Distributed Systems Lab 03

Event-driven shared-nothing architecture

During the previous lab, we presented how to code multi-threaded programs in Rust and discussed how Rust facilitates assuring memory safety and thread safety of multi-threaded programs. This lab presents a relatively efficient and easy-to-implement architecture for distributed computing, and discusses how to implement in Rust its simplified version. We will use files contained in this package, which you should download and extract locally.

Learning Section

Event-driven shared-nothing architecture

The **event-driven shared-nothing architecture** is based on modules. A **module** is a standalone unit that provides particular functionality and operates on some internal data. When some module, *A* (let us introduce names to simplify the description), requires functionality or data provided by some other module, *B*, it explicitly sends a **message** to *B*. *B* processes the message, executes the requested operation or fetches the required data, and, if necessary, sends back a reply message to *A*. Importantly, following the **shared-nothing** paradigm, all messages must be self-contained. They cannot reference any data or structures internal to the modules. In particular, when *A*'s private object, *O*, is required by *B*, *A* should send in the message to *B* a copy of *O* (presumably in a serialized form). In other words, a state of a module is always accessed only by this module. As a result, no synchronization between the modules is required when they are accessing their states.

An actual module can be implemented in many ways, depending on one's needs. For instance, in a simple system, modules can be implemented as Rust `structs`, and their functionalities being executed in multiple threads. The architecture can be also applied to model large-scale systems, in which, for example, modules are implemented as services running on different machines and exchange messages over a network. The shared-nothing paradigm greatly simplifies management and synchronization of the modules and facilitates distribution transparency.

From the architecture's perspective, any module is just a set of message handlers that operate on some private data. The messages, in turn, are generated as a result of some events taking place in the system. In particular, any code of the module is executed only in response to a message received by the module. This message, in turn, is also a result of some earlier event, in particular, the initialization of the entire system. This **event-driven** paradigm allows for eliminating any synchronization also within a module itself. More specifically, the module can be modeled as a **monitor**, being designed to process only a single message at a time.

A drawback of this monitor-based approach to completely eliminate any synchronization is, however, a need to make all operations performed by the module **asynchronous**. In other words, after sending a request message to module *B*, module *A* cannot block itself waiting for the reply from *B* to complete its operation. Instead, it shall suspend the current operation by saving any relevant data (e.g., details of the request) in its private state and start processing another message it receives. When module *A* finally receives the reply message from *B*, it shall load necessary data from in its private state (e.g., the details of the request), and complete the operation using the data received in the reply message

from *B*. As a result, the private data of a module usually contain **finite state machines** that represent the operations being performed by the module.

Executor system

The described above logical architecture facilitates efficient realizations. One of them, an **executor system**, is a performant physical architecture, well-suited for relatively non-intricate multi-threaded programs (but not only).

An executor system is based on **executors** that are mapped one-to-one to operating system threads. Modules are assigned to the executors, and each executor can serve multiple modules. However, each module is assigned to exactly one executor, as this simplifies implementing the modules as monitors. The assignment is done when the system boots and does not change throughout the system's lifetime.

Each executor has a **queue** to which messages targeting its modules are inserted. The executor operates in a simple infinite loop: (1) removes the first message from the queue, (2) dispatches the message to an appropriate module by executing the module's handler specified for this message. If executing the handler involves sending a message to another module, the module appends the message at the end of the queue belonging to the executor of the destination module (this operation may require inter-thread communication). Consequently, a message shall contain such information as its source module, its destination module, and the message type (the last information is necessary for the executor to invoke the right handler).

The executor system decouples operating system threads from modules, as this can improve efficiency. In particular, the number of actual threads can be adjusted to match the number of available CPU cores, rather than to match the number of modules, and thus minimize the overhead due to thread switching. Moreover, little synchronization between the executors (threads) is required, as the only concurrent access occurs when the executors' queues are accessed, and it can be managed relatively inexpensively. Finally, all executors (threads) are created only at startup. This way problems related to memory management and garbage collection, which are common in multi-threaded applications, are avoided.

Implementing the architecture in Rust

We will learn about fully-fledged asynchronous programming in Rust in the next labs. Now, we'll focus on implementing a Proof of Concept version of the aforementioned architecture.

