



CMPE 207-01
Network Programming and Applications
Spring-2016

Video Streaming using RTP/RTSP
Project Report

Group-10

Name	SJSU ID	Class ID
Sashank Malladi	010466651	34
Venkata Adithya Boppana	010731201	08
Sudheer Doddapaneni	010328123	17
Vinay Puttanna	010699858	56

Contents

1	Introduction	3
	Objective	3
2	RTP / RTSP protocols	3
2.1	Real-time Transport Protocol	3
2.1.1	RTP Header	3
2.1.2	RTP Control Protocol -- RTCP	4
2.2	Real-Time Streaming Protocol	4
2.2.1	RTSP States	5
3	RTP/RTSP Server (localhost)	5
3.1	Overview	5
3.2	Flow Chart	5
3.3	Implementation	7
3.3.1	Concurrency in java	7
3.3.2	Explanation of Server implementation using Pseudo code	7
4	RTP/RTSP Client	10
4.1	Overview	10
4.2	Flow diagram	10
4.3	Implementation	11
4.3.1	main() method	11
4.3.2	Media player GUI	11
4.3.3	UI Control Event Handlers	11
4.3.4	Sockets and Request / Response	13
4.3.5	Frame Synchronizer	14
4.3.6	Timer Listener	14
5	Client-Server Interaction	15
6	Project Execution in local Instance	15
6.1	Running the server	15
6.2	Running the client	15
6.3	Execution Result	16
7	Deployment on cloud	16
7.1	Creating instances on AWS	16
7.2	Transferring source code for deployment	17
7.3	Using 'Xming'	18
7.4	Running the application on AWS	18
8	Conclusion	19
9	Acknowledgement	19
10	References	19

1 Introduction

Video streaming is becoming more popular in recent years than ever. Globally, IP video traffic will be 80 percent of all IP traffic (both business and consumer) by 2019. It has helped governments, organizations and people around the world to bring down operating costs to a considerable extent. Rapid increase in the number of mobile device (tablets and smartphones) users has been adding more people to the IP video traffic. In our project we have implemented a client and a concurrent-server application that provides video streaming service to multiple users. Development environment comprised of Java programming language, Linux operating system and Amazon Web Services (AWS) cloud service.

Objective

The objective of this network application development project was to develop a client application and a server (concurrent server) application using RTP/RTSP protocols, that provides video streaming service to the user. Project tasks involved understanding of the RTP & RTSP protocols functioning, implementation of multi-threading to achieve real concurrency, and team collaboration.

2 RTP / RTSP protocols

2.1 Real-time Transport Protocol

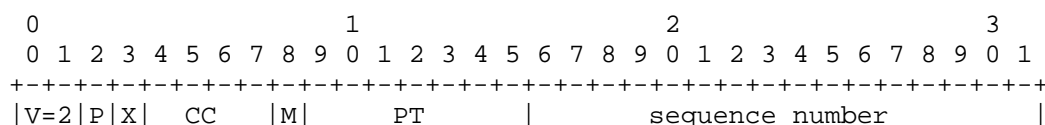
The Real-time Transport Protocol provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, timestamping and delivery monitoring. Applications typically run RTP on top of UDP to make use of its multiplexing and checksum services; both protocols contribute parts of the transport protocol functionality. RTP supports data transfer to multiple destinations using multicast distribution if provided by the underlying network.

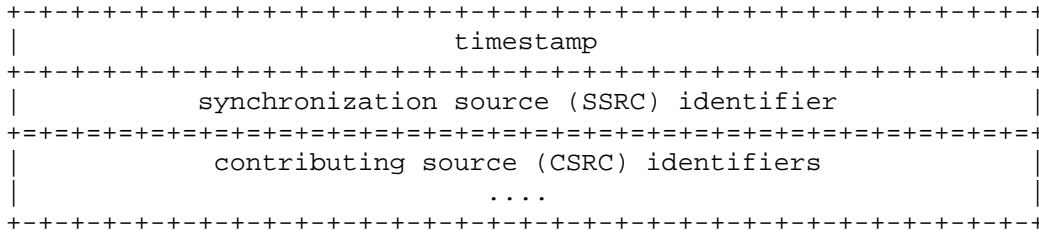
RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do so. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence.

RTP payload type defines a set of rules that is followed by the RTP packetizer to generate RTP packets from an audio or a video file to stream to the client. These packets can be transmitted to the client either using UDP or TCP protocol. UDP is a lightweight protocol that utilizes minimum bandwidth and provides unreliable connectionless service. In our application, the server sends RTP packets over UDP to the clients. This design reduces the overhead imposed by the TCP protocol in providing reliable delivery.

2.1.1 RTP Header

The RTP header has the following format:





The first twelve octets are present in every RTP packet.

- version (V): 2 bits
This field identifies the version of RTP.
- padding (P): 1 bit
If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload.
- extension (X): 1 bit
If the extension bit is set, the fixed header MUST be followed by exactly one header extension.
- CSRC count (CC): 4 bits
The CSRC count contains the number of CSRC identifiers that follow the fixed header.
- marker (M): 1 bit
It allows significant events such as frame boundaries to be marked in the packet stream.
- payload type (PT): 7 bits
This field identifies the format of the RTP payload and determines its interpretation by the application.
- sequence number: 16 bits
The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.
- timestamp: 32 bits
The timestamp reflects the sampling instant of the first octet in the RTP data packet.
- SSRC: 32 bits
The SSRC field identifies the synchronization source.
- CSRC list: 0 to 15 items, 32 bits each
The CSRC list identifies the contributing sources for the payload contained in this packet

2.1.2 RTP Control Protocol -- RTCP

The RTP control protocol (RTCP) is based on the periodic transmission critical to get feedback from the receivers to diagnose faults in the distribution. Sending reception feedback reports to all participants allows one who is observing problems to evaluate whether those problems are local or global. With a distribution mechanism like IP multicast, it is also possible for an entity such as a network service provider who is not otherwise involved in the session to receive the feedback information and act as a third-party monitor to diagnose network problems.

