VoIP Chat Program

R. van Tonder, 15676633
A. Esterhuizen, 15367940

October 5, 2011

Contents

1 Description

This project comprised of implementing a Voice over IP chat program. It enables users to engage in a VoIP conversation over a local network, as well as chat. The architecture is that of a client-server model. The server concerns itself primarily with introducing clients to one another.

Consider that host A wishes to call host B, but that it has no knowledge of host B's address and port. The server will provide this information to host A, such that host A and B can initiate a direct connection and proceed to a VoIP conversation. Furthermore, chat messages from clients are relayed via the server.

The client is the front-end interface that initiates call operations with other hosts. Upon connecting to the server, it provides it's host-name. The server can then provide this information to other hosts wishing to engage in VoIP. The client provides a GUI which updates messages received, as well as a user-list maintained by the server.

2 Features

2.1 Included

2.1.1 Server

The server has the following capabilities, in accordance with the project specification.

1. The server accepts connection requests from clients, and updates the user-list with the host-name of connected clients.
2. When clients wish to initiate a voice conversation, the server responds with the appropriate host-name of the recipient. In this manner clients receive permission to call other hosts.
3. Informs clients if a call or conference is already active between hosts.
4. The server relays all messages to all clients, if they are not recognized as a command such as `\call`. See 3 for more.
5. Multiple users can connect simultaneously.
6. Whispering and global chat message relaying.
7. Call and conference channels. This is done by keeping a list of current active calls and conferences. See 3 for more.
8. A GUI which displays
    - All messages sent through the server
    - Client connections to the server
    - Client disconnections from the server

- The current user-list
- Responses to commands received by the server

2.1.2 Client

The client has the following capabilities, in accordance with the project specification.

1. Clients may connect and disconnect without incident.
2. Commands that are processed by the server for various functionalities, such as `\call`, `\callc`, and `\msg`. See 3 for more.
3. Initiating voice transmission when in a call with another host.
4. Playing Voice output and receiving Mic input.
5. A GUI which displays
   - The current user-list
   - All global messages, and whispers when applicable
   - Whether a call has been initiated with the client
   - Whether a call cannot be established, if a host is already in a conference or call.

2.2 Features Not Included

While all features of the specification has been included, the sound quality of conference calls is rather low. Conference calls are implemented by making use of UDP multi-casting, and experiences some "dead-zones" in terms of audio.

The back-end server, however, still supports the full use of conference channels and VoIP groups despite the under-performing sound quality.

2.3 Extra Features

The back-end server is cross-platform compatible with windows and Ubuntu and written in PyQt. The back-end server sends notification messages to clients upon connections, disconnections, and when a user joins a conference call.

3 Design

3.1 Server Design

The server implementation was written in Python. It parses all incoming messages for the following commands, and performs the appropriate action.

- `\call <host-name>` Initiate a voice conversation with a host. The server responds with the host IP, or a message that a call is already in progress with this host.

- `\callc <host-name>` Initiate a voice conversation with multiple hosts who are already in a call. The conversation will be carried out with hosts which belong to `host-name`'s conference.
- `\msg <host-name> <message>` Whisper a message to a host-name which exists.
- `\dc` Disconnect from a call or conference if engaged in one.

All messages received by the server that do not conform to these commands are broadcast to all clients as a message.

Furthermore, the server maintains a list of threads associated with active connections, and a 2-dimensional list of calls and conferences currently in progress. For example, consider the 2-dimensional list:

```
[[146.232.50.1, 146.232.50.20, 146.232.50.41], [146.232.50.5, 146.232.50.81]]
```

This is representative of the fact that hosts `146.232.50.1, 146.232.50.20,` and `146.232.50.41` are in a conference call, while hosts `146.232.50.5` and `146.232.50.81` are in a separate call. The call/conference list can grow indefinitely, as well as the hosts within a conference. All this information is kept server side, and although clients are informed of connecting hosts, they maintain no data of the hosts in the call/conference.

The server sends an updated user-list to all clients when a change in the user-list takes place. This is done by serializing the host-names associated with the active connections list of thread objects.

The server GUI was written in PyQt.

3.2 Client Design

The client implementation was written in Java. There are two separate clients, one for normal calls and one for conference calls. The implementations of both clients are essentially the same, the only exception being that the conference call client makes use of multi-cast sockets as opposed to regular datagram sockets.

