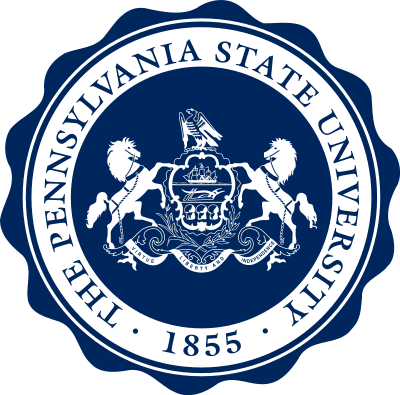
**CSE 511: Operating Systems**

**Spring’20**



Project 2: Replicated

Linearizable Key-Value Store - Report

2nd May 2020

Harshit Passi

Sanaz Mahmoodi Takaghaj

Arjun Menon

Contents

[System Description 3](#_Toc39349476)

[Design 4](#_Toc39349477)

[Server 4](#_Toc39349478)

[Non-Blocking ABD Protocol 4](#_Toc39349479)

[Blocking Protocol 5](#_Toc39349480)

[Experimental Evaluation 5](#_Toc39349481)

[Number of Servers (N) 6](#_Toc39349482)

[Client Crashing 7](#_Toc39349483)

[README (How to execute) 7](#_Toc39349484)

[To run 7](#_Toc39349485)

[To test linearizability 7](#_Toc39349486)

# System Description

The system is built using Python, with the salient features being:

* All servers are implemented as simple web servers using the Bottle framework.
* We use HTTP calls for communication between clients and servers, however, calls are not retried upon failures, and we use the requests-futures library to send simultaneous asynchronous HTTP calls.
* The requests-futures library implicitly spins up threads to send HTTP requests. Responses are handled by an as\_completed iterator, which runs whenever a response is received, and breaks when a majority of responses are received.
* The key-value store is implemented as a MongoDB collection, completely local to each server. We simply use MongoDB as a basic key-value store, without using any of its replication features. The only interaction with each MongoDB store is through the server API running on the node.
* Each key-value pair is stored as a document in the collection. The structure of the object is:

{

'key': 'string based key',

'value': 'string based value',

'ts': {

'id': 0, # client ID - integer

'integer': 0 # Logical timestamp - integer

}

}

* Linearizability is tested using log traces fed to the Knossos framework. All specified test cases are proven to be linearizable. Instructions to test it are in the readme file, as well as the readme section of this document.
* In essence, we simultaneously run multiple clients as processes on a single machine, use the machine’s clock to create a timestamp for each log entry, finally merging and sorting the output. This simulates a global clock in respect to the clients.
* There are two separate client drivers, one using the ABD protocol, and the other using a majority voting-based blocking protocol. The server-side code is common for both protocols.

# Design

All servers for both blocking and blocking protocols were hosted on AWS as shown in Figure 1.

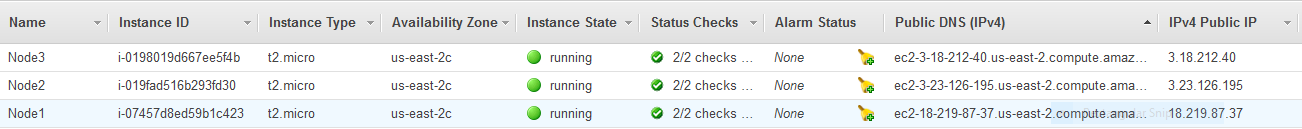


Figure 1: Servers hosted on AWS

All servers run the Server code which is described below:

## Server

Creates a connection with local MongoDB which serves as key-value store. The public server API is described in the readme file.

**read\_kv:**

If key already exists, returns it. Otherwise, returns a blank object.

**create\_kv:**

If key does not exist, creates a new key-value pair with current timestamp. Otherwise, updates the value and timestamp of the existing entry if the integer part of the stored timestamp is less than the integer part of the current timestamp or if the stored client-id part of the stored timestamp is less than the current client-id part of the stored timestamp when the formers are equal.

**acquire\_write\_lock:**

For the blocking protocol, a lock is acquired for each key and a lock-map is used to store per-key. When a client acquires a lock, it stores the client-id in the lock-map and then returns the server-id.

**release\_write\_lock:**

When the client releases the lock, it deletes the key and the client-id from the lock-map.

## Non-Blocking ABD Protocol

**write(key, value):**

Sends write-query to all servers using asynchronous HTTP calls and waits for response from the majority. Then creates and sends a request payload with the new value and an updated timestamp (highest timestamp+1). It returns “success” after receiving acknowledgment from the majority quorum.

**read(key):**

Sends read-query to all servers using asynchronous HTTP calls and waits for response from majority to get the item with the latest timestamp. Then, it sends the latest item to all servers simultaneously and waits to receive acknowledgement from the majority quorum. It then returns the latest value.

**query\_all\_servers:**

Initiates read API calls to all servers simultaneously and handles the calls as they are asynchronously completed by the majority quorums. It compares received timestamps (integer, id) from each server to determine the latest item:

First, integer values are compared to find the highest timestamp and then client ids are compared to break the tie when the two integers are equal. This handler function returns the latest item.