2.2 Real-Time Streaming Protocol

The Real-Time Streaming Protocol (RTSP) establishes and controls either a single or several time-synchronized streams of continuous media such as audio and video. It does not typically deliver the continuous streams itself, although interleaving of the continuous media stream with the control stream is possible. There is no notion of an RTSP connection; instead, a server maintains a session labeled by an

identifier. An RTSP session is in no way tied to a transport-level connection such as a TCP connection. During an RTSP session, an RTSP client may open and close many reliable transport connections to the server to issue RTSP requests. Alternatively, it may use a connectionless transport protocol such as UDP. In our application, TCP is used for exchanging RTSP messages (setup, play, pause, teardown) between the client and the server. The streams controlled by RTSP may use RTP, but the operation of RTSP does not depend on the transport mechanism used to carry continuous media.

An RTSP session is established between the client and the server once the client sends a Setup request to the server. Each session has its own identifier. Video playback occurs as a result of a series of exchange of RTSP messages between the client and the server.

2.2.1 RTSP States

RTSP controls a stream which may be sent via a separate protocol, independent of the control channel. For example, RTSP control may occur on a TCP connection while the data flows via UDP. Thus, data delivery continues even if no RTSP requests are received by the media server. Also, during its lifetime, a single media stream may be controlled by RTSP requests issued sequentially on different TCP connections. Therefore, the server needs to maintain "session state" to be able to correlate RTSP requests with a stream.

Many methods in RTSP do not contribute to state. However, the following play a central role in defining the allocation and usage of stream resources on the server: SETUP, PLAY, RECORD, PAUSE, and TEARDOWN.

- **SETUP:**
Causes the server to allocate resources for a stream and start an RTSP session.
- **PLAY and RECORD:**
Starts data transmission on a stream allocated via SETUP.
- **PAUSE:**
Temporarily halts a stream without freeing server resources.
- **TEARDOWN:**
Frees resources associated with the stream. The RTSP session ceases to exist on the server.

RTSP methods that contribute to state use the Session header field to identify the RTSP session whose state is being manipulated. The server generates session identifiers in response to SETUP requests.

3 RTP/RTSP Server (localhost)

3.1 Overview

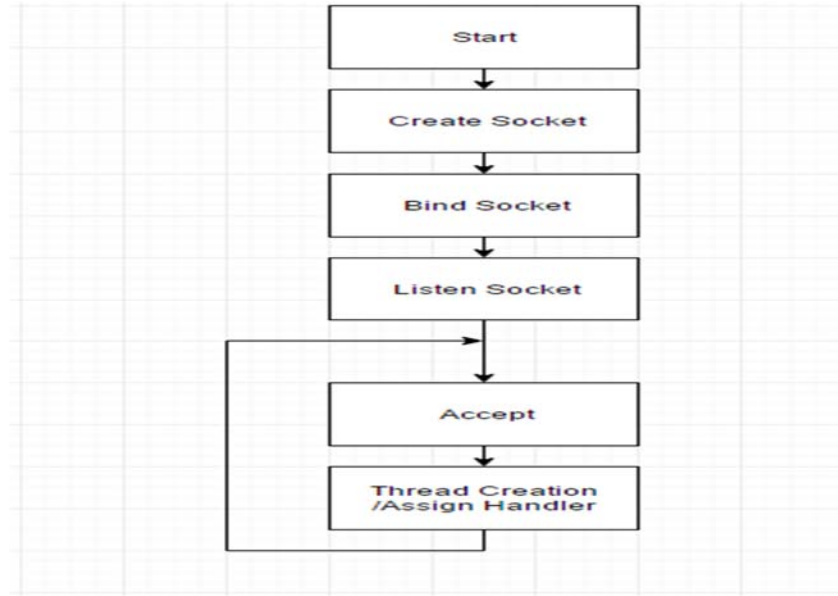
Video streaming server plays an important part of the project by serving requests made by the client. There is a possibility of handling requests from multiple clients at a time. The following are few important characteristics of the RTSP/RTP server.

- Handle requests from multiple clients at same time.
- Maintain session of each client for provision of RTSP commands like Setup, Play, Pause, Teardown etc.
- Synchronize frames and their corresponding rate of transfer appropriately.

