Distributed Systems – Internet Chat System
Project documentation

Project documentation content:

- *Software Engineering Requirement*
  - Functional Requirement
  - Non Functional Requirement
    - Language
    - Reliability
    - Efficiency
  - Security
  - Priority

- *Algorithm Design*
  - *Description of sequence / flow chart*
  - *UML diagram*
  - *Classes*

- *Testing*
  - JUnit test

- C*omparison With a UDP implementation*

- *Appendix:*
  - *Source code*
    - TCPServer.java
    - TCPServerThread.java
    - TCPClient.java
    - TCPClientThread.java
  - *Installation and usage Guide*
    - usage of server.jar
    - usage of client.html

Software Engineering Requirement:

An internet chat system has two basic components: A server and a client.

We will analyze the requirement as is applicable to the system as a whole as well as for each component. We are provided with the TCP protocol to connect our system.

Functional Requirement for the Server:

- A server should run on a specified port.
- It should listen for and accept connections from clients over a TCP connection.
- It should maintain a list or collection of such clients.
- It should receive data from the client and pass it to all the other clients.

Functional Requirement for the Client:

- Client should try to connect to the server when given an ip address and a port number.
- It should receive messages from the server and display it.
- It should read data from the keyboard and send it to the server.
- It should not mix up the users own typing and the message received from the server.

Non Functional Requirement:

Language and library:

The Java 1.5 API and only standard Library will be used.

Efficiency for Server:

The Server must be able to handle simultaneously a large number of clients. A 300 clients maximum limit would be appropriate. It must be able to run on a modest hardware.

Efficiency for Client:

Any java virtual machine capable of running a TCP protocol and the command line, eg. smartphones, laptops, thin clients.

Reliability

This section describes the reliability of the server and the client.

Reliability for Server:

The server must be able to manage its system resource. It must create a lot of connections and close them without running out of resources.

The connections may break, and the server must recover and reuse the resource. The server must not block on any of the connections.

Reliability for Client:

A problem with a single client must not affect the other users connected to the server.

Security:

The system opens a serversocket and large number of sockets on a network. Therefore it is imperative that the sockets are protected from outside malicious users.

It may implement an authentication and/or data encryption.

Priority:

Must have:

- Open a Server Socket on the server
- Add clients into a list on the server
- Open a socket on the client and on the server
- Send a message to the Server from the client
- Receive a message from the Client to the server
- Send copy of a message to all the Clients on the server
- Drop a client whose socket has been blocked or closed

Should:

- Server should only allow legitimate chat clients to establish connections.
- When the user is typing, incoming messages must be buffered in order to avoid any overlap. They will be displayed later, once the user has entered his/her message.
- Server should manage the nick names and prevent duplicates.

May:

- Server and client should encrypt the data.
- The chat client may use a GUI.
- The chat client may implement a filter so that only selected users message are displayed.

Algorithm design

This section describes the sequence of the program, shows the UML diagram and the classes being used.

Description of sequence / flow chart:

quit command: "\exit", this string is entered by a user on his client application when he wish to terminate his connection. It is send to the TCPServer, which will echo this message to the sender alone. The server will close the associated socket,and the serverThread object, and remove it from its list of clients.

On the Client side, either socket exception or reception of this keyword will call the exit method which will close all connections and quit the client program.

TCPServer

1. Open a serverSocket.
2. Accept client connecection and handover to a TCPserverThread.the main returns to listening for serverSocket.
3. Maintain a list of thread.
4. Send message to all client by calling the send message() defined in the serverThread class.
5. On user input close the server terminating all threads.

TCPserverThread

0. Open the datainpoutStream on this socket.
1. Receive messages from a client though the datainputstream.
2. Call the server to send the message to all the Clients.

