**User Manual**

***Compiling and Installation:***

Standard GStreamer and GTK+ packages were used for implementation. Version 0.10.30 of Gstreamer was used.

***Server:***

Once both servers have started then they can be left alone. Nothing else is done with the servers unless some feedback is required. Then just observe the server terminal for session monitoring.

***Client:***

Connecting

Once the client opens you will have a UI with video controls and stream settings for both feeds. To connect to the servers you have to click the Play button which will connect to the server and start streaming the video.

Playback

Once the video has begun playing you can affect the playback in different manners. Each server has its own set of controls. There is a Pause button which will pause the stream until Play is selected again. There is a Rewind button which will start Rewinding the video. And there is a Forward button which will start Forwarding the video.

There is also a choice of resolution. You can decide whether you want to have a 240p resolution or a 480p resolution from a drop down menu.

Stream Control

There are a few options that allow you to control the flow of the stream. The first is the ACTIVE and PASSIVE modes. ACTIVE mode will play video at 25 frames per second or 15 frames per second depending on you bandwidth and play audio at 8 kHz. PASSIVE mode will just play video without audio at 10 frames per second.

You can also control the bandwidth. You type in the bandwidth available to you (in B/s) and click update bandwidth. This tells the server how much bandwidth you have available to you which affects how many frames per second you receive in ACTIVE mode.

**Development Documentation:**

***Server***:

The servers serve 2 pre-recorded files, one in 480p and another in 240p. The video for both is encoded in MJPEG at 30fps, and the audio is encoded using MULAW at 8000Hz.

The flow for video pipeline in both Active and Passive modes is as follows:

File -> Decoder -> Queue -> Video Rate -> Encoder -> RTP Pay -> UDP Sink

In Active there’s also sound, its flow is:

File -> Decoder -> Queue -> Audio Resample -> Encoder -> RTP Pay -> UDP Sink

The pipelines are built on connection, when clients are confirmed to have enough bandwidth and their networking information becomes available. The server runs different threads for incoming connections, serving media files, and listening for session control signals. The REWIND and FAST FORWARD signals from client cause server to send seek events in the pipeline, which eventually reflects in the client streaming session.

***Client:***

The client simply runs the GUI on initialization. On Play button click, it connects to the Server (IP address of the server must be specified as the first argument), and starts Resource Negotiation. If successful, it starts a GStreamer pipeline based on the current mode.

The Basic pipeline:

UDP Source -> Jitter Buffer ->

Tee J –> Video Queue -> RTP Depay -> Decoder ->

Tee D -> Queue -> Video Sink

-> Queue -> Video App Sink

-> Queue -> Jitter App Sink

The Active pipeline includes the Basic pipeline and an identical pipeline for Audio (minus the Jitter Buffer)

The Audio and Video streams n Active mode are synchronized using the RTP components. The Jitter Buffer handles QoS client-side. The App Sinks are used to monitor the streaming session.

**Resource Negotiation and Signaling:**

***Server Side:***

The server starts by first getting the current bandwidth available from a resource.txt kept in the project directory. The server then begins to listen through port 6000 and waits until a client connects. Once the server receives a signal from a client, it begins the resources negotiation by receiving the desired mode, frame rate, and resolution from the client, calculates the total amount of bandwidth needed to send this type of request, and reports an accept signal back to the client if there is enough bandwidth available, or a reject request if there is not enough bandwidth and closes the connection. If the accept signal was sent, the server opens a new port starting at 5001, incrementing upwards for each client that connects, sends the new ports for messaging, video and audio to the client through port 6000, and creates a new thread to handle that client. The main thread then goes back to listening through port 6000 for more clients.

In the newly created thread, the server begins by creating the pipeline, and waits for the client to send a play signal before starting the pipeline. While the client is still connected it continues to listen through the messaging port for any pause, stop, etc signals from the client. Whenever a switch mode or a change rate signal is received, the server goes through the resource negotiation process again, and checks if the there is enough remaining bandwidth for the new request. When the client wishes to stop the stream, a stop signal will be received, and the server will close the current messaging port, as well as stopping and cleaning up the pipeline. The server finishes by adding the bandwidth reserved to the client back to the available bandwidth and ending the thread that was dedicated to that client.

***Client Side:***

The client begins by getting the current bandwidth available from a resource.txt kept in the project directory. The GUI will start and the program will wait until the client hits play. Once the client does hit play, the current settings will be used to calculate the bandwidth needed to receive the request. If the client does not have enough bandwidth, an error is sent to the user reporting this to them, otherwise the client begins by requesting to connect to the server through port 6000. Once it has connected, it sends the desired mode, rate and resolution to the server and waits for either an accept signal or a reject signal depending on whether or not the server has enough bandwidth available. A reject signal will result in the client reporting a connection error to the client. If an accept signal is received, the client waits to be given a new messaging port, a video port and an audio port from the server. Once these are received, the client connects through the new messaging port, closes the old socket and begins setting up the pipeline, and setting it to play.

When the user changes either the mode or the rate while the pipeline is currently running, it first takes these new settings and calculates the new bandwidth required. If the client has enough bandwidth to support the new request, it sends either a change mode or a change rate signal to the server. Otherwise, it will report a resource error back to the user.

When the user clicks on any stream commands, such as pause, rewind or stop, a signal is sent to the server telling it what command was clicked and the server adjusts the stream data accordingly. In the case that stop was clicked, the client sends a stop signal to the server, then stops and cleans up the pipeline.

***Session Monitoring:***

The session monitoring works by creating appsinks at several places along the client’s pipeline. An appsink after the jitter buffer takes in buffers and uses the buffer’s timestamp and the current clock time to find the end to end delay between the server and the client. An event handler after the jitter buffer receives a do-lost signal from the jitter buffer that increments the total number of packets lost. The total packets lost is then divided by the total number of packets sent to find the packet loss rate. An appsink after the video decoder takes in buffers, increments the number of successful packets, and uses the buffer’s timestamp and the current clock time to find the time it took from the server encoding the buffer to the client decoding the buffer. An appsink after the audio decoder takes in buffers and uses the buffer’s timestamp and the current clock time to find the time it took from the server encoding the buffer to the client decoding the buffer. These two differences are then subtracted from the other in order to find the audio/video synchronization skew.