3.2 Flow Chart

Following are the flow charts for implementation of RTSP/ RTP server

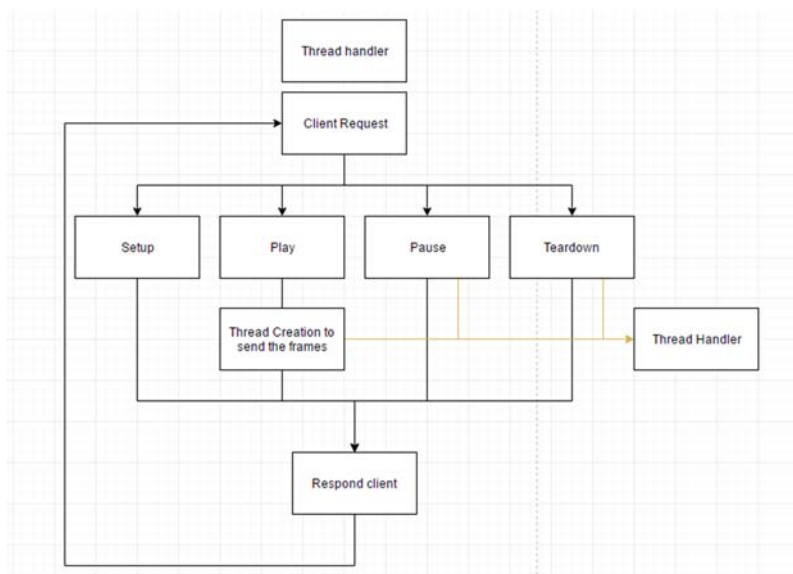
- Server implementation



As part of server implementation, the following steps are involved according to the flow diagram description

- Initially a TCP socket is created and a port is initialized accordingly
- Once binding is done to a port, server will be in passive mode listening to the port for any incoming requests
- If it gets a request from any client for connection, it immediately accepts the request.
- After acceptance, a new thread is created for the accepted socket to handle all the requests from connected node
- Then the iterates back to the listening mode to accept more clients

- Client thread handler



As part of server implementation, the next big thing is to handle client requests individually in a thread. The following steps are involved according to the flow diagram description

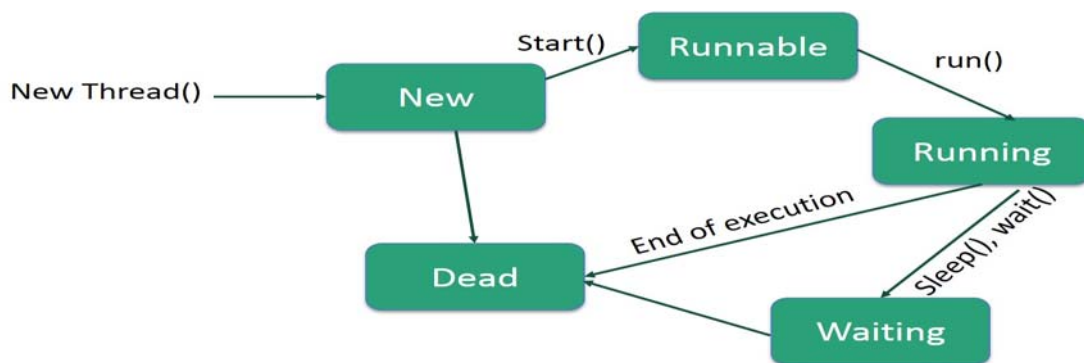
- Initially the thread will be waiting for setup request from client after accepting connection.
- On receiving client request for set up it creates a UDP socket for further communication or sending the video data.
- On play request the thread will start sending the video packets at corresponding frame rate using another thread.
- Then the thread iterates back to listening for any future requests from the client to handle functions like pause, teardown etc.

3.3 Implementation

3.3.1 Concurrency in java

Java is a multi-threaded programming language. A multi-threaded program can contain two or more tasks running concurrently at the same time. This makes optimal utilization in available resources especially when your computer has multiple CPUs.

The following diagram shows thread life cycle.



One of the way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

- You will need to override run() method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

- Once Thread object is created, you can start it by calling start() method, which executes a call to run() method. Following is simple syntax of start() method:

```
void start( );
```

3.3.2 Explanation of Server implementation using Pseudo code

In this section, we will observe pseudo code implementation of the server. The server code utilizes following header files and their corresponding implementation for successful execution.

- VideoStream.java

b. RTPpacket.java

As part of initial server implementation steps, server will be creating a TCP socket and start listening to the defined port. The following code shows how a RTSP socket is created and start listening to clients through the port

```
ServerSocket listenSocket = new ServerSocket(RTSPport);
int id = 0;
while (true) {
    try{
        theServer.RTSPsocket = listenSocket.accept();
        ClientServiceThread cliThread = new
        ClientServiceThread(theServer.RTSPsocket,id++);
        cliThread.start();
    }
}
```

In the above code it can be observed that a thread with name `ClientServiceThread` is created to handle client requests in new thread. `ClientServiceThread` is a class with implementation to handle all kinds of RTSP requests from the client. From the below code it can be observed clearly how this class extends the base thread class.

```
class ClientServiceThread extends Thread
```

Inside the `ClientServiceThread` class a method name `run` is declared which will be initialized or triggered on call of `CliThread.start()` method.

```
public void run() {}
```

Once a client request is received and a corresponding handler is assigned, the following code describes the way how Setup request is handled. It is very clear that a datagram socket is created for RTP communication with the client. `parse_RTSP_request` is a method to parse the incoming RTSP request from client to the understandable format.

```
while(!done) {
    request_type = parse_RTSP_request(); //blocking
    if (request_type == SETUP) {
        RTPsocket = new DatagramSocket();}}}
```

Once setup is done, the client may request with a file name to play the requested video. The playing of the requested video is handled from the below code.

```
if ((request_type == PLAY) && (state == READY)) {
    send_RTSP_response();
    if(running){
        //send back response
        playthread = new stream();
        playthread.start();
    }
    else{
        test.resume1();
        running=true;}}
```


It can be clearly observed a new thread is created for handling the play request from the client. Playthread is an instance of the `stream` class which will send the RTP packets of requested video to the client.