Consider an exemplary system consisting of two *PingPong* modules that alternately send messages to each other: when one module receives a message, it replies to the other module with a new message. The *PingPong* module is implemented in Rust as a `struct`:

```rust
struct PingPong {
    /// Name of the module.
    name: &'static str,
    /// Number of messages received by this module.
    received_messages: usize,
    /// Reference to the queue of the other module's executor.
    other_queue: ???, // We will shortly see what the type must be.
}
```

When the system is being initialized, two such objects are created and passed to the executor. The message and the *PingPong*'s message handler are implemented as follows:

```rust
// The message, Ball, is just a single String
// as it suffices for this simple system:
type Ball = String;

impl PingPong {
    fn handler(&mut self, msg: Ball) {
        self.received_messages += 1;
        println!(
            "{} receives {}. It has received {} message(s) so far.",
            self.name, msg, self.received_messages
        );
        self.other_queue.put_msg(Ball {}) // ???
    }
}
```

The above attempt to implement the message passing highlights an important issue related to the ownership: if the executor is the owner of the queue and of the modules, how can the module issue a message to the other module? Conceptually, the reference to the queue must be `Send`, `Sync`, and easy to clone (it needs to be passed to all modules requiring it). It must also allow of inserting messages to the queue, of course. However, the reference stored in the module cannot own the queue entirely because the executor needs to be able to retrieve messages from it.

Channels

In Rust, the queue can be implemented using a **channel**. A channel, in short, is a queue having its input and output ends decoupled and satisfying all the aforementioned requirements. Such multi-producer, single-consumer FIFO queue communication primitives are available in Rust standard library: `std::sync::mpsc`.

However, during this course, we will use channels provided by the crossbeam crate (library), which support several extra features. We can add an external dependency to a Cargo project by executing in a shell:

```
cargo add crossbeam-channel
```

or simply listing the crate name in `Cargo.toml`:

```
[dependencies]
crossbeam-channel = "0.5"
```

You can find more crates in the official repository at crates.io. Please do not include extra dependencies in your solutions, unless explicitly allowed.

With the new dependency added, we can use it just as if it were in scope:

```rust
fn crossbeam_example() {
    // Create a channel of unbounded capacity.
    // Two channel's ends are returned:
    let (send_end, receive_end) = crossbeam_channel::unbounded();

    // To send a message use the sending end:
    send_end.send("Message").unwrap();

    // To receive a message, use the receiving end:
    println!("A new message is received: '{}'", r.recv().unwrap());
}
```

More examples are provided in `examples/crossbeam.rs`.

Simplified executor system

A rough idea how the executor system with the *PingPong* modules can be implemented, including a demonstration how to use a channel as the queue, is provided in `examples/tennis.rs`. The example, however, is not a full-featured executor system, as it contains multiple simplifications. For instance, there is only one executor (the system is single-threaded), it supports only one module type (*PingPong*) and only one simple message type (*Ball*), the modules are registered statically, it assumes that there are exactly two modules, and they have particular names, it runs for a specified number of steps, and so forth.

Small Assignment

Your task is to implement in Rust a more advanced executor system (although still not a full-featured one) with two modules calculating the Fibonacci sequence by exchanging messages between each other.

The system shall: - consist of one executor, - support a single type of modules (`Fibonnacci`), - a single, but complex, type of messages (`FibonacciSystemMessage`).

It shall support dynamic module registration (when created, a module is registered in the executor by itself) for any number of modules. In other words, even though only two modules make sense when calculating the Fibonacci sequence in the algorithm presented next, the executor system *shall not make any assumptions about the number of modules*. Furthermore, it should not involve any logic of computing the Fibonacci sequence apart from handling the `RegisterModule` message to register a module and the `Done` message to shut down the executor. The executor shall run in a separate, dedicated thread.

To compute the Fibonacci numbers, one module, *A*, shall be initialized with `0` (the 0th Fibonacci number), and the other one, *B*, with `1` (the 1st Fibonacci number). The calculation begins when module *B* receives an *Init* message. Module *B* sends its number to module *A*. Module *A* calculates the 2nd Fibonacci number, saves it locally, and sends it to module *B*. Module *B* receives the message, calculates the next Fibonacci number (by adding the received number to the stored number), saves it locally, and sends it to module *A*. And so on, back and forth. The process continues until the required index in the Fibonacci sequence is reached (`FibonacciModules` should track the current index). When one module notices that the required number is calculated, it sends a `Done` message to the executor, and the whole program should be terminated gracefully.

For details about the message types, refer to the provided template. The template also asks you to include in your implementation a special `println!()` statement to log progress of the calculations. An exemplary correct output is provided in `main.rs`. Remember to **not change the public interface** defined in the template, as your solution will also be unit tested.

Solutions that calculate the Fibonacci numbers directly (e.g., without a separate thread serving the modules, accepting the messages, and delivering them to the modules) will be discarded.

Additional Homework

The first Large Assignment will be dedicated to implementing a full-featured executor system, so it is worth to review the concept again to make sure that you understand it.

If you want to see some real-world implementation of the event-driven shared-nothing architecture, we recommend browsing the Riker framework. The **actor model** it implements is essentially what we call the event-driven shared-nothing architecture, and the **actor** is what we call the module.

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