RPC Resource Management and Remote Objects with Rust

Class Project for CS 244b
Spring 2022 quarter, Stanford

Tim Chirananthavat
(NetID: timch)

June 2022

Abstract

Resource management (e.g., deallocating unused memory) is difficult, even in a program on a single machine. When it comes to distributed settings using RPCs, existing RPC protocols do not provide much help, and require programmers to manually and explicitly manage resources. Rust has language features that make resource management more manageable while allowing explicit control by the programmer. We present an RPC system[1] that leverages Rust's features to allow for pointers that reference remote objects, similarly to network object systems, and deallocating said remote objects exactly when they're finished being used.

1 Background

The Cap'n Proto RPC protocol[3], unlike other popular RPC protocols, is capable of sending opaque objects over the networks, effectively resulting in a remote object system or distributed object system. However, it is unclear from the documentation how memory management is done in Cap'n Proto. Manual memory management seems unwieldy, so the other two options are either leaking all unused memory until the connection is closed, which is bad for long-running connections, or some sort of distributed garbage collection algorithm is run, which is probably inefficent.

Rust provides an alternative. Rust uses compile-time checks to automatically determine exactly at which point in the code an object should be deallocated, while ensuring that each object is deallocated exactly once. This gives us the efficiency of manual memory management, but some of the convenience and all of the safety and security of garbage collection.

We use Rust's unique features to implement an RPC system which has remote objects (called *services* in our system), while properly deallocating unused objects.

2 Design

2.1 Usage

In this section, we'll examine how our RPC system could be used.

In our current design, the RPC system is used between two parties, the server and the client. The server creates a TCP listener, and waits for the client to connect to it. The client can then make RPC calls to the server, and ends the interaction by closing the TCP connection. The server stores one set of state data per client connection, with each set of state data being separate from each other.

I will use the term *server* to either mean one of the two communicating parties, or mean the concrete implementation on the server side of a certain service instance.

[1] Code can be found at `https://github.com/theemathas/rusty_rpc`

```
struct Foo {
    x: i32,
    y: Bar,
}
struct Bar {
    z: i32,
}
service MyService {
    foo(&mut self) -> i32;
    bar(&mut self, arg: i32) -> i32;
    baz(&mut self, arg1: i32, arg2: Foo)
        -> Foo;
}
```

Listing 1: File `hello_world.protocol`

2.1.1 Basic usage

Like most if not all RPC systems, the user of our system needs to write a file specifying what RPC calls are allowed. We'll call this file a *protocol file*. An example of a simple protocol file can be found at listing 1.

The user defines `struct` data types. Currently, the supported types of fields are `i32` (Rust's 32-bit integer type), and other structs. The user also defines `service`s. These are translated into Rust traits. If a Rust object implement that trait, then other code can call the methods.

A protocol file is translated into generated Rust glue code and user-facing Rust types and traits. The user-facing portion of the translation for `hello_world.protocol` can be found at listing 2. The `#[async_trait]` attribute (from an external library[2]) is used to allow async functions being used in traits, which Rust does not currently support natively yet. The return types are wrapped in `io::Result`, signalling that the function might result in an I/O error, since this trait might be called from the client side. The `Send + Sync` trait bounds is required for usage with the `tokio`[5] async runtime.

Client code using this protocol file can be seen at listing 3. It uses the `interface_file!` macro to load the interface file. It uses the `await` keyword to asynchronously wait for the calls to finish, and it uses the `.unwrap()` method

```
use std::io;
use async_trait::async_trait;
#[async_trait]
pub trait MyService: Send + Sync {
  async fn bar(&mut self, arg: i32)
    -> io::Result<i32>;
  async fn baz(&mut self, arg1: i32,
    arg2: Foo) -> io::Result<Foo>;
  async fn foo(&mut self)
    -> io::Result<i32>;
}
```

Listing 2: User-facing portion of the translation of `hello_world.protocol`

to (poorly) handle I/O errors by crashing. In the `start_client` function call, the client needs to specify that it expects the server to (initially) provide the `MyService` service. For the `.close()` method calls, see section 3.4. Note that the client can call the `service` like any other method, as though the service was running locally.

Server code using this protocol file can be seen at listing 4. It shares some similarities with the client code, so we will describe only the differences here. The code defines `MyServiceServer` to be a concrete type implementing the abstract `MyService` interface. The implementation is marked with the `#[service_server_impl]` attribute, which is a macro that generates the necessary glue code. The `MyServiceServer` type can contain data necessary for the server, but in this case it does not. To call `start_server` for a type, that type has to implement the `Default` trait, which means that our RPC system can create more values of the `MyServiceServer` type on demand, one per each client.