## Blocking Protocol

**Algorithm:**

* Client write protocol:
  + Sends per-key lock acquisition requests to all servers, breaking after receiving a response from a majority of servers.
  + If all responses are true, the client proceeds to query the majority servers for the key and chooses the highest timestamp among them.
  + It then increments the timestamp, updates it in its payload, and sends a write request to all servers.
  + After receiving acknowledgements from a majority, it breaks and sends a release lock request to all servers.
  + Upon receiving a response from the majority, it finishes execution.
* Client read protocol:
  + Sends per-key lock acquisition requests to all servers, breaking after receiving a response from a majority of servers.
  + If all responses are true, the client proceeds to query the majority servers for the key and chooses the highest timestamp among them.
  + It then performs write requests to all servers with the latest key-value pair as payload.
  + After receiving acknowledgements from a majority, it breaks and sends a release lock request to all servers.
  + Upon receiving a response from the majority, it returns the latest item’s value.
* Server protocol:
  + The server maintains a map data structure called lock\_map. The keys for this structure are the keys in the key-value store. The values are the client IDs of the client that has a lock on that key.
  + On receiving an acquire lock request, it checks whether the key exists in the lock\_map. If not, it adds the key-client ID pair to it and returns its server ID (so that the client knows which server granted it the lock). If it already exists in the lock\_map, it returns FALSE.
  + On receiving a release lock request, the server deletes the key from the lock\_map.
  + Reads and writes are the same as the ABD protocol.

**write(key, value):**

Acquires write-locks from majority quorum and saves the granting server ids in an array. It then sends queries to the majority servers to get the latest timestamp. It then creates a payload with the new value and initiates write API calls to simultaneously send the new value to all the servers. After receiving acknowledgement from the majority, it releases the lock and returns “success”.

**read(key):**

Acquires read-locks from majority quorum and saves the server ids in an array. It then sends query requests to these servers to get the item with the latest timestamp. It then creates a payload with the latest item and initiates write API calls to send the latest item to all the servers simultaneously. After receiving acknowledge from the majority, it releases the lock and returns the latest value.

**query\_majority\_servers:**

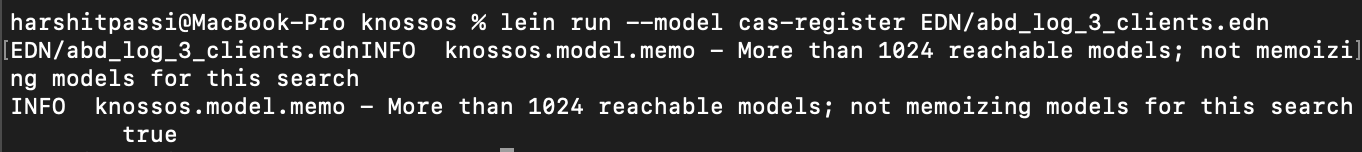
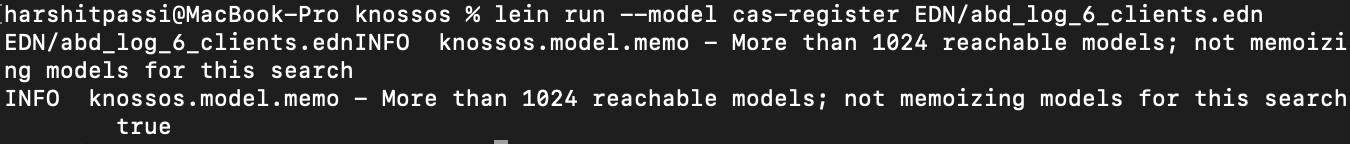
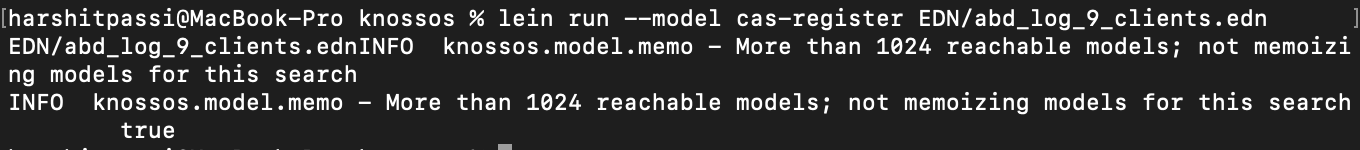
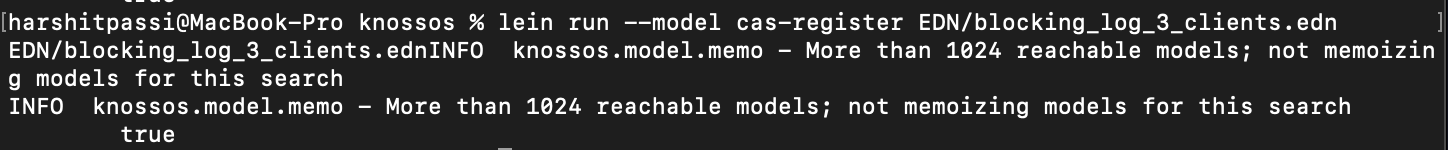
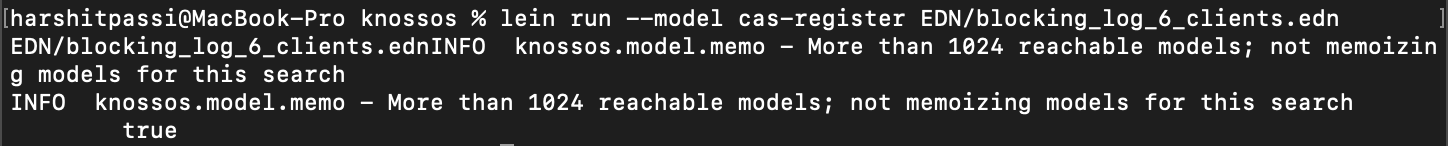
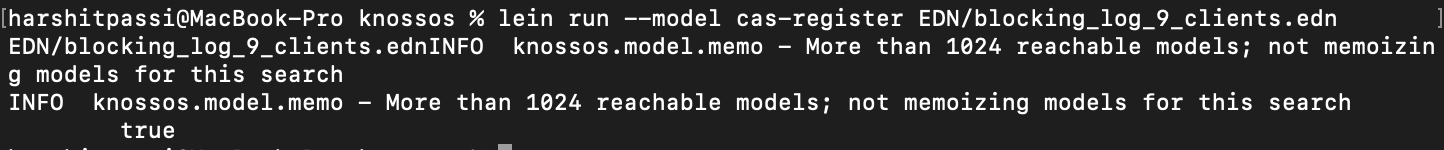
Initiates simultaneous read API calls to the majority servers and handles the calls as they are asynchronously completed by the majority quorums. It compares received timestamps (integer, id) from each server to determine the latest item. This function is similar to “query\_all\_servers” function but it only sends query to the majority of the servers to get the latest item.

