Project Report: Key Value Web Server

Distributed Systems. Advanced Course.

Sofia Krylova <krylova@kth.se>, Iosif Koen <iosif@kth.se>

April 7, 2023

1 Introduction

In this course, as a mandatory task, we were asked to implement a highly available Key-Value web server, using OmniPaxos library, that was developed in KTH.

OmniPaxos is an in-development replicated log library implemented in Rust. OmniPaxos aims to hide the complexities of consensus to provide users a replicated log that is as simple to use as a local log. Similar to Raft, OmniPaxos can be used to build strongly consistent services such as replicated state machines. Additionally, the leader election of OmniPaxos offers better resilience to partial connectivity and more flexible and efficient reconfiguration compared to Raft.

The developed web server is fault-tolerant. It is accessed via REST and stores key-value data that can be read in a consistent manner.

The project's main goal was to give us a hands-on experience with OmniPaxos, consensus, leader election and other components that were covered by the course.

2 Main problem

In this project, we were supposed to build a web server that is made fault-tolerant using omnipaxos. The web server should be accessed via REST or gRPC and store some data that is accessed in a consistent manner. Apart from standard GET and POST operations, our web server also supports compare-and-swap (CAS).

3 Solutions and Evaluation

3.1 Architecture

We have implemented the following architecture:

