Netsketch: A Collaborative Whiteboard

Assignment Report

Giorgio Grigolo

Contents

1 Introduction **2**
1.1 Project Structure 2
1.2 Libraries Used 2

2 System Design **3**
2.1 Shared Library: `ns-core` 3
2.1.1 A blank slate - The Whiteboard 3
2.1.2 A way of communicating - `TcpPackets` 4

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 A blank slate - The Whiteboard

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 brings us to introduce 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.

2.1.2 A way of communicating - `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.

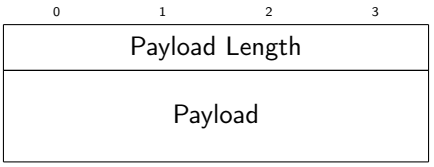

Figure 1: TCP Packet Datagram

Over the wire, any element of the `TcpPacket` enum is sent as bits, or more conveniently placed in a `Vec<u8>`, a vector of bytes. 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 encoding 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 this *hint*. Sinc 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`.

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