ubuntu Desktop 18.04 virtual machine on VMWare 16.0 with 3GB of RAM and 1 processor.
2. remotely on 2 machines connected to each other, the specification of each machine is the same as above.

5.2 Basic interactions of the client

The interaction between the client and the server is exactly as intended, all the buttons are responsive and functional, the video is streamed without any noticeable problems in the process, and also terminated correctly. Even though there seems to be a bit of a delay in the stream for both experiments, I used a stopwatch to calculate the "real" streaming time, and it was a little bit higher than 25 seconds (which is the duration of the video) by a small amount (around 1 sec).

5.3 Loss rate and data rate

The average RTP packet loss rate and video data rate for each experiment (each experiment was evaluated 5 times) is as follow:

1. Locally: loss = 0%, data rate = 146084 bytes/sec.
2. Remotely: loss = 0%, data rate = 136187 bytes/sec.

6 User Manual

6.1 Server

To start the server, use the following command (server port should be greater than 1024):

`python3 Server.py <server_port>`

6.2 Basic client

To start the basic client with 4 buttons:
`python3 ClientLauncher.py <server_host> <server_port> <RTP_port> <video_file>`

Where `server_host` is the IP address of the server, `server_port` is the port passed to the server above, `RTP_port` is the port to receive incoming RTP (should also be greater than 1024), and `video_file` is the name of the requested video to be streamed.

For the basic client, users must first click the Setup button to initialize the connection, then the stream can be started using the Play button. The stream can be paused at anytime by using the Pause button, and quit at anytime using the Teardown or the exit buttons.

6.3 Three-button client

To start the improved client with 3 buttons:
`python3 ClientLauncher3Btn.py <server_host> <server_port> <RTP_port> <video_file>`

The options are the same as the basic client. For this client, users don't need to setup the stream before playing, simply press Play to start the stream. The Pause button works exactly the same, and the Stop button is equivalent to Teardown.