CONSOLE-BASED TLS CHAT

TECHNICAL REPORT

Kirill Ivanov
Innopolis University

Kirill Glinskiy
Innopolis University

May 17, 2021

ABSTRACT

We describe the implementation of a broadcast chat application that supports multiple simultaneous encrypted connections between clients. The application is implemented by using Rust sockets, and TLS security protocol.

Keywords Chat · Rust · Client-Server · TLS

1 Introduction

We describe the implementation of a multi-client chat, with a special focus on network communication mechanisms, as well as communication security with TLS protocol. In addition, we describe the design of a terminal user interface and authorization process. The project is developed in the scope of the Computer Networks course at Innopolis University.

2 Implementation

The implementation includes the following parts.

2.1 Technologies and Configurations used

The application is implemented in Rust programming language by using the libraries in Table 1.

Rust Library	Description
`native_tls`	An abstraction over platform-specific TLS implementations
`net`	Networking primitives for TCP/UDP communication
`crossterm`	Design of a command line interface
`chrono`	Date and time library

Table 1. Rust Libraries Overview

We choose Rust mostly because of the absence of common C/C++ memory problems and a wide range of libraries for network development.

To provide a secure data communication we use OpenSSL with TLS protocol. The default network configuration is listed in Table 2.

Address Part	Numerical Representation
Server IP	127.0.0.1
Server Port	8080

Table 2. Network Configuration.

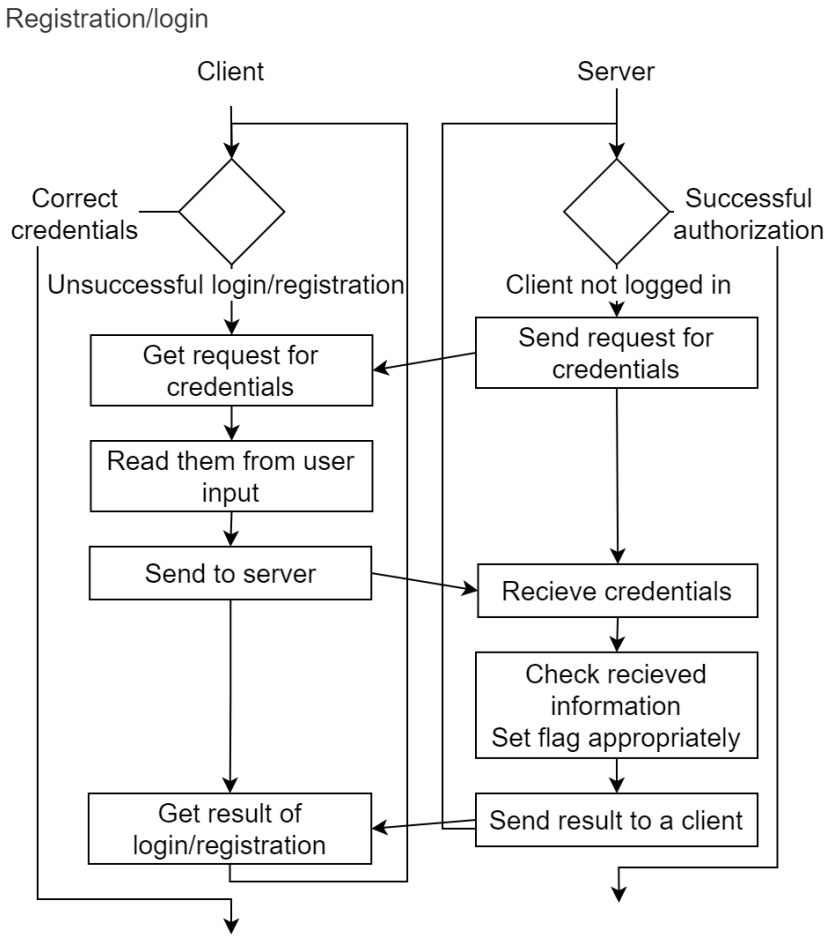

Figure 1: Registration

2.2 Key Functionality

We next describe the functionality implemented on server and client sides of our application.

Server stores the information of network clients, including their IP addresses and user credentials, implements authorization checks, along with receiving and broadcasting the messages.

Client part implements user registration, sending and receiving text messages, and a communication with the centralized server.

To communicate with each other, all clients establish a TCP/TLS connection to the centralized server first. The key stages of establishing such a connection are depicted at Figure 2.

After the connection to a server is established successfully, a client start exchanging messages to a server. To provide a simultaneous message sending and receival by the same client, we introduce the sender thread shown at Figure 4 and the receiver thread shown at Figure 3. Instead, on a server side, we create a dedicated new thread for every client joined the chat (see Figure 5).

