Distributed Key-Value Store

Weifeng Han

May 3, 2022

1 Introduction

This is my design of a key-value storage system. I used multithreads in python to mimic a distributed environment where each thread represents a replica of server for storing key-value pairs. The broadcast strategy applied to ensure the consistency is total order broadcast. The tree structure of the repository can be found in **Screenshot 5** in **Appendix**.

2 Design

2.1 Overall Design

2.1.1 Open Servers In Different Mode

To open server in eventual consistency mode, one can call:

$ python3.9 servers.py –consistency=eventual

And the same for any other consistency modes. Currently, the system only supports eventual consistency and sequential consistency.

2.1.2 Distributed Servers In Threads

The number of replicas created when running **servers.py** depends on a variable called **numberOfReplicas** in **servers.py**. Basically, for each replica that we want to create, the program will generate two threads. One thread listens to a port where messages from clients would be sent to. The other thread listens to another port where broadcasts between replicas would be sent to. So, for example, if we want to have 2 replicas, there would be 4 threads and 4 ports. Please see section 5.1 for limitations on this implementation. **Due to the limitation discussed in section 5.1, for all the test cases, I use the default number of servers to be 3.**

2.1.3 Syntax For Messages

A user will enter memcached-like commands in the following syntax:

```
command := set var_name byte
         | get var_name
# set and get are literals, var_name and byte are parameters.
```

A *set* command is parsed into the following syntax:

```
msg = ['set', key, value, byte]
```

A *get* command is parsed into the following syntax:

```
msg = ['get', key]
```

A *broadcast* message is parsed into the following syntax:

```
msg = ['broadcast', key, value, byte, timestamp, 'message']
# timestamp is a tuple in the form of [serverID, Time]
```

A *acknowledgement* to a broadcast message is parsed into the following syntax:

```
msg = [broadcastTS, key, value, byte, timestamp, '
                                  acknowledgement']
# timestamp is a tuple in the form of [serverID, Time]
# broadcastTS is the timestamp of the initial message
```

2.2 Eventual Consistency

2.2.1 Get Command

The client side parses the user input into the *get* format shown in **section 2.1.3** and send the parsed message to a port. The server side will find the value associated with the given key in the local file **storage.txt**.

2.2.2 Set Command

The client side parses the user input into the *set* format shown in **section 2.1.3** and send the parsed message to a port. The server that listens to that particular port then starts to broadcast the message. After the broadcast, the server **does not wait** until the the total order broadcast finishes. It immediately send the reply, `'STORED'`, back to the client.

2.2.3 Total Order Broadcast

When a server receives a *broadcast* message, it will encode the timestamp and then add the encoded timestamp to a priority queue. Roughly speaking, for instance, when `msg = ['broadcast', 'x', '42', '4', '[0,2]', 'message']` is received, `heappush(pqueue, encoding([0,2])` will add the timestamp to the priority queue. When a server receives an *acknowledgement* message,

it will add this message to a dictionary that maps the broadcast timestamp (see *acknowledgement* in **section 2.1.3**) to the number of acknowledgements that has been received. For instance, let's assume that `msg = ['[0,2]', 'x', '42', '4', '[1,4]', 'acknowledgement']` is received. Also, assume that acknowledgement of this message has been received 4 time prior to this one, i.e., `acknowledgementDictionary =['[0,2]': 4]`. Then, after this operation, the count will be incremented by 1, from 4 to 5, i.e., `acknowledgementDictionary =['[0,2]': 5]`. Eventually, when the first timestamp in the priority queue has received acknowledgement from all replicas, it will be popped out of the priority queue and write that key-value to the local file **storage.txt**.

2.3 Sequential Consistency

2.3.1 Get Command

The same as **section 2.2.1**.

