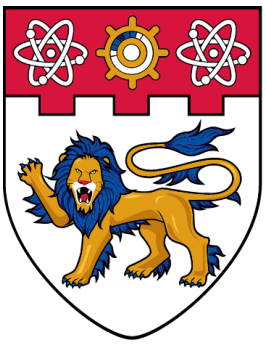
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

NANYANG TECHNOLOGICAL UNIVERSITY

College of Computing and Data Science (CCDS)

SC4051: Distributed Systems



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Academic Year: 2024-2025, Semester 2

Assignment Report

M Hisham B Khairul A (U2121992E)

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Contents

1. Introduction

Objective

The project involves the construction of a client and server program for managing a facility booking system through UDP sockets. This report explains the details of the project's UDP communication, marshalling/unmarshalling, and reliability mechanisms, and briefly explores the mechanisms' part in enforcing at-least-once and at-most-once semantics.

Environment/Execution

Both client and server are developed in Rust. They can be compiled on any target that has Rust installed. The executables should be runnable on any machine of the same architecture.

Below are instructions for building and running:

```
## server
# build it
cd server
cargo build --release
cd ../target/release
# see definition for args
.\server -h
# ...for eg, run with caching + 30% packet drops
.\server -u -p 0.3

## client
# build it
cd client
cargo build --release
cd ../target/release
# see definition for args
.\client -h
# ...for eg, run with retries + 60% packet dupes
.\client -u -d 0.6
```

Assumptions

The following assumptions were made:

- **Packet integrity** - Any UDP packets successfully received are assumed not to be corrupted
- **Endianness** - Endianness of data consumed is assumed to be same as platform endianness (most modern architectures use little endian)
- **Single-threaded** - Both client and program handle requests/responses one at a time
- **Storage** - Server data is not persisted
- **Message layout** - A successfully unmarshalled message is assumed to be correct
- **Client monitoring** - A monitoring client is responsible for halting its monitoring on its own; the server does not tell the client to
- **Booking times** - A booking always starts and ends on the same day