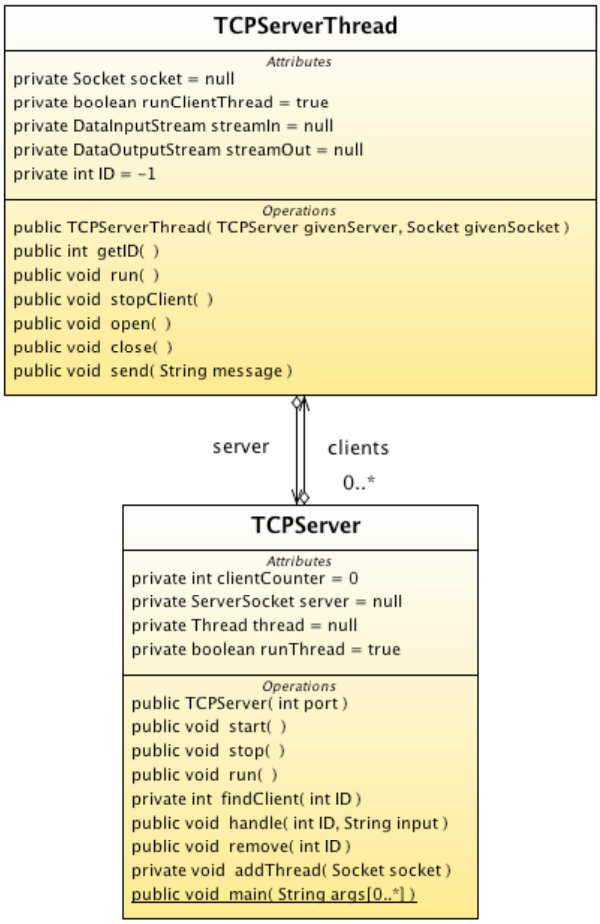
2b. Provide a message send method which will write a data on the dataoutput stream on the socket.
3a. On receiving a quit message from a client call the server to remove this thread from the list of serverThread.
3b. Close the socket.

Client

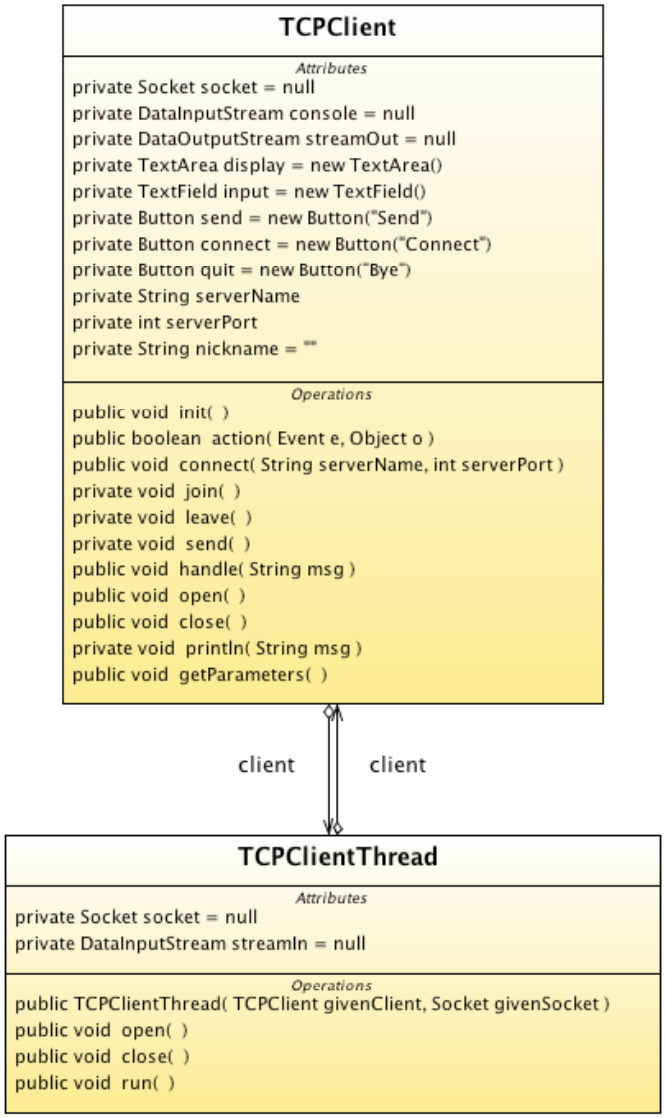
1. Read the address of the server.
2. Establish tcp socket connection.
3. Open dataOutputstream.link it to the keyboard.
4. Open datainputstream. Read data from it.
5. If data is quit command,quit the program.Otherwise print the data on to the system.out
6. If the user types a "\q" , pause the data display function ,and wait for user's input.
7. User may enter some string and hit the enter. This will enable the data display function to resume writing on the screen, while the string is written to the dataOutputStream of the socket.

