COMP3331 WebForum Report

System Overview

I've implemented a command line discussion forum in Java that uses a simple custom protocol over UDP for control messages and TCP for file transfers. The client (Client.java) begins by prompting the user for a username, sending a FIRST_CONN packet (action 0) to check whether the user exists or needs to register. If the server replies to SUCCESS, the client then prompts for a password, sending a LOGIN packet (action 1) to verify or set the password. Until authentication succeeds, the client loops on these two exchanges. Once logged in, the client offers the user the commands CRT, MSG, DLT, EDT, LST, RDT, UPD, DWN, RMV and XIT. All non file operations use UDP with a 600 ms timeout and up to 16 retransmissions; for UPD and DWN the client opens a short-lived TCP socket on the same port to push or pull a binary file (100 KB), then waits for a final UDP acknowledgement. This happens by opening a TCP connection to transfer raw bytes with no extra reliability logic. After the TCP transfer, the server (for UPD) or the client (for DWN) writes a short success message back over UDP so the client can confirm success to the user. The rest of the command simply print out a command designed in the client after affirming the packet is either a SUCCESS or FAILURE packet.

Every control exchange uses one line of ASCII comprising of an action, a status, a username and message content, where action ranges from a number correlating to a command from 0 - 11 (e.g. CRT=2, XIT=11), status {0 = FAILURE, 1 = SUCCESS, 3 = UNAUTHENTICATED}, username is the current user, and content is command specific (thread title, message text, filename or error).

On the server (Server.java), a single UDP DatagramSocket and a TCP ServerSocket both listen on the same port. Incoming UDP packets are dispatched to a fixed thread pool of ten worker threads, each wrapping the packet's data, source address and port into a ClientTask. In process(), the task parses action, username and content, checks that any command beyond LOGIN comes from a logged in user (else responds with UNAUTHENTICATED), then invokes MessageHandler to perform the operation and generate a response.

MessageHandler uses two main helpeer Classes:
UserList maintains the credentials.txt file and an in-memory List<User> wrapped by Collections.synchronizedList. This was the only kind of Mutex guard I used in the entire codebase. On initialisation, it loads every username/password pair from the credentials.txt file and marks them offline. All mutation methods, add(), exists(), setOnline(), are declared synchronized to prevent race conditions when multiple clients log in concurrently. When a new user is created, add() appends to credentials.txt using BufferedWriter in append mode.t his is atomic by default because it is a blocking function.

ThreadManager keeps a Map<String, ForumThread> guarded by synchronized methods. Each ForumThread mirrors its on disk file. The first line is the thread creator's username; each subsequent line is either “<number> <username>: <message>” or “<username> uploaded <filename>”. All operations: creating a thread, posting, editing, deleting, attaching files or removing a thread, are synchronized at method level so the in-memory state and on disk file remain consistent under concurrent access. File attachments are stored next to the thread file, named ‘*threadtitle-filename*’, and an attachment entry is appended to the actual thread file.

I deliberately chose a simple synchronization of methods over fine-grained thread locking using mutexes to reduce complexity and prevent risk of deadlocks. In marking scenarios there will be at most three active clients, so the contention among different threads accessing resources is low and lock hold times are brief. The fixed thread-pool of ten threads handles bursts of requests without dynamic resizing or queuing overhead. I omitted a forced shutdown mechanism in the client to catch Ctrl C-while it could automatically send XIT before exit, that feature wasn’t necessary as the specficication states “*During testing, we will not*

abruptly terminate the server or client processes (CTRL-C)", and I preferred to focus on the core functional correctness.

Concurrency in WebForum

I handled all UDP requests through a fixed-size thread pool (Executors.newFixedThreadPool(10)), so each incoming packet is wrapped in its own ClientTask and processed concurrently. Every piece of shared state, whether user sessions in UserList or threads and posts in ThreadManager, is only ever accessed through a synchronized method. This ensures that two threads trying to create, delete or modify the same thread will queue at the lock rather than corrupt each other's data through data race conditions.

