CSE411 - Advanced Programming Techniques

Homework-2

Department of Computer Science & Engineering

Lehigh University

Prepared for,

Professor Corey Montella

Prepared By,

- Badrinadh Aila
- Deven Bhadane
- Nikita Chaudhari

1. Introduction

In the ever-evolving landscape of digital communication, the need for robust and efficient client-server interactions is more pressing than ever. As our world becomes increasingly interconnected, the challenges of managing concurrent connections, ensuring data integrity, and providing a seamless user experience have become central concerns in software architecture. Traditional client-server models often falter under the weight of these demands, struggling to maintain shared resources securely and efficiently.

In response to these challenges, our implementation harnesses the power of Rust's advanced multithreading capabilities and concurrency mechanisms. We have meticulously designed a client-server communication system that not only handles a myriad of client requests concurrently but also ensures the secure management of a shared database.

In the upcoming sections, we will explore the intricacies of our implementation, multiple analysis used to establish a secure, and highly responsive client-server architecture. By leveraging Rust's unique features, we have crafted an adaptable and efficient system capable of meeting the demands of modern, interconnected digital ecosystems.

2. System Desing

The implemented client-server communication system is a robust and efficient model, allowing multiple clients to connect concurrently and interact with a shared database resource. Utilizing multithreading in Rust, the server handles incoming client connections in a scalable manner. The server's local address and port are predefined, ensuring a standardized point of access for clients. Central to this communication model is the implementation of a shared database, safeguarded by Rust's Mutex and Arc. This ensures exclusive access to the database, allowing only one client at a time to perform operations like PUT, GET, or DELETE.

Upon accepting client connections, the server spawns individual threads to handle communication with specific clients. Each thread operates within a loop, reading client messages and processing them based on their operation type. Requests are parsed to determine if they are PUT, GET, or DELETE operations. For PUT requests, the server checks if the key already exists in the database, inserting the new key-value pair if not. GET requests retrieve values associated with a specific key, and DELETE requests remove key-value pairs from the database. Furthermore, the server's response mechanism is well-structured. Responses are categorized as either "OK" or "ERROR," reflecting the success or failure of the client's request. These responses are sent back to the clients via a message-passing channel, ensuring that all clients receive consistent and accurate feedback regarding their requests.

In summary, this implementation not only achieves concurrent and secure client-server communication but also provides a foundational structure for further enhancements and features. By employing Rust's multithreading capabilities, Mutex, and Arc, the system maintains robustness and reliability, laying a solid foundation for future developments in the client-server interaction.

3. Implementation

Client Code:

The client code is a rust program that generates random PUT, GET, and DEL requests and sends them to a server using TCP/IP.

In client side, LOCAL is a string constant representing the IP address (127.0.0.1) and port number (1895) that the client will use to connect to the server. MSG_SIZE is a constant of type usize representing the maximum size of the message buffer used for reading data from the server. KEY_RANGE is a constant representing the range of possible key values that can be generated by the client with 5 possible values. IN_NUM_OF_WORDS is a constant of type i32 specifying the minimum and maximum number of words that can be generated for a PUT request.

Client Main Function:

1. Initialization:

TCP Connection: The client establishes a TCP connection to the server at the address specified by the `LOCAL` constant (`127.0.0.1:1895`). And the client sets the connection to non-blocking mode, enabling communication with the server.

2. Channel Setup:

Message-passing Channel: The client creates a message-passing channel (`mpsc::channel`) to facilitate communication between the main thread and the spawned thread. Where `tx` used to send messages to the spawned thread.and `rx` is used to receive messages from the spawned thread.

3. Spawned Thread:

For each client a new thread is spawned to handle communication with the server. The thread attempts to read data from the server into the `buff` buffer. The received data is

parsed into a string and split into lines. Each line is printed for visualization. The thread checks for messages in the channel (`rx`). If messages are present, they are sent to the server after conversion and then thread sleeps for 100 milliseconds to avoid busy-waiting, enhancing efficiency.

