Computer Networks, Spring 2022

Final assignment

Due Tue May 10 5:00:00 PM EST 2022 to Gradescope.
Last updated Wed Apr 20 06:23:25 PM EDT 2022

For your final assignment, you will implement a streaming video server and client that communicate using the Real-Time Streaming Protocol (RTSP) and send data using the Real-time Transport Protocol (RTP). Your task is to implement the RTSP protocol in the client and implement the RTP packetization in the server. You will also answer questions about your implementation, and provide a Wireshark capture showing that the application works.

This assessment can be completed individually, or in pairs of two. If you choose to be in a pair, we expect a reflection on who completed which portions of the assignment. More information on our expectations for pairs is found later in this document. Collaboration outside of this is prohibited.

Code

We will provide you code that implements the RTSP protocol in the server, the RTP de-packetization in the client, and takes care of displaying the transmitted video. The remainder of the code is yours to complete.

- **Client, ClientLauncher**: The ClientLauncher starts the Client and the user interface which you use to send RTSP commands and which is used to display the video. In the Client class, you will need to implement the actions that are taken when the buttons are pressed. You do not need to modify the ClientLauncher module.
- **ServerWorker, Server**: These two modules implement the server which responds to the RTSP requests and streams back the video. The RTSP interaction is already implemented and the ServerWorker calls methods from the RtpPacket class to packetize the video data. You do not need to modify these modules.
- **RtpPacket**: This class is used to handle the RTP packets. It has separate methods for handling the received packets at the client side and you do not need to modify them. The Client also de-packetizes (decodes) the data and you do not need to modify this method. You will need to complete the implementation of video data RTP-packetization (which is used by the server).
- **VideoStream**: This class is used to read video data from the file on disk. You do not need to modify this class.

As you develop your code, you may find it wise to use Wireshark to debug how the individual parts of the code interact. At the end of your implementation, you will submit a Wireshark capture of your application in action. Information on submission requirements are later in this document.

Running the code

After completing the code, you can run it as follows. First, start the server with the command:

```
python3 Server.py server_port
```

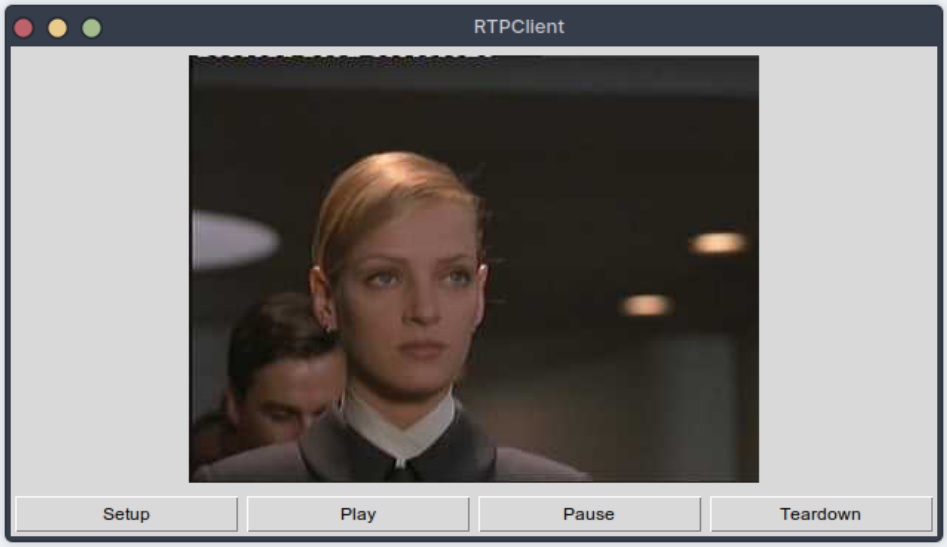
where `server_port` is the port your server listens to for incoming RTSP connections. The standard RTSP port is 554, but you will need to choose a port number greater than 1024.