2. Design

Client

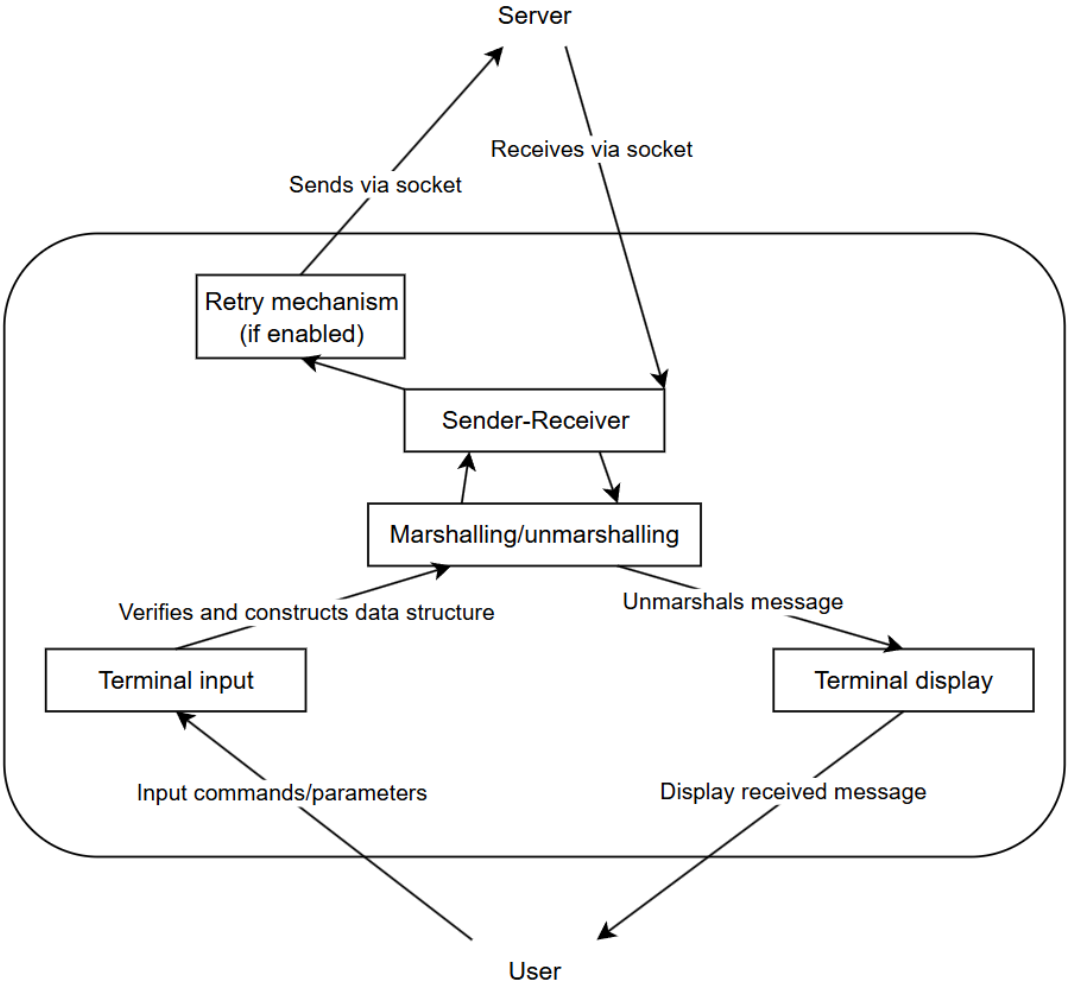


Figure 1.1 - Client architecture

The client code can be found in ***/client/src***. It consists of 2 main components, the input/output in ***main.rs*** and the socket wrapper in ***socket.rs***.

The input/output is responsible for obtaining commands and parameters from the user via the terminal, verifying them, constructing the corresponding message, and passing it to the socket wrapper. Then, when the response arrives, it displays it and waits for the next input.

The socket wrapper wraps the UDP socket with (un)marshalling and retry functionality. If a response isn't received for a sent message after a certain timeout, it is assumed lost and the retry mechanism triggers. The retry mechanism and a packet duplication rate (for testing) can be configured via command-line arguments.

The (un)marshalling is implemented directly on message data structures and is explained further below.

