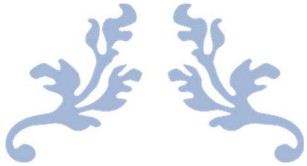

VOIP CHAT PROGRAM

Project 4

SEPTEMBER 30, 2022

Markus Sass (22548890@sun.ac.za)
Chris Langeveldt (23632135@sun.ac.za)

Table of Contents

Overview

Due to the ability of multiple users being able to have conversations over the internet, the popularity and use of the internet has paved a new way of communication. The aim of this project was to implement an application whereby multiple users can interact through sending messages and voice notes, but more importantly to implement a Voice over Internet Protocol (VOIP). This must be achieved through having a server whereby multiple clients can connect over a local network and communicate within those methods mentioned.

Required Features Implemented

Server:

1. Handles multiple clients and acts as a mediator between them
2. Handles connections/disconnections
3. Connect users simultaneously and updates user lists
4. Implement a simple GUI

Client:

1. Connects users to server
2. Show online users
3. Chat, voice message and call users
4. Conference calls (using UDP)
5. GUI showing information mentioned above

Unimplemented Features

We managed to implement everything, but the conference call aspect. We found it difficult to test and instead emphasized on a good working model that we could sufficiently test.

Additional Features Implemented

Apart from having all, but conference calls implemented, we have some additional features:

- GUI:
  The server GUI followed a shell based looked, incorporating the visibility of rooms. Furthermore, the client GUI had a dynamic custom designed button for voice notes, where it changes icons depending on the status of the voice note.

- Private direct calls:

  The user could specify a direct call to someone, with a prompt popping up showing ‘incoming call from <user>, do you wish to accept’, which would only allow that user to join the call.

- Extensive commands and /help:
  - /call <ip>
  - /answer - accept incoming call
  - /exit - shut down application
  - /listen - listen to voice note
  - /leave - to leave call
  - /create <name> - create room
  - /join <room> - join that room
  - /help - show help

- Ability to create custom rooms on demand with exclusive chats.
- Whispering to multiple clients

Description of files

Server.java

Handles connections between multiple clients.

Client.java

This class allows users to communicate. It hosts the platform for the user’s interface between text and voice messages as well as VOIP. Also has the responsibility to playback the received voice messages.

CallerThread.java

This class allows for outgoing voice calls to be handled on a separate thread recording the audio and sending the packets through UDP.

ReceiverThread.java

This class allows for receiving incoming call audio, handled on a separate thread. The packets are received and converted into the correct audio format and relayed to the client.

ClientHandler.java
This class keeps track of the rooms, users and messages received and sent. Checks username uniqueness and differs messages to the correct destination. Also handles multiple whispering.

Message.java
This class contains the object used for sending and receiving messages. This object contains text and the username of the message.

ClientListenerThread.java
This class allows for simultaneous receiving of messages from specific rooms and users. It is responsible for the interpretation of message receiving.

Timeout.java
This class allows for keeping track of client inactivity and proceeds to gracefully disconnect once their time has exceeded.

Program Description

The program is initiated through starting up the server, loaded with a GUI interface. Thereafter, clients may connect to the server's IP using a unique username. Clients are now logged into the application with a GUI, whereby they have the ability to send text messages, voice messages or use the commands to create/join a room or start a call with another client.

We used our project 1's foundation for the text messaging, however, added the feature of being able to whisper to multiple clients. The aspect surrounding rooms was also put into place, which allowed for exclusive room content to be shared and communicated.

When sending a voice note/message, the client has the neat functionality of clicking the button, which indicates the status of the voice message. The voice message, similar to text messages, will only be send to the room that the sender is in. The command '/listen' on the receiver's side can be used to listen to the voice note/message.

When requesting to call another client, the user has to use the command '/call <ip>" – while specifying the receiving clients' IP address. The receiver will then receive a dialog message box, prompting that they have received an incoming call from <user>, which they then can accept or decline. Once they accept, the call be transferred and will initiate simultaneous and synchronous voice calling. Messaging

can take place as per usual and to end the call, the user must type the command '/leave' to end the call successfully. Recurrent calls can then be taken thereafter.