4. User Input and Message Sending Loop:

In the main thread, the program generates random requests (`generate_put_req()`, `generate_get_req()`, or `generate_del_req. The request is sent to the spawned thread through the channel (`tx`) for processing and transmission to the server. If the user inputs `:quit`, the loop breaks, and the program terminates. The program prints "bye bye!" indicating the end of its execution.

Client Functions:

In the client application, there are three essential functions: `generate_put_req()`, `generate_get_req()`, and `generate_del_req()`, each responsible for generating specific types of requests to interact with the server.

The `generate_put_req()` function in the client application prepares a formatted "PUT" request to store a key-value pair in the server's database. Initially, an empty string `put_req` is created to store the formatted request. The function calculates the length of the key and appends it to `put_req`, then adds the key itself to the request string. Subsequently, the length of the value is calculated and appended to `put_req`, followed by adding the value to the request string. Finally, the function returns the generated `put_req` string, which is intended for transmission to the server. This modular approach ensures the proper formatting of "PUT" requests, allowing for efficient communication between the client and the server..

The `generate_get_req()` function constructs a "GET" request message, indicating a request to retrieve a value associated with a specific key from the server's database. It starts by initializing an empty string, `get_req`, to store the formatted request. The function generates a random key within the specified `KEY_RANGE`, calculates its length, appends the length and the key to `get_req`, creating the formatted "GET" request string. Finally, the function returns the generated `get_req` string, intended for transmission to the server. This function encapsulates the logic for creating well-formed "GET" requests, facilitating seamless communication between the client and the server.

The `generate_del_req()` function generates a "DEL" request message, indicating the client's intention to delete a key-value pair from the server's database. The function initializes an empty string, `del_req`, to store the formatted request. It generates a random key within the specified `KEY_RANGE`, calculates the length of the key, appends the length and the key to `del_req`, creating the formatted "DEL" request string. The function then returns the generated `del_req` string, which is ready for transmission to the server. This function encapsulates the logic for creating valid "DEL" requests, facilitating the removal of specific data entries from the server's database.

These functions serve as the backbone of the client application, enabling it to communicate effectively with the server.

Server Side:

The server-side code presented implements a concurrent server in Rust, showcasing fundamental concepts of multi-threading, non-blocking I/O, and concurrent data management.

1. Initialization and Setup:

In the initialization, the server defines crucial constants and functions for its operation. `LOCAL: &str = "127.0.0.1:1895"` specifies the server's local address and port number, allowing clients to connect. `MSG_SIZE: usize = 512` establishes the maximum message size for communication between clients and the server. The utility function `sleep()` pauses the current thread for 100 milliseconds, preventing busy-waiting and excessive CPU usage.

2. Main Function Execution:

A fundamental part of the server's architecture is the database, represented as an empty HashMap (database: HashMap<i32, String>) initialized to store key-value pairs. To enable secure and synchronized access from multiple threads, the HashMap is wrapped in an Arc (atomic reference-counted) and a Mutex, guaranteeing concurrent threads can interact safely with the database.

To manage multiple clients simultaneously, the server employs a dynamic clients vector, allowing it to store client sockets efficiently. Additionally, a message-passing channel (mpsc::channel) is established, consisting of a transmitter (tx) and a receiver (rx). This channel acts as a communication bridge between the main thread and client-handling threads, empowering the server to send responses to clients. These foundational elements form the backbone of the server's concurrent architecture, ensuring secure data handling and seamless communication with clients.

3. Handling Client Connections:

In the process of handling client connections, the server operates within a perpetual loop, consistently accepting incoming client connections. Upon receiving a connection request (`server.accept()`), the server initiates a new thread for each client. Within this thread, a continuous loop is established. This loop is responsible for processing client requests and formulating suitable responses, ensuring seamless communication between the server and individual clients. Through the utilization of multithreading, the server can concurrently manage multiple clients, fostering efficient and simultaneous interactions.