2.2 Returning service references.

Our RPC system has a unique feature, inspired by Cap'n Proto[3], where a service can spawn child services (which might even be other instances of that same parent service). A parent service must wait for the child service to be *dropped* (terminated) before the parent service can be dropped. Furthermore, while the child

```rust
use tokio::net::TcpStream;
use rusty_rpc_lib::start_client;
use rusty_rpc_macro::interface_file;

interface_file!(
    "path/to/hello_world.protocol"
);

#[tokio::main]
async fn main() {
  let stream =
    TcpStream::connect("127.0.0.1:8080")
      .await.unwrap();
  let mut service = start_client::
        <dyn MyService, _>(stream).await;
    let foo_result = service.foo().await
        .unwrap();
    assert_eq!(123, foo_result);
    service.close().await.unwrap();
}
```

Listing 3: Client-side code using `hello_world.protocol`

service has not been closed yet, the parent service cannot be used (but see section 3.1). This is enforced both at runtime (using dynamic checks, see section 2.3.1) and compile time (using Rust's features, see section 2.3.2).

To define a method that returns service references, the following syntax is used in the protocol file:

```
child(&mut self) -> &mut service ChildService;
```

The server code and the client code for using service references can be seen in listings 5 and 6, respectively.

Note that now the return type of the method has a `ServiceRefMut<...>`. This type is a proxy type. On the server side, it's just a no-op wrapper around a server. However, on the client side, it holds a `&mut dyn ChildService` (a reference to a *trait object* in Rust) and when used as a `ChildService`, will send the method calls over the network.

In the server code, the child server is given a reference to the parent server. That is, the child

```rust
use std::io;
use tokio::net::TcpListener;
use rusty_rpc_lib::start_server;
use rusty_rpc_macro::{interface_file,
  service_server_impl};

interface_file!(
  "path/to/hello_world.protocol"
);

#[derive(Default)]
struct MyServiceServer;
#[service_server_impl]
impl MyService for MyServiceServer {
  async fn foo(&mut self)
  -> io::Result<i32> {
    Ok(123)
  }
  // Other methods omitted.
}

#[tokio::main]
async fn main() {
  let listener =
  TcpListener::bind("127.0.0.1:8080")
    .await.unwrap();
  start_server::<MyServiceServer>(
    listener).await.unwrap();
}
```

Listing 4: Server-side code using `hello_world.protocol`

```
async fn child(&mut self) ->
io::Result<ServiceRefMut<
dyn ChildService>> {
  Ok(ServiceRefMut::new(
    ChildServer::new(self)
  ))
}
```

Listing 5: Server-side code for returning services.

server is borrowing data from the parent server. This means that until it is dropped, the child server has *borrowed* exclusive access to the parent server's data (unless a grandchild is spawned, in which case the chain of borrowing of access rights continues).

Note that in the client code, the child services must be dropped before the parent service can be used again. In Rust, an object can be manually dropped (as in `child_service_1`), or it can be automatically dropped when it goes out of scope (as in `child_service_2`). In either case, Rust keeps track of when objects are dropped, in order to maintain the invariants mentioned at the beginning of this section.

Again, for the `.close()` method calls, see section 3.4.

2.3 Implementation

In this section, we'll examine how our RPC system was implemented.

The RPC system is implemented in two parts. One part is a normal library, and the other is a *procedural macro*, which is a Rust feature which allows code to generate code. Our RPC system has two procedural macros: `interface_file!(...)` and `#[service_server_impl]`. These two macros generate the glue code necessary for the RPC system to function.

The server and the client both uses the tokio[5] runtime to run asynchronously. The server, once it gets a connection from the client, sends and receives a series of messages. The messages are delimited by a header specifying how large each message is, and each message is serialized using

```
let mut parent_service = ...;

let mut child_service_1 =
  parent_service.child()
  .await.unwrap();
child_service_1.do_something()
  .await.unwrap();
drop(child_service_1);
/* Compilation will fail if
the above line is omitted.
Compilation will also fail if
child_service_1 is used after
this line. */

// Equivalent to above.
{
  let mut child_service_2 =
    parent_service.child()
    .await.unwrap();
  child_service_2.do_something()
    .await.unwrap();
  child_service_2.close()
    .await.unwrap();
}

parent_service.close()
  .await.unwrap();
```

Listing 6: Client-side code for returning services.

the MessagePack[4] format.

When the client makes an RPC call, it sends a message containing a service ID, a method ID, and the serialized arguments. The server then finds the specified service instance, runs the RPC call to completion, and sends a message back to the client containing the return value. RPC calls are processed in sequence serially (but see section 3.3).

2.3.1 Service references

Each service instance is assigned a 64-bit service ID. The initial service is assumed by both parties to have ID zero. When a subsequent service is spawned, the next service ID is incremented and assigned to this service. On overflow, the ID wraps around and keeps being incremented until an unused ID is found.

In Rust, a *mutex guard* (written as `MutexGuard` in code) is a type or a value of that type, representing the rights to access the data behind a locked mutex. Creting a mutex guard requires locking that mutex. When a mutex guard is dropped (deallocated), the mutex is unlocked.

The server side maintains a *server collection*, which is a hash map, mapping from service IDs to servers. Each server in the server collection is guarded by a mutex. Each server in this collection (except the initial server) is associated and stored with a mutex guard representing access rights to that server's parent. As a result, as long as the child server has not been dropped, nobody else can access the parent server, and the parent server also cannot be dropped.

