Netsketch: A Collaborative Whiteboard

Assignment Report

Giorgio Grigolo

Contents

1 Introduction

This implementation of Netsketch, a collaborative whiteboard, is built entirely in Rust, a statically typed, memory-safe, idiomatic, systems programming language.

1.1 Project Structure

Netsketch was divided into three main components: the *server*, the *client*, and a *shared library*. The server is responsible for most of the business logic, such as managing the state of the whiteboard and broadcasting changes to all connected clients. The client is a graphical user interface that allows users to draw on the whiteboard and see the changes made by other users. The shared library contains the data structures and algorithms used by both the server and the client.

To conform with the Rust ecosystem's conventions, the above components are manifested as a Rust workspace, with the server and the client as separate crates, and the shared library as a library crate, initialized by the `cargo init --lib` command.

1.2 Libraries Used

A considerable effort has been made to use as few external libraries (or *crates*) as possible. The only crates used are

- `clap` for the command-line interfaces of the server and the client,
- thiserror for better error handling,
- `bincode` for encoding and decoding data structures into and from bytes,
- `macroquad` for the client's graphical user interface, and
- `tracing` for pretty server side logging.

All other functionality has been implemented from scratch, or by using the Rust standard library `std`.

2 System Design

It is clear, as specified, that this game is a client-server application. Upon reading the requirements of this project, it was immediately clear that most of the logic should take place on the server side, whereas the client, must be kept as simple as possible; it must send clear and concise instructions to the server. With this in mind, while planning the layout of this system, I concluded that I'd like to keep the client as light as possible, while letting the server doing most of the heavy lifting.

2.1 Shared Library: `ns-core`

In this section, I will describe the main features of the core of Netsketch, the library that is used by both the client and the server. I will also attempt to shed some light on the most important data structures in this library.

2.1.1 Don't panic: `Error`

One of the undesirable outcomes during the lifetime of a Rust program is the `Panic`, more commonly known as crashing. To avoid this, I slightly modified the common Rust concept of a `Result<T, E>`, where `T` is some Type, and `E` is some Error type.

In the shared library, with the help of `thiserror` crate, I renamed the default `Result` type, essentialy forcing the `Error` to be our own custom `ns_core::Error`, and avoid filling it in everytime. The advantage of this is that we can define *sub errors*, in an attempt to categorize them under the situation where they may be thrown. Most notably, we have at least an entry for each error returned by the crates we're using, as we want our `Result` type to understand as many errors we can, even if the `From` trait must be implemented manually. All this is to use the `?` notation instead of using too many `.unwrap()`s.

2.1.2 A blank slate: `Canvas`

The central part of Netsketch is the interactive whiteboard, or *canvas*, as I will refer to it from now on. It is defined as an array of *canvas entries*, and a monotonically increasing counter, to keep track of the order in which the actions were performed, as well as to serve as a current pointer to the last entry that was inserted into the canvas.

```
struct Canvas {
    entries: Vec<CanvasEntry>,
    counter: usize,
}
```

This calls the introduction of the `CanvasEntry`. The `CanvasEntry` stores the action's ID, the object drawn (the `CanvasElement`), the username of who drew it (the `author`), and a shown flag to be used by the client to determine whether the entry should be displayed or not.

```
struct CanvasEntry {
    id: usize,
    element: CanvasElement,
    shown: bool,
    author: String,
}
```

The `CanvasElement`, finally is just an enum that contains all the possible objects that can be drawn on the canvas, and the required information to draw them such as sets of coordinates, colour and so on. The objects supported are `Line`, `Rectangle`, `Circle`, and `Text`.

Note

The `Canvas` is further wrapped in another struct, which varies depending on the context in which it is used. In fact, the way it is mutated and accessed is different in the server and in the client. This will be discussed in the following sections.

2.1.3 First signs of communication: `TcpPackets`

Whilst planning how I'm going to transfer data between the client and the server, I remembered I'm using Rust, which boasts an incredible type system. Indeed, enums in Rust are quite flexible, in the sense that one can define any arbitrary struct within the same enum entry. The `TcpPacket` system I implemented makes heavy use of this. At the network level, the packet structure is quite simple, as can be seen in figure 1.

