CSE 511 Project 3 Report

Ata Fatahi, Sushrut Shringarputale, Quinn Burke

December 11, 2018

Contents

1 Introduction

In this project, we extend our implementation of our key-value store to function in a distributed system. Specifically, we replicate our key-value store across multiple machines, and investigate two methods for maintaining consistency between client requests to the database. In the first part, we implement the non-blocking ABD protocol [1], which maintains clients that receive requests from end-users and communicates with the servers that are the final arbiters of what gets written to the database. This protocol needs to communication between any clients or server nodes, hence it is non-blocking - meaning that a slow client will not cause the rest of the system to stop progressing. In the second part, we implement a blocking protocol. A slow/unresponsive node in this implementation will cause the entire system to slow down, but requires no special implementation by the "client". This way places more work on the server side in order to maintain a distributed lock allowing writes to the database.

We choose to further extend our p1 multi-threaded server as the listener for client requests, and also implement a modified client to support the ABD protocol. Exact details on the implementation are as follows.

2 Building and Running

To build the project simply run `make`. This will build the server as well as all the clients that we tested with the server.

To run the server, use

```
./server [1] [2 <node_id> [<ip>:<port>]*] [r]
```

To invoke either of the servers, you must specify either of option 1 or 2. Specify option **1** to invoke the server supporting the ABD protocol. Then, you must invoke an ABD capable client. Specify option **2** to invoke the blocking server. Then, you must specify a <node_id> in order to participate in the consensus algorithm for the distributed lock. Also, you can optionally specify <ip>:<port> combinations of servers (peers) in the system you would like to connect to on startup of the current server. Lastly, you may optionally specify the **r** option in order to enter recovery mode before starting the server and listening for new peer or client connections.

To run a client, just run one of the generated client binaries, although in case the clients are not on the same system as the server, the IP address of the server may have to be manually changed in the code for that specific client.

3 Design and Implementation

3.1 ABD

In part 1 of this project we implemented the ABD algorithm that has been presented in *Lecture* 18. As the algorithm itself is, the server nodes only maintain the state of each key. That is, each server has the value, time-stamp, and the client-id for each key. Without too much modification of Lab 2, we extended the persistent value field by appending the tag to it. Each entry in the database is of form *(key value time-stamp client-id)*. A key assumption here is that the *value* has no spaces in it.

3.1.1 On server side

On server side, if server receive *GET* request, it simply returns the entry (if that exists). Upon receiving *PUT/INSERT* request, server checks if the key exists, if so, it then compares the time-stamp. If the new time-stamp is larger than the key's time-stamp it updates both the time-stamps and the value, otherwise it ignores the request.

3.1.2 On Client Side

- *GET* **requests:**
  For *GET* request we need to implement the read part of ABD protocol. The *abd_read* function first uses the *abd_read_query* function which launches #server many threads to get the largest tag [(time-stamp, client-id)] for the given key. The main thread of the request waits for a majority of these threads using a semaphore (viz. (#servers/2)+1). Once the main thread is woken up, it finds the largest tag and returns the value for that.
- *PUT/INSERT* **requests:**
  For *PUT/INSERT* requests, similar to the implementation for *GET* requests, we first launch #servers many threads to get the largest tag for the given key. A new tag is generated by incrementing the largest received tag by 1. This tag is now associated with the key and value for that client and is broadcasted to all the nodes. Once each thread receives an ACK from server node, it signals the main caller thread and the caller threads finishes.

3.2 Blocking

In part 2 of this project, we place most of the work for maintaining consistency and linearizability in the distributed database on the server nodes. In this system, we assume clients only communicate with a single server, and all servers are connected to one another forming a complete graph. With this, there are two main components that support our system: client-server listening threads and server-server peer listening threads. The exact details on the implementation are as follows:

3.2.1 Client-Server threads

Just as before, each server maintains a single listening thread that awaits client connections. When client connections are received, they are handled by separate threads to keep the server responsive. Upon receiving a client request, the following occurs:

1. As before, the helper thread parses the request and ensures it is formatted correctly. If formatted correctly, it proceeds to serve the client request accordingly.
2. In the case of a GET request, we attempt to serve the client from the cache, then c0, then c1 as before. Since the algorithm invariant ensures that the local database always has the most up to date data, there is no need to communicate between nodes for a GET. In the case of a PUT/INSERT/DELETE, since these operations will cause a *write* to our database, they must first acquire a lock. In this setting, they must acquire the distributed lock across the entire system, implemented with a Lamport mutex, with a call to `distributed_lock()`. We decided to implement a system-wide instead instead of a per-key lock, although that is possible with a bit more overhead. The details on

maintaining the lock are described below in the next section. Assuming the helper thread has acquired the lock, they execute the write on the system utilizing the same functions as provided in p2.