Similar to the client thread described, the Playthread also contains a run method from which the data packets will be transferred to the client. Following are three important methods to handle on play thread to act according the client request

- Run
- Pause
- Resume

In the Run method the frame bytes are send to the client through requested port. The following code describes how data is sent using UDP. This uses the `VideoStream` class to get the frames to be transferred.

```
image_length = video.getnextframe(buf,imagenb);
RTPpacket rtp_packet = new RTPpacket(MJPEG_TYPE, imagenb,
imagenb*FRAME_PERIOD, buf, image_length);

int packet_length = rtp_packet.getlength();

byte[] packet_bits = new byte[packet_length];
rtp_packet.getpacket(packet_bits);

senddp = new DatagramPacket(packet_bits, packet_length, client_ip_addr,
RTP_dest_port);

RTPsocket.send(senddp);
```

When a pause request is received in the client thread, `pause1` method in the in the following code is executed in the stream thread to pause the thread until further notice.

```
public void stop1() {
    keepRunning = false;}

public void pause1() {
    isPaused = true;}

public synchronized void resume1() {
    notify();}
```

Similarly, the streaming is continued by triggering `resume1` method in the stream thread when a resume request is received from the client.

Similar to play request other RTSP requests from the client are handled as follows in corresponding client threads.

```
while(true) {
    //parse the request
    request_type = parse_RTSP_request(); //blocking
    else if ((request_type == PAUSE) && (state == PLAYING))
    else if (request_type == TEARDOWN)
    else if (request_type == DESCRIBE)}
```

For every client request, client will be waiting for acknowledgement as RTSP is a TCP connection. So, for every RTSP request from the client corresponding response is sent using following code.

```
private void send_RTSP_response() {  
    RTSPBufferedWriter.write("RTSP/1.0 200 OK"+CRLF);  
    RTSPBufferedWriter.write("CSeq: "+RTSPSeqNb+CRLF);  
    RTSPBufferedWriter.write("Session: "+RTSP_ID+CRLF);  
    RTSPBufferedWriter.flush();  
    System.out.println("RTSP Server - Sent response to Client.");  
}
```

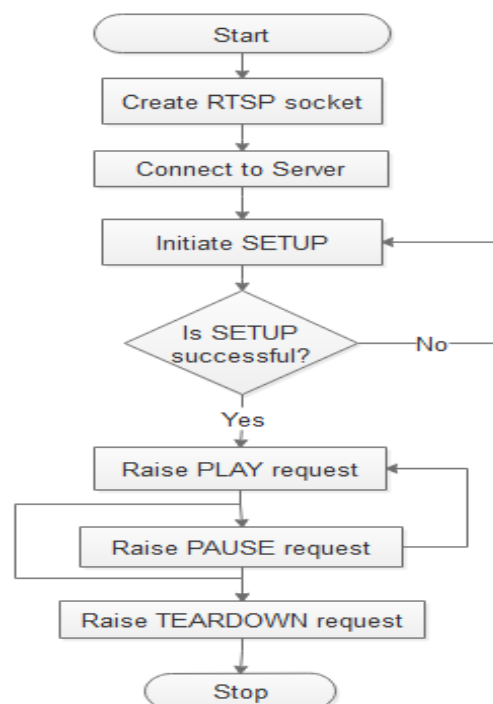
Along with the code few statistics were also maintained to trace the loss of the packets. So based on each transaction corresponding packet length, payload type, sequence number were displayed onto the terminal.

4 RTP/RTSP Client

4.1 Overview

The primary function of the client network application is to allow the user to stream and playback videos from the server. The client application is a standalone application developed using the Java programming language that supports Graphical User Interface (GUI) needed for user interaction. It raises a series of RTSP message requests to the server to establish an RTSP session, followed by play, pause and teardown events. Several different libraries have been utilized to implement the streaming functionality. Libraries (packages) include java.net for network sockets, java.awt and java.swing for GUI.

4.2 Flow diagram



4.3 Implementation

4.3.1 main() method

The main method is the entry point of the application. It takes the server IP address, server protocol port number, local protocol port number and the media file name to be streamed over the network as command line arguments. A new RTSP socket over TCP protocol is created using the socket API. Filters for Input and Output streams are created from `BufferedReader` and `BufferedWriter` classes. The initial state of RTSP is set to "INIT".

Stream Filters: The `java.io` package provides a set of abstract classes that define and partially implement filter streams. A filter stream filters data as it's being read from or written to the stream. The filter streams are `FilterInputStream` or `FilterOutputStream`. A filter stream is constructed on another stream (the underlying stream).

4.3.2 Media player GUI

There was a need of a media player as a standalone application to meet our requirements- for video playback. Hence we have used the Abstract Window Toolkit and Swing GUI library packages provided by Java to develop the video player. It is a basic player with a frame control, image icon control to display data frames, and buttons to control session, play, pause and close operations. Following are the GUI controls used in the client application:

- **JFrame**
A Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. The dimensions of the border area may be obtained using the `getInsets` method. It is implemented as an instance of the `JFrame` class.
- **JButton**
creates interactive button controls. Four buttons are created for setup, play, pause and teardown operations. The user generates different button events by clicking on the button control.
- **JLabel**
creates a label instance, initializing it to have the specified text/image/alignment. Three text label controls have been used for statistics and one label for image icon.
- **JPanel**
The `JPanel` class provides general-purpose containers for lightweight components.
- **ImageIcon**
Many Swing components, such as labels, buttons, and tabbed panes, can be decorated with an icon — a fixed-sized picture. An icon is an object that adheres to the `Icon` interface. Swing provides a particularly useful implementation of the `Icon` interface: `ImageIcon`, which paints an icon from a GIF, JPEG, or PNG image.