Experiments

UDP packet sizes (calling)

- **Hypothesis:** If the packet size is too big, the delay would be longer, and if the packet size is too small, there would be distortion.
- **Variables used:** The packet size used (independent) and listening to the voice on one end of a call (dependent).
- **Method:** Sending voice with packet sizes 20, 512 and 4096 and observing the outcome.
- **Results:** With packet size 20, the distortion was quite bad. With packet size 512, everything worked, the voice quality was good, and the delay was around 1 second. With packet size 4096, the quality was good, but the delay was just over 2 seconds.
- **Conclusion:** The hypothesis holds. If the packet size is too big, the delay was too longer, and if the packet size is too small, there was distortion. We concluded that packet size 512 was sufficient.

Sample rate (calling)

- **Hypothesis:** The bigger the sample rate, the higher the voice quality will be.
- **Variables used:** The sample rate used (independent) and listening to the voice on one end of a call (dependent).
- **Method:** Sending voice with sample rate 8000, 44100 and observing the outcome.
- **Results:** With sample rate, 44100, the sound quality was good but inconsistent. After about 10 seconds of talking, the voice would become distorted. With sample rate, 8000, the quality was still good (less so), but the audio was more consistent and reliable.
- **Conclusion:** The hypothesis does not hold. If the sample rate gets too big, we got quite a lot of distortion even though the quality was better for the first few seconds. We concluded that sample rate 8000 was a good choice.

Sample rate (pre-recorded voice)

- **Hypothesis:** The bigger the sample rate, the higher the voice quality will be and the bigger the file size.
- **Variables used:** The sample rate used (independent) and listening to the voice note (dependent).
- **Method:** Sending voice with sample rate 8000, 44100 and observing the outcome.

- **Results:** With sample rate, 44100, the sound quality was good, and the file size was quite big. With sample rate, 8000, the quality was still good (less so), and the file size was smaller.
- **Conclusion:** The hypothesis holds. The bigger the sample rate, the higher the voice quality will be and the bigger the file size. We also concluded that sample rate 8000 was a suitable choice.

Sending/Receiving voice concurrently

- **Hypothesis:** During a one-to-one call, the two clients can talk at the same time and all sound is transmitted correctly and concurrently.
- **Variables used:** Talking at the simultaneously (independent) and listening to the results on both clients (dependent).
- **Method:** Connecting two clients (on different computers) to the server, initiating a call, and talking at the same time.
- **Results:** Both clients could hear the other while they themselves were busy talking.
- **Conclusion:** The hypothesis holds. During a one-on-one call, two clients can indeed talk at the same time and all sound is transmitted correctly and concurrently.

Calling Voice Quality (eduroam vs mobile data hotspot):

- **Hypothesis:** During a one-to-one call with both clients connected to a hotspot, the voice quality, in terms of distortion, echoes and dead zones, will be better than a one-to-one call with both clients connected to eduroam.
- **Variables used:** Both clients connected to eduroam (independent), both clients connected to a hotspot (independent) and listening to the voice quality with regards to distortion, echoes, and dead zones (dependent).
- **Method:** Connecting two clients on different computers, both connected to a mobile data hotspot, to the server, initiating a call and having a conversation. Then also connecting two clients on different computers, both connected to eduroam, to the server, initiating a call and having a conversation.
- **Results:** During the eduroam call a connection was made between the clients, but there was only a scrambling noise with no indication of someone talking on the other end. During the hotspot call a connection was made between the clients and voice was transmitted correctly with good quality.
- **Conclusion:** The hypothesis holds. During a one-to-one call with both clients connected to a hotspot, the voice quality, in terms of distortion, echoes and dead zones, was better than a one-to-one call with both clients connected to eduroam.

