New Bulgarian University

NETB378: Java Programming Project

VARIANT № 1: SIMPLE CHAT SYSTEM

Write a chat server and chat client. The server allows multiple clients to connect to the server. Use multi-threading to handle different clients that are connected to the server. Implement GUI for the client part.

Biser Perchinkov
F44307

Semester 4

July 11th, 2015

Overview

The application described here is self-contained client-server textual communication solution. It is comprised of two parts - server and client. The server allows for many clients to connect and talk to one another through it. The client must connect to a server in order to communicate with other clients.
The client has the option to spawn a server and connect to it, after which other clients can connect to the same server.
Both the client and the server are written in the Java programming language, as it is specified by the task requirement.
The client and server communicate with each other by following a custom text protocol, inspired by the IRC protocol.

Consumer usage

Important: In order for the application to run, the consumer must have the latest version of Java installed!

Starting

There are several ways to start the application.
* The most common way to start the application is to just double click the distributed .jar file
* Another way is to start it from the command line by issuing the following command:
  * `java -jar NETB378-ChatClient.jar`

Connecting to a server

When the application has started, you will be presented by the Connect to server dialog *(fig. 1)*

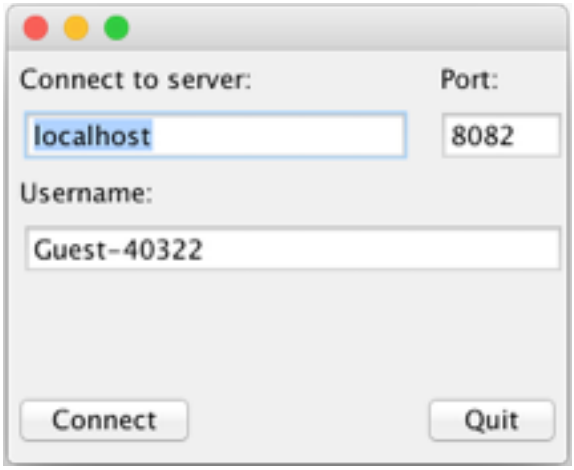

Fig. 1

Here you can specify your connection parameters - server host, server port and username to use to connect.

The server host and port are the main components of a successful connection. You need to specify a proper hostname and port where a server is awaiting connections.
The username is generated randomly on each start of the application, following the format: "***Guest-<random number 0-99999>***". You can change the username freely but it must contain only letters, numbers, dashes.
The **`Quit`** button will quit the application.

To continue and connect to the specified server, click the **`Connect`** button.

If there was a problem with connecting to the destination server, an error message will be shown in the space between the text fields and the buttons *(fig. 2 and 3)*

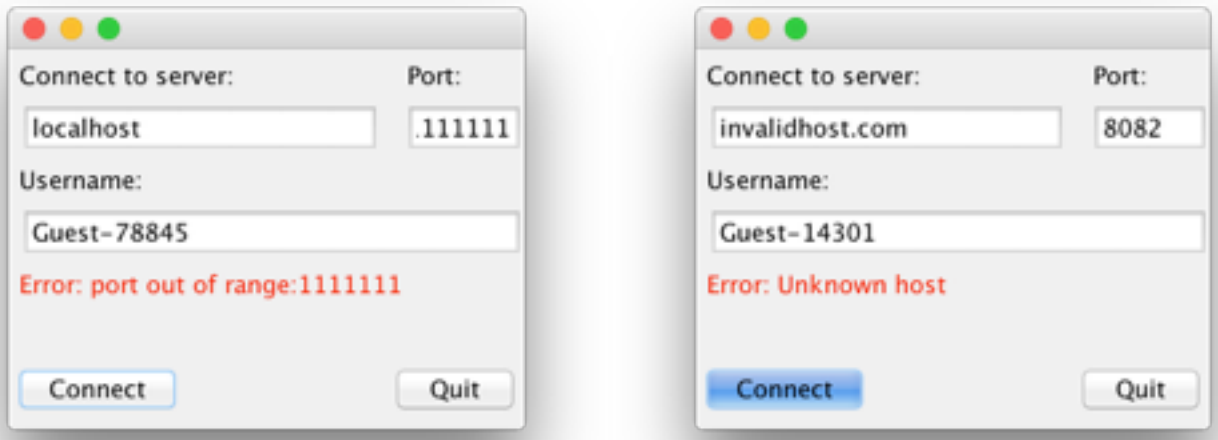

Fig. 2 and 3

If such a problem occurs, please correct the connection parameters and try to connect again.

A well hidden feature of the client application is that it can start a server. In order to do this, set the hostname to "**localhost**" and the port to **8082** and then press the **`Connect`** button. When you do that, a new popup will show up, that will ask you if you want to start a server *(fig. 4)*

Fig. 4

If you chose the **`No`** button, the popup will close and you will return to the previous screen. Otherwise a server will be started and you will automatically be connected to it.