When a client starts up, it immediately establishes a connection with the server at the host address and port number provided as command-line arguments. Clients issue commands to the sever in the format specified in 3.1 Incoming messages are parsed for the following commands, after which the appropriate action is taken.

- `ca__ <host-name>` Initiate a voice conversation with the host.
- `ul__ <host-names>` Receive the user-list.
- `dc__` Disconnect from the current call or conference.

All other text is assumed to be a chat message and is displayed as such. The user-list is received as a serialized list of host-names.

As per the specification, all client-server communication takes place through a TCP connection and transmission of audio data between clients is sent as UDP packets. Upon receiving the `ca__ <host-name>` command from the server, clients commence with streaming of audio data to their respective addresses. In the case of the conference call client, each client subscribes to a multi-cast group. By disabling the multi-cast socket's loop-back mode, the clients are prevented from receiving their own audio data. In this way the audio data sent by each client is received by all other clients in a simplistic manner.

Separate threads are used for updating the GUI, recording and sending audio data, receiving and playing audio data, and listening for messages from the server. Each time a call is initiated, the threads used for audio recording and playing are launched. When instructed to disconnect from a call, the playing and recording threads are terminated. A dedicated thread listens for messages from the server, and updates the GUI accordingly. The GUI components for displaying text and the user-list are manipulated either directly with methods which are thread safe.

4 Sound Quality

Audio recording and playback is done using the Java Sound API. Sampled audio is captured into a buffer, packaged into UDP datagrams and then sent. On the receiving end, the sampled audio is read from a buffer and played back. Audio is recorded by obtaining the sampled audio data from the `TargetDataLine` interface into a buffer. Similarly, audio data is played back by writing to the `SourceDataLine` interface from a buffer.

The format of the sampled audio is controlled by a an `AudioFormat` object. The quality of the audio can be controlled by adjusting parameters such as sample-rate, sample-size, and the number of audio channels. A sample-rate of $8000Hz$, a 16 Bit sample size, and a single channel (Mono), in combination with a 4000 byte buffer for reading into UDP packets was found to deliver the best sound quality. The low data rate and small UDP packets greatly decreased delay in audio data transmission whilst still providing adequate audio quality.

5 Complications

This project presented a number of complications. In terms of technical complications, it was found that current Linux sound library ALSA is not supported by Java or Python without third-party tools. Thus, Windows was resorted to for implementing the client. The Windows sound library presented no problem in implementing VoIP. Fortunately, the server could still be implemented in Python, on Ubuntu.

Furthermore, it was challenging to test the quality of the VoIP conversations and conference calls, due to lack of sufficient equipment. See A for more information.

6 Conclusion

In this project a VoIP implementation was successfully produced, and afforded the understanding of the Java Sound API, as well as how to send sound as data over the UDP protocol.

Further avenues that could be explored is the process of encoding sound data sent over the network, rather than sending raw data. Conference calling via multi-casting did not yield entirely desirable results, and other options could be considered to achieve reliable conference calling.

A Appendix

A purchase necessitated by this project.

CASH SALE

TAX INVOICE

NAME : Edcon (Proprietary) Limited
ADDRESS : Private Bag 1
: Crown Mines
: 2025
VAT REG NO : 4460236773
HCR REG NO : NCRCP62
NAME : CNA STELLENBOSCH
ADDRESS : CNA Stellenbosch
: Shop 26, Elkestad Mall
: Beyers Street
Tel : 021 861 3560

DATE CODE TRM TRN
S/N : 19092011 00329 00008 080147
CASHIER : Sandy Williams
DATE : 19/09/2011

ITEM	QTY	VALUE
1.247180070014 XGR Back Phone Headset	1	59.95
ROUNDING		0.00
TOTAL Excl VAT		52.59
VAT		7.36
TOTAL Incl VAT		59.95
TOTAL PAID		59.95
CHANGE		0.00
EFT [**********2327]		59.95

SERIAL NUMBER : CT567790

AX RATE	CONSIDERATION	TAX
4.00%	59.95	7.36
OTAL TAX		7.36

TEMS PURCHASED 1

ME:17:10:46

RCODE VALUE : SD190911032900080147