**release\_all\_locks:**

This handler function releases all acquired locks.

# Experimental Evaluation

## Linearizability Proof

* ABD Protocol – 3 clients, 501 requests
* ABD Protocol – 6 clients, 1002 requests
* ABD Protocol – 9 clients, 1503 requests
* Blocking Protocol – 3 clients, 501 requests
* Blocking Protocol – 6 clients, 1002 requests
* Blocking Protocol – 9 clients, 1503 requests

## Number of Servers (N)

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=1 | N=3 | N=5 |
| Throughput  (requests/sec) |  |  |  |
| Latency (Median) with get/put ratio of 10% |  |  |  |
| Latency (95th Percentile) with get/put ratio of 10% |  |  |  |
| Latency (Median) with get/put ratio of 90% |  |  |  |
| Latency (95th Percentile) with get/put ratio of 90% |  |  |  |

Table 1: Performance Evaluation of ABD algorithm

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=1 | N=3 | N=5 |
| Throughput  (requests/sec) |  |  |  |
| Latency (Median) with get/put ratio of 10% |  |  |  |
| Latency (95th Percentile) with get/put ratio of 10% |  |  |  |
| Latency (Median) with get/put ratio of 90% |  |  |  |
| Latency (95th Percentile) with get/put ratio of 90% |  |  |  |

Table 2: Performance Evaluation of the Blocking algorithm

## Client Crashing

|  |  |  |
| --- | --- | --- |
|  | ABD Algorithm | Blocking Algorithm |
| Throughput  (requests/sec) |  |  |
| Latency (Median) with get/put ratio of 10% |  |  |
| Latency (95th Percentile) with get/put ratio of 10% |  |  |
| Latency (Median) with get/put ratio of 90% |  |  |
| Latency (95th Percentile) with get/put ratio of 90% |  |  |

Table 3: Impact of client crashing on ABD vs the Blocking algorithm

# README (How to execute)

## To run

1. Make sure Python 3 and pipenv are installed on your system.
2. Go into the project directory and run pipenv install
3. Once the dependencies have finished installing, run pipenv shell
4. Make sure the mongo connection string in server.py is pointing to a local mongo db.
5. Then you can run python server.py to run the server. Enter a server ID matching the order of the server inside the config.
6. If you're running a client instead, then run python <client\_file\_name>.py
7. The config file should contain the addresses of the servers, in order of the server IDs given to each server.
8. There are two clients offered: ABD and Blocking

## To test linearizability

1. There are EDN files of the executions in the EDN folder.
2. The test folder contains test scripts that can be executed to generate new EDN files.
3. Make sure you have no edn files in the test folder.
4. The clients can be run concurrently using the bash/zsh & operator. For example, if you want to concurrently run 3 blocking clients, you would have to run:

python3 blocking\_client\_api.py&python3 blocking\_client\_api2.py&python3 blocking\_client\_api3.py.

1. This would generate 3 timestamped edn files, that you can combine and sort using the unix sort function like so:

sort -u --files0-from=<(printf '%s\0' \*.edn) -o output.

1. output would contain sorted and timestamped log entries from all processes. The timestamps and colons can be removed using a macro in your favorite text editor. Add a [ in the beginning of the file and ] at the end.
2. Add a .edn extension, and your file would be ready to run against the knossos framework.