4.3.3 UI Control Event Handlers

UI Control events have to be handled appropriately for the user actions to be captured and to generate necessary response. In the media player, four buttons- Setup, Play, Pause and Teardown generate click events that have been handled by corresponding button event handler APIs as described below:

- Setup button event handler

```
btnSetup.addActionListener(new BtnSetupEventHandler());
```

```

class BtnSetupEventHandler implements ActionListener{
    public void actionPerformed(ActionEvent e){
        if (rtspState == INIT) {
            rtpSocket = new DatagramSocket(null);
            rtpSocket.setReuseAddress(true);
            rtpSocket.bind(new InetSocketAddress("127.0.0.1", rtpPort));
            RTCPsocket = new DatagramSocket(null);
            RTCPsocket.setReuseAddress(true);
            rtpSocket.setSoTimeout(5);
            packetSequenceNo = 1;
            SendRtspRequest("SETUP");
        }
    }
}

```

Initially, the RTSP state is set to “INIT”. An UDP socket (datagram socket) is created for RTP packet reception from the server. The socket is bound to the local IP address and protocol port. Then, a suitable timeout is set for the socket. Later, the RTSP sequence number is set to 1, which refers to the first frame of the video to be streamed and then an RTSP “SETUP” request message is sent to the server. The response from the server is parsed and validated. The RTSP state is set to “READY” if the call succeeds.

- Play button event handler

```

btnPlay.addActionListener(new BtnPlayEventHandler());

```

```

class BtnPlayEventHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        statStartTime = System.currentTimeMillis();
        if (rtspState == READY) {
            packetSequenceNo++;
            SendRtspRequest("PLAY");
        }
    }
}

```

This handler sets the start time for the current streaming session. The RTSP state must be 'READY' as a pre-requisite for the play button, which starts the video streaming. The RTSP frame sequence is then increased and an RTSP “PLAY” request message is sent to the server. The response of the server is parsed and the RTSP state is set to “PLAYING” if the streaming has begun successfully. The timer is started for the current session.

- Pause button event handler

```

btnPause.addActionListener(new BtnPauseEventHandler());

```

```

class BtnPauseEventHandler implements ActionListener {
    public void actionPerformed(ActionEvent e){
        if (rtspState == PLAYING){
            packetSequenceNo++;
            SendRtspRequest("PAUSE");
        }
    }
}

```

While the streaming is active and the RTSP state is “PLAYING”, a pause request would send a temporary halt request message to the server in order to pause the video streaming. Upon receipt of the “PAUSE” request, the server response is parsed and the RTSP state is reverted back to “READY”. The timer is halted to continue at a later stage based on the client request to resume the streaming.

- Teardown button event handler

```
btnTeardown.addActionListener(new BtnTeardownEventHandler());
```

```
class BtnTeardownEventHandler implements ActionListener {  
    public void actionPerformed(ActionEvent e){  
        packetSequenceNo++;  
        SendRtspRequest("TEARDOWN");  
    }  
}
```

The teardown button is similar to the pause button except that it requests the server to end the current streaming session completely. Hence, the RTSP state is reverted to “INIT” and the timer for the current session is stopped.

4.3.4 Sockets and Request / Response

The client application requires both RTP socket and RTSP socket. RTP socket is a UDP socket to intended to receive the RTP packets from the server that holds the media content. While RTSP socket is a TCP socket is used to generate RTSP requests to control the video streaming session. RTSP socket need not remain connected to the server constantly.

- RTP socket

```
DatagramSocket rtpSocket= new DatagramSocket(null);
```

- RTSP socket

```
Socket rtspSocket = new Socket(objRtspClient.serverIpAddress,  
serverPort);
```

- **SendRtspRequest** method

This method writes to the RTSP socket with the requested service message, to which the server is expected to respond. It uses the buffered filter streams for this purpose.

```
private void SendRtspRequest(String requestType){  
    rtspWriteBuffer.write(requestType + " " + mediaFile + "  
RTSP/1.0" + newLine);  
    rtspWriteBuffer.write("CSeq: " + packetSequenceNo +  
newLine);  
    switch(requestType){  
        case "SETUP":  
            rtspWriteBuffer.write("Transport: RTP/UDP; Client  
RTP Port: " + rtpPort +"; Client RTCP Port: "+ RTCP_RCV_PORT + newLine
```

```
);
```

- **ReadServerResponse** method

This method reads the response from the server to the requests made by the client. Based on the response appropriate handler is invoked.

```
private int ReadServerResponse(){
    int replyCode = 0;
    String status = rtspReadBuffer.readLine();
    StringTokenizer tokens = new StringTokenizer(status);
    tokens.nextToken();
    replyCode = Integer.parseInt(tokens.nextToken());
}
```

