Danmarks Tekniske Universitet

Voix-er: VoIP/IM application

Kursus:
34319: Programming Projects in IT and Communication Technology

Forfatter(e):

Martin Madsen (s124320)
Christian Laustsen (s124324)

28. Juni 2013

Introduction

The program features a server and a client. It supports basic IM with personal messaging and chat rooms that allow several users to interact with each other in real time. Furthermore, the application also implements VoIP (voice over IP), which allows the users to talk directly to one another.

The server-side part of the application is mainly written by Christian (s124324), and the client-side of the application is mainly written by Martin (s124320), although some sparring has been going on, as is natural.

Requirements

The following is a list of requirements that were specified for the program when the project started. Some have been dumped or had failed attempts of implementation.

Implemented

- A server that will allow multiple clients to connect
- A way to keep track of connected clients and their information (nicknames etc)
- Support for IM in the application
- A protocol for establishing a connection with the server and for the IM part of the application
- Chat rooms where several people can chat together at the same time
- A default channel where people are put when they connect

Not implemented

- Group VoIP chat

Design & Implementation

The server-side of the application is divided into several files which each hold their own class. Each class is called the same as its file (ie. channel.py holds the class Channel, and talk_action.py holds the class TalkAction).

Server

tcp_server.py

The TCPServer class is the main class of the server. It keeps track of connected clients and when they are ready to be read from or written to. It also provides an easy access to queue

messages on the client objects. The Server class is also responsible for PINGing the clients periodically. The server class is the one the instantiates new Client objects.

tcp_client.py

The TCPClient class takes care of the TCP connected clients (usually IM, and also used to initiate the UDP client). It keeps track of when it was pinged, its own User object, the message queue for the socket/client etc. The TCPClient is also the one that instantiates new Channel objects when needed (or reuses them if the channel already exists).

parser.py

The Parser class handles the underlying protocol. It manages joins, pings, messages, connections, disconnections etc. It interacts with the Server and the Client/User object. Lastly, the Parser is also responsible for instantiating new Talk objects in new threads when necessary (and uses TalkAction objects to handle actions related to the talk protocol).

channel.py

The Channel class gathers information about the channel, and what users are in it, along with giving quick methods for etc cleaning up a channel. The channel object holds a static dictionary (hashmap) of all the channel names pointing to their channel objects. This provides an easy way to avoid channel name duplication.

user.py

The User class holds the user data, and provides easy access to leaving or joining a channel. The User class holds a static list of all user nicknames (represented in strings), which allows the server to quickly determine if the nickname is already is in use.

talk.py

The Talk class takes care of the VoIP chat sessions, and manages the UDP clients that are connected/linked to it. It has a session key, to provide a quick way of referencing what Talk object (what conversation) the client needs to enter. It waits for a TalkAction object to perform some action (if it doesn't get one it 30 seconds, it closes since the connection is seen as dead).

talk_action.py

The TalkAction class simply holds the action and the target of said action, that is used in the Talk object.

Client

The client has the following architecture as seen on the UML diagrams below. Each class is described in details in the sub-sections.

