CS3102 Live Audio Streaming

Overview

The aim of this practical was to create a server and client capable of streaming audio over a network and save this audio to file. The program is written in java, using UDP as the transport layer protocol.

Usage :
As per the specification the server will not take a request from the client, only one song can be streamed and this is input as a command line argument. Running server.sh will initiate the server on a specified host, the subsequently running client.hs with the same host argument will connect the pair. Music will be streamed to the client, and an audio file will be output with the specified name.

This practical will stream and output to file simultaneously.

Design and implementation:

The communication is performed with Datagrams, the java implementation of UDP. The server will open up a socket and listen on this port until the client connects to it.

The client sends a hello message to initiate the communications, this allows the server to get its port and address information for further messages. The server opens the audio file so it can be read and sends the music format information as a message to the client. This is a necessary step as the javax sound library won't auto detect this, so it needs to be explicitly input to create a SourceDataLine which allows music to be played from a buffer.

The server uses a buffered input stream to read data, this was chosen as it simplifies the process of reading the file as we don't need to maintain a pointer ourselves. We read 2032 bytes per packet and as a 16 byte header, giving each packet a size of 2048 bytes. This was chosen as it is not so small that huge numbers of messages need be sent but no so large that individual packets being lost causes an issue.

These packets are then read into an ArrayList in order, this allows us to iterate over them and send packets in the correct sequence. This allows us to use piping to our advantage and send packets until we are told there is an issue. Originally I had implemented this with only the buffered input stream and sending messages straight away, and waiting for acknowledgements for each packet. This proved to be far less effective than the current version.

On receiving packets the client will determine if it is the next packet it was waiting for or if it is incorrect. If it is correct it will read it into a buffer for later usage, if not it will send a message to the server to say what the last correct packet it received. This will then arrive at the server and the server will reset back to its last correct position and start sending messages again. This proved to be simpler and more effective than sending which packets were missing and resending them, as it just requires that the server go back a bit then continue as before.

For the file output the program also ensures that the first packet, seq number 0, always arrives as it contains the header information needed to play the file.

An extra problem I had to solve was whether every packet needs to arrive, I decided that for streaming we don't need a perfect replica of the file, so we only request that the stream be reset to a

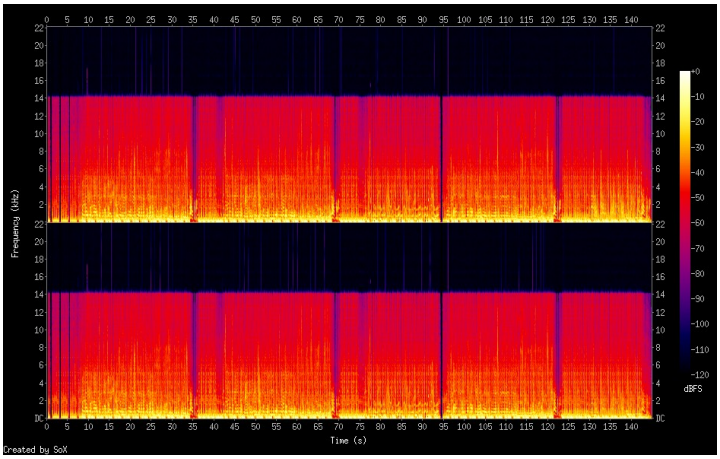
specific point once to prevent the same packet request causing the system to slow to a halt. The only exception to this is if the stream progresses massively ahead of where we expect it to be, I put this number at 100 packets as in practice it seemed appropriate, this number of course can be changed if further experimentation proves that another is more effective.

Once the server reaches the end of the file it sends and end message and closes the connection. On receiving this end message the client closes the buffers and writers and exits.

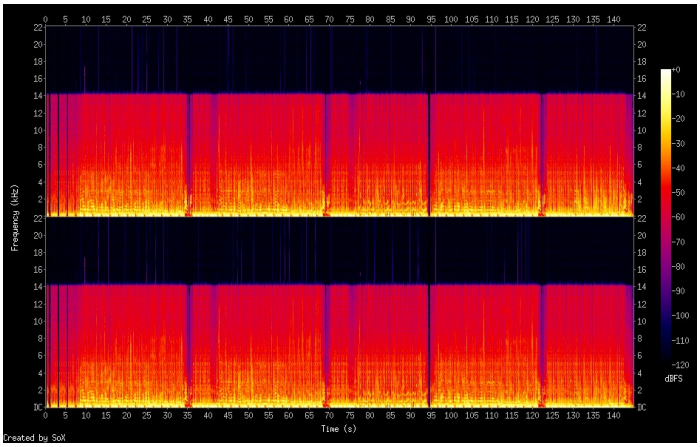
Testing:

For testing I went through all of the network emulation issues specified, listening to each I could describe how they affected the audio streaming, but this would only be qualitative and I felt it would be more appropriate to also include graphically representations of the output file. All images and audio files have been included with this submission.