Then, start the client with the command:

```
python3 ClientLauncher.py server_host server_port RTP_port video_file
```

where `server_host` is the name of the machine where the server is running, `server_port` is the port where the server is listening on, `RTP_port` is the port where the RTP packets are received, and `video_file` is the name of the video file you want to request (we have provided one example file `movie.Mjpeg`). The file format is described in the Appendix.

The client opens a connection to the server and opens up a window like this:



You can send RTSP commands to the server by pressing the buttons. A normal RTSP interaction goes as follows:

1. The client sends SETUP. This command is used to set up the session and transport parameters.
2. The client sends PLAY. This command starts the playback.
3. The client may send PAUSE if it wants to pause during playback.
4. The client sends TEARDOWN. This command terminates the session and closes the connection.

The server always replies to all the messages that the client sends. The code 200 means that the request was successful while the codes 404 and 500 represent FILE_NOT_FOUND error and connection error, respectively. In this lab, you do not need to implement any other reply codes. For more information about RTSP, please see RFC 2326.

The client

Your first task is to implement the RTSP protocol on the client side. To do this, you need to complete the functions that are called when the user clicks on the buttons on the user interface. When the client starts, it also opens the RTSP socket to the server. Use this socket for sending all RTSP requests. You will need to implement the actions for the following request types.

SETUP

- Send a SETUP request to the server. You will need to insert the Transport header in which you specify the port for the RTP data socket you just created.
- Read the server’s response and parse the Session header (from the response) to get the RTSP session ID.
- Create a datagram socket for receiving RTP data and set the timeout on the socket to 0.5 seconds.

PLAY

- Send a PLAY request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.
- Read the server's response.
- As media is received, calculate statistics about the session, and print them to the terminal. Relevant statistics are the RTP packet loss rate and the video data rate (in bits per second).

PAUSE

- Send a PAUSE request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.
- Read the server's response.

TEARDOWN

- Send a TEARDOWN request. You must insert the Session header and use the session ID returned in the SETUP response. You must not put the Transport header in this request.
- Read the server's response.

Note: You must insert the CSeq header in every request you send. The value of the CSeq header is a number which starts at 1 and is incremented by one for each request you send.

An example

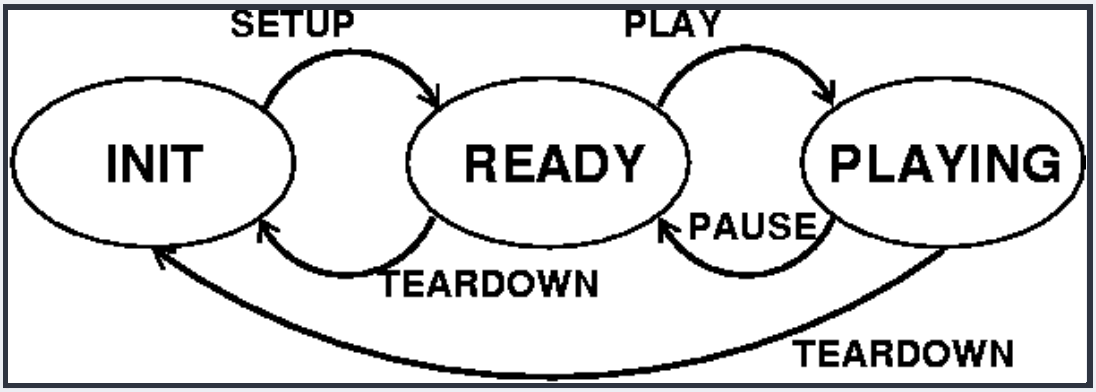
Here is a sample interaction between the client and server. The client's requests are marked with `C:` and server's replies with `S:`. In this lab both the client and the server do not use sophisticated parsing methods, and they expect the header fields to be in the order you see below.

```
C: SETUP movie.Mjpeg RTSP/1.0
C: CSeq: 1
C: Transport: RTP/UDP; client_port= 25000

S: RTSP/1.0 200 OK
S: CSeq: 1
S: Session: 123456

C: PLAY movie.Mjpeg RTSP/1.0
C: CSeq: 2
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 2
S: Session: 123456

C: PAUSE movie.Mjpeg RTSP/1.0
```

```
C: CSeq: 3
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 3
S: Session: 123456

C: PLAY movie.Mjpeg RTSP/1.0
C: CSeq: 4
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 4
S: Session: 123456
```

