Group 14 Project 4: VoIP

Dylan Callaghan: 21831599@sun.ac.za
Stephen Cochrane: 21748209@sun.ac.za

October 22, 2020

1 Introduction

The goal of this project was to implement a full VoIP application with both voice and text communication. The communication was to be between one or many clients in the form of personal chats or conference calls (and chats). All calls were to be done direct from client to client, with a server for only messages and setting up of communications. As said above, the calls were to include one or multiple clients, and the quality of these calls was to be improved by minimising echo, distortions, and noise.

2 Unimplemented Features

3 Additional features

- In addition to the IP addresses used for each client, a nickname assignment was added to allow for personalization of the client Gui. This nickname is transparent to the user, and is translated to the IP addresses when used for sending over the applications network.

4 Description of files

- Client Holds all logic for the communication between the separate client functionalities. This file contains methods that are called by other components of the client, and calls components in these methods, providing a bridge between the different aspects.
- ClientFuncPass The abstract interface that the Client uses to facilitate two-way communication between it's components.
- Chats This class represents the object that holds all the chats of a user. The chats include any number of single user chats or groups, made from conference calls. The class contains methods for concurrently updating the list of chats and its members.

- Group This class represents a group object. The group object holds multiple users which are connected to a certain group/conference call. The group has a root member which is used to facilitate the call.
- User Class for the User object. This object is a simplistic structure mainly used for Gui displaying and for the transformation between nickname and IP address.
- Gui The Gui component of the client. This is a modular Gui which communicates with the client when receiving or sending updates. The Gui handles only drawing of the client graphically and does not have any logic for the client.
- JGroup A class used by the Gui to display multiple strings concatenated together, where each string can have a different colour.
- AudioHandler The audio component of the client. This class handles all audio related functions, which includes input from the microphone, and output to the speakers. Components wishing to interact with the audio of the computer must interact with this class using byte arrays.
- ByteArrayIOStream This class represents a custom object used for the purpose of storing the byte arrays for audio in the project. The object is an extension of both the InputStream and OutputStream in java, and includes all the methods for both of these. The purpose in this, is to have a structure that can exactly control bytes being written to and read from the stream, used as a buffer for the application.
- MultiStream This class facilitates a collection of ByteArrayIOStreams, and for the specific purpose of the audio handler, allows for writing to multiple streams, and reading (and merging by audio) from the streams. The reading is done by reading from each ByteArrayIOStream, and the output is merged using DSP into one signal to be sent to the audio handler.
- ServLink This class provides a link to the server, acting as a form of structured message passing, using the `send` and `recv` functions. ServLink communication has a guarantee of a send, and recv is blocking. Servlink just sends and receives streams of bytes, and hence makes use of the helper class `Message` to assist in this.
- Message The `Message` class acts as a wrapper for byte arrays allowing for easy creation of messages (or reading of) for the `ServLink`. This class has 3 fields, in order they are:
  - tag The tag specifies the type of message.
  - len The length of the payload.
  - payload The payload being sent.

- GroupMessage `GroupMessage` Allows for easy encoding a `Group` into a `Message`, and also decoding a message into a `Group`. When we want to send a group message (i.e we are encoding a `Group`) we can construct the `GroupMessage` and then use `sendAll` to send the `GroupMessage` to everyone in the group. Conversely, we can pass a message into `GroupMessage` to obtain a `Group`.
- UDP This class allows for easy sending and receiving of UDP packets, this class is only used for sending VoIP packets to other clients.
- Server This class is used to run the server, when started it blocks on accepting new client connections. When a client gets accepted a handler process gets forked (and the server will continue blocking until a new client attempts to connect), and all this process does is forward messages from one client to other connected clients.
- ServerGui A gui for the `Server`, it flushes events that occur to the gui as well as to standard out.
- optargs Simple argument parsing, makes handling command line arguments trivial. Very similar to `getopt` in C.

5 Program description

For the client, we implemented a mediator design pattern to facilitate communication between separate modular components of the system. The mediator in our case was the general client class, which had components:

- Audio handler
- Server link
- Gui
- UDP object

The client had a general interface which it used for two-way communication between each component. Each component was therefore responsible for it's own functionality, and passed "messages" to other components when they had finished their tasks. This allowed for a very simplistic design, which allowed us to abstract away the full functionality of the system, and just consider the individual modular components. Each component was responsible for interacting with it's part of the system, and in a sense communicating with it's object. In the case of the Gui, this was the graphical display, the audio handler communicated with the computers audio, the Server link with the server, and UDP with the other UDP client. We used a very simplistic server, that just facilitated communication and sent messages. The server only receives messages from clients and passes them on to other clients. These messages can either be text messages,

user joining or leaving requests, or call configuration messages. In this way, the server was used as a look-up and facilitator to the connection between clients. It deals with all set up between clients whether it be for calls or for connecting to the server. For client connection, the server forks a thread that will set up a connection with the client and maintain this. The server also notifies all other clients of the new user. For calls, the call initiation messages are sent through the server, as well as updates when a client leaves a call, and when the root user updates the call information.

6 Experiments

6.1 AudioFormat sample rate

6.1.1 Expectations

It is expected that increasing the sample rate for the call will increase the quality of the call as the call would have more accurate data to work with.

6.1.2 Findings

The findings were different between local and hamachi testing. On the local network the audio quality increased when the sample rate was increased. This was detected as a clearer call quality, with less static and noise. On the hamachi network however, the audio quality decreased with an increase in sample rate. This was detected by more noise, and less audibility.

