CS3211 Assignment 3

David Zhu (E0958755), William Zi Ang Wang (E1496974)

AY24/25 Semester 2

1 TCP Server

Our TCP server utilizes the `tokio` library to asynchronously handle sending and receiving messages on TCP connections.

2 Concurrency Paradigm

We used a couple different concurrency paradigm/techniques:

- We worked with asynchronous programming by turning the connection handling and value parsing into async functions. A tokio thread is spawned for every connection.
- We also used thread pools using both tokio and rayon to handle I/O task and CPU task parallelism respectively.
    - We use an ARC pointer to pass the rayon pool down the async functions `handle_connection` and `get_task_value` so that rayon threads may be spawned for CPU tasks.
    - We use message passing using a one-shot channel to bridge rayon to tokio.

3 Concurrency Level

Our server achieves task-level concurrency.

1. Multiple clients can run concurrently. Each connection is wrapped in tokio::spawn, and tokio schedules these tasks across its 6 worker threads.
2. I/O and CPU tasks can execute concurrently. I/O tasks run on tokio's async threads, while CPU tasks are offloaded to rayon's 10-thread pool

4 Server Parallel Tasks

Our server will run tasks in parallel.

- CPU-intensive tasks are distributed across rayon threads, so two CPU tasks from different clients execute in parallel.
- I/O tasks are async and concurrent on tokio threads. Multiple tokio threads can handle separate I/O tasks in parallel.

However, tasks within a single client are processed sequentially as enforced by the protocol.

5 Evolution of Implementation

- Initial solution: uses tokio, runs async tasks asynchronously, and cpu tasks on its own thread using tokio's `spawn_blocking`. However, tokio docs suggest using rayon for CPU tasks.
- Final solution: as described in this report, uses rayon thread pool. This also allows to avoids the runtime error of spawning too many tokio threads too by the CPU tasks.