File attachments likewise pose no race conditions. Each upload writes to its own file (for example, threadA-file1 or threadB-file2), and the only shared side-effect is appending a metadata line in the corresponding thread file. Because ForumThread.addAttachment(...) is synchronized, even if two uploads complete at the same instant, one thread holds the monitor, writes its line, releases it, and only then does the next write proceed.

On the network side, I use a single DatagramSocket for all outgoing UDP replies. The main server loop calls socket.receive(...) and immediately passes off a copy of that packet to a worker, so, no worker ever calls receive(). Each send() maps directly to a native sendto() call, which is atomic, and because each ClientTask has its own DatagramPacket, there's no shared buffer to contend over. Concurrent send() calls simply queue up in the OS without interfering.

TCP transfers are equally isolated by Java's blocking I/O. As soon as a worker sends the "ready" UDP response for UPD or DWN it blocks in tcpListener.accept(), waiting for that one client's connection. Once accepted, the upload (bis.read() / bos.write()) or download loop blocks on the socket's streams but never holds up other transfers, since each lives in its own thread. Finally, the main thread always does socket.receive() then immediately pool.execute(...), so if all ten workers are busy new tasks just queue up in the ExecutorService's unbounded work queue rather than blocking the receive, only receive() ever waits for a packet, not for a free thread. (which is fine according to the assignment specifications anyways).

Design trade offs

- The risk of undelivered UDP packets is handled by 'stop and wait' retransmissions with fixed timeouts. Even though a simple mechanism and sufficient on localhost on my PC, it was way quicker to implement than other algorithms like using sliding windows.
- Synchronized collections (for the list of Users) guarantee thread safety for user and thread state, at the cost of potential deadlocks. Given our low concurrency use case, this is unlikely to happen.
- Single TCP port for all file transfers avoids dynamic port negotiation. The client simply connects to the server's known port after receiving UDP READY, and the server calls accept() with a 1 second timeout to guard against stale requests.

Testing and limitations

I tested sequential and concurrent scenarios on VLAB. Some of the harder edge cases to test and resolve included: extending malformed commands and command formatting (client checks syntax before sending), timeouts due to missed TCP-initialising UDP parckjet (client retries up to 16 times because of 1 missed packet), missing files on UPD/DWN requests, and concurrent registration of the same username (guarded by synchronized UserList). I did not implement persistent shutdown hooks or automatic reconnection after server restart, such features could have been added with minimal effort though, if it was required.

Future improvements

- Replace coarse locks with ReadWriteLock to allow parallel reads of threads with Mutex guards and manual thread blocking for more reliable thread-safety.
- Encrypt credentials/messages or use TLS for more secure packet transfers.
- Implement a fail-safe mechanism that logs a user from the server out (call XIT automatically) on either manual (CTRL-C) or error disconnect, allowing client to reauthenicate with complications.

Documentation and References

The UDP packet transfer and TCP upload/download logic came together after going through all the given files from lectures and labs in COMP3331, using this as a starting point for both my Client and Server classes helped a ton. I had a look through the multithreading example they gave for the Server, and while that implementation was sufficienct, I felt as though it lacked robustness so I chose to use a ExecutorService thread pool which would ensure mutiple runnables (threads) are properly handled.

I picked up the basics of raw Java threads from GeeksforGeeks' to understand simple multithreading: https://www.geeksforgeeks.org/multithreading-in-java/. (which used your guys' implementation).
To see how to wire those threads into a managed pool I followed Oracle's ExecutorService tutorial: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html.
For making my user list thread-safe with a synchronized wrapper I leaned on Oracle's collection wrappers guide (look for Collections.synchronizedList): https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List-.
Finally, to turn a file into a byte[] and back into a file over a socket I used this StackOverflow write-up on Java file transfer to see an example of its implementation and how to solve this specific bug: https://stackoverflow.com/questions/17876824/java-file-transfer-socket-write-error.

UML Design Diagram

Below is UML diagram illustrating the classes Client, Server, MessageHandler, UserList, ThreadManager, ForumThread and their key fields and methods.