Distributed Systems: Large Assignment 1

Module system

Your task is to implement a module system supporting arbitrary user-provided types of modules and messages. The solution shall take the form of a library and follow the asynchronous programming paradigm. A template and public tests are provided in this package.

The module system is a bit different from the executor system described in Lab 03. For details, see the Aditional Requirements section.

Module system specification

Your solution shall have the same public interface as the one provided in `solution/lib.rs`, except for adding fields to the `TimerHandle`, `System`, and `ModuleRef` structs. You can add private items and change the file structure or split it into multiple files.

The module system interface provides the following functionality:

- Creating and starting new instances of the system (`System::new()`).
- Registering modules in the system (`System::register_module()`). The `Module` trait specifies bounds that must be satisfied by a module. Registering a module yields a `ModuleRef`, which can be then used to send messages to the module.
- Sending messages to registered modules (`ModuleRef::send()`). A message of type `M` can be sent to a module of type `T` if `T` implements the `Handler<M>` trait. A module should handle messages in the order in which it receives them.

  A message is considered as delivered after the corresponding `ModuleRef::send()` has finished. In other words, the system must behave **as if** `ModuleRef::send()` inserted a message at the end of the receiving module's message queue.
- Creating new references to registered modules (`<ModuleRef as Clone>::clone()`).
- Scheduling a message to be sent to a registered module periodically with a given interval (`ModuleRef::request_tick()`). The first tick should be sent after the interval elapsed. Requesting a tick yields a `TimerHandle`, which can be used to stop the sending of further ticks resulting from this request (`TimerHandle::stop()`).

  `ModuleRef::request_tick()` can be called multiple times. Every call results in sending more ticks and does not cancel ticks resulting from previous calls. For example, if `ModuleRef::request_tick()` is called at time 0 with interval 2 and at time 1 with interval 3, ticks should arrive at times 2, 4, 4 (two ticks at time 4), 6, 7, …
- Shutting the system down gracefully (`System::shutdown()`). The shutdown should wait for all already started handlers to finish and for all registered modules to be dropped. It should not wait for all enqueued messages to be handled. It does not have to wait for all Tokio tasks to

finish, but it must cause all of them to finish (e.g., it is acceptable if a task handling `ModuleRef::request_tick()` finishes an interval after the shutdown).

It is undefined what happens when the system is used after a shutdown. However, you must ensure that a shutdown will not cause any panics in handlers or Tokio tasks (e.g., if a handler is already running when `System::shutdown()` is called, calls to `ModuleRef::send()` in that handler must not panic).

The `Message` trait defines what is expected from messages in the module system. It must be also possible to use `ModuleRef` as `Message`.

The `public-tests/tests/modules.rs` file contains an example of how the module system can be used. In the example there are two modules: `Ping` and `Pong`. After being registered in the system, they are sent references to each other so that they can communicate. Then, one of the modules sends a `Ball` message to the other module. That module replies to it with a new `Ball` message, to which the first module replies with the next `Ball` message, and so forth.

Additional requirements

Your solution should be **asynchronous** and allow multiple modules to execute **concurrently**. It will be run using Tokio. Keep in mind you can run as many Tokio tasks as you want.

In this assignment, you are **not** supposed to implement an executor system like the one from Lab 3. In particular:

- You should not create any threads or Tokio runtimes. You solution will be run in an already existing Tokio runtime.
- You should not use a system-wide message queue for all modules or a system-wide loop for executing all modules' handlers.

You do not need to remove a module from the system when its last `ModuleRef` is dropped.

Ticks requested by `System::request_tick()` must be delivered at specified time intervals. There shall be no drift, that is, the difference between the expected and actual number of ticks sent so far in any moment after `ModuleRef::request_tick()` is called must be bounded.

The module system must not panic unless a registered module panics (because of a user-provided message handler).

Varia

You can use logging if you want to, but do not emit a large amount of logs at levels $\geq$ `INFO` when the system is operating properly. All logging must be done via the `log` crate.

You can only use crates specified in the provided `Cargo.toml` file.

Hints

You might encounter the following problem: what type to use for "some message that can be handled by a module of type `T`"? One way of dealing with this issue is to introduce a private helper trait:

```
1 #[async_trait::async_trait]
2 trait Handlee<T: Module>: Message {
```

```
    async fn get_handled(self: Box<Self>, module_ref: &ModuleRef<T>, module: &mut T);
}

#[async_trait::async_trait]
impl<M: Message, T: Handler<M>> Handlee<T> for M {
    async fn get_handled(self: Box<Self>, module_ref: &ModuleRef<T>, module: &mut T) {
        module.handle(module_ref, *self).await;
    }
}
```

You can then use a trait object of type `Box<dyn Handlee<T>>`. The traits `Handler` and `Handlee` are related in a way similar to `From` and `Into`. An important difference is that here we use `self: Box<Self>`. Thanks to that when calling `get_handled()` we can move the box rather than the boxed value. The latter would be impossible, because in `Box<dyn Handlee<T>>` the boxed value has a statically unknown size.

Testing

You are given a subset of official tests. Their intention is to make sure that the public interface of your solution is correct, and to evaluate basic functionality.

Your solution will be tested with the stable Rust version `rustc 1.82.0 (f6e511eec 2024-10-15)`.

Grading

Your solution will be graded based on results of automatic tests and code inspection. The number of available and required points is specified in the Passing Rules described at the main website of the course. If your solution passes the public tests, you will receive at least the required number of points.

Asking questions

Questions **must** be asked on a dedicated Moodle forum. This way everybody will be able to read the answers. Try to ask questions early if there are any. We will try not to require any changes to existing solutions when providing answers.

Submitting solution

Your solution must be submitted as a single `.zip` file with its name being your login at students (e.g., `ab123456.zip`). After unpacking the archive, a directory path named `ab123456/solution/` must be created. In the `solution` subdirectory there must be a Rust library crate that implements the required interface. Project `public-tests` must be able to be built and tested cleanly when placed next to the `solution` directory.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski, M. Matraszek.