4/24/2023

Project 3

VOIP Chat, Group 17

Computer Science 313

Clinton Elves, Jayden Abrahams, Cindi-Ann Jeffery, Langton Chikukwa, Konanani Mathoho

Members

Name	Student Number	Email	Roles
Clinton Elves	24007676	24007676@sun.ac.za	UDP
Jayden Abrahams	23559322	23559322@sun.ac.za	TCP
Cindi-Ann Jeffery	24695823	24695823@sun.ac.za	Project leader
Langton Chikukwa	23607769	23607769@sun.ac.za	GUIs
Konanani Mathoho	24774898	24774898@sun.ac.za	Tester

Introduction

This project is a VoIP (Voice over Internet Protocol) and Chatroom program designed for different users to make use of the Client User interface (UI) to connect over a network using Hamachi. The VoIP functionality is implemented using User Datagram Protocol (UDP), and the Chat and Voice note functionality with Transmission Control Protocol (TCP). The program makes use of the connectionless and connection orientated protocols to offer the user a full range of chat functionality.

Features

- It is possible to play a previous voice note as it its saved in a .wav in a file for a client
- There is a makefile for not only compiling the .class files, but also for running the program

Description of Files

Caller.java:

The Caller class handles the calling functionality between clients. It does this by utilising two methods, namely: the sendVoice() function, responsible for creating a thread to format and send it in Datagram Packets via a TargetDataLine; and the listenVoice() function, responsible for creating a thread to receive audio through a MulticastSocket, again using AudioFormat to format the audio.

Client.java:

This file contains the Client Class responsible for the Graphical User Interface (GUI) and all backend functionality of the Client. The main function creates a thread on which the startUpWindow() function is called. This function sets up a GUI to get the username and relevant port numbers/IP addresses as input. This GUI closes, a TCP socket is created, the Client constructor is called and the listenForMessages() function is called on the client to open an Input Stream to the client. The GUI is created by the chatWindow() function, which is called in ListenForMessages().
Functionality is implemented through the GUI button (ActionListeners) and a textField on the GUI. This input is all taken in through the chatWindow() function.
Incoming data is taken in and analysed for further use by the ListenForMessages() function. If a user is put into a call then after implementing the backend functionality (See Caller.java), ListenFoeMessages() will call the callGUI() to open a separate GUI for the call. This callGUI() function is also written in this file.
This file also contains methods such as recordVN, vnSender, vnReceiver, playVN, which support the voice note functionality used in ListenForMessages().

ClientHandler.java:

The ClientHandler Class is responsible for handling all the connected clients and groups. It has an ObjectInputStream/ObjectOutPutStream for accepting/sending packets. The constructor reads in a packet from the socket argument (the packet containing the username), checks to see that it is valid, and updates connected users list in the global array of clienthandlers. The run() function forwards messages, handles requests to create groups, sends updated inCallLists, and forwards packets with regards to calls between clients. This file also contains functions for generating random IP addresses and Port numbers, a writer function for writing objects to the ObjectOutputStream, the message forwarding functions, a removeUser() function and an offTheSystem() function for closing the streams and the socket. These functions are made use of in the run() function.

Group.java:

This file contains a Group class responsible for making a Group Object to keep track of the Properties of a Group of clients. Global variables include the name, group members, number of members, address and port number. There is a Group constructor and the appropriate getters and setters for the global variables.

makefile:
A makefile for creating the necessary class files, in addition to running the server and up to two clients.

MulticastGroup.java:
The MulticastGroup Class contains a constructor to set up a separate window (UI) (created from the chatWindow() function) for the user associated with a group that said user is a member of. The constructor calls the groupListen() function which listens for incoming data addressed to the groups IP address and port number. There are functions written for the handling of voice notes that are called in groupListen(). These functions, namely recordVN(), vnSender(), vnReceiver() and playVN() can also be found in this file.

Packet.java:
The Packet Class implements Serializable. This Class contains different Packet constructors for the creation of the appropriate packets to be used in communication between the clients, clienthandlers, and the server. Basically a packet would contain the sender name, the receiver name, the type, a byte array of data, the filename of where the data is from, and the file length. This my change depending on the constructor used for more specialised use. Then there are all the relevant getters and setters for the global variables (packet properties).

Servializer.java:
This Class was created for the sake of convenience. It contains only two functions. toBytes(), which takes in an object that is converted to a byte array which is then returned, and toObject(), which takes in a byte array which is then converted into an object which is returned.

Server.java:
This file contains a main function which instantiates a ServerSocket which is parsed into a call to the Server constructor. The main function then creates a thread on which the server Graphical User Interface (GUI) is run. The file also contains a offTheServer() function to close the serverSocket.