4. Handling Client Requests:

In the client-handling process, the dedicated thread reads incoming data from the client socket and interprets the received messages. These messages, indicating operations like "PUT", "GET", or "DEL", are parsed to discern their type. Depending on the operation, corresponding actions are executed on the shared database. Subsequently, the server crafts responses ("OK" or "ERROR") reflecting the success or failure of the operation. These responses are then sent back to the client via the message-passing channel (`tx`), ensuring seamless communication and synchronization between the server and its clients.

5.Handling output messages:

Here, the main thread monitors the message-passing channel (`rx`) for any messages, which usually consist of responses from the server intended for the clients. If there are messages available, indicating responses ready to be sent, the server broadcasts these messages to all connected clients. This broadcasting mechanism ensures that all clients receive the same responses simultaneously, guaranteeing uniform and consistent communication between the server and its clients.

6.Format_msg function :

The `format_msg` function in the client-server system plays a crucial role as a message parser. It dissects incoming client messages, extracting operation types ("PUT", "GET", or "DEL"), key lengths, keys, and, for "PUT" requests, value lengths and values. This parsed data is organized into a HashMap for efficient server-side processing. By providing a structured representation of client requests, this function ensures accurate interpretation of client intentions and enhances the server's ability to handle various operations effectively.

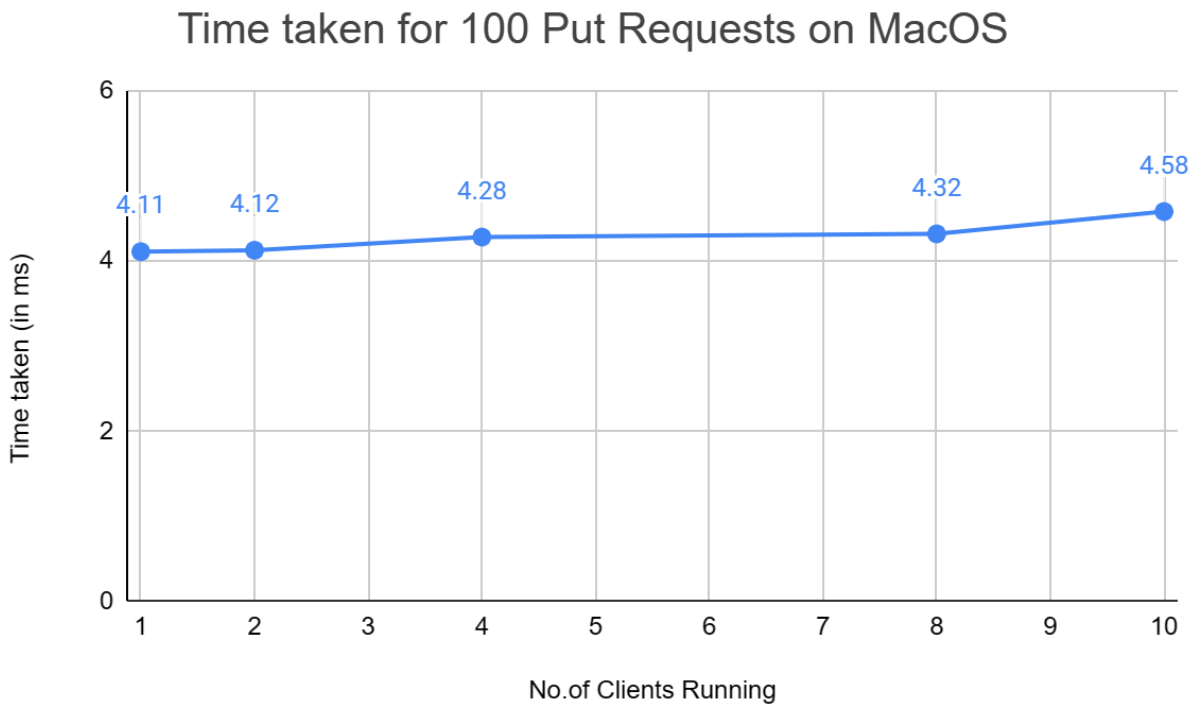
4. Analysis

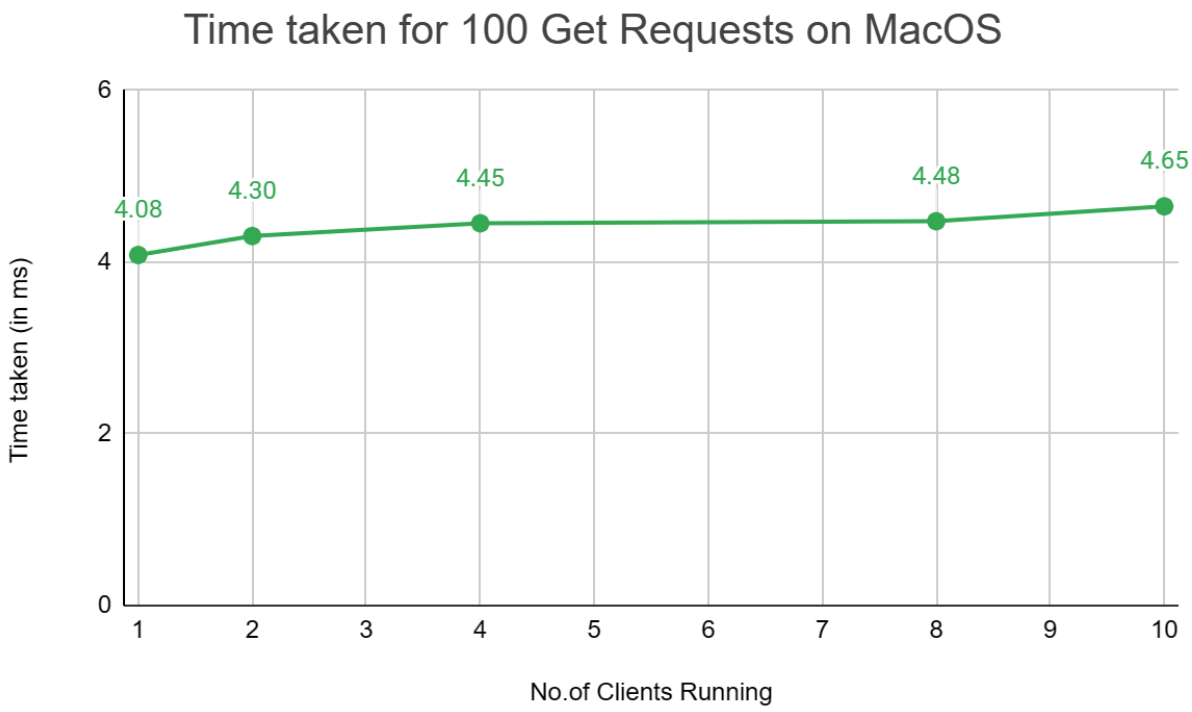
For the analysis a benchmark is set to evaluate the performance of handling "Delete," "Put," and "Get" requests in a client-server communication system. These benchmarks simulate clients connecting to a server, generating requests, and measuring the communication time.