Tracking client state

One of the key differences between HTTP and RTSP is that in RTSP each session has a state. In this assignment, you will need to keep the client's state up-to-date. The client changes state when it receives a reply from the server according to the following state diagram.

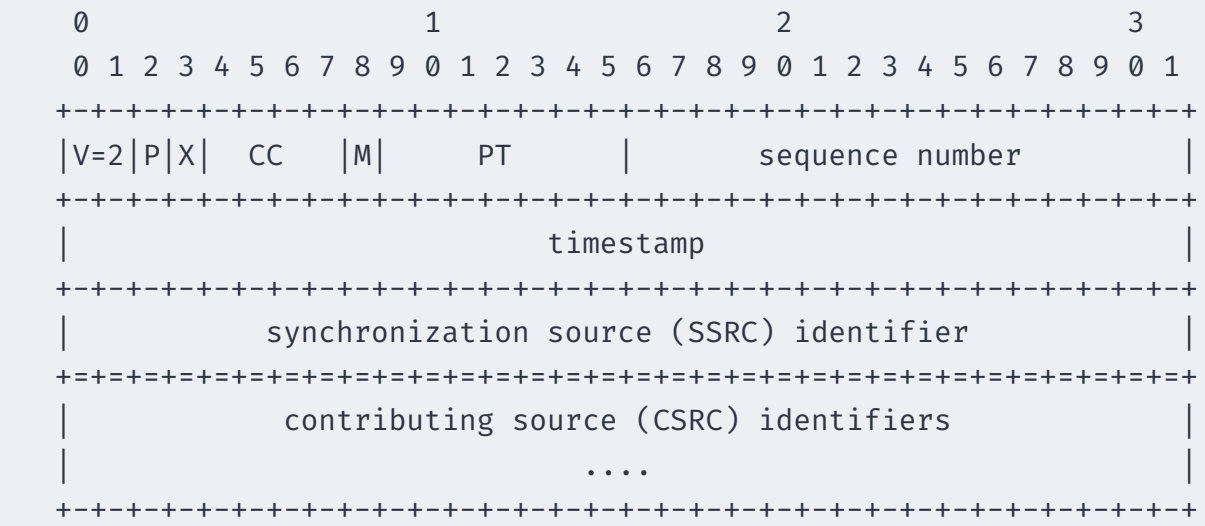


The server

On the server side, you will need to implement the packetization of the video data into RTP packets. You will need to create the packet, set the fields in the packet header and copy the payload (i.e., one video frame) into the packet.

When the server receives the PLAY-request from the client, the server reads one video frame from the file and creates an RtpPacket-object which is the RTP-encapsulation of the video frame. It then sends the frame to the client over UDP every 50 milliseconds. For the encapsulation, the server calls the encode function of the RtpPacket class. Your task is to write this function. You will need to do the following: (the letters in parenthesis refer to the fields in the RTP packet format below).

- Set the RTP-version field (V). You must set this to 2.
- Set padding (P), extension (X), number of contributing sources (CC), and marker (M) fields. These are all set to zero in this lab.
- Set payload type field (PT). In this lab we use MJPEG and the type for that is 26.
- Set the sequence number. The server gives this the sequence number as the frameNbr argument to the encode function.
- Set the timestamp using Python's `time` module.
- Set the source identifier (SSRC). This field identifies the server. You can pick any integer value you like.
- Because we have no other contributing sources (field CC == 0), the CSRC-field does not exist. The length of the packet header is therefore 12 bytes, or the first three lines from the diagram below.