4.3.5 Frame Synchronizer

The frame synchronizer employs an image queue. It enqueues and dequeues the frames in proper order, based on the frame sequence index, which provides sufficient time and hence handles the synchronization of the frames.

```
class FrameSynchronizer {
    public FrameSynchronizer(int bufSize) {
        currentSequenceNo = 1;
        bufferSize = bufSize;
        imageQueue = new ArrayDeque<Image>(bufferSize);
    }
    public void addFrame(Image image, int sequenceNo) {
        if (sequenceNo < currentSequenceNo) {
            imageQueue.add(lastImage);
        }
    }
}
```

4.3.6 Timer Listener

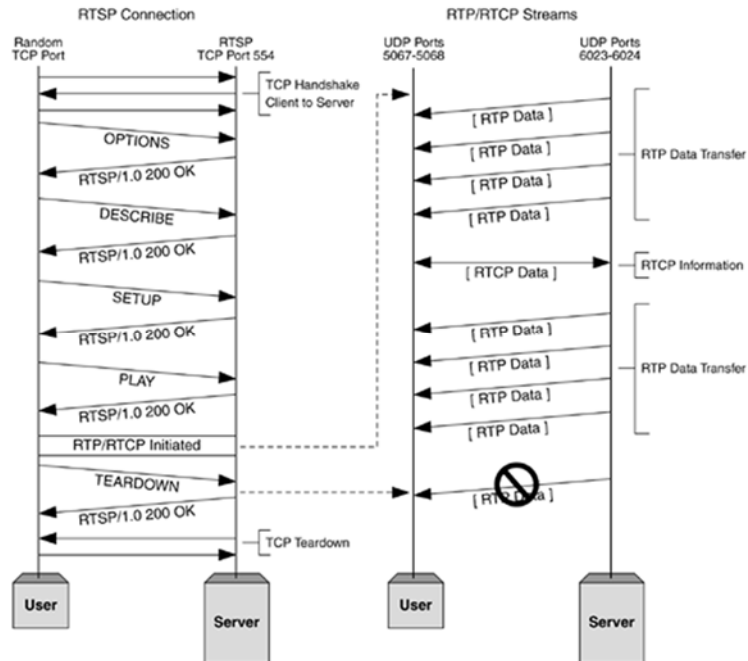
Timers are constructed by specifying both a delay parameter and an ActionListener. The delay parameter is used to set both the initial delay and the delay between event firing, in milliseconds. Once the timer has been started, it waits for the initial delay before firing its first ActionEvent to registered handlers.

```
class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        rtpPacket = new DatagramPacket(buf, buf.length);
        rtpSocket.receive(rtpPacket);
        double curTime = System.currentTimeMillis();
        statTotalPlayTime += curTime - statStartTime;
        statStartTime = curTime;
    }
}
```

The timer handler updates the total time of the streaming. It extracts the payload data and other information such as the frame sequence index, payload type, time stamp etc. from the received UDP packet. It also checks if the packet received is the latest and updates the statistics indicators like missed

packet count, total streaming time, streaming data rate, fraction lost, payload length etc. Finally, it displays the received frame as the icon in the GUI.