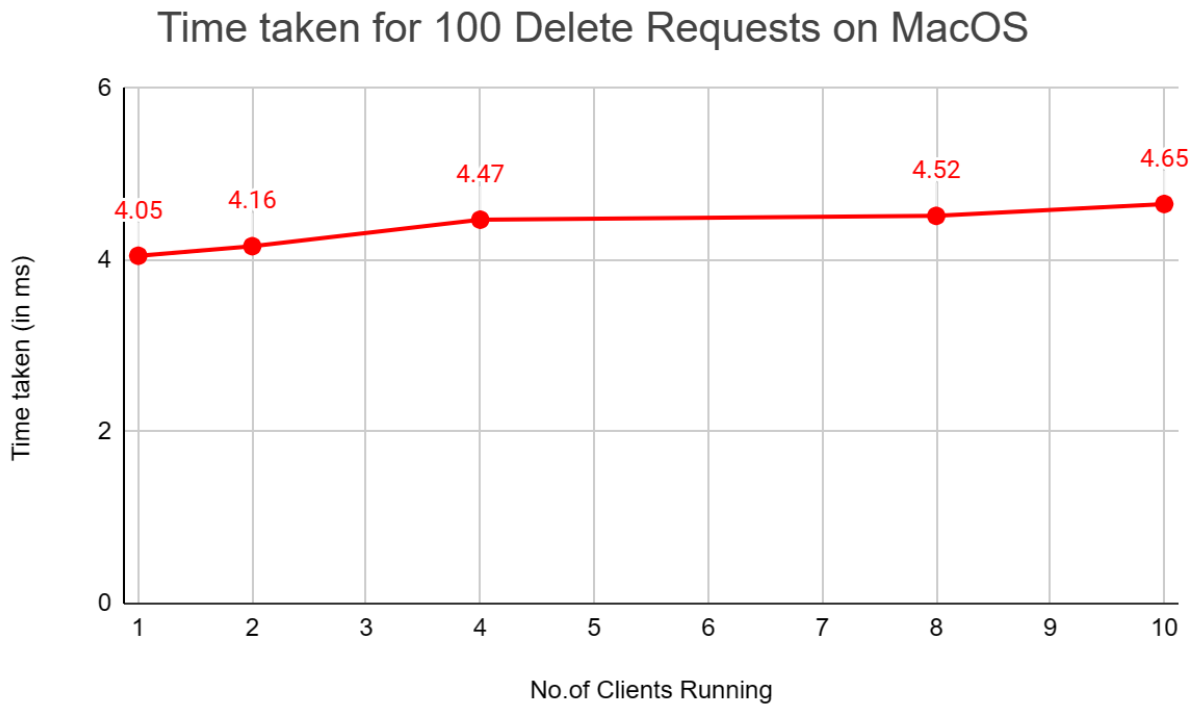
The "Delete Request Benchmark" assesses the server's efficiency in processing "DEL" requests. It focuses on non-blocking communication and processing when handling delete requests under load. The "Put Request Benchmark" evaluates the server's performance under load when processing "PUT" requests and the "Get Request Benchmark" focuses on the efficiency of handling "GET" requests, using random keys to simulate diverse client requests

These benchmarks collectively offer a comprehensive analysis of the server's responsiveness, scalability, and efficiency in handling various number of client requests (e.g. 1,2,4,8,16 clients) on different operating systems (windows and iOS). The results obtained from these benchmarks are essential for optimizing the server's performance in real-world scenarios with concurrent client interactions.

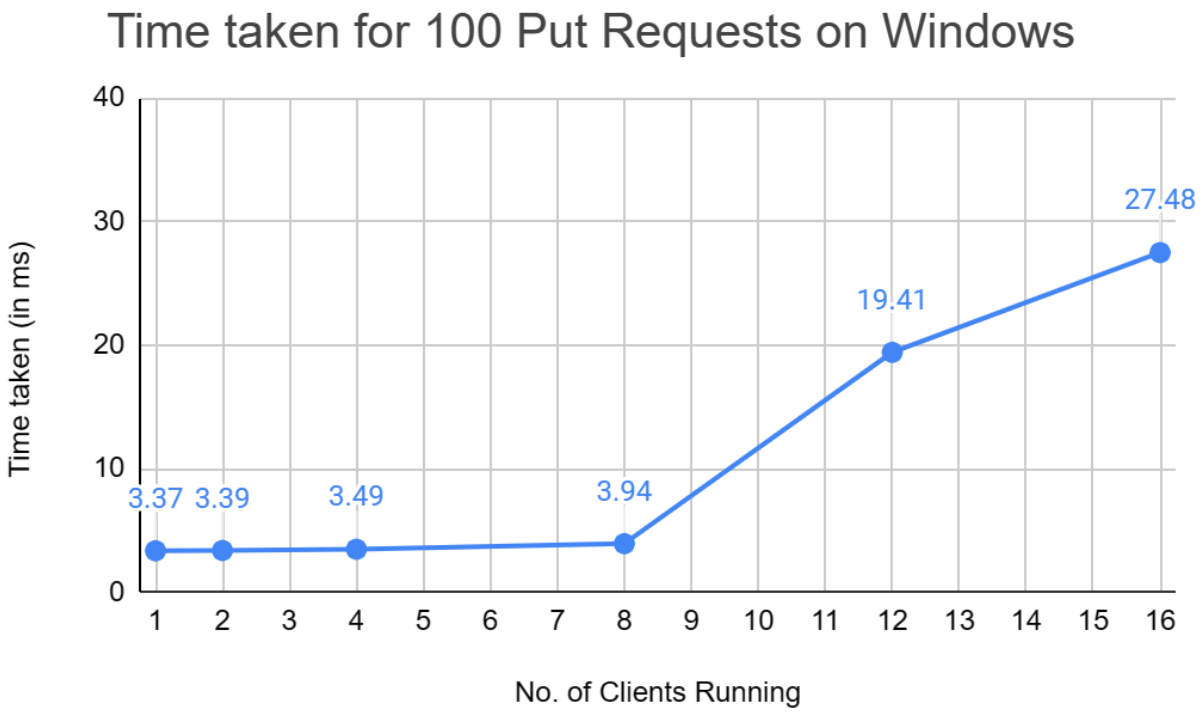
The following graphs demonstrate the required time to process the different requests with multiple clients on **Mac OS operating system**.