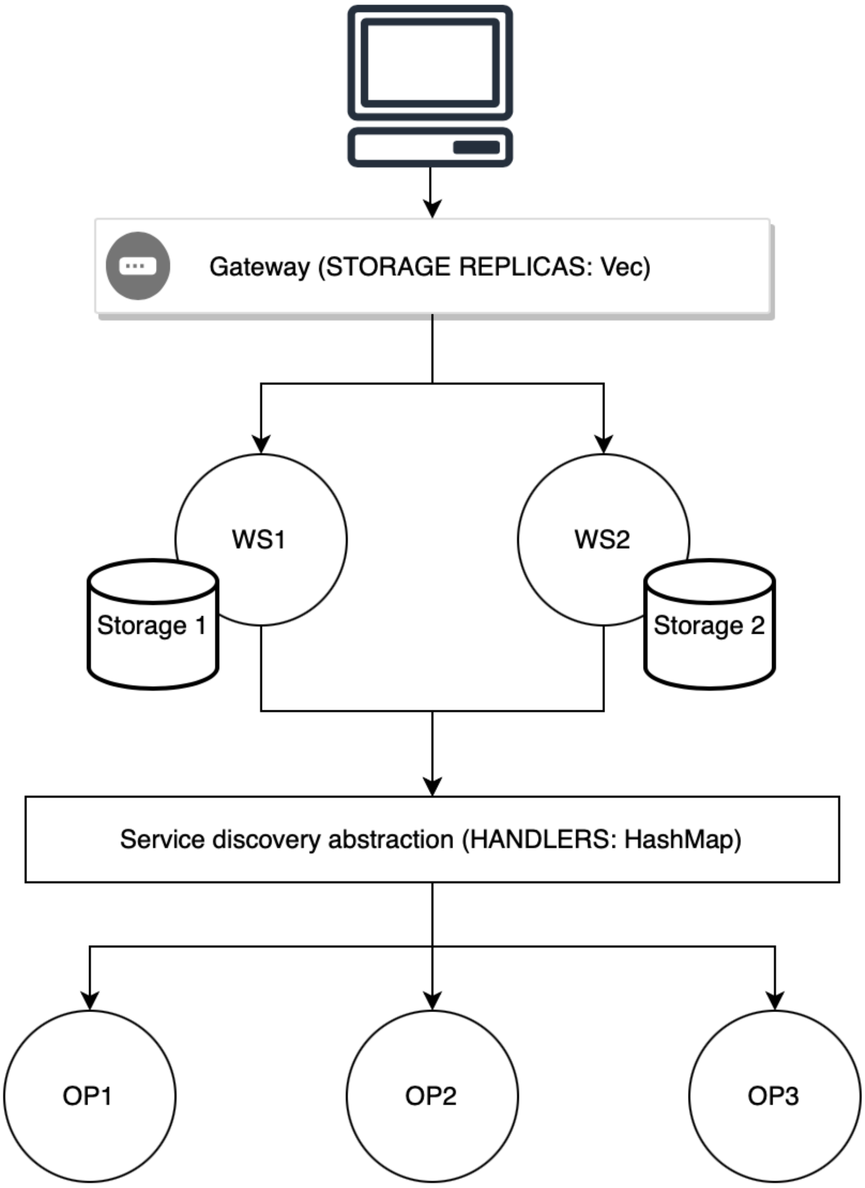

Figure 1: Web server architecture

Web server application is deployed on a separate machine or container and is mainly responsible for processing incoming requests and synchronising data with OmniPaxos servers.

Each web server has a local storage that keeps the key-value state after each synchronisation. In case of a failure, we can just start a new web server process and run an initial synchronisation, without losing any data.

3.2 Features

The developed web server supports the following operations:

- Read a value buy its key
- Write a new key-value record
- Compare and swap value for the given key

The detailed workflow for each operation is presented on a corresponding diagram.

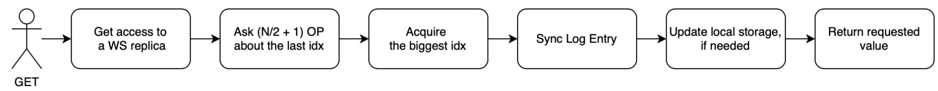

Figure 2: Read a value buy its key

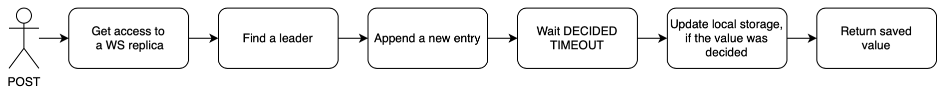

Figure 3: Write a new key-value record

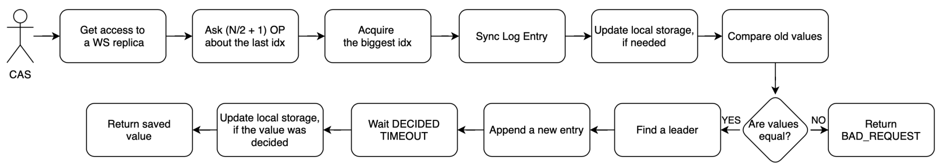

Figure 4: Compare and swap value for the given key

As you can observe on the given diagrams, we synchronise logs and a local storage with each GET/CAS operation, which ensures a linearisable read.

—- I WOULD DELETE FROM HERE —-

The `utils.rs` module defines several constants which are used in the project, specifically for buffering and timing. Those constants are:

```
use std::time::Duration;

pub const BUFFER_SIZE: usize = 10000;
pub const ELECTION_TIMEOUT: Duration = Duration::from_millis(100);
pub const OUTGOING_MESSAGE_PERIOD: Duration = Duration::from_millis(100);
```

```
pub const WAIT_LEADER_TIMEOUT: Duration = Duration::from_millis(500);
pub const WAIT_DECIDED_TIMEOUT: Duration = Duration::from_millis(250);
```

- The `BUFFER_SIZE` is a `usize` constant with a value of 10,000, representing the buffer size for network communications.
- The `ELECTION_TIMEOUT` is a Duration constant set to 100 milliseconds. This is the duration before a new election is triggered in the Paxos system if a leader is not detected.
- The `OUTGOING_MESSAGE_PERIOD` is a Duration constant, which is set to 100 milliseconds. This duration indicates the interval at which outgoing messages are sent by the Paxos system.
- The `WAIT_LEADER_TIMEOUT` is a Duration constant also witch set to 500 milliseconds. This duration is the time the system waits for a leader to be detected before proceeding with other tasks.
- Last but not least the `WAIT_DECIDED_TIMEOUT` is a Duration constant set to 250 milliseconds. This duration is the time the system waits for a decision on a key-value pair before checking again.

The project uses these constants to ensure consistent timing and buffering settings. They help manage the behavior of the distributed Paxos system, ensuring that elections, message sending, and decision-making are handled with appropriate timing.

