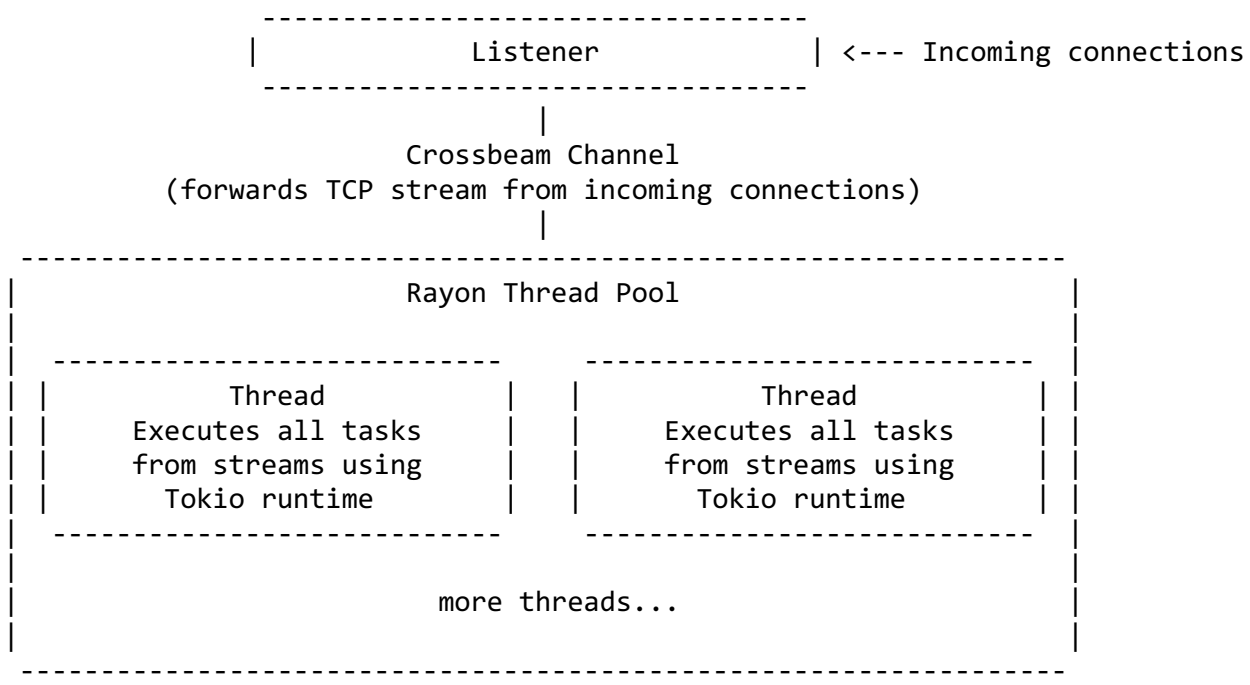
Concurrent TCP Server in Rust

This is a concurrent TCP Server that will manage simulated client requests, in Rust. It utilizes the Rayon library for thread
pooling and parallelism, and the Tokio runtime for asynchronous I/O operations.

Architecture



Outline

The server follows these steps:

- It creates a TCP listener to accept incoming client connections.
- It sets up a thread pool using Rayon running multiple threads.
- Each thread is spawned coonnected to an MPMC channel in Crossbeam to handle new client connections, and a separate runtime in Tokio that manages asynchronous task execution.
- When each thread receives a new connection from the channel, it creates a new task containing the TCP stream within the Tokio runtime to handle all task execution from that client.
- The server processes the tasks from the TCP stream based on their type and writes the results back into the stream.
- A global atomic counter is used to limit the total number of concurrent CPU-bound tasks.
- When the limit on CPU bound tasks is hit, the runtime awaits on a notification that will be sent once any CPU bound task is executed. If the executing manages to increase the counter, it will start executing the CPU bound task.

Concurrency paradigms

Thread Pool: Multiple threads are spawned in a thread pool to handle multiple clients. This allows for parallel execution of tasks.
Asynchronous Concurrency: Each thread in the thread pool runs a separate Tokio runtime. The runtime schedules tasks from each client asynchronously.

Level of concurrency

The server contains task-level concurrency by scheduling more threads than CPUs, with each executing thread concurrently execute CPU-bound and I/O-bound tasks. Threads from the thread pool are scheduled for concurrent execution on CPUs, each thread executing tasks received using the Tokio runtime. When there are many I/O bound tasks, each thread concurrently executes I/O bound tasks with CPU bound tasks interleaved.

The level of concurrency may decrease when there are large amounts of incoming client connections or when there are many CPU bound tasks.

1. There is only 1 thread listening and sequentially forwarding TCP streams from incoming connections. When there is a large number of incoming connections, and/or when each client makes only a few requests, this sequential execution becomes more apparent which causes the level of concurrency to drop.

2. While the Tokio runtime schedules tasks for execution without blocking the thread, Tokio will execute CPU bound tasks
   to completion or each thread's time slice is used up without preemption. This means that when any CPU bound task executes, I/O bound tasks will have to wait for the currently executing CPU bound task to finish executing first or the time slice expires, reducing concurrency within each thread. However, there is still concurrent execution of CPU bound and I/O bound tasks if an I/O bound tasks executes first and is awaited since other CPU bound task can execute in the meantime. Furthermore, context switch between threads can still occur which allows execution of I/O bound tasks from other threads on the same CPU while a CPU bound task executes.

Parallel execution

If multiple CPUs are used, the server will spawn multiple threads that can be scheduled on multiple CPUs, providing parallel execution.

References

- Using Rustlang's Async Tokio Runtime for CPU-Bound Tasks

- Reddit: Tokio for CPU intensive work