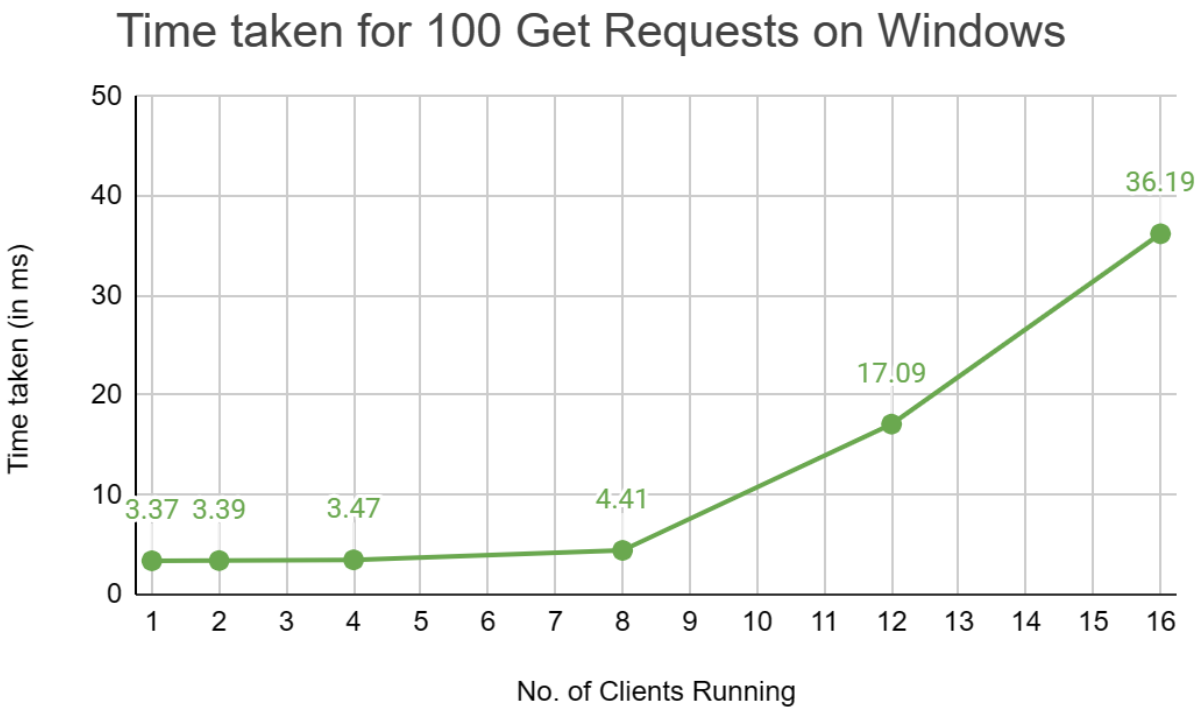


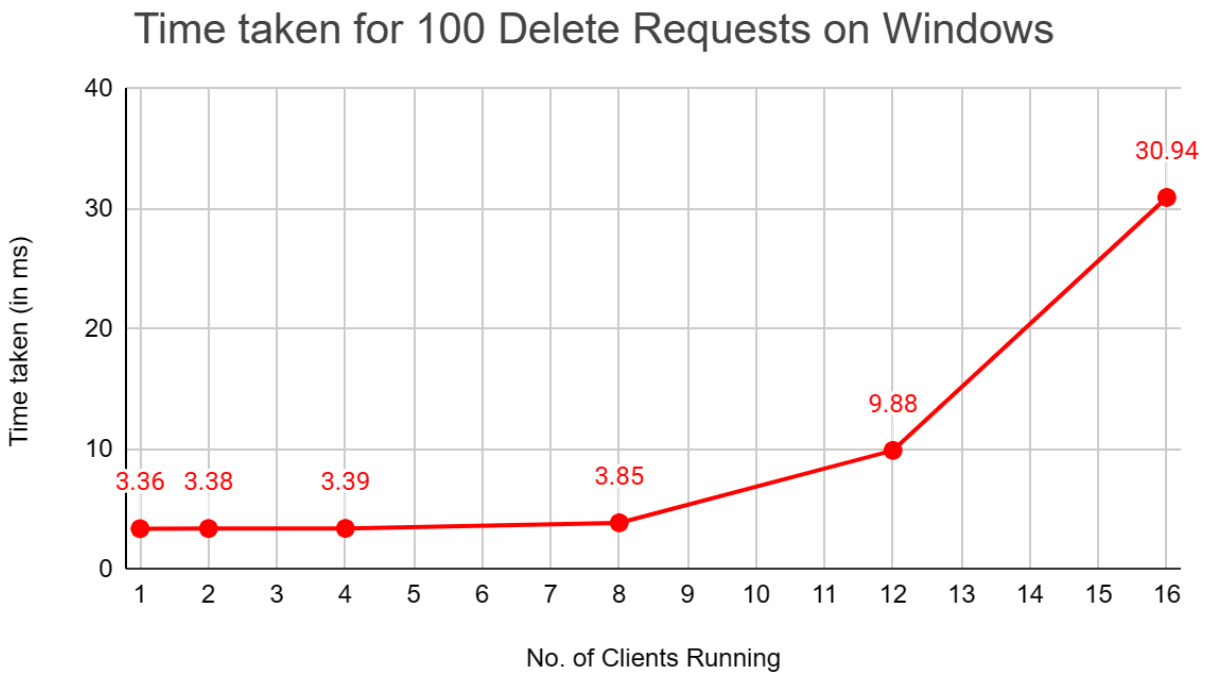

Time taken for 100 Get Requests on MacOS
Time taken (in ms)
No.of Clients Running
4.08
4.30
4.45
4.48
4.65
0
2
4
6
1
2
3
4
5
6
7
8
9
10


Time taken for 100 Delete Requests on MacOS
Time taken (in ms)
No.of Clients Running
4.05
4.16
4.47
4.52
4.65
0
2
4
6
1
2
3
4
5
6
7
8
9
10

The following graphs demonstrate the required time to process the different requests with multiple clients on **Windows operating system**.







5. Conclusion

In conclusion, the designed client-server communication system in Rust, implemented with multithreading, was rigorously tested across different platforms, including Windows and Mac. The system's performance was evaluated with varying numbers of clients, ranging from 1 to 16. The tests revealed that the server's performance experienced a notable decrease as the number of requests increased. Additionally, the tests highlighted platform-specific variations, indicating that Mac outperformed Intel-based systems in handling multithreading tasks. These insights underscore the importance of thorough cross-platform testing and careful consideration of hardware differences when designing multithreaded applications. Furthermore, the observed performance degradation with an increase in requests emphasizes the need for continuous optimization and scalability considerations to ensure the system's responsiveness, especially under high load conditions. These findings serve as valuable guidelines for further refining the client-server architecture and enhancing its overall efficiency.