3.3 kv.rs and kv_controler.rs

The kv.rs module is responsible for creating a key-value store that allows for easy access and modification of data. To ensure thread safety, it uses a HashMap and a mutex. Moreover, it also incorporates a Snapshot trait that enables the creation and merging of snapshots of the key-value store. We also made sure the storage is a singleton Value to make read and write operations consistent.

We used the `fn get_storage()` method that returns an `Arc<Mutex<KVStore>>`

```
pub(crate) fn get_storage() -> Arc<Mutex<KVStore>> {
       unsafe {
           match KV_STORE {
               None => {
                   let kv_store = Arc::new(Mutex::new(KVStore {
                       key_value: HashMap::new(),
                       decided_idx: 0,
                   }));
```

```
                    KV_STORE = Some(kv_store.clone());
                    kv_store
                }
                Some(ref kv_store) =>
                    kv_store.clone(),
            }
        }
    }
```

The module kv_controller.rs is in charge of managing HTTP requests that are linked to the key-value store in a Rust-based web application that utilizes Actix Web, a well-known Rust web framework. Its primary function is to offer two API endpoints that allow for creating and retrieving key-value pairs.

To obtain a value for a specific key from the store, we use the `get(key: Path<String>)`, a GET request can be made to /key-value/key. The URL path should include the key parameter, and the response will provide a KeyValueResponse structure or a `"404 Not Found"` status if the key is not found in the store. It is necessary to include the key parameter in the URL path to retrieve the value for that specific key.

```
    #[get("/key-value/{key}")]
pub async fn get(key: Path<String>) -> HttpResponse {
    let response = get_kv(key.into_inner()).await;

    return if response.key.is_empty() {
        HttpResponse::NotFound()
            .content_type("application/json")
            .status(StatusCode::NOT_FOUND)
            .finish()
    } else {
        HttpResponse::Ok()
            .content_type("application/json")
            .status(StatusCode::OK)
            .json(response)
    };
```

3.4 servers.rs

The servers.rs component describes an OmniPaxos server that utilizes the Paxos distributed consensus algorithm through the OmniPaxos library. This server is responsible for managing node communication, processing incoming messages, and dispatching outgoing messages.

The OmniPaxos server is executed in a continuous loop by the run() method, which operates in an asynchronous manner. This method employs two tokio intervals - election_interval for managing election timeouts and outgoing_interval for transmitting outgoing messages. Additionally, it monitors the incoming Receiver channel for any new messages and handles them by utilizing the handle_incoming() function of the omni_paxos instance.

```
pub(crate) async fn run(&mut self) {
    let mut outgoing_interval = time::interval(OUTGOING_MESSAGE_PERIOD);
    let mut election_interval = time::interval(ELECTION_TIMEOUT);
    loop {
        tokio::select! {
            biased;
            _ = election_interval.tick() => { self.omni_paxos.lock()
            .unwrap()
            .election_timeout(); },
            _ = outgoing_interval.tick() => { self.send_outgoing_msgs().await; },
            Some(in_msg) = self.incoming.recv() => { self.omni_paxos.lock()
            .unwrap()
            .handle_incoming(in_msg); },
            else => { }
        }
    }
}
```

The function configure_persistent_storage() requires a String type path as input and produces a PersistentStorageConfig object as output. It operates by generating a LogOptions object and a sled::Config object using the path provided, which it then utilizes to create a default PersistentStorageConfig.

```
      pub(crate) fn configure_persistent_storage(path: String) ->
      PersistentStorageConfig {
        let log_opts = LogOptions::new(path.clone());
        let mut sled_opts = Config::new();
        sled_opts = Config::path(sled_opts, path.clone());

        // generate default configuration and set user-defined options
        let persist_config = PersistentStorageConfig::with(
            path.to_string(), log_opts, sled_opts);

        persist_config
    }
}
```

The servers.rs module provides the necessary structure and methods to run an OmniPaxos server, manage communication between nodes in the network, and handle timeouts and message processing.

3.5 storage.rs

The management of `key-value storage`, including the creation and retrieval of key-value pairs, as well as synchronization between local storage and the distributed Paxos system, is handled by the `storage.rs` module.

