Concurrency in Rust : The Case of Connect Four.

The aim of this lab is to work on an application that is both compute-bound and IO-bound. We will create a Connect Four game capable of network play (IO-bound), and a robot agent to play against you (compute-bound).

The lab will be evaluated based on the code that you create. **Add your name in the authors section in Cargo.toml.** Your code should be formatted with `cargo fmt` and have as few `cargo clippy` warnings as possible.

Most of the project is already coded, but parts are left for you to complete. Only modify sections marked by `todo!` or `//TODO`. There are a few tests available with `cargo t` and `cargo t --release`. By the end of the lab, you should pass all of them.

If `cargo` is not available on your machine, you can get it from rustup.rs.

I Architecture of the Project

The project is divided into modules:

`game` This module contains the game logic, where no modifications are necessary.
`network` This module implements network logic.
`evaluation` This module represents robot agent logic for multiple cases.
`caches` This module contains knowledge caches implemented using memoization.
`blocking_future` This module provides a future to execute compute-bound tasks in an async context.

To familiarize yourself with the game, you can use the `cargo r --release --bin local` command to play against yourself. Just give the letter corresponding to the column, and press enter. **Don't play for too long!**

As a reminder, to launch `bin/foo.rs` with the parameters `ham spam`, you can either
- Use `cargo b --release`, then `./target/release/foo ham spam`.
- Or use `cargo r --release --bin foo -- ham spam`.

The latter allows you to be sure that the program is recompiled when launched.

II Playing with a Friend

Here we will implement network logic. Please note that *VS Code* allows you to split the terminal in two, which is very useful for testing against yourself.

The aim is for you to launch `manual host localhost:4444` in one terminal and `manual client localhost:4444` in another, and be able to play each player in each terminal.

Complete the three functions in `network.rs` then play a game against yourself with manual. We use Serde to turn Rust objects into String on the sender (serialization) and String into Rust objects on the receiver (deserialization). Note that turning an object into a String, then into `[u8]` to be sent over the TCP stream cannot fail, but turning `[u8]` into a String then into an object can. Either the slice is not valid UTF-8, or the string is not in the correct format.

If the game goes smoothly, you can continue to the next part.

III Playing without a Friend

Connect Four is a zero-sum game, like chess. This means that we can use an algorithm called MinMax to look a few moves into the future to guess which move is best. The number of moves looked ahead is called depth. With a depth of 1, the agent will look at each possible move and choose the best one. With a depth of 2, for each of its moves, the agent will look at each possible move of the adversary… This means that the algorithm has complexity $O(\text{width} * \text{depth})$. The algorithm is already implemented; it's not the point of this lab to modify it.

Robot agents are represented by the trait SyncEvaluator. We can use robot to have a robot player instead of a human one. Try creating a robot agent with `robot host localhost:4444 <depth>` and play against it with `manual client localhost:4444`. With a depth of 2, you can still easily win. With a depth of 4, the agent becomes better. With a depth of 8, the agent is pretty strong. But at such a depth, the agent is also pretty slow. You can make two robot agents play each other and see that deeper agents are better players.

To see what a robot sees, add the `-r` flag to the command line.

III.1 Add Threading

One solution to resolve this issue is to use multiple threads to parallelise the workload since we are faced with a compute-bound task. Our choice is to use a wrapper around an existing `SyncEvaluator`.

To activate the wrapper, add the -t flag to the command line. Now you can stop playing and watch robot fight with `robot host localhost:4444 6 -t` and `robot client localhost:4444 7 -tr`. Notice that the robot with the larger depth does not lose, only stalls or wins.

Implement `threaded_wrapper.rs`. Use `std::thread::spawn` to create threads. You can look into `min_max.rs` for inspiration. You only need to parallelise the first move, then delegate evaluation to eval member.

You can test your work by comparing `robot host localhost:4444 6 -t -r` to `robot client localhost:4444 7 -r`. Both should take the same reflection time, but the first goes one move deeper in its analysis.

III.2 Knowledge Cache

To speed up our analysis, we can use a cache to store already computed values for later use, also called *memoization*. To add caching, add the `-c` flag to the robot command line. **In `bin/robot.rs`, complete code to have threading and caching at the same time.**

Notice that the compiler will not let you use `KnowledgeCacheSingleThread` since it uses `RefCell`, which is not `Sync`. We need to implement a thread-safe cache. **Implement `KnowledgeCacheMultiThread`.**

There are two possible ways:
- Using `std::sync::mutex` for a classical mutex.
- Using `parking_lot::RwLock` for a mutex with multiple readers.

Choose the one you prefer and take inspiration from `KnowledgeCacheSingleThread`.

Please note that caching here is not very effective.

III.3 Making our Agent Async

Add the `-a` flag to the command line. This will create an async task that will write to the terminal at fixed time intervals. **Does `robot host localhost:4444 -ta 7` have the intended behavior? Why?** Why does the `AsyncEvaluator` resolve this issue?

But for this, we will use a `Future` that:

- Creates a thread
- Launches an expensive calculation inside the thread
- Returns pending until the calculation is terminated
- Joins the thread and returns the value

Implement this in `blocking_future.rs`. Be very careful when joining the thread to prevent any blocking calls. Also, thread scheduling is not deterministic, so your `Future` might work most of the time but break sometimes, so try to think about every possible interweaving.

Remember that Futures are only polled if they are awaited. You may want to look inside the futures crate for useful Futures built from other Futures. Complete `BlockingTaskWrapper` in `async_wrapper.rs` in a similar way as ThreadedPolicy. You can use your own BlockingFuture or use Tokio's spawn_blocking.

To use `AsyncEvaluator`, use the `async_robot` command instead of `robot`. Is the behavior correct now?

IV Further Work

This is not mandatory and is only here as a complement to better match the true implementation of the blocking future in real work framework. **You can try to implement this, but it's not required.** Create a structure that will hold a fixed number of threads, called `ThreadPool`. Implement `ThreadPool::new(n: usize) -> Self` to initialize the structure. Implement `ThreadPool::execute<F : FnOnce () -> X>(&self, f: F) -> Future<Output = X>`, which returns a future similar to blocking_future but instead of starting a new future, it will take a future from the pool and return it after calculation.

You can add other constraints on `F` or `X`.