You must fill in the header fields in the header byte array of the RtpPacket class. You will also need to copy the payload (given as argument data) to the RtpPacket's payload data field.

The above diagram is in the network byte order (also known as big-endian). Python uses the same byte order, so you do not need to transform your packet header into the network byte order. For more details on RTP, please see RFC 1889.

Bitwise operations

Here are some examples on how to set and check individual bits or groups of bits. Note that in the RTP packet header format smaller bit-numbers refer to higher order bits, that is, bit number 0 of a byte is 2^7 and bit number 7 is 1 (or 2^0). In the examples below, the bit numbers refer to the numbers in the above diagram.

Because the header-field of the RtpPacket class is of type bytearray, you will need to set the header one byte at a time, that is, in groups of 8 bits. The first byte has bits 0-7, the second byte has bits 8-15, and so on.

To set bit number `n` in variable `mybyte` of type byte:

```
mybyte = mybyte | 1 << (7 - n)
```

To set bits `n` and `n + 1` to the value of `foo` in variable `mybyte`:

```
mybyte = mybyte | foo << (7 - n)
```

Note that `foo` must have a value that can be expressed with 2 bits, that is, 0, 1, 2, or 3.

To copy a 16-bit integer `foo` into 2 bytes, `b1` and `b2`:

```
b1 = (foo >> 8) & 0×FF
b2 = foo & 0×FF
```

After this, `b1` will have the 8 high-order bits of `foo` and `b2` will have the 8 low-order bits of `foo`.

You can copy a 32-bit integer into 4 bytes in a similar way.

Bits example

Suppose we want to fill in the first byte of the RTP packet header with the following values:

```
V = 2
P = 0
X = 0
CC = 3
```

In binary this would be represented as

```
1 0 | 0 | 0 | 0 0 1 1
V=2   P   X   CC = 3
2^7 . . . . . . . . 2^0
```

Code questions

After completing your implementation, answer the following questions, and submit them to Gradescope as a PDF.

1. The user interface on the RTPClient has 4 buttons for the 4 actions. If you compare this to a standard media player, such as Quicktime or Windows Media Player, you can see that they have only 3 buttons for the same actions: PLAY, PAUSE, and STOP (roughly corresponding to TEARDOWN). There is no SETUP button available to the user.
    a. Given that SETUP is mandatory in an RTSP-interaction, how would you implement that in a media player? When does the client send the SETUP? Explain.
    b. Is it appropriate to send TEARDOWN when the user clicks on the STOP button? Why or why not?

For this assignment, we expect both students in a pair to complete about half of the work. After you complete the assignment, please answer the following reflection questions, and submit them to Gradescope. If you are in a pair, this is the only part of the final assignment that must be done by yourself. This reflection is not required for students working individually.

1. What parts of the codebase did you complete, and what parts did your partner complete?
2. Do you feel like the allocation of work between you and your partner was fair? Why or why not?

Submission

The following files must be submitted to Gradescope by the deadline. Include your name(s) and JHU email address(es) in the headers of your PDF submissions, and as comments at the top of your code submission.

- A ZIP file with the completed Client.py and RtpPacket.py.
- A PDF with responses to the code questions.
- A Wireshark capture showing correct streaming of `movie.Mjpeg`, with SETUP, PLAY, PAUSE, and TEARDOWN messages in the capture.
- For students working in pairs, a PDF with responses to the pair reflection questions, submitted individually.

Appendix

MJPEG Format: In this assignment, the server streams a video which has been encoded into a proprietary MJPEG file format. This format stores the video as concatenated JPEG-encoded images, with each image being preceded by a 5-Byte header which indicates the bit size of the image. The server parses the bitstream of the MJPEG file to extract the JPEG images on the fly. The server sends the images to the client at periodic intervals. The client then displays the individual JPEG images as they arrive from the server.