The `sync_decided_kv()` method is responsible for synchronizing the local key-value storage with the distributed Paxos system. This function locks the `KVStore` and selects a server from `OP_SERVER_HANDLERS` randomly. Afterward, it retrieves the decided entries from the server and updates the local `key-value storage`. This method ensures that both snapshotted entries and decided entries are appropriately handled.

```
async fn sync_decided_kv() {
    let kv_store = KVStore::get_storage();
    let mut storage = kv_store.lock().unwrap();

    let handler = OP_SERVER_HANDLERS.lock().unwrap();
    let server_id = rand::thread_rng().gen_range(1..PEERS);
    // println!("Chosen server {}", server_id);
    let (server, _, _) = handler.get(&server_id).unwrap();

    let last_idx = server
        .lock()
        .unwrap()
        .get_decided_idx();
    println!("Last index {}", last_idx);
    println!("Local index {}", storage.decided_idx);

    let mut overlap = false;
    if storage.decided_idx == 0 { overlap = true; }
    if last_idx > storage.decided_idx {
        let committed_ents = server
            .lock()
            .unwrap()
            .read_decided_suffix(storage.decided_idx as u64)
            .expect("Failed to read expected entries");

        for (_, ent) in committed_ents.iter().enumerate() {
            match ent {
                LogEntry::Decided(kv_decided) => {
```

```
                    storage.decided_idx += 1;
                    storage.key_value.insert(kv_decided.key
                    .clone(), kv_decided.value);
                    println!("Adding value: {:?}, decided idx {} via server {}",
                             kv_decided.value, storage.decided_idx, server_id);
                }
                LogEntry::Snapshotted(kv_snapshotted) => {
                    for (k, v) in &kv_snapshotted.snapshot.snapshotted {
                        storage.decided_idx += 1;
                        storage.key_value.insert(k.clone(), *v);
                        println!("Adding value: {:?}, decided inx {} via server {}",
                                 v, storage.decided_idx, server_id);
                    }
                }
                _ => {} // ignore not committed entries
            }
        }
    }
```

Overall the storage.rs module provides the functions to create and retrieve key-value pairs while ensuring that the local key-value storage is synchronized with the distributed Paxos system.

3.6 main.rs

The `main.rs` module is as its name says our main program and serves as the entry point for the application of `server`. It sets up the OmniPaxos `key-value` store, initializes the servers, and starts the HTTP server. The HTTP server exposes two routes for creating and getting `key-value` pairs.

Firstly let's see the `main()` function witch sets up the environment and starts the HTTP server. It first calls the `cleanup()` function to remove any previous storage directories. Next, it initializes the OmniPaxos servers and handlers by calling `initialise_handlers()`. Finally, it starts the Actix HTTP server with two routes for creating and getting key-value pairs.

```
async fn main() -> std::io::Result<()> {
    // Clean-up storage
    cleanup();

    OP_SERVER_HANDLERS
        .lock()
        .unwrap()
        .extend(initialise_handlers());

    HttpServer::new(move || App::new().service(create).service(get))
```

```
        .bind(("127.0.0.1", 8080))?
        .run()
        .await
}
```

We can see the methods/functions in our `main` module.

- The `cleanup()` function removes any previous storage directories.
- The `initialise_channels()` creates sender and receiver channels for the OmniPaxos system.
- The `initialise_handlers()` initialize the OmniPaxos servers, handlers, and configurations.
- Finally the `recovery()` is use to recover a failed node by recreating its storage and rejoining the network.

3.7 test.rs

Finally, in the `test.rs` module containing tests for the OmniPaxos key-value store implemented in the main application. It uses the `restest` library to test the HTTP endpoints and `tokio` for asynchronous execution. The tests include creating and getting key-value pairs, both individually and concurrently. Namely, the test functions are:

- `test_create()`
- `test_multiple_create()`
- `test_get()`

To run these tests, one would need to start the server first. Unfortunately, the `restest` library do not provide embedded application testing setups.

—- I WOULD DELETE TO HERE —-

4 Running the server

4.1 Start the server

In this section, we will explain how to run the server on your local machine.

First of all, choose a free port on your computer, which can be used for the server, and update the configuration. Usually, it is 8000 or 8080, but one may choose whatever port they wish.