2.3.2 Set Command

Almost the same as **section 2.2.2**, but the server **does wait** until the the total order broadcast finishes. I acquired a semaphore right after broadcasting the message so that the wait feature can be achieved. It's worth noticing that this is a local semaphore that other servers are not supposed to access. The reason for this semaphore is because each server would have two threads as discussed in **section 2.1.2**. We want to freeze the thread that handles client request until the thread that handles broadcasts between servers finish writing to the local **storage.txt**. When the total order broadcast is done and the write instruction has been issued successfully, the semaphore will be released and then the reply could be sent.

2.3.3 Total Order Broadcast

Almost the same as **section 2.2.3**. The only extra content is that when a message is popped out from the priority queue, the semaphore will be released (see **line 106** in **servers.py**).

3 How to Run Tests

All bash script should under the **a3 dist** repository. Otherwise, the code doesn't work properly. After each test, you may want to reset the value of x to be 0 by typing the following.

```
$ ./client-xto0.sh
```

3.1 Functionality of Each Bash Script

If you ever get confused on what each bash script does, please refer to this section.

- **client-normal.sh** : It opens a client that allows two memcached-like commands - set and get (syntax can be found in **section 2.1.3**). All instructions are sent to the first server (port 9889).
- **client-read.sh** : It reads the value of x. All instructions are sent to the first server (port 9889).
- **client-stale.sh** : It first sets x to be 42, and then reads x. All instructions are sent to the first server (port 9889).
- **client-xto0.sh** : It sets x to be 0. All instructions are sent to the first server (port 9889).
- **server-normal.sh** ***mode*** : it opens 3 replicas of the given ***mode*** that operate normally.
- **server-stale.sh** ***mode*** : it opens 3 replicas of the given ***mode*** that sleeps for 5 seconds before writing to **storage.txt**.

3.2 How to Run Staleness Tests

3.2.1 Eventual

First, start eventual servers that sleep for 5 seconds before finish any writing command by typing the following. This will start 3 eventual servers.

```
$ ./server-stale.sh eventual
```

Next, open another session and run a example client by typing:

```
$ ./client-stale.sh
```

This client does the thing mentioned above in section ?. It contacts port 9889, which is the first of server, and send x=42 then read x. By expectation, we should get x=0, which is its initial value. Wait for 5 seconds, if you look at **storage.txt** in any of the three folders, we should see that x's value has been set to 42. Thus, the staleness of the eventual consistency can be demonstrated. You can check this scenario in **Screenshot 1** in the **Appendix** section.

3.2.2 Sequential

First, start sequential servers that sleep for 5 seconds before finish any writing command by typing the following. This will start 3 sequential servers.

```
$ ./server-stale.sh sequential
```

Next, open another session and run a example client by typing:

```
$ ./client-stale.sh
```

This client does the thing mentioned above in section ??. It contacts port 9889, which is the first of server, and send x=42 then read x. By expectation, we should get x=42, which is not its initial value. Thus, in contrary to eventual consistency, we have demonstrated that the latency in set commands will not affect the consistency in a sequential model.You can check this scenario in **Screenshot 2** in the **Appendix** section.

3.3 How to Demonstrate Inconsistent Reads In Sequential Consistency

First, start sequential servers that sleep for 5 seconds before finish any writing command by typing the following. This will start 3 sequential servers.

```
$ ./server-stale.sh sequential
```

Next, open another session and run an example client by typing:

```
$ ./client-stale.sh
```

Lastly, again, open another session and run a different example client by typing:

```
$ ./client-read.sh
```

This client contacts the second server and read x from it. If you run this command within five seconds after you run **client-stale.sh**, you should see that this clients return the initial value of x instead of the value set by **client-stale.sh**.You can check this scenario in **Screenshot 3** and **Screenshot 4** in the **Appendix** section.

3.4 How to Run Custom Tests

First, start servers of any **mode** (either **eventual** or **sequential**) by typing the following. This will start 3 servers of the designated mode.

```
$ ./server-normal.sh mode
```

Next, open another session and run a client by typing:

```
$ ./client-normal.sh
```

This client contacts port 9889, which is the first of server. You can experiment with any set or get commands now.