For an increased communication security, every client is required to register on a server side first. The registration is depicted schematically in Figure 1.

The overall application organization is summarized in Figure 6.

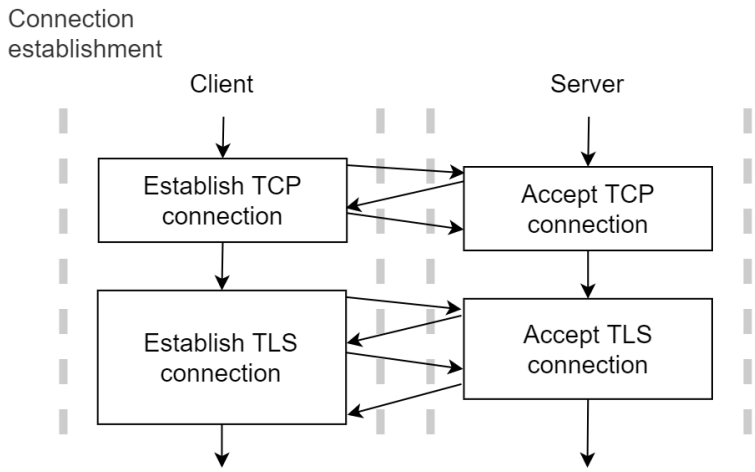

Figure 2: Connection process

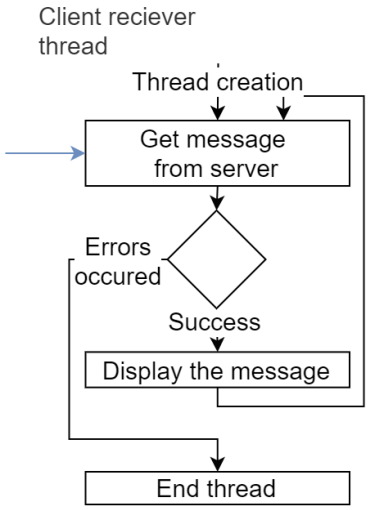

Figure 3: Receiver client thread

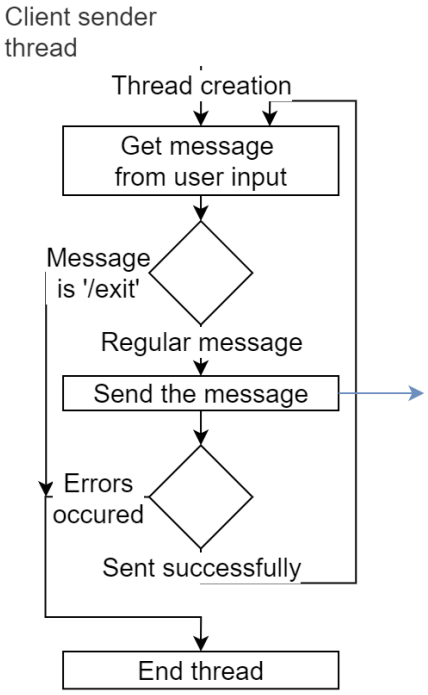

Figure 4: Sender thread in client

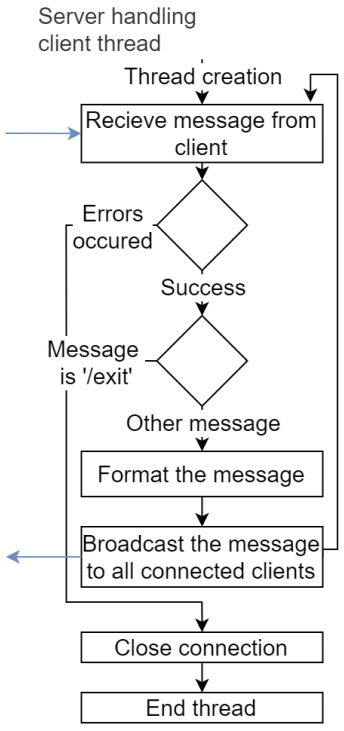

Figure 5: Server thread for client

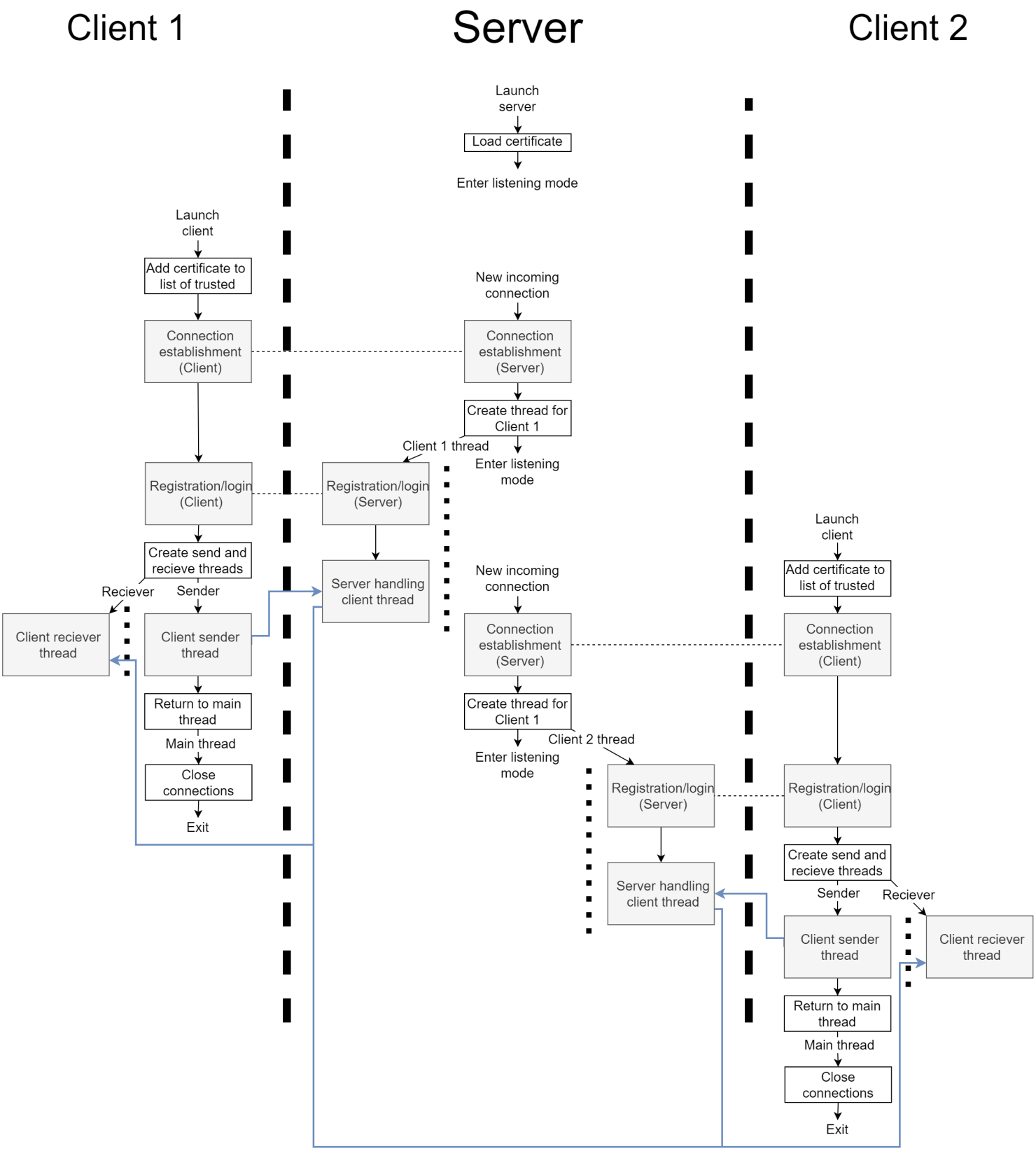

Figure 6: Example of application logic with 2 connected clients

3 Experience with Rust

We finally share our experience of a network programming by using Rust sockets.

To ensure maximum memory safety, we used thread-safe structures. However, they overlapped, resulting in big mess (eg. `Arc<Mutex<Vec<Arc<Mutex<TlsStream<TcpStream>>>>>>`). It decreased readability of the code, so we decided to encapsulate such structures.

Speaking of comparing socket programming in Rust and C, there are analog functions in both languages. However, most of the function arguments in Rust are hidden. For example, we explicitly declare what type of socket we are using (e.g. `UdpSocket::bind("127.0.0.1:34254")`)

Since it was one of our first experiences with the language, it resulted in developing being more time-consuming. Rust's policy is strict to programmer in terms of code writing. However, as we said before, it prevented typical memory bugs, and therefore, saved time for fixing them.

4 Conclusion

We have implemented a simple console messaging application with a high degree of security. The project is implemented in the scope of Computer Networks course at the Innopolis University. It is publicly available on the Internet at GitHub repository, both for server[1] and client[2].

There are several limitations in a current implementation that need further development, for example addition of a database with confidential user data to achieve non-volatile server memory and password hiding during keyboard input. We aim to resolve them for the next release using SQLite integration into Rust code and changing the registration logic.

[1] https://github.com/PurplePegasuss/Rust_SC_Chat
[2] https://github.com/bragov4ik/Rust_SC_Chat_Client