By splitting a packet in two main sections, we are able to exactly determine how many bytes to read from the stream provided by a TCP socket, and therefore also the size of the buffers which will be populated with the data.

In this way, we can make use of `read_exact(&mut [u8])`, a wrapper over the `read` system call, which keeps on reading from the stream until the mutable slice whose reference we pass to the function is full.

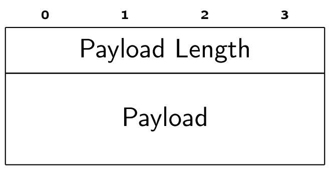

Figure 1: TCP Packet Datagram

Over the wire, any element of the `TcpPacket` enum is sent as bits, or more conveniently as a `Vec<u8>`, byte-array. How do we decide the order in which to place the bytes of the data contained in each `TcpPacket` entry? Since we already noted that enums are quite flexible, in fact, completely arbitrary, it makes the contents quite unpredictable.

Fortunately, using some meta-programming (or Rust procedural macros) we are able to uniquely encode structs into bits, given a translation table, or mathematically speaking, a bijective function that recursively maps the types of the members of the struct to the memory layout of the struct itself.

As per the `bincode` specification, we note that by default, all structures are encoded in little-endian, with various patterns for each data type. For example enums are encoded with their variant first, followed by optionally the variant fields. The variant index is based on the `IntEncoding` during serialization. For the `IntEncoding`, by default, `bincode` uses a variable integer encoding which means that the size of the integer is not fixed, but rather depends on the value of the integer itself.

All the rest is abstracted out in the `Encode` and `Decode` derive macros respectively, which when applied to our `Packet` struct, they allow us to use the `bincode::encode_to_vec` on the `Packet` on the sender's side, and `bincode::decode_from_slice` on the receiving end, whilst casting the decode function call to the `Packet` enum. This is because `bincode` is just receiving bytes: it is not aware how they are to be arranged, until we give it a type hint. Due to the fact that we are allowed to cast any bytes to any struct, this process might fail, which is why the `try_from_bytes` function implemented for a `Packet` returns a `Result`.

```rust
impl Packet { ...
fn try_from_bytes(bytes: &[u8]) -> Result<Self> {
    Ok(bincode::decode_from_slice(bytes, config::standard())?.0)
}
... }
```

2.2 The Server: `ns-server`

During the initial brainstorming sessions, it was clear to me that whenever a client drew on the canvas, sending the entire data structure to everyone over and over again was going to be quite inefficient. In this section, I will outline the main execution flow of the server, how it handles incoming packets, and most importantly, how it concurrently manages its state safely, amongst all clients.

Thankfully, it was an early stage where I made the foundational design decision whereby only the updates to the canvas had to be propagated to the clients' canvases. It was also apparent that I had to implement a cached version of the canvas for the client, as well as an *update receiver* to keep the canvas up to date. We shall talk about the client-specific implementations in a subsequent section.

2.2.1 Sharing is caring: `Arc<Mutex<T>>`

When handling incoming connections, the sever spawns a new thread for each connection. This is done to avoid blocking the main thread, and to allow the server to handle multiple clients concurrently. However, at least in Rust, it is not possible to share mutable data between threads without some form of synchronization and a method to allow us to share value ownership. The `Arc<Mutex<T»` pattern is a common way to achieve this.

The `Arc` is an *atomically reference-counted pointer*, which allows us to safely share the ownership of a value across multiple threads.When cloning an `Arc` an new instance that points to the same heap allocation as the original Arc is created, whilst incrementing the reference count. Finally, when the last Arc pointer to a specific allocation is disposed of, the value in that allocation is also dropped.

Note

`Arc` does not ensure thread safety by itself, it only protects against race conditions when sharing ownership. An example of this is when a thread tries to clone an `Arc` while another thread is dropping it.

