VoIP Chat Program

R. van Tonder, 15676633
A. Esterhuizen, 15367940

October 4, 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.

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 assocaited 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

4 Sound 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-dparty 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 **??** 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.