3. After the write has successfully executed (at least placed in the c0 tree), the helper thread must *broadcast* the write in the form of a SERVER_WRITE message to the rest of the server peers in the system in order to synchronize their version of the database. A node will keep attempting to distribute the write to other nodes since there is no rollback of the local write at this point.

4. Once the broadcast has finished, the helper thread issues a `distributed_unlock()` to release the lock, which can subsequently be acquired by another helper thread on the same server or another thread on another peer in the system, depending on the contents of the *lock_queue*.

Once the helper thread completes these steps, they can begin listening for more requests on the socket, and allow other client requests to be served.

3.2.2 Server-Server threads

In part 2, we refer to our servers as *peers*. We assume clients connect to a single server, and all servers are connected forming a complete graph. The server maintains another thread that opens and listens on a different port than client connections. This port is meant for peers to connect to, in which the main peer listener thread will subsequently spawn a new thread to listen for data on each peer socket. After initial connection, peers may send four types of messages to eachother:

1. **REQUEST_LOCK.** This message type will be issued by the client-server helper threads on invocation of `distributed_lock()`. The server will send out the message to all of its peer sockets indicating a request for the lock, in order to be allowed to issue writes to the database.

2. **RELEASE_LOCK.** This message type will also be issues by the client-server helper threads on invocation of `distributed_unlock()`. The server will send out the message to all of its peer sockets indicating that the server (really, a helper thread on the server) is finished its write required to serve the associated client request.

3. **REPLY_LOCK.** This message type will be issued by the *peer* listener/helper threads on receipt of a REQUEST_LOCK message. These messages are required by the thread requesting the lock, in order to confirm its current right of ownership of the lock. The peer helper threads will conditionally issue these requests on receipt of replied to any previously issued REQUEST_LOCK message by itself. If it receives a request and has not received a REPLY_LOCK for its own previous request, it will set a flag that will notify the peer thread later on to issue the REPLY_LOCK once it actually receives a REPLY_LOCK for its own lock request.

4. **SERVER_WRITE.** This message type is used by the client-server helper threads in order to synchronize the databases across all of the server peers in the system. After completion of a PUT/INSERT/DELETE, the thread will *broadcast* the write to all peers. Then the peers will validate that the server is at the head of the lock_queue (i.e. has the lock), and issue the write to its own database.

It is important to note that there are two distinct components that participate in the distributed protocol. The helper threads that handle client requests may require help from the peer helper threads in issuing lock requests, as well as broadcasting writes to the database. The helper threads that listen for messages from peers must maintain the distributed lock properly by managing a lock_queue, as well as ensuring that it is synchronized with the rest of the system by serving broadcasted writes to its own database.

3.3 Caveats

As explained previously, the failure or latency of a single node in the peer network causes the entire system to slow down/fail. In rare cases messages are lost or take a long time to reach and may cause one of the peers to keep blocking for responses. In such cases, the progress of the entire network is halted until the necessary message arrives.

4 Evaluation

In this section, we explain the evaluation we did for both ABD and the blocking algorithm.

4.1 Experimental Setup

Infrastructure. We used AWS virtual machines for our experiments. The VMs we used are all *t2.micro* that each has 1 vCPUs and 1 GB of memory and running Ubuntu Server 16.04 LTS.

Workload. We use two types of workloads where GET/PUT ratio varies and is either 0.1 or 0.9. The arrival rate is following a uniform distribution with the average 50 requests per seconds with a concurrency level of 25. The Key-Value pair size also is 1KB. The popularity distribution for keys follows a Zipf distribution with 1.345675 which results in having a 90:10 popularity fraction. That is, 90 percent of the workload is targeting 10 percent of keys.

Methodology. For each experiment, we have a set of 1000 unique keys which their popularity is described above. As the workloads we use consist of GET and PUT requests, we first INSERT all the keys into the database. This is two-folds, first, it fills the cache and hence brings the system into a steady state, and it prevents from getting errors and misses upon PUT and GET requests respectively. Since 90 percent of the whole workload is targeting 10 percent of keys (100 keys), and the cache hit rate should be 90%, we set the cache size to be 100 resulting a 90% hit ratio.

