report.pdf

z5360925 COMP3331 assignment

This report outlines the important design considerations that were involved in the implementation of the assignment, and a justification of how the system works, in terms of both the custom application protocol and multi-threading functionality.

1. Program design

The client and server were initially designed to communicate over TCP, as with this connection type the server stores an instance of the Client in its own thread, allowing for seamless communication until one or the other disconnects. It also reduces the chance of packet loss. However, after re-reading the initial specification, I had to switch the entire implementation to use UDP instead. This was better designed to facilitate faster connections without storing Client instances in the server, and instead focus on processing individual packets.

All active users are stored in a HashMap data type. This data type in Java acts as a nested dictionary which stores key-value pairs. I let the username of the user be the key, which is determined during the login phase. The value of the user would be another HashMap with the key being the client address and the value being the client port number. This is essential for storing the connection port and addresses of each client with their username, in order to validate usernames in manipulation of threads and messages, and printing updates to the server console. In other words, the HashMap simply exists to check if a username is in use by another client, in the case of concurrency. It should be noted that this is very different to the TCP protocol, which would involve storing each client instance in its own thread.

There are many helper functions used to get specific file names, thread names, contents inside files, print messages to the console and send packets between client and server. All constants are declared (as static final in Java) at the top of the file. These are designed to reduce maintainability.

2. Application layer message format

In the application layer, each message is handled in the server before being sent back to the client. Each message the server receives starts with a command. If the command is invalid, an invalid command error message is sent back to the client. Otherwise, most commands are in the exact same format as what is passed in to the client from the user, defined in the specification. Below are just a few exceptions:

- HEALTHCHECK: this command takes in no arguments, and is used to establish connection between client and server (almost almost like a 2-way handshake) by returning an “ALIVE” status message
- USERNAME: this is part 1 of the login process, which takes username as argument and verifies if the username given is being used by another client
- PASSWORD: this is part 2 of the login process, which takes both username and password as arguments and verifies that the password matches username. It also takes in a boolean (true or false) determining if a new user should be created, in the case of

registration. This is usually sent by the client based on the status message returned after processing the USERNAME command

After successful login, every single command will display the list of available commands to the user, to remind them what actions they may like to perform. The client may expect messages from the server in some of the following formats:

- ERROR: this response message indicates an error with the previous command
- LOGIN ERROR: same as above but specifically related to login, in order to tell the client to reset to the “Enter Username” stage
- SUCCESS: this response message indicates a successful processing of the command previously sent

Of particular interest is the behaviour of the system and format of application layer messages related to upload and download file commands:

1. Client sends UPD command to the server
2. Server sends UPD message back to client to indicate to client that they have opened a TCP socket (2-way handshake), in order for client to open their TCP connection to be accepted by the server socket
3. Once server socket accepts client connection, the client sends the file packets until the entire file is sent. Then it closes the connection.
4. The server receives the entire file in packets, and once received closes the connection. Then the server sends one final message back to the client to confirm received message

The same process happens for download (DWN) command, with the server sending packets to the client instead.

Finally, just like other terminal environments, a prompt is used to indicate where the user’s scanned text will appear. For *fun*, I decided to use a sock emoji as prompt, because I named the forum application “SockForums”.

5. Description of how system works

As each packet contains metadata about the client’s address and port the packet was sent from, the server can use this to reply with a packet of its own directly to the client, containing the response message. Thus, as is consistent with most UDP implementations, there server needs not store an instance of the client in its own thread, and is thus a connectionless network.

When the client first starts up, it establishes a 2-way handshake with the server by first sending a health check request to the server and expecting a response to prove successful connection. Then the client enters the login process. Usernames and passwords are compared with the credentials.txt file to validate logins. If a username is not found, it assumes a new user is registering and appends the username and password to this file as a new entry.

After this, the behaviour of commands closely matches that of the specification. Threads are stored in their own files with the username of the creator as first entry, then each message and upload notification listed below, and messages having their own consistent message number. If a message is deleted, the message number still continues upwards from the last message added, for simplicity. Files are added to the server with the thread

name prefix and the file type suffix retained. Only the writer of the message can remove the message, and only the creator of the thread can remove the thread.

Every single instance of a file reader or writer, buffer reader or writer, or UDP or TCP socket input or output stream, is closed after it is used. This is to prevent memory leaks in the system.

An error is displayed to the user in the event that the server crashes. However, a safe closure mechanism has been implemented to prevent the error from displaying during "safe" termination. The termination process is simply a normal termination, where the client sends the "XIT" request to server, and then shuts down. The server receives this and invalidates the user from the activeUsers HashMap. It does not send a confirmation back to the client, since the client has already shut down and closed the connection. This ensures fast termination performance, necessary for UDP implementations.

4. Handling concurrency

To handle multiple concurrent messages, multi-threading was used. Each packet is stored in its own thread, which allows for multiple packets to be handled simultaneously, so the users are always presented with a smooth, low-latency experience.

However, then comes the case where two clients request to login with the same username at the same time. This also includes other cases that involve accessing the same data structure simultaneously. To solve this, another Java library was used: `java.util.concurrent.locks.ReentrantLock`.

This library uses locks to lock data structures, to prevent other threads from accessing them, essentially forming a queue. It is used while accessing and manipulating the HashMap of active users, and was even used in the cleanup stage. The lock ensures one user is processed at a time. So if two users wanted to login with the same username at the same time, the first user in the "queue" receives the username while the second user receives the error message "username is already in use by another client."

Once the interactions are performed, after each instance of a lock we must unlock the lock, to allow other threads to access the HashMap. This is Java's simplest instance of handling concurrency, which is robust and used in many real-world UDP applications.