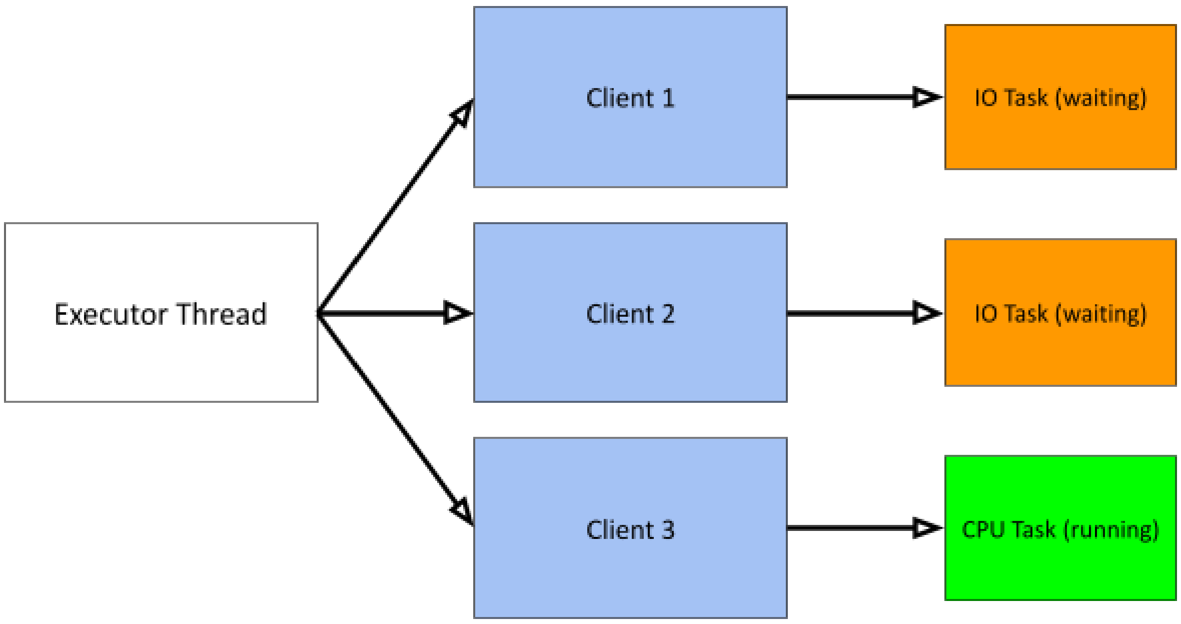
Team: e0774645_e1324860

TCP Server implementation and Concurrency Paradigm:

1. The TCP Server is implemented with asynchronous programming, with the Tokio library. When the TcpListener receives a connection, the connection is handled asynchronously. Each connection is handled with a Future, and tokio::spawn is called to manage the Future.
2. In handling a connection, the execute_async function is used instead of the execute function. The execute_async function executes the IO intensive tasks in a non blocking manner, allowing for other tasks to execute on the same CPU while waiting for IO.
3. The asynchronous implementations of TcpListener, TcpStream and BufReader are used. These implementations are non-blocking, and will allow other tasks to execute if we are await-ing on a connection or data to be written into the buffer.
4. To enforce the constraint where the server runs at most 40 CPU intensive tasks concurrently, we used Tokio's non blocking semaphore implementation with a starting count of 40. Each time a CPU intensive task is executed, acquire is called on the semaphore to ensure that at most 40 CPU intensive tasks are being run concurrently. The non blocking nature of the semaphore allows other things to be done while a task waits for its turn to be executed. In addition, when the task completes at the end of the tokio::spawn, the semaphore is automatically released.



Example of an executor thread handling 3 clients concurrently

Team: e0774645_e1324860

Level of concurrency:

Our implementation achieves **Task-Level concurrency**.

1. Multiple clients can be handled concurrently, as each client connection is handled in a non blocking manner. This means that the executor will not be blocked from accepting and handling multiple connections from clients.
2. IO and CPU intensive tasks are also able to be handled concurrently on the same CPU. Since IO tasks are handled in a non blocking manner, CPU tasks can be executed while waiting on IO tasks.

Cases where concurrency level decreases:

1. In the case where there are only IO intensive tasks or only CPU intensive tasks, it would not be possible to observe IO and CPU tasks being handled concurrently.
2. In the case where the server is using a single threaded run time and only CPU intensive tasks are being sent from the client, the server will not be able to handle multiple clients concurrently. Since asynchronous tasks are cooperative and not pre-emptive, if the CPU intensive tasks of a current client do not give up control of the executor thread, the executor thread will not be able to handle the next client. This results in the executor thread handling one client at a time.
3. If the task to be executed is not programmed correctly and has a synchronous blocking call such as sleep, which will actually cause the executor thread to sleep, hence unable to process other clients.

Running in parallel:

In the case where the server is run on a CPU with multiple cores, tasks will be able to run in parallel. By default, without specifying any parameters in the #[tokio::main] macro, Tokio will use a multi-threaded runtime that creates a number of worker threads that is equal to the number of CPU cores available, and the multiple cores will be able to run the multiple threads in parallel, allowing tasks to be executed in parallel.

Other Implementations:

Threadpool:

We initially tried to use a thread pool implementation where each client is handled by one thread. To limit the number of concurrent CPU tasks, we limit the number of threads to 40, but the issue is this limits the maximum number of threads to 40, which means we can only run a maximum of 40 tasks regardless whether it's CPU or IO intensive, which is not ideal. Especially if all 40 tasks were I/O bound tasks, the threads would have to wait idly for the tasks to complete instead of executing the CPU intensive tasks. We could increase the number of threads in the pool, but it would also mean increasing overhead due to the amount of context switching and additional memory needed.

The best way to tackle the overhead of increasing the number of threads is to make use of non-blocking I/O. This would allow us to enable concurrency for each thread. Hence, the limitations of the thread pool model led us to explore asynchronous programming, and its key feature is its non-blocking nature, where a single thread can handle multiple tasks concurrently. We decided to choose Tokio because it provides a well tested and robust framework for writing asynchronous programs in Rust easily while abstracting the complexity of dealing with futures and poll.