A Study on Rust Client-Server Applications

Andrei Cătălin Clicinschi
Department of Artificial Intelligence
Politehnica University Bucharest
andrei.clicinschi@stud.acs.upb.ro

Abstract

Client-server architectures are popular for a wide array of applications, including but not limited to multiplayer video-games, chat and email clients, distributed compute platforms, and other types of online services. Rust is a relatively new multi-paradigm, general-purpose programming language that emphasizes performance, type safety, and concurrency. This paper describes a framework for implementing a client-server application in Rust and discusses the difficulties in evaluating its performance and methods of overcoming them.

1 Introduction

When starting working on an application, one of the first decisions that must be made is to choose the programming language in which to code the application in. Many different criteria can influence this decision, for example a website has a completely different purpose and requirements compared to a video game, which in turn is very different compared to a command line utility that needs to be as portable as possible. The list of specific factors includes, but is not limited to:

- the optimization demands: if maximum performance is required, low-level languages such as assembly can be a good option
- the time constraints: if a fast implementation is required, high-level languages such as python perform best
- the skill set of the programmer or team
- the hardware configuration of the system targeted by the application: applications targeting less powerful systems should be less resource-intensive
- the software configuration of the system targeted by the application: some systems have much better support for coding in certain languages only, for example Android which mainly supports Java and Kotlin

With these factors in mind, we decided to implement the current project in Rust. Rust is a relatively new programming language, having been created in 2006 but widely adopted in 2015, at the time of its first stable release, by companies such as Amazon, Discord, Dropbox, Google (Alphabet), Meta, and Microsoft [1]. As a testament of its reliability and usefulness, in December 2022, Rust became the first language other than C and assembly to be supported in the development of the Linux kernel.

Rust is a multi-paradigm, general-purpose programming language that emphasizes performance, type safety, and concurrency. It enforces memory safety, meaning that all references point to valid memory, without requiring the use of automated memory management techniques, such as garbage collection. To simultaneously enforce memory safety and prevent data races, its "borrow checker" tracks the object lifetime of all references in a program during compilation. Rust was influenced by

[1] https://en.wikipedia.org/wiki/Rust_(programming_language)

Preprint. Under review.

ideas from functional programming, including immutability, higher-order functions, and algebraic data types. [1]

In order to briefly present some defining features of the Rust language, we have included the following table containing a brief comparison between Rust and one of its main alternatives, C++.

.	Rust	C++
Data ownership	Variables have ownership and lifetime, which means that only one variable can own a specific piece of data at a time	No such system, data races are much more likely
Error handling	The Result type is used to handle errors, which makes it easy to handle and propagate errors in a safe and consistent way	Uses exceptions to handle errors
Concurrency	Built-in support for concurrency through its ownership model and its lightweight threading model	No built-in support for concurrency and it requires more manual work to handle threads and synchronization
Template metaprogramming	Rust does not have a powerful template system and it requires more manual work to perform metaprogramming at compile-time. However, it is possible to achieve similar functionality using procedural macros	Powerful template system that allows you to perform metaprogramming at compile-time
Ease of use	Easier to use due to its well-defined semantics and its ability to prevent unwanted/undefined behavior	More flexible, can sometimes achieve better performance due to this
Community support	Has frameworks for various purposes and plenty of libraries (called crates)	Being an older language, it has more libraries, that tend to be more general purpose, and more educational resources

The study conducted by Pereira et al[2] regarding the energy, time and memory efficiency of programming languages shows that Rust is second only to C in execution time and energy consumption across a variety of tasks, with only a 3-4% difference between them, and uses only 50% more memory compared to the most memory-efficient programming language (Pascal). These findings prove that Rust is a noteworthy programming language, having numerous features and improvements not present in C, with only minor performance drawbacks.

2 Related Work

The client-server architecture, along with the peer-to-peer architecture, are the two main models for distributed environments. In the peer-to-peer architecture, all instances are more or less equivalent, serving the same functions either simultaneously or alternatively, capable of both transmitting and receiving data. Some examples include basic multiplayer games, for example an online card game might not have a dedicated server, with all computations being done by the players' computers. Another example are some peer-to-peer file-sharing applications, in which the users take turns in downloading parts of the desired files from other users, then allowing their computer to idly provide other users with their desired files. No single centralized point of communication exists in this architecture, the clients communicate directly between themselves.

In client-server applications, there are one or more servers that have the dedicated purpose of providing the users, called clients, with a service. The service can be any combination of the following:

- answering queries by retrieving data from a private database (such as online stores, educational platforms, and many others)
- performing computations (online compilers and execution environments)
- facilitating the communication between two or more clients (such as messaging services or multiplayer video games)

Kumar [3] highlights the separation of the client-server applications depending on the implementation of the presentation, application, and data layers. In our case, the client handles the presentation layer

