CS3211
Assignment 3: TCP server in Rust

1) An outline and a brief explanation of your TCP server — include all assumptions, as well as any non-trivial implementation details related to processing requests from multiple clients.

Our TCP server spawns a new asynchronous task for each client, which then calls the handle_connection function, using tokio::task::spawn (Line 47), enabling the server to manage multiple clients concurrently. The execution of tasks is also asynchronous, as we use the Task::execute_async function (Lines 85 and 87). The functions start_server (Line 23), handle_connection (Line 60) and get_task_value (Line 79) are asynchronous, and their inner operations have also been changed to be non-blocking, e.g. using tokio::io::{BufReader, AsyncBufReadExt, AsyncWriteExt}, tokio::net::{TcpListener, TcpStream}, and tokio::sync::{Semaphore}. The general implementation of the server follows the original sequential boilerplate code given.

Regarding the limit of 40 CPU-intensive tasks, we utilise a Semaphore that is initialised to a value of 40, which is then wrapped in an Arc and has its clones passed around. We check if the task type is CPU intensive, and if it is, we acquire the semaphore before asynchronously executing the CPU-intensive task (Lines 83-85).

We split TcpStream into a readable and writable part, to allow for independent ownership of the read and write halves (Line 62). Minor changes are made to read the stream of lines accordingly (Lines 64-67).

2) Explain the concurrency paradigm used in your concurrent implementation and how the clients' requests are handled concurrently.

We use asynchronous concurrency, through the tokio runtime. We use Non-Blocking I/O operations such as tokio::io::{BufReader, AsyncBufReadExt, AsyncWriteExt}, and we use the non-blocking APIs tokio::net::{TcpListener, TcpStream}.

Each client connection is handled by using tokio::spawn to spawn a new asynchronous task, thus each client is dealt with independently and can progress concurrently.

3) What level of concurrency does your server achieve and why? Explain your design. Briefly explain ALL cases when the concurrency level decreases from your claimed concurrency level.

Our server achieves task-level concurrency. Firstly, client-level concurrency is achieved as a new asynchronous task is spawned for each incoming connection/client, allowing the server to manage multiple clients concurrently, each in its task. This setup also allows I/O and CPU tasks to be executed concurrently, as both are executed asynchronously using the execute_async function and both tasks await on the future that is returned. Hence task level concurrency is achieved.
The concurrency level will deviate when the asynchronous tasks have not yet been spawned for each incoming connection/client. E.g. when there is a queue of incoming connections waiting to be accepted

and for their corresponding asynchronous tasks to be spawned. If the CPU-intensive tasks exceed 40, there is also the possibility that concurrency is reduced as they are waiting for a semaphore permit to be released.

4) Will your server run tasks in parallel? Briefly explain.

Yes, our server can run tasks in parallel, facilitated by the tokio runtime, which uses multi-threading to manage and execute tasks across multiple cores. Tokio also utilises non-blocking I/O to ensure that other tasks are not blocked from executing. However, actual achieved parallelism will depend on the CPU and tokio's scheduling of the tasks. Tokio is able to execute tasks in parallel and utilise multiple cores, and by default is able to utilise all the CPU cores available.

5) If you have tried multiple implementations, explain the differences and your evolution to the current submission. There is no need to submit your alternative implementations.

Initially, we achieved a task-level concurrent solution by using tokio's spawn_blocking mechanism, which allows you to spawn blocking threads, where the maximum number of threads spawned is limited by the maximum blocking threads parameter in the tokio runtime. We planned on using this to limit the CPU-intensive tasks to 40. While this did work, we had to modify the main function to run asynchronously, which involved many changes to the main.rs file. Eventually, we decided that using a semaphore was a cleaner solution, which we decided to stick with.