The configuration can be changes in the `main` module at the `main()` function:

`.bind(("127.0.0.1", 8080))?`.

Then you should go to your terminal and run the `cargo run` command or inside from your IDE click the run button on the `main()` function. You should see the following output:

```
help: use `let _ = ...` to ignore the resulting value
   |
72 |     let _ = fs::remove_dir_all("storage3");
   |     +++++++

warning: `web_server` (bin "web_server") generated 10 warnings (run `cargo fix --bin "web_server"` to apply 3 suggestions
    Finished dev [unoptimized + debuginfo] target(s) in 0.41s
     Running `target/debug/web_server`
```

By default, there are 3 running instances of the web server and also 3 instances of OmniPaxos servers.

Now, you can use Postman to perform operations on a web server.

4.2 Create a key-value record

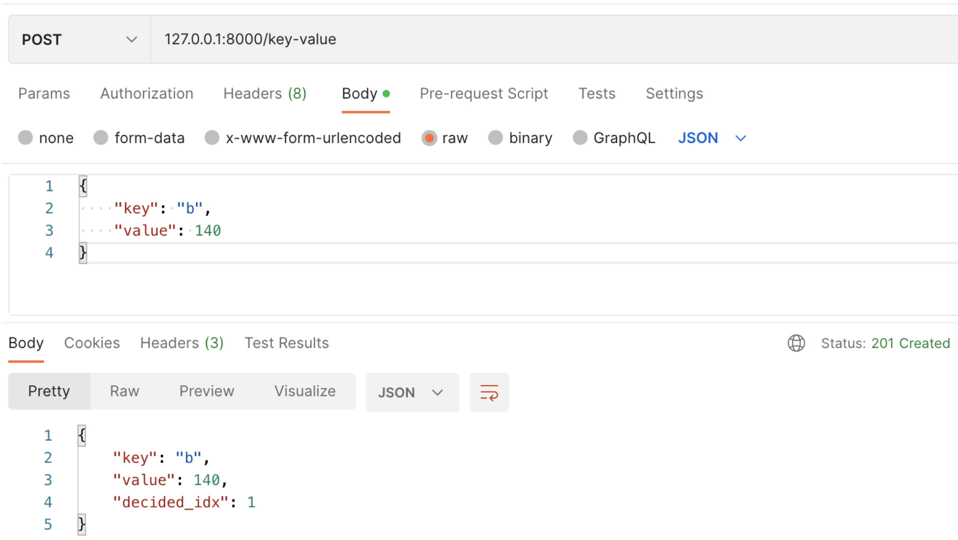