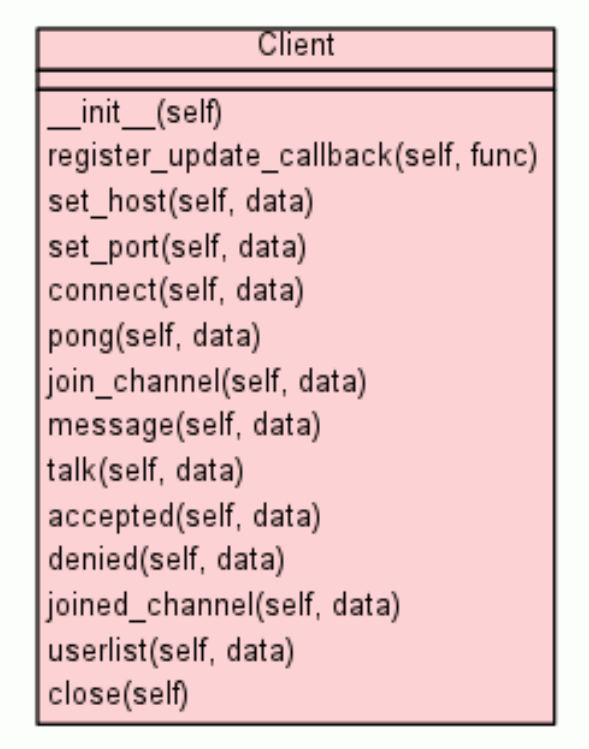

Figure 1: The structure of the CLI class.

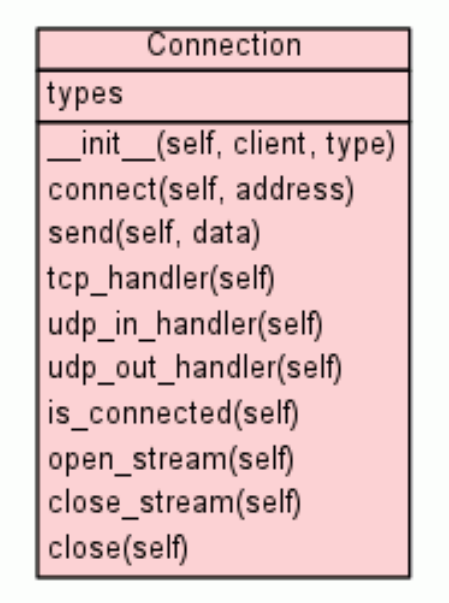

Figure 2: The structure of the Connection class that handles TCP and UDP sockets.

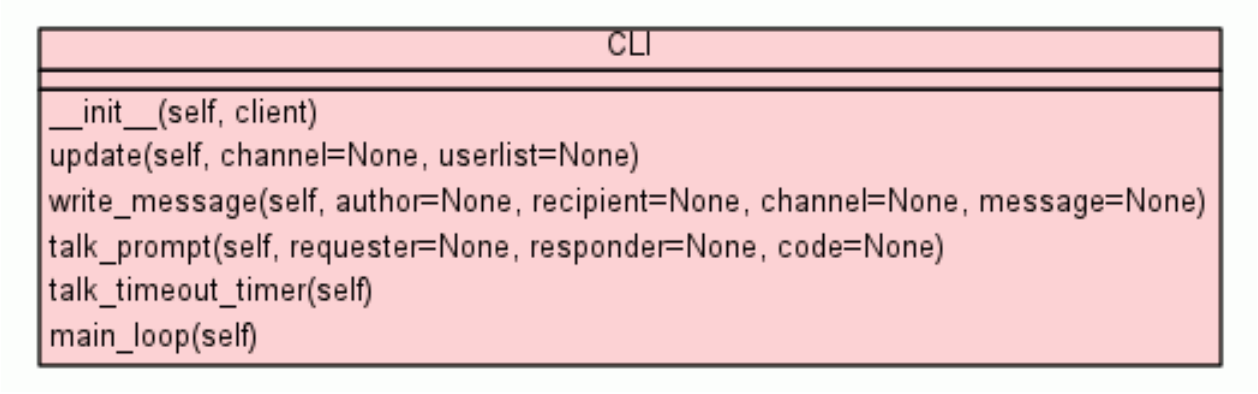

Figure 3: The structure of the CLI class.