4 Performance Evaluation

4.1 How To Run The Performance Test

First, start servers of either **eventual** mode or **sequential** mode by typing the following. This will start 3 servers of the designated mode.

```
$ ./server-normal.sh mode
```

Next, open another session and run a client by typing:

```
$ ./performance.sh
```

4.2 Conclusion

After running 100 pairs of alternating writes and reads, we notice that eventual consistency only took 0.544 seconds whereas sequential consistency took 2.068 seconds. However, the reads were consistent in the sequential modal, but only read up to 27 in the eventual modal. This makes sense because eventual only used $\frac{1}{4}$ of the time compared to sequential. So, there are $\frac{3}{4}$ more content to be broadcast and modified in the background still.

5 Limitation

5.1 Servers In Threads

If we want 3 replicas, we have to make sure that there are folders called **0**, **1**, and **2**. In addition, we have to make sure that the **storage.txt** file in all folders are identical. If this is not met, errors could potentially occur while running this program.

5.2 Encoding Of Timestamps

The encoding of timestamps is for the priority queue. The current implementation only supports up to 100 servers. For more than 100 servers, the encoding of timestamp can potentially produce the same results for two different timestamps.

5.3 Only One Client At A Time

The current implementation renders that each server can only take one client at a time. The next client request will be handle **after** the current client request ends.

5.4 Lack of Interface to Connect to Arbitrary Ports

Although I have supported this feature in **client.py**, I didn't let you choose the port in the test cases. If you really want to do that, reading a couple lines in client.py will serve the goal.

6 Further Implementation

The invariant of this design is that if we want **n** replicas, we assume that there is already folders named from **0** to **n-1**, and the **storage.txt** in all folders are identical. It denotes that the fault tolerance of this system is not good enough. For example, going back to **section 5.1**, if we want 3 replicas whereas we only created folder **0** and **1**, an error would occur while running this program. This could be implemented in the next iteration (if there is still going to be any).