Figure 5: Successful CREATE output

4.3 Read a value by a non-existent key

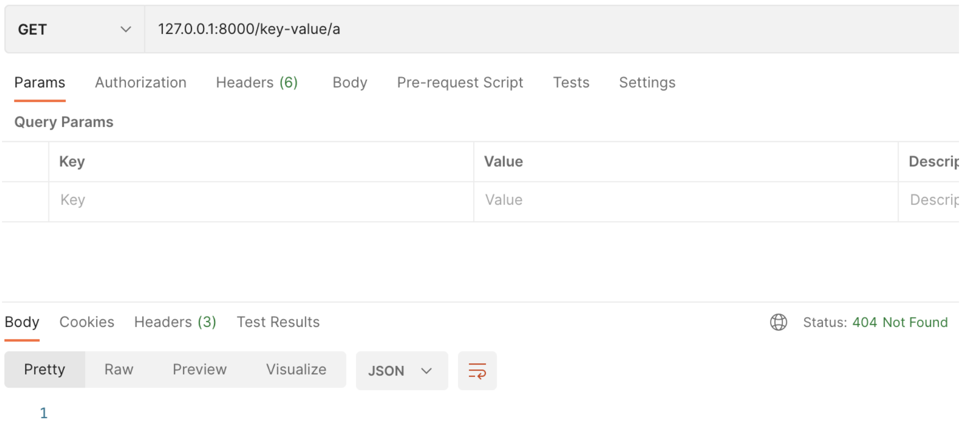

Figure 6: Unsuccessful READ output

4.4 Read a value by an existent key

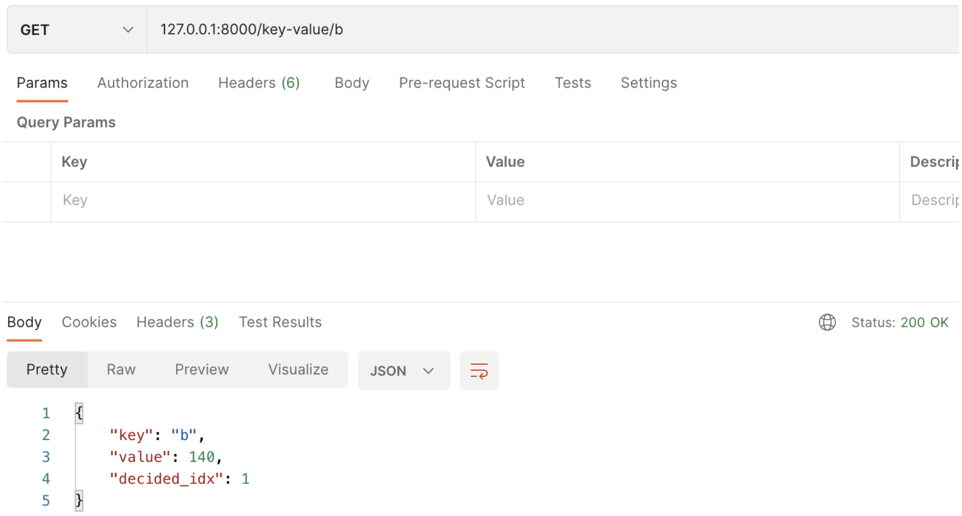

Figure 7: Successful READ output

4.5 Compare and swap value for the given key with a wrong old value

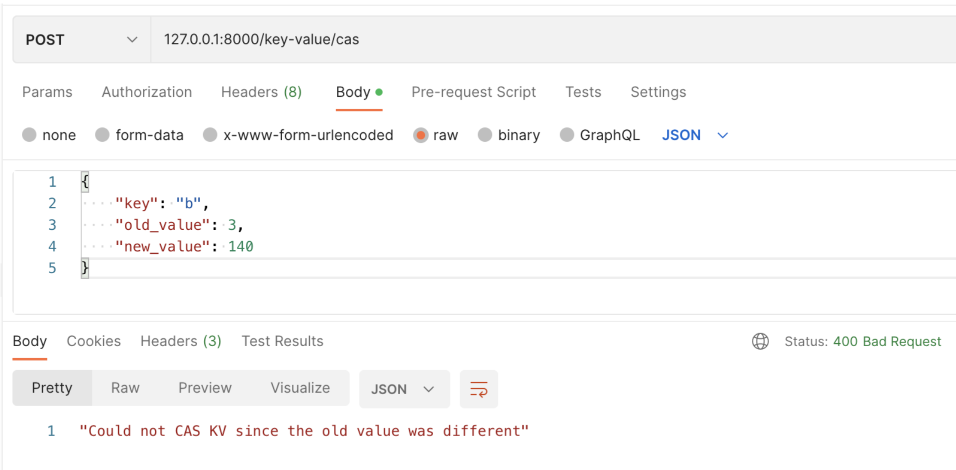

Figure 8: Unsuccessful Compare and Swap output

4.6 Compare and swap value for the given key with a correct old value

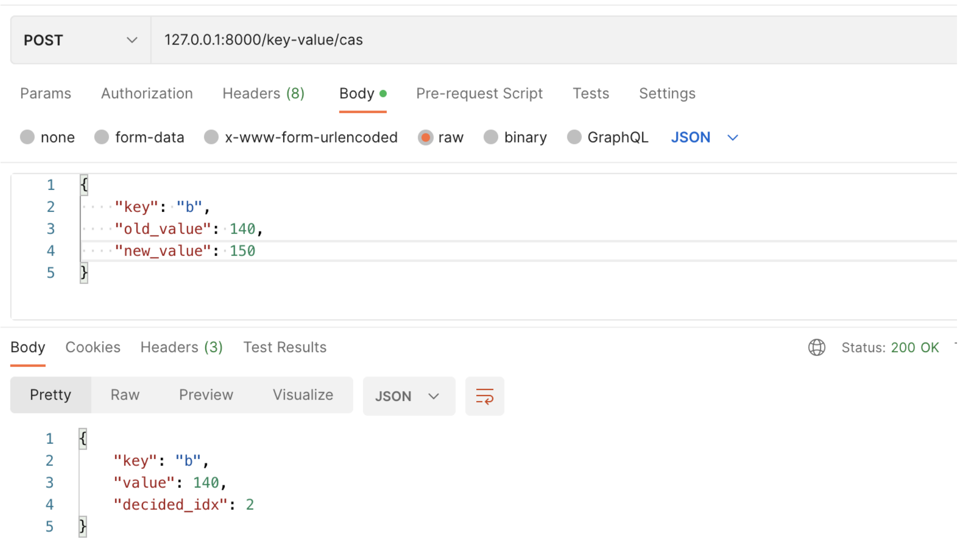

Figure 9: Successful Compare and Swap output

Please, keep in mind that this operation return the OLD value. But if we read this key after CAS, we will get a new value

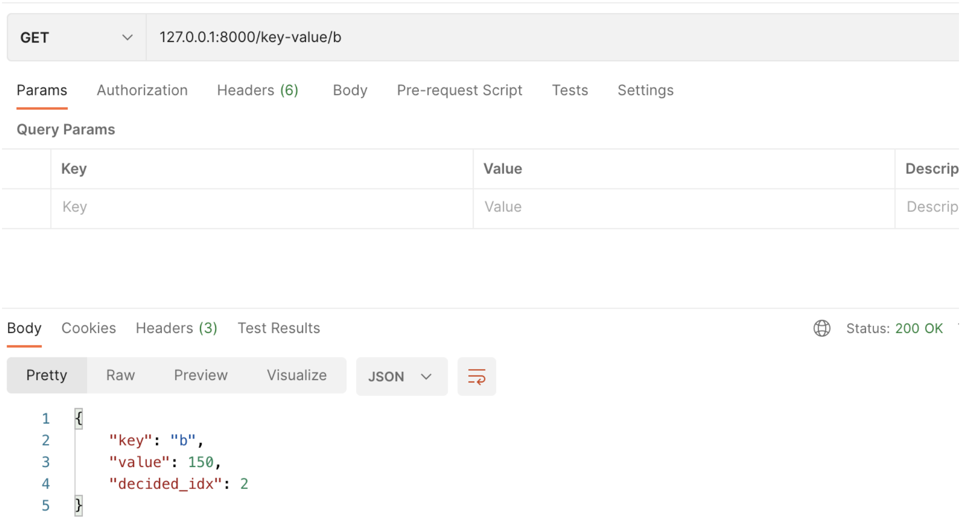

Figure 10: Updated Value after successful CAS

5 Testing

Once you started the server, you can also run tests, that are under test module. You should get the similar output:

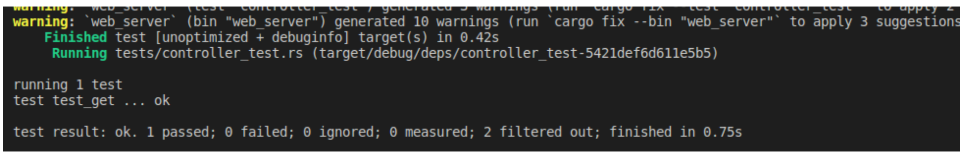

Figure 11: Test output

We have developed the following tests:

- test_create(): tests a successful CREATE operation
- test_multiple_create(): tests multiple successful CREATE operations
- test_get(): tests a successful GET operation
- test_cas(): tests a successful CAS operation
- test_unsuccessful_cas(): tests an successful CAS operation

6 Conclusions

In this project, we have implemented a highly-available fault-tolerant web server, that stores and accesses key-value data in a consistent manner.

We have implemented operations that are sequential (provided by the OmniPaxos library) and linearisable (ensure by our synchronisation mechanism before each operation).

We also acquired new practical skills, that are based on the knowledge, provided by the course. It gave us an excellent grasp of the course outcome knowledge.