UML diagram:

UML diagram for server:



UML diagram for client:



Classes:

There are the flowing classes being used from the server:

- TCPServer.java
- TCPServerThread.java

There are the flowing classes being used from the client:

- TCPClient.java
- TCPClientThread.java

Testing:

JUnit test:

We made a JUnit test for the server, visibile in the file "ServerTest4.java".

Comparison With a UDP implementation:

Alternative UDP implementation Server

The server application will receive UDP datagrams from clients, and add their ip and port address to a list of its clients.
(there are appropriate get methods in UDPdatagrampacket class).

It then forwards the data to all the other clients by creating new datagrams address to all the clients.

To provide some management options it could read the datagrams and look for special keywords. That way a user can declare his nickname, his wish to stop receiving messages etc. At periodic intervals, the server can delete the entries in its list of clients who has been inactive for long time as we can assume the client is either not interested or disconnected.

Client

The client is basically the same as the server, except we maintain the ip and port address of the server machine only.

In addition we need a method to read users input, and display the messages received from other clients. When the client application is started we will read the users nickname from the keyboard and then send it to the server as a UDPDataGramPacket. If we get a message from the server then we display it and run until user wants to quit. The Client program will have the same functionality as the TCP client in being able to block message display when the user is typing. A simple scheme will to separate the display & scan function with a sleep() and block if the user indicate a wish to type a message by entering a special key string. The display function will resume after the sleep() or the block has ended, whichever be the last.

Comparison:

Reliability:

From a top down perspective, an Internet chat System exchanges messages. A message is a string that a user has typed on the console and wants to send to other users.

In the TCP implementation, we are guaranteed that this message is received and transmitted by the server without any corruption and loss.

In a UDP implementation, we have no such guaranties. A single message could be broken up into multiple datagrams, and these may arrive at the destination out of order, or with some of the packet may get loss.

The data grams themselves are broken into segments in the underlying transport layer and error correction is minimal.

At the destination, the message must be recompiled from the multiple packets, and loss of packet will return a corrupt data. Therefore although on a 100% reliable network, a UDP based system will outperform a TCP based system, a real life application using UDP will suffer some perventage of packet loss.

In our chat system, this packet loss will translate into messages with garbled words and spellings and even loss of whole message.

Performance:

The reliability of the TCP protocol comes at the price of slower data transmission. While a TCP server negotiates a connection using the three way handshake, a UDP server can start sending data as soon as the datagram can be created.

The UDP implementation has also the advantage that it is very light on system resources as well as the network.

A TCP Server has to maintain a large number of sockets, one for each client. The UDP server needs only one socket to send its data to all its clients. A TCP server would need a much larger system resource to pay for maintaining error free connection with all its Clients. A TCP server cannot use the multicast system while a UDP could incorporate that as well as some error correction in the application to significantly provide a faster and more efficient network service.

Installation and Usage Guide:

For running the server as well as the client we created JAR files that can be used in the following way.

Usage of "Server.jar":

- Use the command line and be sure that you have the "server.jar" file in the same directory where you like to run it, if you are not currently there.
- Then you can launch the server with the following command:

```
o java –jar server.jar
```

- If you get the following output, the server is correctly running:

```
Server: doing the set up ...
Server: setting port at 7896
Server: started server on
ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=7896
]
Server: waiting for clients ...
```

- You can stop the server by typing ^C

Usage of "client.html":

- Use your web browser to start the client. Be sure that the "client.html" file is in a folder that has a subfolder "client" that contains "TCPClientThread.class" and "TCPClient.class".
- Then you can launch the client by clicking on "client.html" or explicitly launching it by your web browser (if not automatically opened).
- As next you are asked to enter the IP of the server and your user name.
- Then the applet gets launched and you can connect finally to the server which you have entered by clicking on the "Connect" button. Be sure that the server is running if your are testing it on your local machine.
- Now you are able to send messages by typing it into the empty field and clicking on "Send" button for sending it.
- For leaving the chat we implemented two different methods, you either can click on the "Bye" button or type "/exit" into the message field and click on "Send" button. There is a slight difference on it: if you do the "/exit" your are not be able to connect again. If you click on "Bye" you can do e.g. a new connection to another server with another nickname.