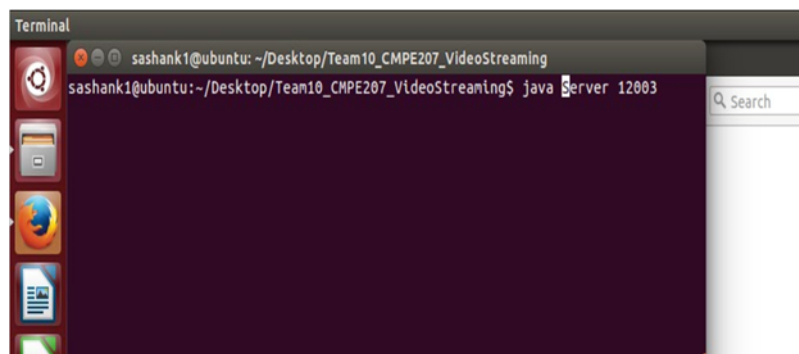
5 Client-Server Interaction



6 Project Execution in local Instance

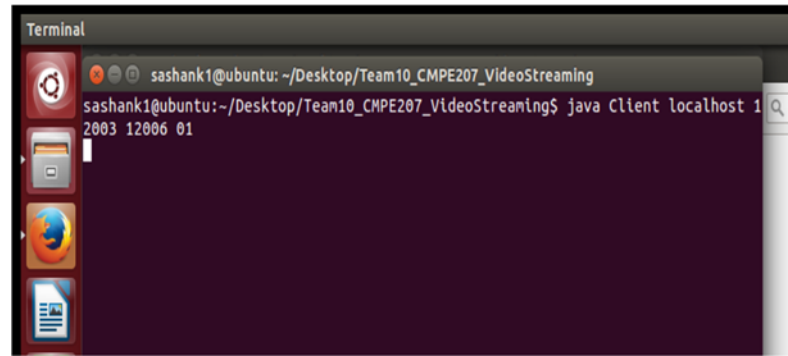
6.1 Running the server

Following Screen shot depicts running of server



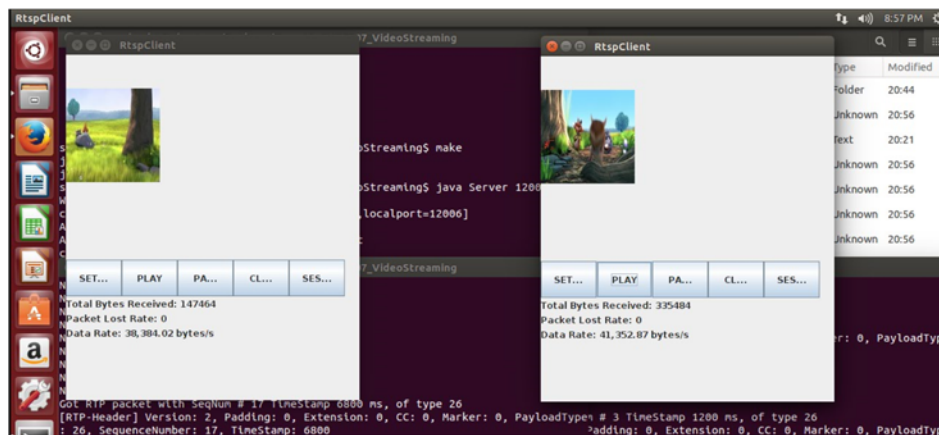
6.2 Running the client

Following Screen shot depicts running of client



6.3 Execution Result

Following Screenshot depicts execution of multiple clients



7 Deployment on cloud

To achieve meaningful utility from the video streaming application developed it is best to host the application in cloud. We choose to use Amazon Web Services (AWS) to host the application on cloud. Among services provides AWS we are using Elastic cloud service (EC2) to host our application as the functionalities offered by the EC2 are in line to our requirements.


7.1 Creating instances on AWS

To deploy the video streaming application on AWS at least two instances of OS are required to effectively check the working of the application. So two instances of Ubuntu have been created, one for client and another for server.

Ubuntu was chosen as the OS of our instance because of mainly two factor: 1) Robustness of the distribution 2) our familiarity with the distribution.



T2.micro type instance is selected as it filled out our requirements.T2.micro type of instance has 1 processing core CPU, 1 GB of memory (Elastic Block Storage).

	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate
---	-----------------	--------------------------------	---	---	----------	---	-----------------

To make things easier we have decided to use elastic IP addresses to our instances which lets us use fixed public IP address for the instances.

<input type="checkbox"/>	Address	Allocation ID	Instance ID	Network Interface ID	Scope	Private Address
<input type="checkbox"/>	52.37.88.140	eipalloc-f0670594	i-0c958958c05521...	eni-2ee95763	vpc	172.31.39.148
<input type="checkbox"/>	52.27.24.118	eipalloc-37690b53	i-025d8dd5b9ec90...	eni-1c57a251	vpc	172.31.43.234

Since the video streaming application we created uses RTP/RTSP protocols it uses both UDP and TCP streams of data respectively. So to use the instances we have to set the security group settings in EC2 such that it allows both TCP and UDP streams. Also RTP/RTSP protocols can be used using any port we specify. So to make sure that instances work no matter what the local settings, port used the security group settings of EC2 have been set such that instances allow all inbound/outbound packets from all the sources and ports irrespective of the protocol used for communication.

Description	Inbound	Outbound	Tags
-------------	---------	----------	------

Edit

Type 	Protocol 	Port Range 	Destination
All traffic	All	All	0.0.0.0/0

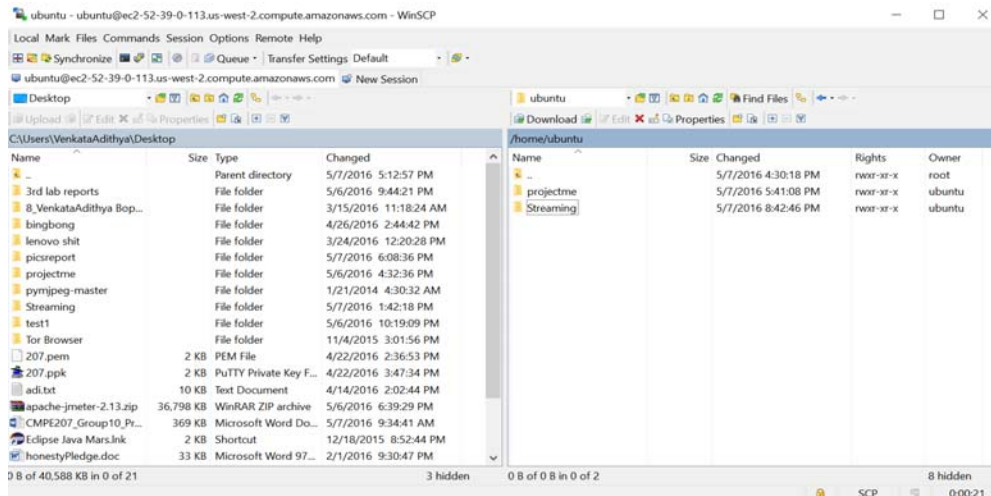
Description	Inbound	Outbound	Tags
-------------	---------	----------	------

Edit

Type 	Protocol 	Port Range 	Source 
All traffic	All	All	0.0.0.0/0

7.2 Transferring source code for deployment

After writing the source code for the video streaming application we are using winSCP application to securely transfer the code from the local machine to cloud instance. WinSCP transfers the code securely using SSH protocol from port 22.



