A Java based Simple Chat Application

Sanket Sharma

Abstract

A very simple cross-platform client-server chat application has been implemented in Java. Its design is described, limitations are discussed, improvements are proposed and a user manual is included.

Introduction

Implementing a chat server application provides a good opportunity for a beginner to design and implement a network-based system. [1] The design is very simple. It is implemented in Java, since is easy to program in, it precludes the need to deal with low-level memory management and includes powerful libraries for sockets and threads.

Related Work/Background

Implementing a chat server application is one of the most popular network programming projects with newbie programmers. Tons of free source code is available on the web. Excellent highly configurable applications are available both as open source well as proprietary software. Some examples of open source Chat applications implemented in Java are Llama Chat, FreeCS, Chipchat and OpenCHAT. [2] With little experience in network programming as well as a short duration for the project, my intension was not to match or improve the existing implementations but to implement a basic version on my own.

Design

TCP is used as the transport layer protocol, since it provides reliable delivery which is critical for the given application. TCP does not provide timing guarantee, which is not very important in the given scenario.

Server

The server is implemented as a singleton class [3]. The main thread opens a server socket on the local inet address and port 2000, which has been arbitrarily chosen and hard coded. It then waits for clients to connect to it. When a client connects to the server, it creates a separate thread as well as a User object dedicated to it. The server maintains a hash map of User objects associated with the clients connected to it hashed against the user name. The thread dedicated to the client opens a buffered

stream to read input messages from the client and a print writer stream to send messages to the client.

The messages going from a client to the server are strings with at least two words separated by a space. The first word is the header, which is the destination username. The messages going from the server to a client are strings with at least two words separated by a space. The first word is the header, which is the source username.

The thread dedicated to the client waits for incoming messages in the input buffered stream, tokenizes the messages into the header and the message string, and checks if the destination user is online by looking into the hash map. If the user is online, it attaches a new header displaying the source user name to the beginning of the message string and copies it to the print writer stream of the User object associated with the destination user. If the destination user isn't online, it sends back an error message to the source client.

Messages destined for the server (control messages) contain the string "server" as the header. Control messages are sent to know who is online and to exit. In case of the former, the thread dedicated to the user, gets all the keys from the hash map and returns a string containing the online user names to the user. When exit is requested, the thread closes the streams and the socket associated with the client and exits.

Client

The Client is implemented using two threads, one each for incoming and outgoing messages. The main thread opens the socket and connects to the server. It then opens input buffered stream and print writer for incoming and outgoing messages respectively. It also creates a buffered stream to read from the console. The main thread takes care of the outgoing message, while another thread deals with incoming messages. In order to send a message, the user types the destination user name following by the message string on the console, which is then read into the buffered stream reading from the console. The main thread creates the formatted message by adding the destination user name to the beginning of the message string and writes it to the print writer stream. Control messages have the string "server" as the header.

The reader thread, waits for incoming messages on the buffered stream. When a message arrives, it tokenizes it into the header and the message string and prints <source user> says: <message string> on the console. Responses from the server are displayed as "server says: <message string>".

When exit is requested, the client sends exit message to the server, waits for its response, and closes the streams and exits.

Limitations

There is no graphical user interface (GUI). So there are no chat windows and the user needs to write destination user name before message string. Also the user needs to enquire, who is online, rather than the server automatically updating the list of users logged in a window.

The absence of unique user identifications will lead to the nickname collision problem.

There is no encryption of message strings being sent across the network.

Possible Improvements

Limitations mentioned above can be addressed in improvements. A graphical user interface (GUI) will preclude the need to write the name of destination user when sending messages and make it more users friendly. Also users will not need to ask server who is online. A window displaying the logged in users, which is periodically updated will be a great enhancement.

In the current design, the Server hashes the users using usernames as keys. This creates problem with multiple clients using the same name. This can be addressed by hashing with IP and port number.

Current design does not encrypt the text strings. An enhanced design can use encryption of string using SSL for improved security.

A database of users containing username and password can be coupled with the existing design to maintain user accounts.

An improved version can include multiple servers, serving different geographical locations, while talking to each other. This will preclude messages between clients located close to each other being routed through a server located in a far off location, thus decreasing the delay.

Manual

Server side

Starting the server

$ java Server

The following message is displayed in the terminal
Listening on <server Inet Address> port 2000

Client side

Establishing a connection to server
$ java Client <serverurl> <username>

The following message is displayed in the terminal
connected to server at <server Inet Address> port: 2000

Once connection is established, to enquire who is online
server whoisonline

Output:
server says: <string displaying user list>

Sending a message
<destination user> <message string>

If <destination user> is not online, following message is displayed.
server says: User <destination user> is not online. Message <message string> not delivered.

Exiting
server exit

The following message is displayed in the terminal
Successfully disconnected from Server

Do not use Control-C to terminate client process, otherwise the server will not know that the given user has disconnected.

References

[1] Open Source Chat Servers in Java http://java-source.net/open-source/chat-servers

[2] The Singleton Design Pattern - Brian D Foy
http://www.theperlreview.com/Articles/v0i1/singletons.pdf

[3] Internet Relay Chat http://en.wikipedia.org/wiki/Internet_Relay_Chat