Chatting

After a successfully established connection, the main chat window will show up *(fig. 5)*

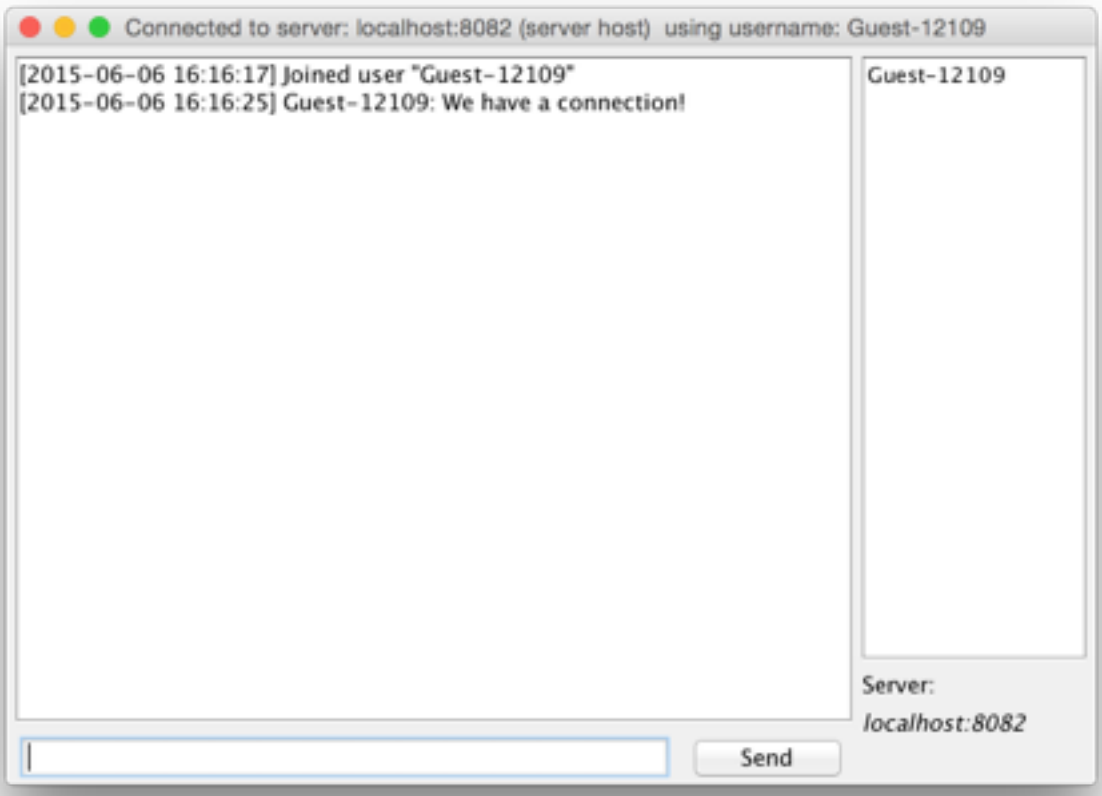

Fig. 5

The main windows have 3 sections - the text area, user list and text input:

* The text area shows the received chat messages as well as the events that occur while we are connected to the server. This includes messages by other users, events (join, quit, username changes, etc.), server notifications.
* The user list shows the users that are currently connected to the server and will see your messages if you send them. Every row represents a different user.
* The text input box at the bottom is where you write your messages or commands.

Secondary information is shown in the window's title bar and below the user list. It includes information about the server you are currently connected to, if the server you are connected to is hosted by you (started by your chat client) and what username you are currently using.

In order to send a message or perform an action, the text box accepts several types of input.

* Command input:
  * Every input that starts with a forward slash (**/**) is regarded as a command. The first part is a command and the rest of the input is payload, e.g.:
    * `/nick <new nickname>`
      * in this example, the nickname will be changed to **<new nickname>**
  * Supported commands:

    * **/nick <new username>** - changes your username to **<new username>**.
    * **/msg <message>** - sends a message to the server and other clients. It is the full command syntax for sending a message. Short syntax is to just type the message in the input box.
    * **/quit** - exit the chat.
* Message input:
  * Every other input, not starting with forward-slash (**/**) will be considered a plain text message and will be sent as is to the server and other users in the chat.

Solution description

The solution for the chat client and server holds two parts in one project. The same binary/jar can be used as both a client and a server.

The project is organised in 3 packages:

* `netb378.chatclient`
* `netb378.chatclient.Client`
* `netb378.chatclient.Server`

And as the names suggest, they take care of the main application, the client and the server respectively.

The code flow is as follows:

Initialization

The code gets initialised in class **NETB378ChatClient**, method **main**. The code there checks the parameters passed to the application and proceeds based on that.

If a **--server** parameter was specified, then the code will jump into the server module and will not start the client module.

If there was no server parameter previously specified, the code will start the client module and GUI.

Server

The server components consist of several sub components that take care of the several different tasks that comprise the server. The server package is **netb378.chatclient.Server**.

Initialisation is done by **ChatClientServer** class. The class handles initialisation of the **ChatClientServerConnectionPool**, which in turn does the listening for new connections and initialising sessions with the chat server.