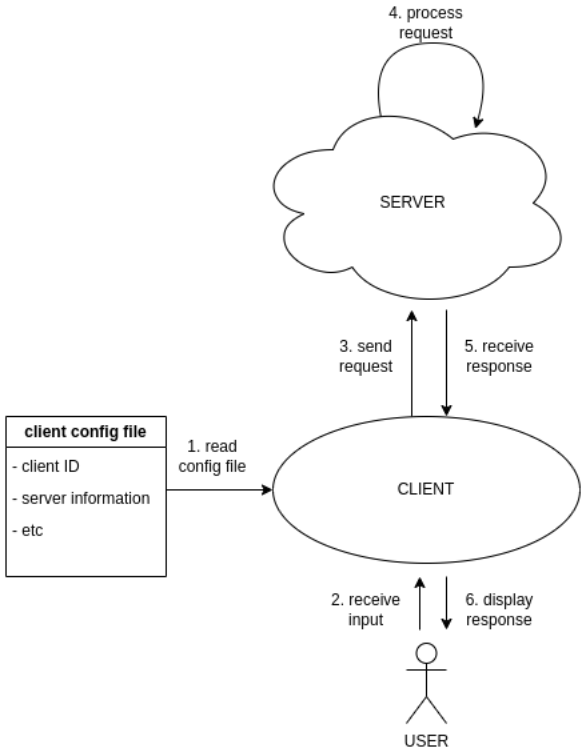
and the server implements the application and data layers, but other architectures might distribute the application and data layers on separate servers, for example. The following table by Kumar [3] showcases some areas of research and evaluation metrics regarding client-server applications. In our work, we focused on the first area of research: performance evaluation.

Research opportunity	Study parameters
Performance evaluation of the client-server system	Response time, Throughput, Workload, Time out
Reliability study of the client-server system	Failure rate, Mean time to failure, Failure Rate, Fault analysis
Trusted client-server system design	Vulnerabilities assessments, Secure system, Authentication, Authorization
Secure client-server system development	Authentication, Authorization, Integrity, Privacy, prevention of cyber attacks

Yu et al [4] show that although Rust emphasizes concurrency and data safety, such bugs are still possible even without bypassing some compiler safety checks by using the "unsafe" keyword. Deadlocks can occur when using double RwLock (Rust's name for Mutex). Since the release is implicit, at the end of the scope, calling read and write in the same scope leads to a deadlock. Message passing between threads can lead to a similar deadlock, since waiting for a message is blocking. Data races can occur for the getrand() function, which employs a caching mechanism because the regular execution time is relatively long. This mechanism is sensitive to compiler reordering of the execution of some lines that check the value of a special variable used for caching.

3 Experimental Setup

3.1 Control Flow



The previous figure shows the control flow of our application.

1. the client reads its configuration file and stores the information

2. the user inputs a command and command-specific parameters
3. the client builds a request with the information from step 2 and sends it to the server read at step 1
4. the server accepts the connection and processes the request
5. the server sends back the response to the request from step 3
6. the response is logged and displayed to the user

3.2 Client Configuration File

The client configuration file is structured as a JSON file with the following mandatory fields:

Field name	Type	Description
Client-ID	u32	Must be unique between all clients
Config-Server	String	Contains the server network address/hostname and port number
Log-File	String	The name of the file that will contain the client log

3.3 Request Format

The client requests are structured as JSON objects with the following mandatory fields:

Field name	Type	Description
Client-ID	u32	Must be unique between all clients
Timestamp	u64	The unix timestamp at the time the request is created (in ms)
Request-Type	u32	See 3.5
Request-Body	Depends on Request-Type	

3.4 Response Format

The server responses are structured as JSON objects with the following mandatory fields:

Field name	Type	Description
Client-ID	u32	Must be unique between all clients
Timestamp	u64	The unix timestamp at the time the request is created (in ms)
Request-Type	u32	See 3.5
Response-Body	Depends on Request-Type	

3.5 Request Types

The following are the two request types that are currently implemented:

Request-Type	Description	Request-Body	Response-Body
0	Echo request, the server will return the same body as the one received from the client	{“hello”}	{“hello”}
1	Configuration request, the server will return a string that can be used to configure the client	{}	{“configuration”}

4 Results

The server and client executables were run on the same machine, which severely limited the relevance of the tests performed. A true client-server scenario would depend on the network round-trip times, and would have asymmetric hardware running the clients and server.

The tests were performed by manually running the server and one or two clients in parallel. The number of clients is limited only by the number of simultaneous connections that can be accepted by the server (hardware and/or software limitations). This low number of clients is also a weak point of our tests, since realistic scenarios that test the server's multitasking capability would contain at least dozens of clients at the same time.

The response time varied between 0ms (so, in reality, lower than the precision of the clock used to perform the time measurements: *SystemTime::now()*, but obviously not a true zero) and 3ms. The variance in reported times for such simple tasks is extremely high, making the measurements essentially meaningless.

Our server is capable of accepting connections from multiple clients at once, and the client executable can be used to run multiple client instances by providing a configuration file id (from which the client will derive the path to its configuration file). The server correctly responds to the requests, and the responses are logged by the client to its log file.

Although we implemented no explicit design patterns in this project, we did explore a few options. For such a basic application, with the client running a loop that awaits user inputs and calling functions for creating, sending requests and receiving responses and the server running a loop that waits for requests and calling functions that create a response and send it to the client, most design patterns felt forced, without any real practical benefit. In this case, Rust being a multi-paradigm language instead of a pure object-oriented language was a detriment, since it meant there existed other options of organizing the code, which we naturally preferred due to their simplicity.

5 Conclusion

In conclusion, although limited in scope, this project was a good introduction to the Rust programming language, and to the more general task of creating an application and documenting the process.

Possible future improvements include:

- run the client and server on separate machines, in order to better emulate a realistic production environment
- expand the scope of the client and server, in order to perform more complex computations and facilitate the implementation of design patterns
- perform larger stress tests, with more clients in parallel

References

[1] J. Blandy and J. Orendorff. *Programming Rust: Fast, Safe Systems Development*. O'Reilly Media, 2017.

[2] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, pages 256–267, 2017.

[3] Santosh Kumar. A review on client-server based applications and research opportunity. *International Journal of Recent Scientific Research*, 10(7):33857–3386, 2019.

[4] Zeming Yu, Linhai Song, and Yiying Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software, 2019.