7 Appendix

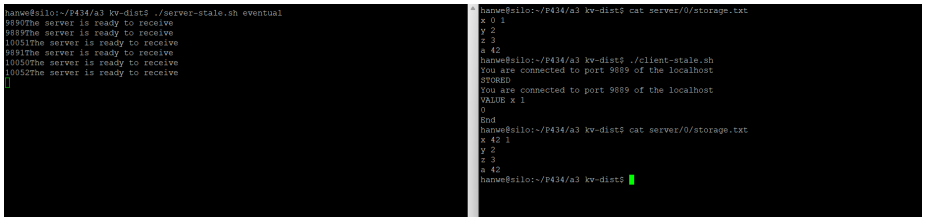

Figure 1: Screenshot 1

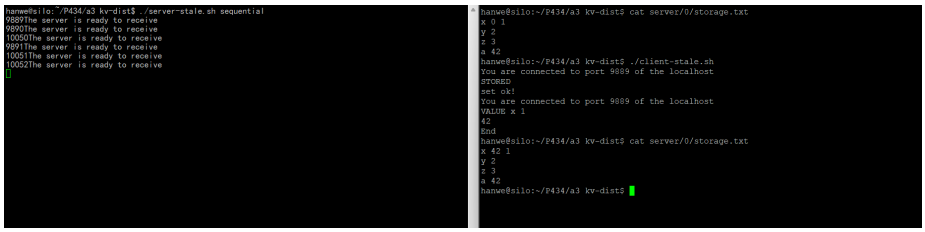

Figure 2: Screenshot 2

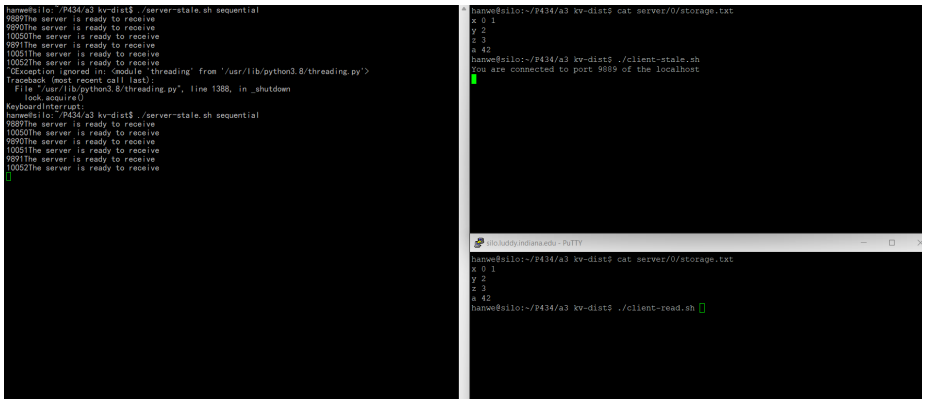

Figure 3: Screenshot 3

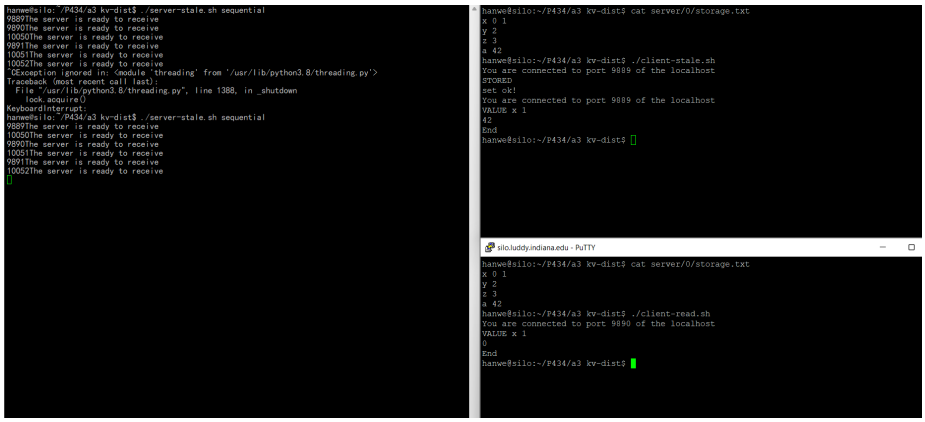