Test.java:
Test.java implements Serializable. It was created for the testing of certain functions. After a function was written and needed to be tested. It would be copied and pasted into this class, and the main would call it with the appropriate methods. The output would be analysed to see if it was correct. This class was used for testing purposes only and not used in any of the actual source code of the Program.

Project Description
There are three main areas in the program. The Client, ClientHandler and the Server. All other Classes all contain functionality that adds to these three key components.

Client:
Opens a GUI from the main (startUpWindow). This GUI just prompts the user for the appropriate IP addresses/port numbers, closes and opens a chat window for the user. If a Group call takes place then a new Window will open for that respective call. (See Client.java)

ClientHandler:
Listens for input data from the Client and processes the information accordingly. It helps to keep track of any Groups the Client is in. (See ClientHandler.java)

Sever:
The server has a GUI which displays information about what is happening between clients, such as When a user connects/disconnects, when a user joins a group or information about a call which is being made. (See Server.java)

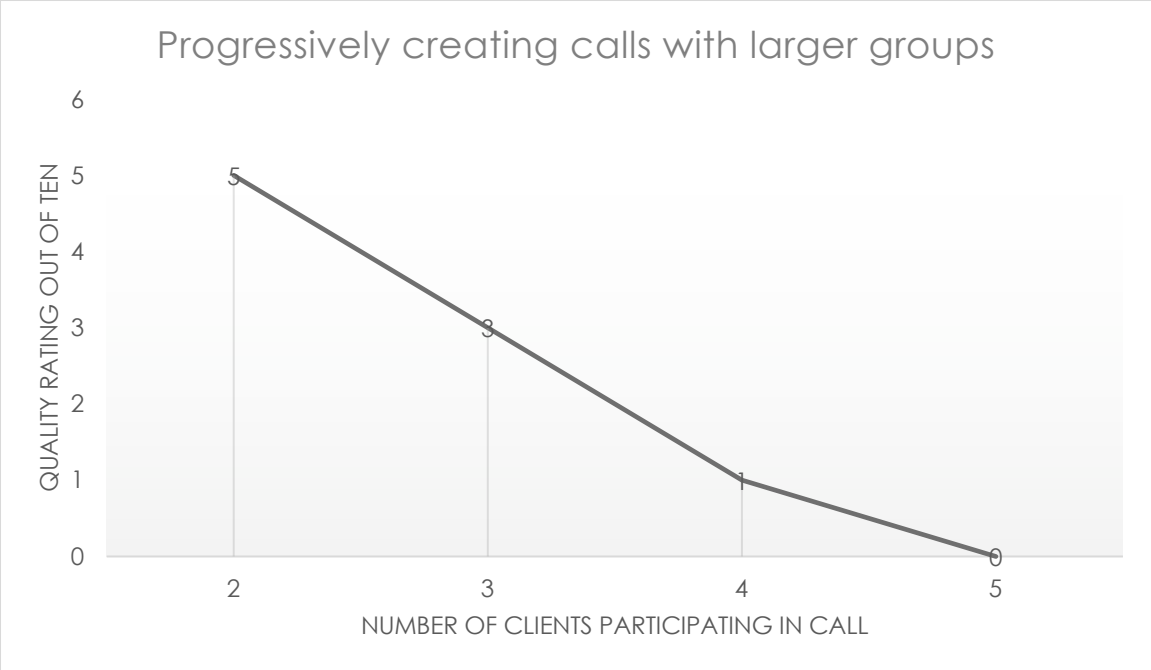
Experiments

Experiment 1: Progressively creating calls with larger groups.

Hypothesis:
As the number of people in a call increases, the call quality will drop.

Methodologies:
On one computer, test a series of calls, where for each new call the number of members increases by one. Each project group member rates the call quality out of ten. The average is then calculated and analysed.

Findings:



Analysis:
The Call quality was not good at the start of the experiment. During the call jitter, echoing, constructive interference would be experienced. When five clients where connected to the call, not much could be understood at all. This shows a decrease in quality with an approximate gradient of -1.

