Ray Smets
3/16/14

Silent Disco Project Report

How to run project:

To run the project one only needs to execute the SDserverApp.jar on the machine chosen to act as the server, and the SDclientApp.jar on any number of machines one wishes to uses as clients. The three requirements are that Java is installed on all the systems, the local network allows multicast packets, and the machine used for the server has a microphone for capturing audio.

Report Preface:

I included what the file structure of the program for your reference under *File Structure* in the Appendix.

Depth of what was accomplished:

The end result is a stateless multicast server / client application that sends / receives and plays audio from the server's microphone. This is a different from my original goal of playing audio from the line in port – the reasons for this are discussed later – however the functionality of the app is more accurately described as a VOIP broadcast system instead of a silent disco.

Server: I created a threaded server with a custom built AudioGrabber class to capture audio from the microphone 512 bytes at a time and a Server class to handle sending of packets to multicast group of clients. I utilized Java's standard .net and .sound APIs built all audio and networking functionality myself, however I did use "Beads", a java library for real-time audio, to determine the buffer size optimal for the audio being captured. This helped ensure respectable audio quality and minimum latency over the network. Although I could've made the project easier on myself by using such a library to do audio capturing and playing for me, I choose to avoid such libraries for educational purposes.

Client: The client app was also built threaded with a Client class to handle receiving the packets and AudioPlayer class to play the audio. Again all functionality was built and implemented myself over standard Java libraries.

Packets: Used UDP to tunnel a custom made RTP packet containing information regarding the audio being sent from the client in addition to the raw audio bytes.

How it works:

Please refer to the Appendix *File Structure* for reference.

I used a java concurrent interface, a BlockingQueue, shared between the server's AudioGrabber class and the server's Server class and between the clients' AudioPlayer class and the client's Client class. This prevented the race condition of the server trying to send audio packets faster than the AudioGrabber could grab the audio bytes. The BlockingQueue functionality served a similar purpose on the Client side between the receiving of the packets and playing of audio. Because the BlockingQueue interface only holds objects I implemented a shared ByteArrayContainer class to hold the raw byte arrays of audio data that I was handling. Inside the ByteArrayContainer I also included a timestamp of the time of creation for debugging and performance analysis purposes.

Additionally I created a shared AudioFormatContainer class and a Custom RTPpacket class. The AudioFormatContainer was implemented for the server's and client's classes to have a shared knowledge of the audio format amongst themselves. This included information such as the sampling rate, frame size, channels, signed or not, and endian considerations. Because I was grabbing audio from a DataLine and not an audio file with a set format I needed to specify this myself. Because the client needed to know this information I included the audio formatting in the custom RTP packet header, which the client could then used to create a shared AudioFormatContainer object to keep track of the audio format between its two threads. Additionally the RTP packet header includes a sequence number used to determine if a packet needs to be dropped due to out of order delivery – a common practice for real time audio streaming.

Functionality Impemented:

Due to only having the application fully functioning with microphone audio the audio format could be implemented on the client before receiving the packet. This allows me to start the audioPlayer thread simultaneously with the client thread, which achieves minimum latency between audio capturing and playback. However for the sake of added networking complexity for the project I have implemented the functionality for the client to be totally unaware of the audio types the server will send prior to starting. However to demonstrate the minimum latency implementation proformance I have included an additional client jar, SDclientAppQuick.jar, which demonstrates the client's ability to play audio without setup overhead, thus with less latency from the time of speech on the server end. The performance difference of ~250ms is best observed when running both server and client jars on the same machine. I have also included a diff of the source code for implementation.

Because I used multicasting clients were able to join or leave the live microphone audio feed without any networking overhead.

Protocols Implemented:

The RTPpacket class I implemented had a header and "payload" section. The header included sequence number, sampling rate, frame size, channels, signed or not, and endian considerations. Because I also began working on implementing sending of audio from mp3 files the header included an encoding flag to signal if an AudioFormatContainer should be created to match a certain audio file type specifications. I ran out of time to finish this implementation, however the encoding field was left in the RTPpacket header anyway, demonstrating the protocol is capable of functionality not yet implemented from the audio perspective.

Challenges overcame:

To implement the intend functionality of broadcasting music audio from any source I ideally should have had a device with a line-in mixer line to use as the server. Unfortunately my macbook does not support this functionality, which is what lead me to grabbing audio from the microphone. Due to the mediocre quality of sound from the microphone I used a 64kbs mono encoding (8000 sampling rate, 8 bits per frame) which I feel is adequate for VOIP.