Server

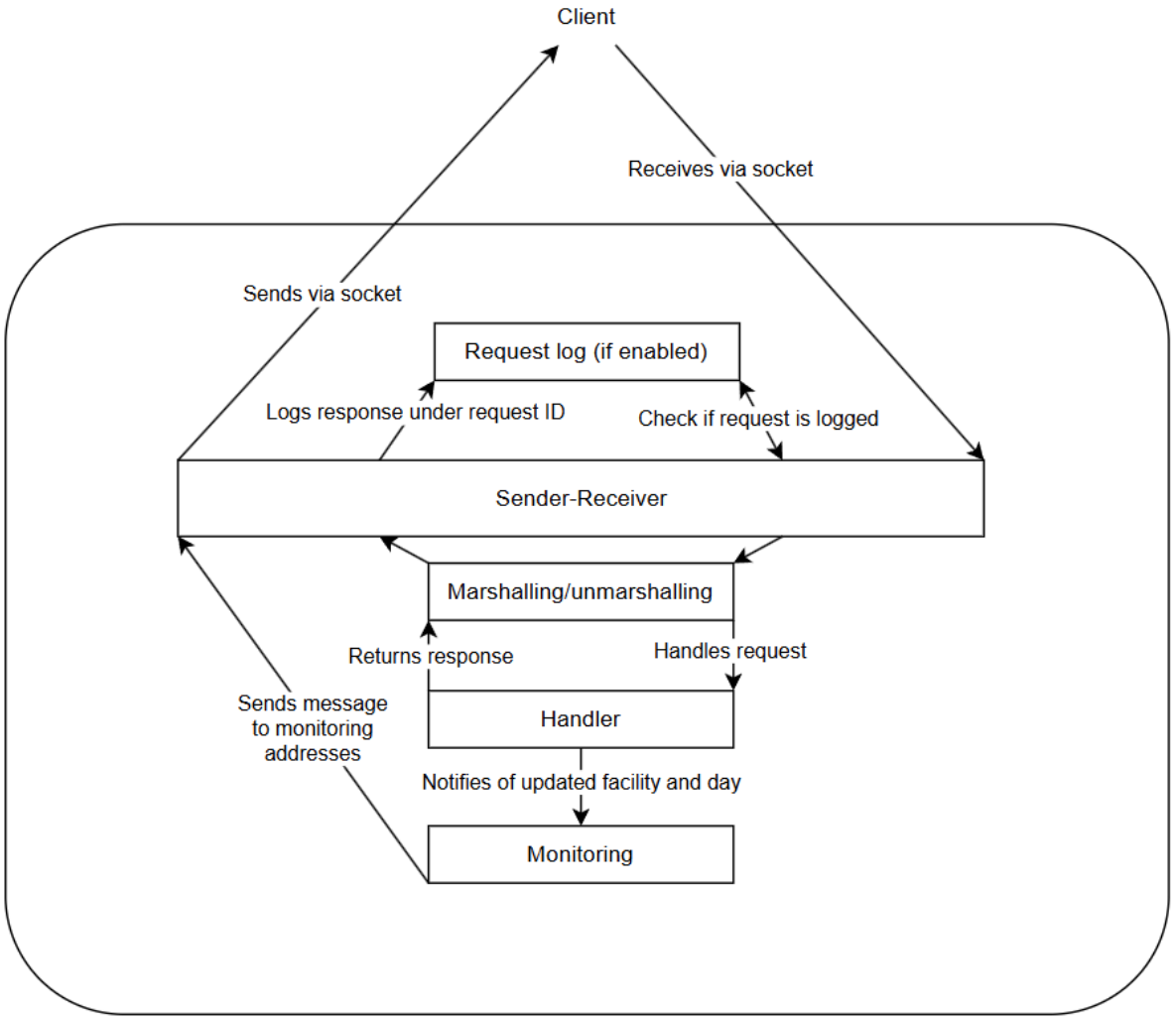


Figure 2.2 - Server architecture

The server code can be found in ***/server/src***. Its components consist of the request log in ***log.rs***, the socket wrapper in ***socket.rs***, and the handler in ***handler.rs.***

The request log stores responses corresponding to request IDs, up to a certain capacity. If an incoming request ID is found in the log, its response data is immediately returned. This is for managing duplicate messages.

The socket wrapper wraps the UDP socket with (un)marshalling and request log. The logging mechanism and a packet drop rate (for testing) can be configured via command-line arguments.

The handler is the largest component. It stores the program's state (i.e. facilities), and provides an interface for handling messages. For state-modifying messages, if successful, it also initiates the sending of updates to any addresses monitoring the involved facility.

Messages, types, and marshalling/unmarshalling

Messages and any involved types such as ***Time*** are found in ***/shared/src***.

Marshalling/unmarshalling functionality is directly implemented on the types themselves. To represent this, a trait (interface) called ***Byteable*** is defined in ***lib.rs***. This trait has 2 methods:

- ***to_bytes*** - Represents writing the type to a buffer of bytes
- ***from_bytes*** - Represents fallibly constructing the type from a buffer of bytes

The trait has the following implementation semantics:

- Static-sized (i.e *bool, u8, u16)* - Write/read the bytes directly
- Variable-length (i.e *String, Vec)* - Write/read the bytelength, followed by the data itself
- Struct - Write/read in top-to-bottom order of fields, where each field is ***Byteable***
- Enum - Write/read the discriminant as a *u8*, followed by the data itself

Byteable implementations for basic types are defined in ***lib.rs***. Request and response types are defined in ***request.rs*** and ***response.rs***. Types for time involved in bookings are defined in ***time.rs***. Time types also have implementations for ***Add*** and ***Sub***, to allow for out-of-the-box arithmetic.

One may note that there is no ***Byteable*** implementation for structs here. This is because the implementation for structs is very straightforward and rote, but also tedious. To reduce work, a procedural macro called ***ByteableDerive*** is defined in ***/derive/src/lib.rs***. By tagging a struct with ***#[derive(ByteableDerive)]***, the macro is invoked to automatically generate a ***Byteable*** implementation, like so:

```
// before...
#[derive(ByteableDerive)]
struct A {
	b: B,
	c: C
}

// ...after (as long as A and B are Byteable)
impl Byteable for A {
	fn from_bytes(data: &mut Vec<u8>) -> Result<Self, String> {
		let b = B::from_bytes(data)?;
		let c = C::from_bytes(data)?;
		Ok(A { b, c })
	}
	fn to_bytes(self) -> Vec<u8> {
		let mut bytes = Vec::new();
		bytes.extend(b.to_bytes());
		bytes.extend(c.to_bytes(());
		bytes
	}
}
```

Thus, once implementations for basic types are written, implementations for ***Byteable***-composed structs can easily be derived.

