Project 4 Report

M Shepherd, 19059020@sun.ac.za
M von Fintel, 20058837@sun.ac.za

21 September 2018

Voice over Internet Protocol

Contents

1 Introduction

Voice over Internet Protocol (VoIP) is a method and group of technologies that allow for voice transmission over Internet Protocol (IP) networks. This enables two users to communicate via voice, without having to use a public switched network. The digital information is packed into packets and is transmitted via IP.

For this project, it was required to create a VoIP system implementation to enable multiple users call each other over the local network.

2 Project Overview

As was said, the aim of this project was to implement a VoIP system. Many clients could connect to the server and could call each other over VoIP. The program was implemented using Java sockets. Users could call each other, make conference calls and message each other.

2.1 The Server

The server coordinates all the user activity. Each user connects to the server via a client. Many clients can connect to the server at the same time. A `clientInstance` is made for each client that connects. This instance of each client controls how the clients connect and disconnect. It also keeps track of the clients' behavior.

The server also sets up the connection for calls, group calls and voice notes. Once the connection is set up though, the clients communicate amongst each other and the server plays no part in the communications, apart from termination of calls.

2.2 The Client

The client is the interface to the user and allows the user to connect to other users via the client. The client provides the interface and the infrastructure to make calls, chat via text and send voice notes with other clients that are connected to the server.

Once the user has specified the IP address of the server they wish to connect to, the client connects to the server and then waits for incoming messages, voice notes and calls from other clients.

3 Features

Clients can send global text messages to all clients connected to the server. They can also text message individual clients with a function known as whispering.

Clients can send voice notes to each other.

Clients can make groups from users that are connected and can text message and send voice notes in these groups. Voice calling is also an feature in these groups. All clients in the group are able to speak and take part in this, hence it is known as conference calling.

3.1 Extra Features

Users can private text message each other, known as whispering.

Users could specify a unique nickname, on top of just using hostnames. This made the appearance in the client GUI neater and more professional. It also made it easier to identify between clients.

3.2 Unimplemented Features

Java mixers was not used to remove echos, distortions and dead zones in the calls.

4 Description of Source Files

Most of the development of this project was done in Netbeans. This means that everything is stored in the Netbeans project format. The source files are located in the package folder within the src folder.

There are four source files:

`SenderPane.java` contains all the code for the server. This creates the initial connection and creates a `clientInstance` for each client that connects.

`clientInstance.java` is the class that controls the server side interactions for each client. The connection is created and stored. It listens for incoming messages from other clients and sends outgoing messages to the client it represents.

`ClientPane.java` is the interface that the user interacts with. It provides all the infrastructure for calling, voice notes and text messages.

`sendPacket.java` is the class that sends voice notes and call sound. It sends byte buffers using UDP.

5 Program Flow Description

Once the server is up and running clients are free to connect. When the client is run, the user is prompted for the IP address (host) of the server to connect to. The client the connects to the server. The clients can then chat.

To make a call, users must make a group by selecting the users or users they wish to speak to from the list of connected clients. The call is then initiated by pressing the start call button and stopped by the end call button.

6 Experiments and Testing

6.1 Text Messaging

Global Chat
We tested the global chat, by connecting to our server with three clients. We sent messages from all users concurrently and monitored the receiver box on all sides.

The messages came through as expected, which shows that our global chat behaved correctly to our expectations.

Group Channel Chat
We tested the group channels by connecting to our server with four clients and creating two groups with two members each. We then sent messages to the groups from all clients to confirm that the correct clients were receiving the correct messages and that only the correct clients were receiving the messages.

We found that our Multicast sockets behaved as expected and that the correct clients received the correct messages. This also shows us that our channels work correctly.

6.2 Voice Notes

Recording and playback
To test this, two clients were connected to the server and they were both added to a group. To test this correctly, we would need to compare the local playback to the playback on the receiver side. This was done by first recording the voice note and then playing it locally, after which it was sent to the other user and played on their side.

This experiment was repeated 5 times for accuracy and we determined that our voice notes were sent correctly and the sound quality on both ends was too similar to find any differences with the naked ear. This voice quality did not display any echos, distortion or dead zones, which showed us that our recording, UDP packet forwarding and playback was working correctly and exceptionally.

Voice Notes and Text Chat
To test this, we repeated the above `recording and playback` tests, but also sent messages through the group channel concurrently.

The results of these tests showed us that our threading correctly separated the tasks and allowed voice to be recorded and played back while sending and receiving messages in a group and globally.