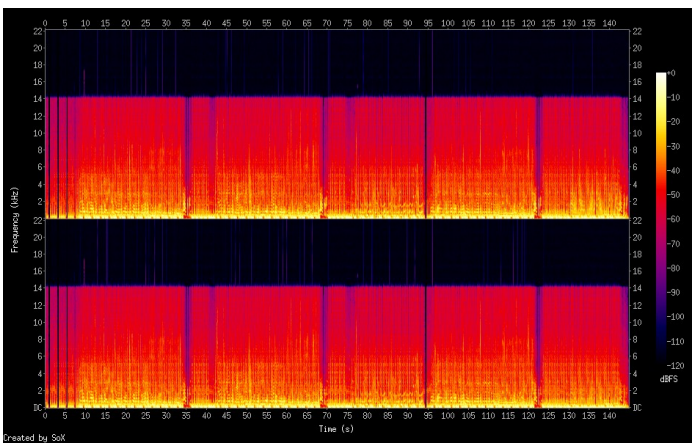
This first spectrogram is the audio file, for the purposes of this it is the Beastie Boys – Now get busy, which thanks to this practical I am over familiar with. I chose it because it is long enough to show that my program can handle large files without issue.



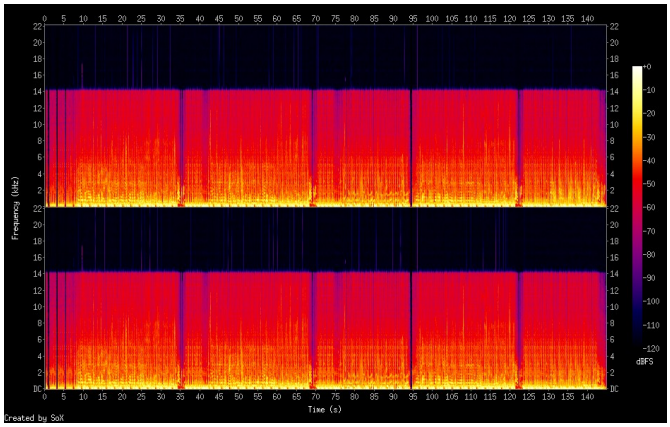
This second file is with 50ms of delay added to the network. It is practically identical, when I listened back it sounded the same as just sending over the network normally.



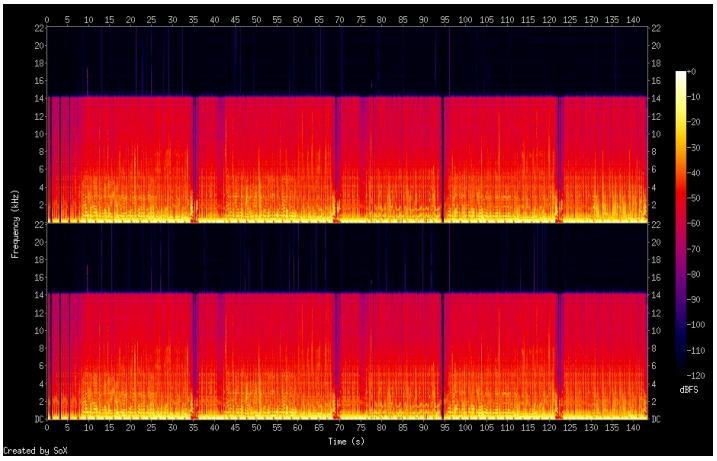
This third image is for jitter. Again there are not perceptible differences. It should be noted that playback is not as smooth, but does not have any distortion only minor pauses as the buffer refills.



The loss of 10% does have a few minor differences if you look extremely closely, but listening to it either on the saved file or the stream doesn’t sound any different from the original.



The playback for `delay 10ms reorder 25% 50%` sounds just like the original as well, and the graph looks identical to the above graphs which is exactly what I was hoping to achieve



Evaluation:

Initially I completed this practical by acknowledging each packet which arrived, this worked but caused huge amounts of delay in the system and made playback stunted. Having continued through the module and learning about piping I reimplemented the system and must say that it was a vast improvement on the initial implementation, but was frustrating to debug. Using a combination of ideas taken from lectures and research as well as applying common sense to specific problems I faced I am very happy with how this practical turned out.

I believe it covers both the basic specification and copes with the various hurdles a network may throw at it.