



Danmarks Tekniske Universitet

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 join

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.py

The `Server` 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 that 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 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.

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.

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) receives this message 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 receives the request (which will be Mike in this case) will get the nickname of the requester in place of nickname. The receiver 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 request (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.

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

Possible features & expansions

Conclusion

User guide