Concurrent Key/Value Store Implementation Evaluated via Thread-Safe Server and Client

Bonor Ayambem

CSE-411: Advanced Programming Techniques

Prof. Corey Montella

October 5, 2023

1. Introduction

Key/value stores stand as pivotal components in the realm of modern distributed systems, as they empower programs to efficiently store and retrieve data on highly available servers. This interface consists of three fundamental operations:

- **get(key)**: Retrieves the value associated with a given key, returning the value if present, or null if not.
- **del(key)**: Removes a key-value pair from the store, returning true if successful (the key existed), or false if the key was absent.
- **put(key, value)**: Adds or modifies a mapping from a key to a value, ensuring that the key-value store remains concurrent.

To implement a concurrent key/value store, we face the challenge of designing a data structure that can efficiently handle multiple threads and concurrent operations. This project implements a hashmap from the collections library and wraps it in `Arc<Mutex<T>>` to achieve concurrency.

In addition to the data structure, we must define a communication protocol for interacting with the key/value store over the network. This protocol, inspired by HTTP, utilizes newline-separated metadata and variable-length content for binary data. The client component constructs key/value requests and transmits them over the network to the key/value server. It also receives responses from the server and presents the results to the user. The server receives key/value requests from clients, executes the requested operations on the map data structure, and generates responses to be sent back. It must efficiently handle multiple client requests concurrently through multithreading.

Both the client and server will communicate over port 1895, adhering to the established protocol for data exchange. To ensure the reliability and robustness of this implementation, thorough testing is imperative. This involves running four clients and one server, crafting test cases that encompass all possible operations and outcomes, and validating that our key/value store operates as expected within a distributed environment.

2. Implementation Summaries

2.1 Concurrent HashMap Implementation

Concurrency is achieved here by wrapping the HashMap inside an `Arc<Mutex<HashMap>>`. This allows multiple threads to share access to the hashmap while ensuring that only one thread can mutate it at a time. The Mutex guards access to the hashmap, preventing concurrent writes and providing a form of synchronization.

The `new` function initializes a new `ConcurrentHashMap`. It creates an empty HashMap, wraps it in a Mutex, and then wraps the Mutex in an Arc (atomic reference counting) to make it shareable between threads. The `put` method inserts a key-value pair into the hashmap. It locks the mutex, inserts the value, and then unlocks the mutex. This ensures that only one thread can modify the hashmap at a time, preventing data races. The `del` method removes a key from the hashmap. It locks the mutex, checks if the key exists, removes it if it does, and then unlocks the mutex. This operation is also protected from concurrent access. The `get` method retrieves a value by its key from the hashmap. It locks the mutex, checks if the key exists, clones the value if it does, and then unlocks the mutex. It returns an `Option<String>`, allowing clients to handle the case where the key is not present in the hashmap.

This implementation achieves concurrency control by wrapping a regular HashMap in a thread-safe Mutex and sharing it across threads using an Arc. This ensures that multiple threads can safely read and modify the hashmap without causing data races.

2.2 Client Implementation

The client implementation is designed to send requests to a server using a TCP stream and receive responses. The code imports necessary modules from the standard library for working with networking (TcpStream) and input/output operations (Read and Write).

The `send_request()` function takes a tuple req as input, containing two strings: the first string is the request type ("get" or "del" or another string which represents a key for the put request), and the second string is the request data (key or key-value pair).

Depending on the request type ("get", "del", or "put"), the function constructs a request string in the specified format. The constructed request string is sent to the server through the provided TcpStream. If an error occurs during the sending process, it is captured and displayed.

The client reads the server's response from the TcpStream. It checks if the response is empty and prints an error message if it is. The response is decoded as a UTF-8 string, and if successful, it is printed to the console. In case of an error while decoding the response, an error message is printed along with the raw response bytes. The decoded response is printed to the console.

2.3 Server Implementation

The server implementation handles incoming client connections, processes various types of requests ("GET," "DEL," and "PUT") from clients, and measures the rate of requests processed per second. The code imports necessary modules from the standard library for working with networking (TcpListener, TcpStream) and input/output operations (Read and Write). It also

imports synchronization and timing-related modules (Arc, Mutex, Duration, Instant). Three constants, `EXPECTED_GET`, `EXPECTED_DEL`, and `EXPECTED_PUT`, are defined to represent the expected request types in ASCII format.

The `handle_requests()` function is the main entry point for handling client requests. It creates a new ConcurrentHashMap (a thread-safe map) and binds a TcpListener to the address "127.0.0.1:1895" to listen for incoming client connections. A new thread is spawned to handle incoming client connections. Inside the thread handling client connections, a loop iterates over incoming client streams. For each client, a new thread is spawned to handle the client's requests. Within a loop, the server reads data from the client stream into a buffer. The server then analyzes the received data to identify the type of request (GET, DEL, PUT). For each request type, the server extracts relevant information, such as key and value lengths. Based on the request type, appropriate actions are taken, such as looking up or modifying the map.

The server sends a response back to the client. The loop continues until the client closes the connection or an error occurs. The code includes error handling for various scenarios, such as reading from the client stream, sending responses, and decoding responses. After processing client requests, the server thread joins, and the main thread remains alive, listening for incoming connections.

3. Testing Setup

3.1 Testing

`cargo run` is run in one terminal to start the server, and `cargo test` is run in another terminal to begin communicating with that server.

This project includes two test functions, test_client1 and test_client2, which test the client's ability to send various types of requests to the server. This test function is designed to simulate a client (Client 1) sending multiple requests to the server. It establishes a TCP connection to the server at "127.0.0.1:1895" using TcpStream::connect. The client sends a series of requests to the server using the send_request function. Each request is represented as a tuple, with the first element specifying the request type ("get," "del," or a key), and the second element specifying additional data (e.g., the key or value). Several requests are sent to test different scenarios, including "get," "del," and "put" requests with various key-value pairs.

The `test_client2` function is similar to test_client1 but represents a different client (Client 2) sending its own set of requests.Overall, these test functions serve as test cases to check how well the client interacts with the server. They simulate various client behaviors, such as requesting data, deleting data, and adding data to the server. The client's ability to establish connections and send requests is evaluated through these tests.

3.2 Benchmarking

Timing testing is performed to measure how many requests can be processed by the server in one second.

At the beginning of each client connection handling thread, a `start_time` variable is initialized using `Instant::now()`. This creates a timestamp representing the current time. A request_count variable is initialized to keep track of the number of requests processed within one second.

The code enters a loop that continuously receives and processes incoming requests from the connected client. For each request received, the code checks whether one second has elapsed since the last time it printed the request count. Within the loop, the code checks the elapsed time

by comparing the current time (using Instant::now()) with the start_time. This is done with the line: if start_time.elapsed() >= Duration::from_secs(1) { ... }

If one second has passed, the code proceeds to print the request count and reset both the `start_time` and `request_count`. For each request processed within the loop, the `request_count` is incremented by one: `request_count += 1;` When one second has elapsed, the code prints the number of requests processed in that second, allowing you to measure the server's request processing rate.

The purpose of this timing testing section is to benchmark the server's performance in terms of request processing throughput. By counting how many requests the server can handle within one second, we assess the server's capacity to handle concurrent requests and its overall performance.

4. Results

Problems in communication between the server and client have made it difficult to obtain results from the written tests.

5 Conclusions

It is difficult to make substantial conclusions without test results, however, timing testing allows us to assess the server’s throughput, which is the number of requests it can process per unit of time (e.g., requests per second). Higher throughput indicates better server performance.