Connection handling is done by a separate thread in an endless loop on a **ServerSocket**. When a new connection appears on the ServerSocket, a new, separate thread is created for the connection. Threading every single connection prevents blocking of the chat server if one of the connection hangs or timeouts.

New connections and socket events are handled in the **ChatClientServerUser** class. The class takes care of passing data to and from the socket. It also notifies the main server object in the case when the user socket gets disconnected, so the appropriate actions can be taken.

Message information to and from the user is handled back in the **ChatClientServer** class.
Every message that is received is passed from the socket to the **handleMessage** method, which takes care of parsing the message and taking the appropriate action according to that information. Protocol message handling is confined to methods in the class and their usage can be seen in the **handleMessage** method.

An interesting point is how the user information is stored and how it is accessed or modified:

ChatClientServer's property **userList** is a Hashtable that stores all the user sessions in the form of **ChatClientServerUser**. The entries are indexed by their port number, taken from **Socket.getPort()** method that returns the remote port, opened for the user. This makes the user identification unique and consistent throughout the server code.
Access to the hashtable is done by the **id** of the entry, which is as specified above.

The users, connected to the server must specify a unique username. If no unique username is specified, then upon adding the user to the user hashtable a check is done, to ensure that the username is unique. If the name is not unique, a suffix is added in the form of a dash, followed by a random number between 0 and 999999. This is done in a loop, to make sure no username is given twice, even with the random element.

Client

The client portion of the project is initialised in class **ChatClientClient**. After the class is initialised and run, the **ChatClientServerConnectForm** form is initialised. It handles the server connection details input from the client. The form can be seen on Fig. 1 listed above in the documentation. The input fields are validated through two validators:

* **ChatClientClientVerifierHostName** - makes sure the input allows only characters that can be found in hostnames. A regex representation of the validator is **^[-.\w]+$** . The validator is used for the username input too.

* **ChatClientClientVerifierNumeric** - makes sure that the input is numeric only. It is used to validate the port number on the connection form.

The Quit button exits the application. The Connect button has some logic hidden behind it. After clicking it, the client tries to initialise a socket connection with the parameters, defined in the input boxes, and checks if there was an error. If there is an error, the error message is displayed in a Label box and one additional check is done. The check detects if the hostname, specified in the input field is a local address. This is a feature that helps the user start a server, without going to great lengths to start a console and running the application with a parameter. The check is done in **isThisALocalAddress** method. If the method returns true, because it is a local host, the user is given an option which asks him if he wants to start a server. If the answer is positive, a separate thread is created, that initialises a server object as if the --server parameter is passed.
After the server is initialised, the client automatically connects to the server.

The connection is handled in a similar fashion to the server - a separate thread is created for it, so we don't have blocking when we wait for new data to come or to send from and to the server.

After successful connection, the control is passed on the class **ChatClientClientMainWindow**. Here the code is separated in two parts - backend and presentation.

The backend handles communication with the server and handling of events that occur throughout the connection lifetime. Events are parsed in a similar fashion to the server code.

After an event is parsed, a call is made to the presentation layer (the form), in order to handle the event and the information relating it.
All the presentation of the protocol events is done in the main form code, no the chat client class. This is done as the code needs to be separated by logic, something like a Model and View pattern.

The user list that can be seen on the right side of the main chat window is represented by a **jList** component, with an attached **DefaultListModel**, which allows for easy manipulation of the user list when needed.

The command input box at the bottom handles user input and has some checks that determine if the input text is a command or a plain text message. This must be done in order to be able to determine if the text should be sent as a message or as a command.

Protocol specification

The protocol is inspired by the IRC protocol. It is plaintext protocol, following a simple format.
The format is as follows:

* `<COMMAND> <PAYLOAD>`

Commands are all uppercase, i.e. **MSG <payload>.** The payload part is a string, containing the command parameters or payload. It can be arbitrary, but if it doesn't follow the proper format, an error will be thrown.

The server sometimes responds with a different format:

* `<COMMAND> <FROM> <PAYLOAD>`

This behaviour occurs when we need to notify the other clients where the event originated from.

Supported commands are as follows:

* `NICK <new username>`
  * When connecting to the server, the command sets the username for the user that is connecting
  * When already connected to the server, the command changes the username
* `NICK <old username> <new username>`
  * This is sent from the server to the clients when the nickname is changed
* `FORCENICK <new username>`
  * When connecting, it is possible that the username the client wants to set is taken. In this case the server returns a new username with this command, so that the client knows what his new username is.

* `MSG <message>`
  * This command tells the server that the current user is sending a message to everyone else in the chat server.
* `NAMES`
  * Lists all the users, currently connected to the server. It is a space separated list.
* `QUIT <quit message>`
  * The client disconnects from the chat server and provides an optional message as a reason. Sent from the client to the server.
* `QUIT <username> <quit message>`
  * The server informs you that a user has quit with an optional quit message.

* `JOIN <username>`
  * When you or another user joins the chat, the server informs all users, currently connected (including the client) that someone has joined or that the client himself has joined.