6.1.3 Conclusion

This led us to the conclusion that the sample rate increase provided more accurate data, and therefore on a local network (or local machine) provided better quality audio. However, when sending packets over hamachi, the network was slower. This meant that the increase in sample rate (which led to an increase of data size) will mean that less packets will go through on the slow hamachi network, some maybe not going through at all. This would explain the audio quality decreasing over hamachi when the audio sample rate is increased.

6.2 Input method change

6.2.1 Expectations

We expected that the features added to allow the user to change the audio input method would give the user multiple input methods that they can switch between, and decide which one works for them.

6.2.2 Findings

On startup, on devices that had different defaults set that may not have worked, the user was able to change their input method using the framework we set up. In most cases, this caused devices with audio not previously working to work if the audio method chosen was the correct one. In some cases, however, the change did not help, and this occurred mainly when experimenting over the hamachi network for calls.

6.2.3 Conclusion

The conclusion for this result is that the audio input method changing that was set up successfully changes the audio method. We also found that on most devices, the "default" method was the best (i.e. working and best quality) for input. The outlying cases with experiments involving hamachi could be attributed to other external factors such as the speed of the network, or firewalls in between devices, and so we concluded that the audio method was not the cause of these problems.

6.3 Call quality over hamachi

6.3.1 Expectations

The hamachi network is not substantially fast, and so calls over the hamachi network were expected to be somewhat different to those over a local network. For this reason, some delay and maybe jittering was expected.

6.3.2 Findings

The call quality over hamachi proved to be quite stable and good. The voice data had very little delay on average and little to no jitter. However, that being said, some calls over hamachi would, for a varying amount of time at the start of the call, not successfully transmit the correct audio packets from one user. This was possibly attributed to hamachi due to the call working perfectly over local networks. Also, at certain times in certain calls, the audio data would be delayed, which would cause the delay in the call to worsen.

6.3.3 Conclusion

These results showed, on average, that the call quality over hamachi was quite stable and good. However, the findings about calls not transmitting data from one user seem to conclude that hamachi, although having its benefits, has downsides too, those being some unpredictable behavior in certain cases.

7 Issues encountered

The only issues we encountered was that of hamachi being hamachi (that is insane packet loss and or connection issues), as well as Dylan Callaghan's mic not always working. We solved this issue by allowing the client to select which mic to use. This is achieved by first typing `/l` will list all devices, and then typing `/a <num>` to use the device labeled as `<num>`.

8 Significant Data Structures

- ByteArrayIOStream:
  This structure is used as an Input/Output stream, and functions as a thread safe buffer. The way this structure works is by having a byte array with various pointers that can be read from and written to (thread safely). The array can be resized, and wraps around after data has been read. This way, the array always has enough size to store the data, and only uses as much as it needs. The structure has two main functions:
  - Writing arrays of bytes to the structure
  - Reading arrays of bytes from the structure

  These functions can be done with any size of array to read/write. In this sense, the stream is both an input stream (allowing for reading of bytes how an input stream would), and an output stream (same for writing). All the other methods in the ByteArrayIOStream are variants of these functions, or helper methods for them. For instance, for reading, there is both a blocking and non-blocking function to read from the structure (based on whether there is enough data to read).

9 Design

- Roots in calls:
  The calling mechanism in our project relies on the fact that a single person is in charge of a call so that any configuration messages can go through them. Say for example, if someone joins the call, the root must be asked to accept the request, and then should tell the other users about this change. The root of a call is initially set as the user who is being called (when the call is just between two people). This means very little for a two-person call as the root user has no real other tasks. However, when someone else joins the call and so creates a conference call, the root attribute takes effect. The most important thing is that a root user is the only user that will send out group messages (configuration messages). This is because the root will be the only user who knows of all the activity in the conference call. In addition, the following conditions apply:

- Any call requests received by anyone on the call will be forwarded to the root.
- Any user leaving the call will notify the root user that they are leaving
- The root user is the only user able to change the name of the group (on starting the group)

On receiving any of the above changes, the root user will update it's group object, and then broadcast this update to the rest of the group.

- Passing of root in calls:
  The root mechanism allows for effective message passing, but presents a problem when the root user leaves a call. For this situation, we have a mechanism in place to transfer the "root" attribute between users.

  This works by having the root user update their group object to exclude them as root of the group, and allocate another random user as the root. They then broadcast this group update message to all users as their last message to the group. When each group member receives this message, they will update their root to be the new root, and remove the root user from their list. If they are the user which is the new root, they will change their state to reflect this. Since all users in the group receive this message, the new root will now receive any changes, and can pass these along in group messages.

10 Compilation

```
$ make
```

11 Execution

11.1 Server

11.1.1 Running

```
$ ./server [options]
```

11.1.2 Help

```
$ ./server -h
```

11.2 Client

11.2.1 Running

```
$ ./client [options]
```

11.2.2 Help

```
$ ./client -h
```

12 Libraries used

- java.awt.*;
- java.io.*;
- java.lang.String;
- java.net.*;
- java.nio.ByteBuffer;
- java.util.ArrayList;
- java.util.HashMap;
- java.util.Hashtable;
- java.util.Iterator;
- java.util.LinkedList;
- java.util.Objects;
- java.util.Scanner;
- javax.swing.*;