3. Services

Basic request/response

A request to the server has the following structure:

- **request_id** - A static-sized UUID to identify the request by
- **request_type** - An enum of the possible request types (defined below)

A response to the client looks like so:

- **request_id** - The initiating request's ID (or if just a monitoring message, a random UUID)
- **is_error** - A bool, indicating whether the response is an error
- **message** - A string carrying the actual response

Note that there is only 1 response type, as opposed to the many request types, as the client doesn't require any structure to its response; it simply prints out whatever it receives, for the user to read.

Availability

This service allows a user to get the availability of a facility for some given days. It is idempotent as no state is changed.

- **Input**: facility name (*String*), selection of days (array of *Day*)
- **Output**: availability for those days
- **Error**: facility doesn't exist

Book

This service allows a user to create a booking for a facility. It is idempotent as booking twice results in the same state (the booking existing).

- **Input**: facility name (*String*), start and end times (2 instances of *Time)*
- **Output**: confirmation of booking, booking ID
- **Error**: facility doesn't exist, overlaps with existing booking(s), or times are invalid

Offset

This service allows a user to offset a given booking. It is non-idempotent as calling this twice causes the booking to be offsetted twice.

- **Input**: booking ID (*UUID*), offset hours and minutes (2 *u8*), whether to offset backwards (*bool)*
- **Output**: confirmation of offset
- **Error**: booking ID doesn't exist, overlaps with existing booking(s)

Monitor

This service allows a user to register their client for monitoring a given facility. It is idempotent as calling this twice results in the same state (the client being added to the list of monitoring addresses).

- **Input**: facility name (*String*), seconds to monitor for (*u8*)
- **Output**: confirmation of registration
- **Error**: facility doesn't exist

Once successfully registered, a client will block and watch the socket for incoming messages, for the given number of seconds. On the server's side, whenever a facility is updated, a message is sent to involved addresses with the availabilities for the updated day until the address expires, at which point it is removed from the list of monitoring addresses.

Cancel - Additional Service 1

This service allows a user to cancel a booking. It is idempotent as cancelling a booking twice results in the same state (the booking being gone).

- **Input**: booking ID (*UUID*)
- **Output**: confirmation of cancellation
- **Error**: booking doesn't exist

Extend - Additional Service 2

This service allows a user to extend a given booking. It is non-idempotent as calling this twice causes the booking to be extended twice.

- **Input**: booking ID (*UUID*), offset hours and minutes (2 *u8*)
- **Output**: confirmation of extension
- **Error**: booking doesn't exist, overlaps with existing booking(s), or times are invalid

4. Invocation Semantics and Experimentation

Invocation semantics

At-most-once means that a packet is transmitted exactly one time. This means that if a packet is lost, it is lost for good. This works for cases where a small amount of data loss is tolerable, such as streaming of metrics.

In contrast, at-least-once means that a packet can be transmitted multiple times. Thus, if a packet appears to be lost (i.e no response was received for it), the sender can resend it. This works for cases where duplicated data is acceptable (i.e idempotent operations), or data can be deduplicated.

In our case, each packet represents a distinct message which may trigger a non-idempotent operation. Furthermore, losing a packet means the intended operation is not executed, which is not ideal. Thus, it is important that the client can confirm successful delivery of the packet, and that the server executes a packet's operation exactly once.

To handle this, we require at-least-once semantics. On the client side, a retry mechanism for failed deliveries is implemented, which retries deliveries with an increasing backoff. On the server side, incoming messages' IDs are checked and logged, allowing for deduplication.

Experimentation

For experimentation, the ability to disable client-side retries and server-side request logging was added. Additionally, the client's packet duplication rate and the server's packet drop rate can be configured. Together, the program can be configured to test behaviours under different semantics.

Idempotent

We first test with an idempotent operation, such as creating a booking.

For at-least-once, the retry mechanism must be used. We also use a high packet duplication rate. With request logging, the server successfully changes duplicated requests and only processes once. Without request logging, each request is independently executed; however, as the operation is idempotent, the end result is the same.

For at-most-once, the retry mechanism is turned off, and a high packet drop rate is used. When a message is dropped, the operation is not executed at all, resulting in a failure.

Non-idempotent

We now test with a non-idempotent operation, such as extending a booking.

We use the previous conditions for at-least-once. The result is the same with request logging. However, without it, duplicated requests are independently executed, with the booking being extended multiple times, resulting in failure.

For at-most-once, the result is the same as before, with a dropped message resulting in no execution, and thus failure.