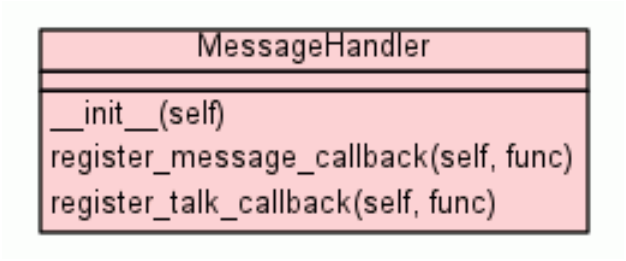

Figure 4: TODO:

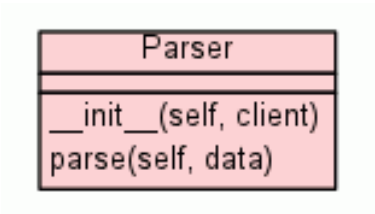

Figure 5: TODO:

client.py

connection.py

message_handler.py

parser.py

cli.py

gui.py

The GUI class is to be defined here properly set up with wxWidgets. At this moment it doesn't have much functionality. Therefore any interaction with the client must happen using the CLI.

Protocol

For the application, a custom protocol has been made. The following is a list with an explanation of all the protocol commands/actions.

CONNECT

From - Client
Syntax - CONNECT: <nickname> <realname> version number
Example - CONNECT: "John87" "John Wilson" 1.0.0

The CONNECT action is sent when the client makes the initial connection. Without this, the client cannot proceed further since all commands will be seen as invalid if the user hasn't authenticated himself with this.

ACCEPT

From - Server
Syntax - ACCEPT: <hostname>
Example - ACCEPT: ip.google.com

The ACCEPT action is sent to the client to acknowledge the CONNECT.

NOTACCEPTED

From - Server
Syntax - NOTACCEPTED: <message>
Example - NOTACCEPTED: Syntax error, please resend the CONNECT request

The NOTACCEPTED action is sent when either there is a syntax error in the CONNECT, or when the CONNECT hasn't been sent.

NICKNAMEINUSE

From - Server
Syntax - NICKNAMEINUSE: <message>
Example - NICKNAMEINUSE: The nickname 'John83' is already taken

The NICKNAMEINUSE action is sent when the nickname that the user sent in with the CONNECT is already in use.

PING

From - Server
Syntax - PING: <random number>
Example - PING: 42313812731

The server keeps track of if clients are still connected by sending them a PING periodically, and expecting a return PONG from them with the same number as the PING had.

PONG

From - Client
Syntax - PONG: <random number recieved from PING>
Example - PONG: 42313812731

The client sends a PONG as a response to a PING to let the server know that it is actually connected, and isn't a faulty connection (or that it has a latency higher than a set amount of seconds, which is 150 by default).

JOIN

From - Client
Syntax - JOIN: <channel>
Example - JOIN: #SuperAwesomeChannel

Send a request to the server to join a channel.

JOINED

From - Server
Syntax - JOINED: <channel>
Example - JOINED: #SuperAwesomeChannel

Respond to the client that it has successfully joined a channel.

USERLIST

From - Server
Syntax - USERLIST <channel>: <comma separated list of users>
Example - USERLIST #SuperAwesomeChannel: Will, John83, Mike

This is send right after the JOINED action, and lists all the users in the channel including the user itself (the client is responsible for filtering this out).

USERJOIN

From - Server
Syntax - USERJOIN <channel>: <nickname>
Example - USERJOIN #SuperAwesomeChannel: Thomas12

Notify the client that a user has joined a channel.

USERLEAVE

From - Server
Syntax - USERLEAVE <channel>: <nickname>
Example - USERLEAVE #SuperAwesomeChannel: Thomas12

Notify the client that a user has left a channel.

MSG

From - Client
Syntax - MSG <recipient>: <message>
Example - MSG #SuperAwesomeChannel: Hey all! How's it going?
Example - MSG John83: Hey John! How's it going?

The client can either send a message to a channel (where all clients in that channel receives it), or directly to a user.

MSG

From - Server
Syntax - MSG <sender> <recipient>: <message>
Example - MSG Thomas12 #SuperAwesomeChannel: Hey all! How's it going?
Example - MSG Thomas12 John83: Hey John! How's it going?