Figure 4: Screenshot 4

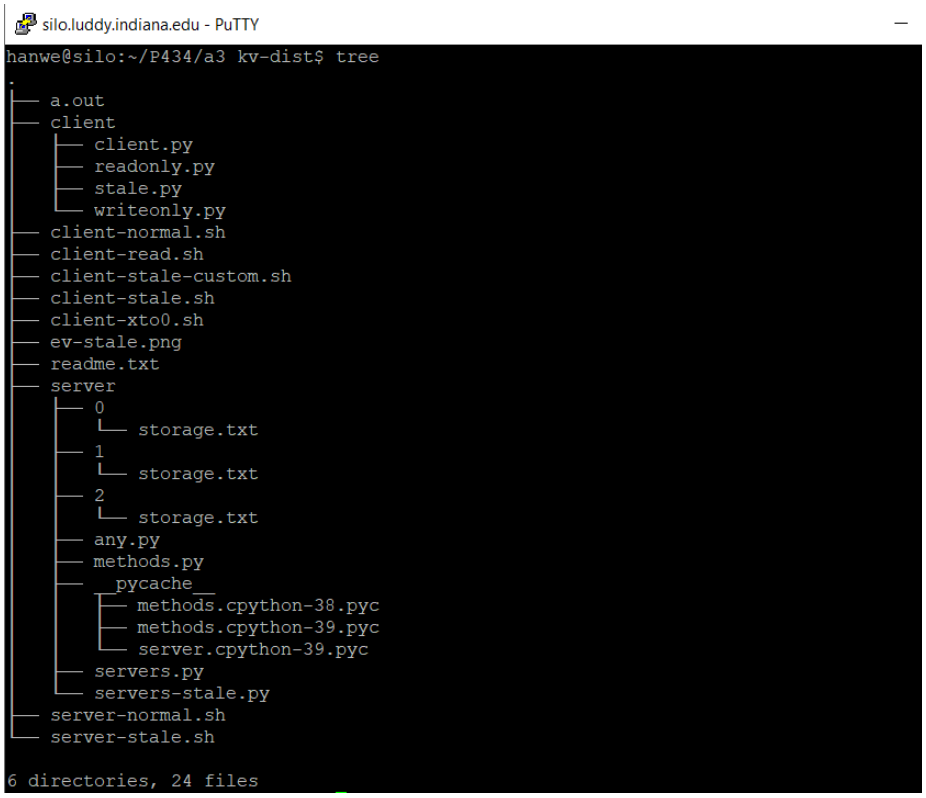

Figure 5: Screenshot 5

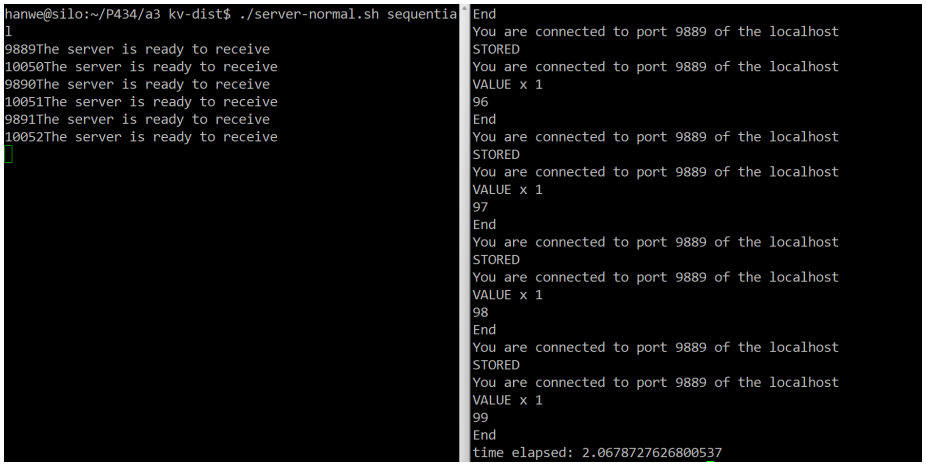

Figure 6: Sequential Performance

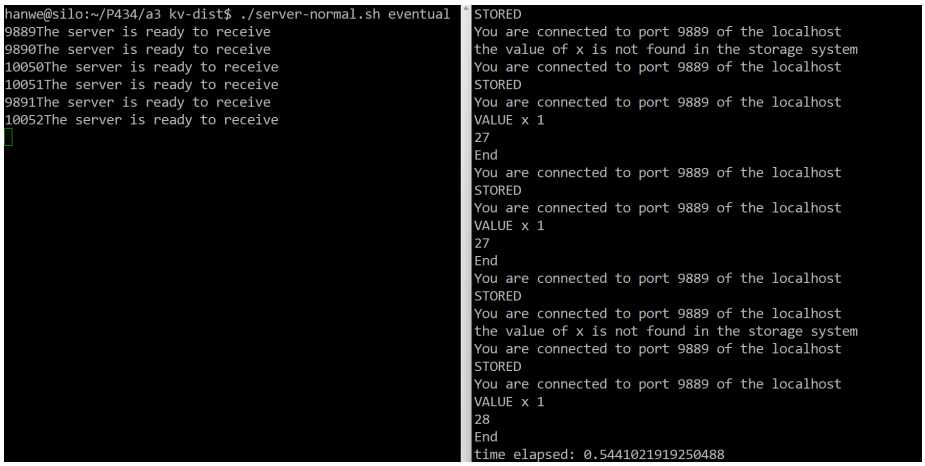

Figure 7: Eventual Performance