Overview

Our TCP server leverages Tokio's async runtime to efficiently accept and process client tasks.

It has 3 main functionalities:

1. **Binds & Listens**

   On startup, the server (running as a Tokio task) binds to the configured address and immediately enters an infinite loop, awaiting new connections with `listener.accept().await?`

2. **Spawns Per-Connection Tokio Task**

   For every accepted client connection, the server calls `tokio::spawn` to launch a new dedicated Tokio Task. This Tokio Task owns the client's `TcpStream`, continuously reads incoming lines, invokes `Task::execute_async(...)` and sends the result back to the client

3. **Startup Coordination**

   To avoid a race where clients try to connect before the server is ready, the (default in template) `mpsc` channel as a simple readiness signal:

   - Once `TcpListener::bind(...)` succeeds, the server sends `Ok(())` on the channel.
   - The async `(main)` function (that is annotated with `#[tokio::main]`) awaits that signal before spawning any client threads

Concurrency Paradigm

We used asynchronous programming as our concurrency paradigm. For each connected client, a Tokio task is spawned that contains the client's incoming `TcpStream`. Inside this Tokio task, every I/O or compute step — such as reading, executing `Task::execute_async(...)`, and writing the response — is itself an `async` operation (i.e. a Future). When `.await` is called on one of these futures, it yields control back to the runtime only if the operation (a Future) is not yet ready to be done.

Tokio's multi-threaded executor polls those fine-grained Futures on a pool of worker threads, where by default, the number of worker threads is the number of logical CPU cores on the machine. When one future is awaiting, the executor dispatches another ready future on the same thread (if there is one), with the intention to keep the cores as busy as possible.

Therefore, multiple clients can be at different stages of processing

(read → compute → write back), executing concurrently on different cores.

Level of Concurrency

Our server achieves task-level concurrency. Every client connection is handled by its own Tokio task, where tasks from different clients can execute concurrently on different cores of the machine, client-level concurrency is achieved.

Because each stage (read, compute, and write back) is its own fine-grained `Future`, Tokio's multi-threaded executor can interleave I/O and CPU work on the same core. When one task hits an `.await` and its future isn't ready (for example, waiting on I/O), it yields control back to the runtime. The executor then polls and runs another task's ready future on that same worker thread (and thus the same CPU core), keeping the core busy. As soon as the original future becomes ready again, the runtime picks it up and resumes execution, allowing multiple clients' requests to make progress concurrently even on a single CPU core. Therefore, achieving task-level concurrency.

There is one main case where the level of concurrency will drop.

1. **CPU-Bound Saturation**

   In this situation, there are as many CPU-intensive tasks running as there are Tokio worker threads (by default one per logical core). Each thread is busy executing compute-bound futures without hitting any `.await` points to yield back to the runtime. As a result, new tasks — whether I/O-ready or freshly spawned — cannot be polled until a thread becomes free. Although every client still "owns" its own task, the effective concurrency is capped by the number of available worker threads.

As it does not run using the CPU, the number of IO-intensive tasks that can be executed concurrently is not bounded by how many cores the machine has and is potentially unlimited (subject to any other hardware requirements the task might have). In contrast, the number of CPU-intensive tasks that can be executed concurrently is limited by the number of Tokio's worker threads, and in turn, the number of the machine's cores.

Will your server run tasks in parallel? Briefly explain.

Our server does run tasks in parallel. As stated above, each client connection is handled by its own Tokio task, and Tokio's multi-threaded executor (which spawns one worker thread per logical core by default), polls those task's futures across the worker threads. This means that ready work from different clients can execute truly in parallel on separate CPU cores.

Other Implementations attempted

In our initial design, we used Rayon's global thread pool to parallelize CPU-intensive work while the Tokio runtime handled I/O. We assumed this hybrid approach would leverage Rayon's data-parallelism, but benchmarking showed that Tokio alone was faster — even with the additional "mystery" task in the bonus submission.

Therefore, we decided to stick with Tokio's multi-threaded executor for both I/O and CPU-bound workloads for simplicity and ease of testing, reserving the (Rayon + Tokio) integration as an optimized alternative in the bonus version.