To grab the microphone I had to get go through the machine's mixer channels. I found some operating systems or soundcard driver APIs do not provide information on the type of the available mixer channels. In these cases, a Java Sound implementation cannot match mixer channels with pre-defined Port types, such as MICROPHONE. Thus the way I grabbed the DataLine audio from the microphone had to be in a slightly roundabout way. I queried the Audio System's Mixers, grabbing each one's name, made it lowercase, and looked for one that included "mic". While this has worked on all test devices I feel it maybe the primary suspect for errors when running on the TAs test machine.

Additionally the audio processing capabilities vary from machine to machine, which caused the quality of audio to vary greatly based on test machines. When running the client one can observe the amount of delay from time of receiving the packet to time of playing by the timeCreated attribute of the ByteArrayContainer object. I printed this time difference to the console on the client for one to determine if latency is introduced form the network or audio processing of the machine.

Performance Analysis:

Upon testing on multiple OS X and Windows machines the broadcast system works with variable audio quality. This is due to the variance in machine capabilities as used as described above. The latency averaged around 500ms, acceptable for VOIP audio. It was also noted that because only used on a local network with a single router the UDP packets very rarely arrived out of order and none were observed to be lost. However the nature my implantation to deal with the case of out of order delivery drops one packet out of every 256, which is unnoticeable to the human ear.

Things that could be improved / implemented:

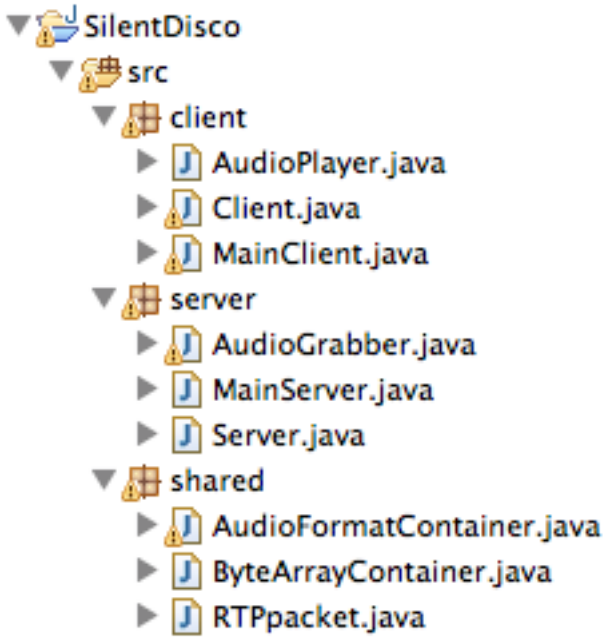
- Develop the app on a device with line in capabilities.
- To get around the line in limitation I could've used a program such as JackAudio to emulate a line in my device
- To implement playing audio from any source directly, not from line in, I could make an audio output device interface that plays audio from any source (Spotify, youtube, itunes, etc) and feed into JackAudio to emulate the line in my device.
- If I was able to capture music audio from these sources the higher bit rate would force the need for compression to reduce congest the network.

Final Remarks:

Gained considerable experience in audio manipulation, creating custom protocols, and how to tunnel a custom protocol through a typical transport layer protocol such as UDP. If I could do this project over again for this course I would start with reading the Java Sound FAQs sooner than later. While the API is decently documented its shortcomings would have been easier to avoid. Additionally, due to my interest in developing a fully functional silent disco project I hope to totally re-implement this project using networking media frameworks and audio libraries to aid in implementing functionality desired. I feel building an almost functional silent disco app without using frameworks and libraries will benefit me greatly in the future to understand what's going on under the hood of many sound and networking APIs.

Appendix:

File Structure:



Changes in source code to for lower latency functionality at the cost of dynamic audio formatting on client side:

6 src/client/Client.java

```
@@ -54,7 +54,7 @@ private void receiveMulticast() throws Ex
54  54          byte[] receiveData = new byte[525]; //1000 for ser
55  55
56  56          //figure out what audio format to use from header
57      -       determineAudioFormat(receiveData, socket);
    57  +       //determineAudioFormat(receiveData, socket);
```

4 src/client/MainClient.java

```
@@ -28,10 +28,10 @@ public static void main(String[] args) throws Unsu
28 28
29 29                 //would use below audioPlayer and start its thread if
30 30                 //it would decrease delay but doing this later allows
31    -              //AudioPlayer audioPlayer = new AudioPlayer(audioQ);
   31 +              AudioPlayer audioPlayer = new AudioPlayer(audioQ);
32 32
33 33                 new Thread(client).start();
34    -              //new Thread(audioPlayer).start();
   34 +              new Thread(audioPlayer).start();
35 35         }
```