However, in Rust, shared references prohibit mutation by default, and since `Arc` is a shared reference, we need to further wrap our inner data structure in a `Mutex` to allow for mutation, particularly amongst multiple threads. The `Mutex` is a mutual exclusion primitive provided by the Rust standard library, which allows us to lock and unlock the canvas, ensuring that only one thread can access and modify it at a time.

In essence, we are putting clients into a metaphorical queue, where they are allowed to access the canvas one by one, and only when they are done, the next client in the queue is allowed to access the canvas. This, inevitably introduces some latency, as well as an inferred order in which the operations on the canvas are performed.

```rust
let mut server_state = match server_state.lock() {
        Ok(server_state) => server_state,
        Err(_) => return Err(ServerError::LockError.into()),
};
```

The code snippet above demonstrates the `handle_client` function. It spawns a thread for client-requested operations, secures the canvas, or errors if the lock fails. This ensures thread safety and mutual exclusivity, for the server's the critical section - canvas mutation. Conveniently, Rust unlocks the mutex automatically when the scope of the lock ends.

2.2.2 What we are sharing: `ServerState`

Now that we know how the server state is passed *safely* to a thread, we can observe what it is storing, and most importantly why. In this section, we will discuss the `ServerState` struct, which is the main data structure that the server uses to manage the state of the whiteboard, as well as the clients connected to it.

```
struct ServerState {
    canvas: Canvas,
    sessions: Vec<Session>,
    users: HashMap<String, UserData>,
}
```

The `ServerState` struct contains the canvas (see Section 2.1.1), a list of sessions, and a map of usernames to the respective user data.

```
 struct Session {
     username: String,
     stream: TcpStream,
}
```

The `Session` struct contains two fields: the `username` of the client and the `TCPStream` that the client is connected to. The `std::net::TCPStreams` found in this list of sessions are what the server uses to broadcast canvas updates to the clients.

```
struct UserData {
        username: String,
        action_history: Vec<Action>,
        last_login: Option<Instant>,
    }
```

The `UserData` struct contains the client's username, their `Action` history (for the *undo* logic) and the last login time, which is used to determine when the server should adopt the client's action history. I have augmented the *draw list* as found in the assignment specification to an action history, which includes not just a series of `Draw` commands, but also `Delete`, `Clear` and `Update` commands.

2.2.3 The main *(for)* loop

With all the data structures in place, we can now discuss the main execution flow of the server.

Firstly, we call `init_server`, which does the following:

- Starts a subscriber for tracing events, which is used for logging.
- Binds the server to the specified address.
- Returns the TCP listener, which is used to accept incoming connections.

Secondly, we initialize the server state, as previously described (see Section 2.2.1).

Finally, we enter the main *for* loop, where we iterate over `tcp_listener.incoming()`, which as the Rust documentation states, is equivalent to calling `TcpListener::accept` in a loop. This `accept` method, is the same as the `accept` system call found in `sys/socket.h` in C, which blocks until a new connection is made to the server.

For this reason, we spawn a new thread for each incoming connection, inside which we loop indefinitely whilst running the `handle_client` function, which is responsible for handling the client's requests. If for any reason, the `handle_client` function returns an error, the server will log the error disconnect the user (by removing the session) and continue to the next iteration of the loop.

Limitation

It is possible, that the server could be overwhelmed by the number of clients connected to it, as it spawns a new thread for each connection. This could lead to a situation where the server runs out of resources, and the system becomes unresponsive. To mitigate this, the server could be modified to use a thread pool, which would limit the number of threads that can be spawned at any given time.

To completely unravel the server's execution flow, all that is left is to implement the `handle_client` function.

Firstly, we set up the client's `TcpStream` to timeout after 10 minutes, which is the maximum time a client can be inactive before being disconnected. We then read the first 4 bytes from the stream, which contain the length of the payload, and then read the payload itself. The thread will block until the payload is received, or until the timeout is reached.

Once the payload is received, we decode it into a `Packet` enum, which contains the client's request. Then we immediately lock the mutex containing the server state, as we're entering the critical section.