Pre-recorded Voice Quality (eduroam vs mobile data hotspot):

- **Hypothesis:** A voice note sent with both clients connected to a hotspot, the voice quality, in terms of distortion, echoes and dead zones, will be better than a one-to-one call with both clients connected to eduroam.
- **Variables used:** Both clients connected to eduroam (independent), both clients connected to a hotspot (independent) and listening to the voice quality with regards to distortion, echoes, and dead zones (dependent).
- **Method:** Connecting two clients on different computers, both connected to a mobile data hotspot, to the server, and sending a voice note. Then also connecting two clients on different computers, both connected to eduroam, to the server, and sending a voice note.
- **Results:** Sending over eduroam did not always work. Sometimes the voice note was never sent, other times it took up to a minute for the message to show. Sending over a hotspot always worked and the message was received within 2-3 seconds. The quality, however, was the same.
- **Conclusion:** The hypothesis somewhat holds. The voice note was not always sent when connected to eduroam. When sent however, the quality was very similar to sending over a hotspot.

Program Stability (consecutive calls)

- **Hypothesis:** Making and receiving consecutive calls works.
- **Variables used:** Making consecutive calls (independent) and listening to the results on both clients (dependent).
- **Method:** Connecting two clients (on different computers) to the server, initiating a call, and talking. Then doing this multiple times from client 1 to client 2 and again and then from client 2 to client 1.
- **Results:** All calls worked.
- **Conclusion:** The hypothesis holds. Making and receiving consecutive calls works.

Conclusion

From these tests we concluded that our implementation acts mostly as we expected it to act. Importantly, we concluded that we would use packet size of 512 bytes, and sample rate 8000. We also concluded that the use of eduroam to run this project was not sustainable. We concluded that we needed to use a hotspot in order to properly use this program.

Issues Encountered

Most issues we encountered were later discovered to be as a result of eduroam or differing operating systems, especially when it came to all voice transfers.

For example, voice notes would fail to send the packets when both were connected to eduroam, but locally all was functional. However, when connected to an external network, it was fully functional. This was the same case when it came to voice calling.

Another issue, we struggled with was the ability to host conference calls. We managed to get multicast sockets to work, however, only when both users were using windows. This made testing extremely difficult, since one member only had ubuntu and the Narga labs, would not allow for Hamachi to be installed. Therefore, we had resulted to falling back on conference calls and instead opted to spend the significant time to buff out and improve the other features. We experimented with different packet and sample sizes, until we were satisfied with the high quality and synchronous, as well as the stability of the program.

Significant Data Structures

- clientHandlers - This is an array list in 'ClientHandler.java' that contains all the thread objects that handle the communication with each client. These elements are 'ClientHandler.java' objects.
- rooms - This is an array list in 'ClientHandler.java' that contains all the names of the rooms created. These elements are String objects.
- usersList – This is an array list used on both server and client side that keeps track of the active users. This is a 'JList<String>' object.
- roomsList – This is an array list used on both server and client side that keeps track of the active rooms. This is a 'JList<String>' object.

Design

The 'Server.java' acted as our server and all clients made a TCP connection to this server. We decided to use TCP for text messages and voice notes, and UDP for direct calling.

The server acts as a middleman when approving calls. The call request and acceptance message are sent as messages through the server. Once approved, the call is made directly between clients.

We decided to have the two thread classes 'CallerThread' and 'ReceiverThread' to ensure that voice sending and receiving occur concurrently.

Regarding rooms, we decided to keep track of all rooms with a String array list. Each clientHandler is tagged with one of these string as an attribute to keep track of the current room.

We also designed our implementation in such a way that allows for simultaneous sending, receiving and other command typing. We did this by having a separate thread for listening/waiting for message to arrive. We also improved usability by adding command features.

Compilation

Server
javac Server.java

Client
javac Client.java

Execution

Server
java Server

Client
java Client

Libraries and Packages

- java.io
- java.net.*;
- java.awt.*;
- java.awt.event.*;
- javax.imageio.ImageIO;
- javax.sound.sampled.*;
- javax.swing.*;