The server notifies either all the clients in a channel, or just one client directly, that a message has been sent to them and where the message stems from.

TALK (REQUEST)

From - Client
Syntax - TALK <nickname>: REQUEST
Example - TALK John83: REQUEST

Request a VoIP conversation with John83.

TALK (REQUEST)

From - Server
Syntax - TALK <nickname> <session key>: REQUEST
Example - TALK John83 827742843590392: REQUEST
Example - TALK Mike 827742843590392: REQUEST

The one that requested the conversation (let's say John83 did this) recieves this messages with the nickname being the one he requested. He then needs to store the session key for later use in the conversation.
The one that recieves the request (which will be Mike in this case) will get the nickname of the

requester in place of nickname. The reciever will also need to store the session key, for when he answers back.

TALK (ACCEPT/DENY)

From - Client
Syntax - TALK <nickname> <session key>: ACCEPT
Syntax - TALK <nickname> <session key>: DENY
Example - TALK John83 827742843590392: ACCEPT
Example - TALK John83 827742843590392: DENY

To continue the example from before, the one that got the request (Mike), will either accept or deny the request, and will send the matching action to the server.

TALK (ACCEPTED/DENIED)

From - Client
Syntax - TALK <nickname> <session key>: ACCEPTED
Syntax - TALK <nickname> <session key>: DENIED
Example - TALK Mike 827742843590392: ACCEPTED
Example - TALK Mike 827742843590392: DENIED

Again, continuing the example, since the one that got the reqeust (Mike) has now answered with either a ACCEPT or DENY, the requester (John83) will now get a response from the server stating whether or not the request got accepted. If the conversation is accepted, the VoIP conversation is started, if it's denied, the process is stopped.

TALKSESSION

From - Client
Syntax - TALKSESSION: <session key>
Example - TALKSESSION: 827742843590392

This is sent by the UDP client after a talk request has been accepted. It initiates the talk session/conversation.

SESSIONERROR

From - Server
Syntax - SESSIONERROR: <message>
Example - SESSIONERROR: Invalid session key '827742843590392'

If the TALKSESSION sends a wrong (nonexistant) or invalid session key, the server responds with a SESSIONERROR.

DISCONNECT

From - Client
Syntax - DISCONNECT
Example - DISCONNECT

Signals to the server that the user is disconnecting, and allows the server to quickly clean up after the user.

Testing

The client has a variety of tests to assure valid data while developing. The client is tested by itself with a dummy server. The dummy server is to make sure data sent from the client reaches the server it is connected to. This does mean, that not all scenarios are testable and therefore communication between the client and the real server may cause unexpected results. In this situation the scenarios are only testable by black-box testing.

The client code is approximately covered 91%. This coverage is distributed on modules as seen in the following scheme.

```
Name                    Stmts   Miss  Cover   Missing
-------------------------------------------------------
voix                        0      0   100%
voix.client               101      7    93%   109, 148, 206, 217-218, 221-222
voix.connection            68      4    94%   103, 107-110
voix.interface              0      0   100%
voix.interface.cli         68      9    87%   38, 92, 104, 117-118, 126-129
voix.interface.gui          6      3    50%   17-19
voix.message_handler        8      0   100%
voix.parser                14      0   100%
-------------------------------------------------------
TOTAL                     265     23    91%
----------------------------------------------------------------------
Ran 16 tests in 6.187s

OK
```

The test methods' names should explain the goal that is wished to assert for. And the runtime of ~6 seconds is caused by delays to overcome threaded operations.

For the server, a fake client is made to overcome the same situation as on the client. Again real scenarios are not tested and the test cases may not be reliable.

Possible features & expansions

Here's a list of additional brainstormed features and things that should be done.

- Major code cleanup (mainly on the client) and possible rewrites of parts of the code that would simplify certain complex tasks.

Conclusion

User guide