6.3 Voice Calls

One on One
This was tested by connecting two clients to our server. The clients were then put into a group and a voice call was started. We then monitored the voice quality and delay on both sides.

In this experiment, we could clearly hear each other talking and could clearly discern separate words, but the quality was compromised by a slight echo. We did note that there were no dead zones, nor was there any distortion. We also experience a very small delay, but as this delay was smaller than one second, it can be seen as negligible. This showed us that our voice transmission had small issues, but we were unable to fix these issues in time for the project demonstration.

Three Clients
This test was almost exactly the same as the One on One test, but with another client added to the group.

We found that the audio quality and delay was exactly the same as in the previous test. This showed us that our project scaled correctly with the transmission and receiving of voice.

Voice Calls and Text Chat
To test this, we ran the previous two voice call tests, but sent messages concurrently from all clients in both the group and global chats.

The results of this test showed us that our threading of sending and receiving voice was done correctly, as there was no interruption to the voice call on any sending or receiving of messages.

6.4 Audio Format

In this section, we experimented with different Sample sizes and bit sizes to find the clearest and most efficient combination.
From this table, we can see that the best audio qualities were found at the sample sizes of 8000 and 44100 bits with bit size 16. We chose to use 8000 as

Sample Size	Bit Size	Observation
8000	8	Constant static
8000	16	Clear quality
16000	8	Slight distortion
16000	16	Constant distortion
44100	8	Slight Static
44100	16	Absolutely Clear

Table 1: Audio Format Permutations

our sample size, as we expected this to cause less problems in the transmission of bytes.

6.5 Conclusions

In conclusion, it can be said that we have comprehensively tested and experimented with our project.

The audio format experiments allowed an informed decision to be made with our specific audio formats and the tests showed us that our data structures and implementations have been executed correctly and satisfy the brief of the stated problem.

7 Issues Encountered

The main issue encountered was the lack of time we had to fully research everything and make sure our implementation worked perfectly. We managed to get a basic implementation running, but it would have taken more time than we had available due to other projects and tests.

There was an initial struggle to get the users to be able to talk concurrently to each other. This was overcome by sitting and eliminating bugs in our code. The final result was that users could talk at the same time.

8 Algorithms and Data Structures Used

8.1 Algorithms

Our notable algorithms would be the algorithms that we have created to record, playback, send and receive voice.

Recording
For recording a voice note, we read from a `TargetDataLine` from the audio input into a temporary buffer. We then write that temporary buffer to a

`ByteArrayOutputStream`, which is then available for playback.
For transmitting voice, the bulk of the work is done by an almost identical algorithm. The difference is that an instructional header is added to the temporary buffer, and the buffer is then sent to all users that subscribe to the `MulicastSocket` that the current user is subscribed to.
Every instance of recording is done in a seperate thread.

Playback
For our playback algorithm, we created `SourceDataLine` attached to the audio output. This algorithm then takes the given input stream and iterates through it, writing to the `SourceDataLine` until the input is exhausted. This is done in a seperate thread to a separate `SourceDataLine` for each instance.
For voice notes, we send the entire `ByteArrayOutputStream` into the playback method, while for voice calls, we send each packet that we receive in separately and sequentially so that it appears as if there is non-stop transmission.

8.2 Data Structures

Java TCP sockets were used for the connection between the server and the client for text messages and alerts. This was implemented using Java's `Socket` and `ServerSocket` classes.

To send voice notes and voice packets in calls, UDP sockets were used. Java's `MulticastSocket` was used for this, as it allowed for easy grouping of messages.

The voice bytes are stored locally in `ByteArrayOutputStreams` and are converted to and from standard byte arrays with instructional headers for transmission.

`javax.swing` was used for the GUI.

9 Compilation

From the commandline, go to the folder where the source code is located, as is indicated in the README. Compile all the files by using the `make` command.

10 Execution

To run a server, find the file called `run_server.sh` (located in the same folder as the source code) and run it by typing `./run_server.sh`. This must be done before a client attempts to connect.

To run a client, find the file called `run_client.sh` (located in the same folder as the source code) and run it by typing `./run_client.sh`. Enter the host IP.

11 Conclusion

The aim of this project was to implement a VoIP system. The aim was successfully achieved as a basic VoIP system was implemented in Java.

This project helped us to learn how the theoretical concepts work out in the practical side of things. We are both now confident in programming these sort of applications now, which fulfills the overall aim of the project.