When a server creates and returns a `ServiceRefMut` object, the RPC system considers that to be a newly created server, and assigns it a new service ID. This service ID is then sent across the network and stored in the client-side `ServiceRefMut` object which has a service proxy inside. This object will then transmit back this service ID when a method is called on this service. Once the `.close()` method is called on a service proxy, the client transmits a request to the server to drop the service, and then waits for the server to respond with a confirmation before proceeding.

A method in a service that returns a service reference is translated to a Rust method such as below:

```
async fn child(&mut self) ->
  ServiceRefMut<dyn ChildService>
```

which is equivalent to below (due to *lifetime elision*)

```
async fn child<'a>(&'a mut self) ->
  ServiceRefMut<'a, dyn ChildService + 'a>
```

Rust is then able to connect the return value with the `self` argument via the `'a` lifetime, meaning that the return value might contain references to data in `self`.

2.3.2 Traits and code generation

The following traits are used in order to bridge the gap (in a type-safe way) between the RPC library and the code generated by the procedural macro:

- `RustyRpcStruct`: This trait is implemented by the `interface_file!()` macro for all structs. The type `i32` also implements this trait. That is, this trait is implemented all arguments and all return values except for return values that are service references.

  Types implementing this trait can be serialized and deserialized.

- `RustyRpcServiceClient`: This trait is implemented by the `interface_file!()` macro for the trait object corresponding to the service trait. For example, in 1, the type `dyn MyService` (which is the dynamic-dispatch version of the `MyService` trait) implements `RustyRpcServiceClient`.

  Types implementing this trait has a corresponding proxy type that implements `RustyRpcServiceProxy`, and is therefore legible for being in a `ServiceRefMut`, e.g. `ServiceRefMut<dyn MyService>`.

- `RustyRpcServiceProxy`: This trait is implemented by the `interface_file!()`

macro for a newly generated type. For example, in 1, a type named `MyService_RustyRpcServiceProxy` is generated which implements this trait.

Types implementing this trait can be constructed from a service Id.

- `RustyRpcServiceServer`: This trait is implemented by the `#[service_server_impl]` macro for server types. For example, in 4, the type `MyServiceServer` implements this trait. The way this is implemented is that it calls a method added to the (for example) `MyService` trait by the `interface_file!()` macro.

  Types implementing this trait can be stored in a server collection. In order for this to work, types implementing this trait can be given a method ID and a bunch of opaque bytes representing the arguments.

  Types implementing this trait can also be used as the initial service in the `start_server()` method.

- `RustyRpcServiceServerWithKnownClientType`: This trait is implemented alongside the `RustyRpcServiceServer` trait. The difference is that, as the name suggests, this trait contains type-level information on what the corresponding service trait is being implemented. For example, in 4, the type `MyServiceServer` implements `RustyRpcServiceServerWithKnownClientType<'a, dyn MyService>` for all lifetimes `'a`.

The above traits should not be manually implemented by the users of the RPC system.

3 Possible future extensions

There are some possible extensions that were envisioned in designed, but were not actually implemented due to time constraints. They are discussed here.

3.1 Shared references

Unfortunately, with the current design, only one service can be in use at a time. This is enforced at runtime by a mutex and at compile time by a chain of `&mut T` references, which are exclusives. It is possible to extend the syntax to allow not only `&mut T` references to services, but also `&T` references, which are nonexclusive shared references. Such references have behavior mirroring a read-lock in a readers-writer lock, and can be enforced at runtime by such a mechanism instead of a mutex.

3.2 Independent (non-borrowed) services

In section 2.3.1, we see that a child service always borrows from a parent service. Therefore, the parent service cannot be used while the child service is active (unless the section 3.1 is used). It might be sometimes desirable to spawn a child that is completely independent from the parent, so both services can be active concurrently.

3.3 Concurrency and multiple clients

Once section 3.1 or section 3.2 is implemented, there can now be multiple active services at the same time. We could then allow for multiple concurrent RPC calls at the same time. This would require adding an RPC call number to each request from the client, and to each response from the server. Once we have this set up, it should be relatively easy to allow for multiple clients to share access to the same server state.

3.4 Async drop

In the current design, the `.close()` method (see section 2.3.1) on a service proxy causes the corresponding server-side object to be dropped. The original design did not have this method, but merely dropping the service proxy would be sufficient. This would require the destructor (a.k.a. drop implementation) to run async code, which is currently impossible in Rust. There appears to be a plan[1] to make this "async drop" feature possible, but it is unclear.

References

[1] *async fn fundamentals initiative: Async drop*. URL: https://rust-lang.github.io/async-fundamentals-initiative/roadmap/async_drop.html.

[2] *async-trait Crate*. URL: https://github.com/dtolnay/async-trait.

[3] *Cap'n Proto: RPC Protocol*. URL: https://capnproto.org/rpc.html.

[4] *MessagePack Serialization Format*. URL: https://msgpack.org/.

[5] *Tokio Crate*. URL: https://tokio.rs/.