4.2 ABD Evaluation

In this section, we present the results for MT architecture for 1KB sized keys, and different GET/PUT ratio including 0.1 and 0.9.

4.2.1 Number of server nodes N: 1

For $N = 1$, the results are as follows:

Table 1: Response times for ABD protocol, N=1

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	2000	6.43	6.89	6.9	7.2
0.9	9000	5.23	5.09	6.2	6.35

The request response time is as follow:

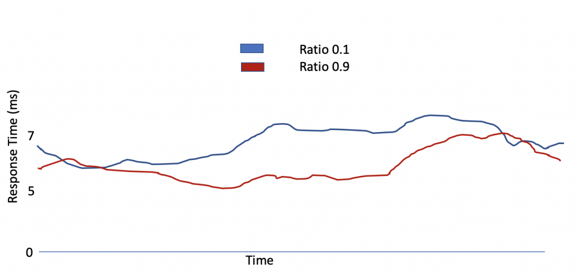

Figure 1: Response time for different GET/PUT ratios and N=1

4.2.2 Number of server nodes N: 3

For $N = 1$, the results are as follows:

Table 2: Response times for ABD protocol, N=3

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1800	7.3	7.29	7.39	7.5
0.9	8350	6.17	6.31	6.23	6.37

The request response time is as follow:

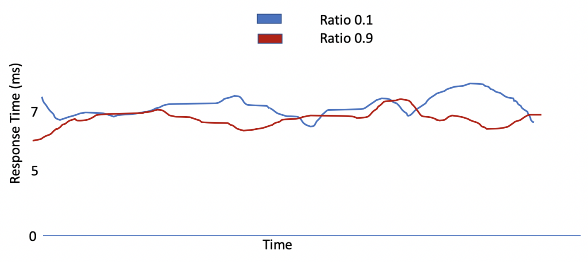

Figure 2: Response time for different GET/PUT ratios and N=3

4.2.3 Number of server nodes N: 5

For $N = 1$, the results are as follows:

Table 3: Response times for ABD protocol, N=5

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1820	7.22	7.05	7	7.15
0.9	8210	6.67	6.64	6.53	6.71

The request response time is as follow:

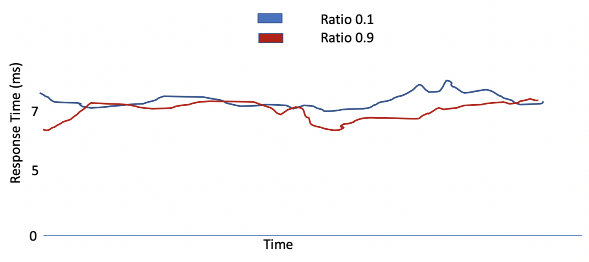

Figure 3: Response time for different GET/PUT ratios and N=5

4.3 Blocking Algorithm Evaluation

4.3.1 Number of server nodes N: 1

For $N = 1$, the results are as follows:

Table 4: Response times for blocking protocol, N=1

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	2100	6.13	6.39	6.74	7.12
0.9	6740	6.37	6.43	6.7	6.98

The request response time is as follow:

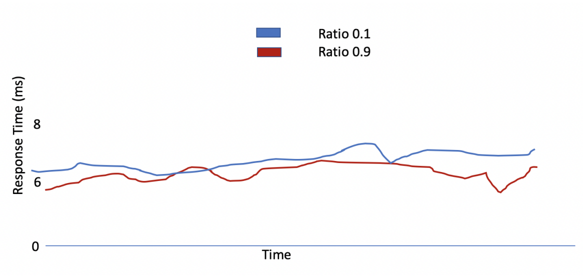

Figure 4: Response time for different GET/PUT ratios and N=1

4.3.2 Number of server nodes N: 3

For $N = 3$, the results are as follows:

Table 5: Response times for blocking protocol, N=3

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1600	8.34	8.45	8.65	9.03
0.9	6405	7.37	7.51	7.68	8.1

The request response time is as follow:

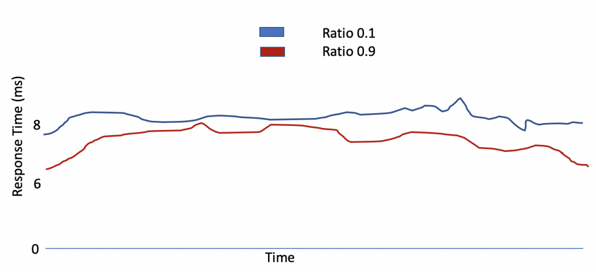

Figure 5: Response time for different GET/PUT ratios and N=3

4.3.3 Number of server nodes N: 5

For $N = 5$, the results are as follows:

Table 6: Response times for blocking protocol, N=5

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1320	9.53	9.64	9.88	10.1
0.9	5920	7.27	7.52	7.77	8.05

The request response time is as follow:

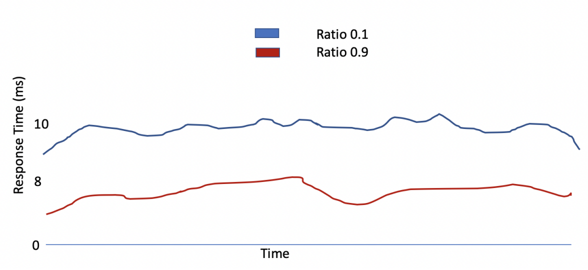

Figure 6: Response time for different GET/PUT ratios and N=5

4.4 Impact of crashing and/or slow clients

We also evaluate the performance of our system in the presence of crashing or slow clients. There are two settings under which we consider our evaluation of the performance and impact. They are as follows.

4.4.1 Crashing lock holder

In the case of ABD, clients do not maintain locks and clients do not know about other clients in the system. Clients only connect with servers, and so a crashing client will have no performance impact on on the rest of the client in the system. The servers will simply observe the closed socket and clean up the connection and continue.

In the case of the blocking protocol, crashing clients may have a significant impact on the performance of the system. In fact, since servers maintain a distributed lock that a single client connection may have control of at any given point in time, if the client crashes then there is no way that the servers will be able to "realize" the release of the lock from the client, since they cannot receive a RELEASE_LOCK message from him. In this scenario, the system will be in deadlock. However, a simple solution is to put a fair timeout on lock ownership (or a timeout from the last write from a client owning the lock), which will allow the system to recover in real-time in the event of a client crash. The impact on the performance of the system will be directly proportional to the timeout then. In the case of a client *not* currently owning the lock crashing, the system would not have any performance impact, as the helper thread polling the socket will recognize the socket close and cleanup the connection.

4.4.2 Slow clients

In the case of ABD, slow clients do pose a negative performance impact on the system. The reason is because most of the processing for ABD is done on the clients, and although clients are not connected to eachother, the consistency of each servers database is dependent on efficient clients. Each query issued to server is either a read or write query, which they are accustomed to. Long waits between issuing of subsequent read/write requests in order to establish consistency in the system will create time epochs where the system should have certain values for keys, had the messages been delivered in a timely manner. In the case that messages from clients are delivered to some servers, and others are left waiting for a significant portion of time before receiving the message (if at all), there will be inconsistency in the system. Although this problem arises, it does not destroy the integrity of the system, since clients will issue requests to all servers, and as long as a single server has the most recent value version for specific keys, then correctness exists. This may pose issues for crashing servers, however, and with large differences in database versions between servers, possibly make recovery to the correct state of the system after a crash impossible.

In the case of the blocking protocol, slow clients do pose a significant threat. Since most of the processing is done server client, and all we need is a p2 capable client, a slow client will certainly have a negative impact on the end user it is serving. More importantly, however, a slow client that has ownership of the global distributed lock, will force other client helper threads, local and on peers, to wait unfair amounts of time. This will cause the system to spend most of its time managing the locking and unlocking of the distributed lock, since servers must communicate this information in order to establish the right to issue a write to the database. Again, this issue can be resolved with a fair timeout on lock ownership, and the servers would have to be modified to support the possibility of incompleted client requests as a result of lock ownership revocation, and be able to finish serving the requests once they are able to acquire the lock again at a later time.

5 Conclusion

In this project we make our project in lab 2 linearizable using the non-blocking ABD protocol. Towards that, the server side has been changed slightly to keep the $(time-stamp\ client-id)$ per each key. Most of the work is in the client for ABD protocol where it applies its two-phase read/write. We also implemented a blocking protocol where clients hold a lock to perform the writes. Experimental evaluation has been done and the result show that where the number of nodes grows, non-blocking ABD outperforms the blocking protocol.

References

[1] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 363–375, New York, NY, USA, 1990. ACM. 2