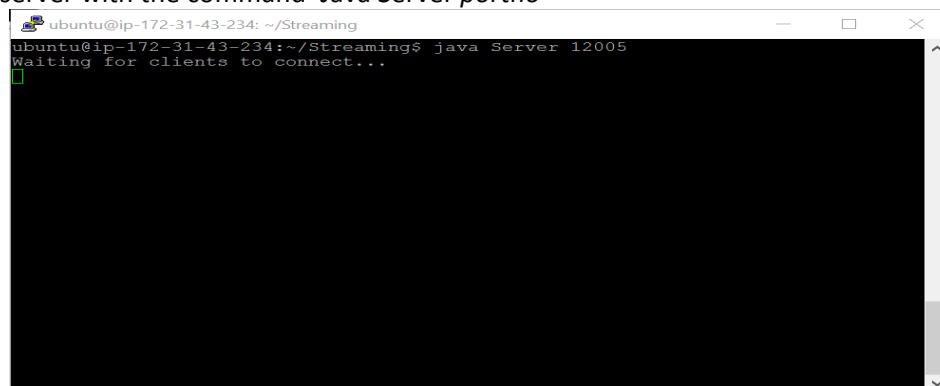
7.3 Using 'Xming'

Since playing video on the client requires a video player which cannot be achieved by putty. 'Xming' program is used. 'Xming' is an X11 display server for Microsoft Windows operating systems it can be used to run the GUI applications. It is used to display the video player application.

7.4 Running the application on AWS

As the video streaming application is written in java, java jre, jdk and other essential programs are installed onto the instance for deployment. By connecting to the instances using putty we started the server on one of the Linux instance.

We start the server with the command 'Java Server *portno*'



As the server running in the 1st instance supports concurrency we can interact with multiple clients and stream video concurrently to multiple clients. In the similar way as we connected to server instance with putty. We use two putty clients running on the same instance to issue a request to stream video to them.

As per the source code we can request the server to stream video by initiating the client code in the following way 'java Client *serverIP Serverport clientport video_selection*'.

Since server is running on the port 12005 and IP address of server is 52.37.88.140 in this particular scenario, so we ran 'java 52.37.88.140 12005 1006 1' for first client and 'java 52.37.88.140 12005 12006 1' for the second client.

```

ubuntu@ip-172-31-39-148: ~/Streaming
ubuntu@ip-172-31-39-148:~$ cd Streaming
ubuntu@ip-172-31-39-148:~/Streaming$ java Client1 52.37.88.140
Error: Could not find or load main class Client1
ubuntu@ip-172-31-39-148:~/Streaming$ java Client1 52.37.88.140 12205 1006 1

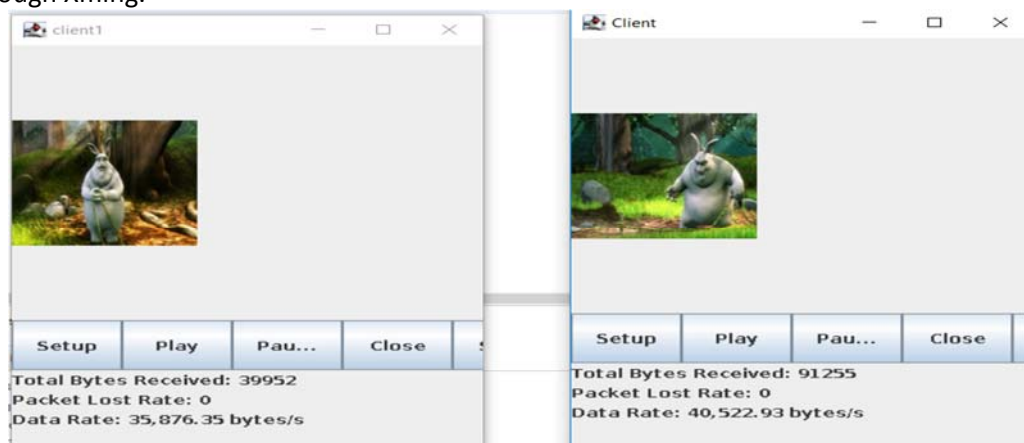
```

```

projects | driving | data structures | jobs | 207 | 208 | 294 | learning
ubuntu@ip-172-31-39-148: ~/Streaming
ubuntu@ip-172-31-39-148:~/Streaming$ java Client 52.37.88.140 12005 12006 1

```

Output through Xming.



8 Conclusion

This course project assignment has helped us understand the RTP and RTSP protocol functioning in providing video streaming service to the end user. Analysis of various statistics such as packet count, packet sequence number, timestamp etc. provided deeper understanding of the network system functioning. We could implement multi-threading to achieve concurrency on the server side using Java threads library to provide service to multiple clients simultaneously. We got an opportunity to learn about Amazon Web Services and its deployment methods. This project also improved our team collaboration skills.

9 Acknowledgement

We would like to thank Prof. Dr. Younghee Park for introducing and instructing various steps involved in socket programming and server side concurrency. Also, we would like to thank her group of research assistants for meaningful discussions.

10 References

- <https://tools.ietf.org/html/rfc3550>
- <https://www.ietf.org/rfc/rfc2326>

- <https://github.com/mutaphore/RTSP-Client-Server>
- <http://www.straightrunning.com/xmingnotes/>
- <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>
- A fully functional RTSP/RTP streaming server hello world example in C for experimentation with JPEG payloads. (n.d.). Retrieved December 6, 2015, from <https://www.medialan.de/usecase0001.html>
- How Streaming Video and Audio Work. (2007, October 11). Retrieved December 6, 2015, from <http://computer.howstuffworks.com/internet/basics/streaming-video-and-audio.htm>