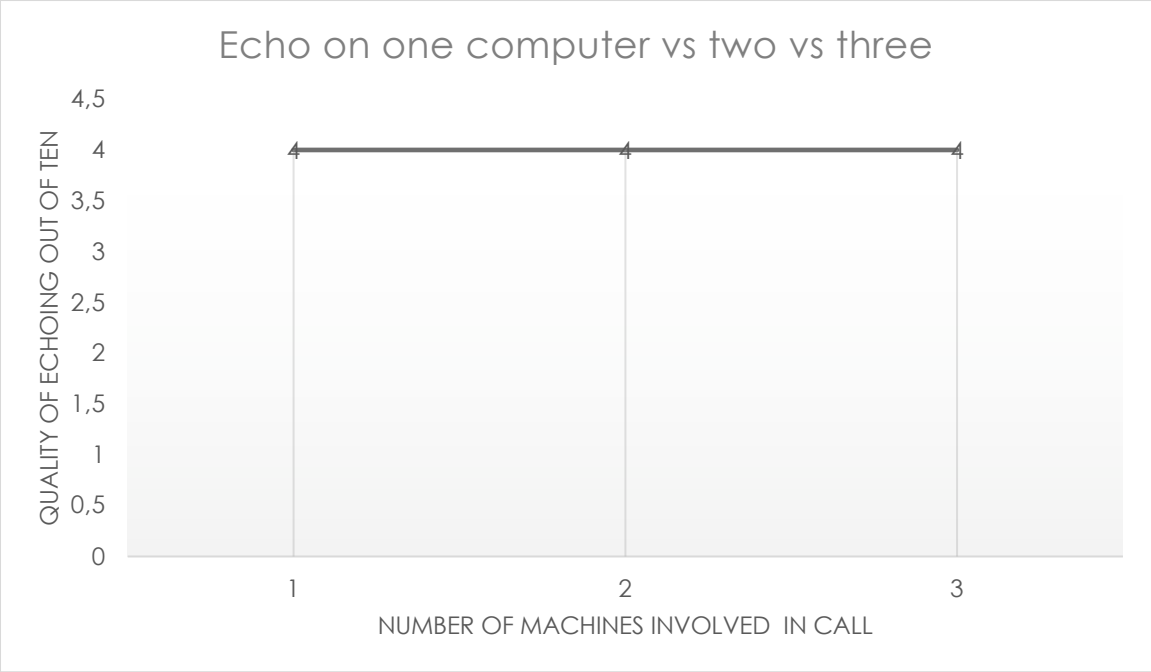
Conclusion:
Hypothesis was proven to be true. As the number of people in a call increased, the call quality did drop. If this experiment had to be redone elsewhere, it would be recommended that multiple machines be used (on for each client).

Experiment 2: Echo on one computer vs two vs three

Hypothesis:
As the amount of users of a call over a Hamachi network increase, so will the amount of echoing in the call.

Methodologies:
A call is tested three times. Once over one machine, then two, then three. Each project group member then rates the call out of ten in terms of echoing and the averages are further analysed to prove/disprove the conclusion.

Findings:



Analysis:
Although as the number of machines increased, the amount of problems with being able to connect over Hamachi increased, the echoing effect did not changed and stayed constant. However, this constant rate is 4 out of the possible 10, so still the quality in terms of echoing did not inherit a good user experience.

Conclusion:
The hypothesis was disproven. Showing that although the number of machines being used in a call was increased, the quality in terms of the echoing remained constant. It did not go down. If this experiment had to be redone elsewhere, it would be recommended that it should involve more machines.

Issues encountered

Echoing:
A prevalent issue that becomes apparent when using the calling functionality is the echoing of the voice over the network. This is yet to be resolved.

Constructive Interference:
There are times when the echoing in a call is produced and overlaps itself when taken in as input. This results in a pulse like "beep" that becomes louder after every iteration. This is also yet to be resolved.

Significant Data Structures
Numerous arrays and arrayLists are used throughout the code for this project. ArrayLists were used for storing clientHandlers and more complex objects as they can dynamically shrink or grow and provide efficient access to elements via index.

Design
The following design decisions were made for the sake of ease of implementation of larger functions in other Classes.

Caller class:
Contains sendVoice() and listenVoice(). The two functions necessary for making calls. Easier to have in a separate class than to nest in other classes.

Serializer Class
After starting to code, it became apparent that it would be easier to have a separate class responsible for converting a byte array into an object and the reverse. These two functions can be found in this class.

Packet Class

The prevalent need for various different types of packets to accommodate for the different types of communication between clients (calls, messages, group calls, voice notes, voice notes on group, etc) created a need for the corresponding various packet constructors. Rather than to have multiple nested packet classes, it is easier to maintain a Packet.java file with a single Packet Class with multiple constructors.

Compilation

To compile the program there is a makefile available for use with the following commands:

- make (compiles and runs server)
- make run (runs the server and 2 clients)
- make clean (deletes class files)
- make runServer (runs Server)
- make runClient1 (runs Client)
- make runClient2 (runs Client)