# Multi-Paxos Distributed KV-Store

Evan Wireman, wireman@wisc.edu
Aanandita Dhawan, dhawan6@wisc.edu
Safi Nassar, nassar2@wisc.edu
CS, UW–Madison, WI

## ABSTRACT

In this project, we implemented a strongly-consistent replicated database using Multi-Paxos consensus [4, 5, 7] We tweaked some of the functionality to simplify our code base. We provide two key-value store operations (GET and SET), which have been designed to provide strong consistency and high availability. Our design can handle up to $(n/2) - 1$ failing replicas, where $n$ is the original size of the cluster. A failed node is capable of rejoining the cluster, repopulating its key-value store to match the leader's. In this paper, we will discuss our key design choices as well as highlighting some of our implementation's results.

## 1 DESIGN

Right from the beginning, we strove to develop modular code and easily reproducible setup. The entire project code, tests, documentation, results and releases are available on Github [1]. Details describing the intended roles of the datastructures and each class can be found in the repo's `README.md` file. *Unfortunately, as the deadline approached, we violated separation of concerns and wrote instead long functions that don't stick to the modularity guidelines originally set.*

In our implementation, there are two RPC services, ConsensusRPC and DatabaseRPC, running at all times on separate threads. They run on separate ports, and serve unique purposes. The DatabaseRPC service handles communication to and from the client. Its actions are guided by the ConsensusRPC service. A software architecture diagram with the RPC call flow is presented in *Figure 1*.

When a client submits a request to GET a value, the Database thread of the replica who received the request (replica 'a') will first check to see if it is the leader. If so, it will respond to the client with the value they requested. Otherwise, the request will be forwarded to the leader's Database thread. This thread will also check, then realize it is the leader, and reply to replica 'a' with the value found in its key-value store, which will be forwarded to the client. If replica 'a' does not get a response from the leader, it will trigger an election under the assumption that the leader is down. Upon reaching consensus with a new leader, replica 'a' will request the value from the new leader (which may be replica 'a' itself), consequently forwarding the response to the client.

When a client submits a SET request to a replica (replica 'a'), it will first check if it is the leader. If not, it will forward the request to the leader. Once the leader receives the request, it will attempt to reach consensus with the replicas. The only situation in which this could fail is if there are not enough live nodes for a quorum, since we are only allowing the leader to act as a proposer, thus eliminating concurrent proposals. Once consensus is achieved, the leader informs all replicas the newly accepted value.

Consensus is achieved through a slightly modified version of standard Multi-Paxos [2]. The proposal, acceptance, and inform stages remain unchanged from standard Paxos. However, we add a ping stage before the proposal. This is because we decided to remove the periodic heartbeat messages. Instead, the ping stage will essentially send heartbeats to all replicas, counting the number of live nodes to assess if we have a quorum. The number of nodes that are accounted for during the ping stage are the only nodes that will receive propose and accept requests. This prevents nodes who are recovering from failure from rejoining the cluster in the middle of consensus-seeking.

Because nodes cannot rejoin the cluster in the midst of seeking consensus, we forego forwarding the Paxos log to recovering nodes. Instead, the leader sends recovering nodes a copy of its key-value store. This gives the recovering node the most recent snapshot of the state of the database. The inform stage of our implementation

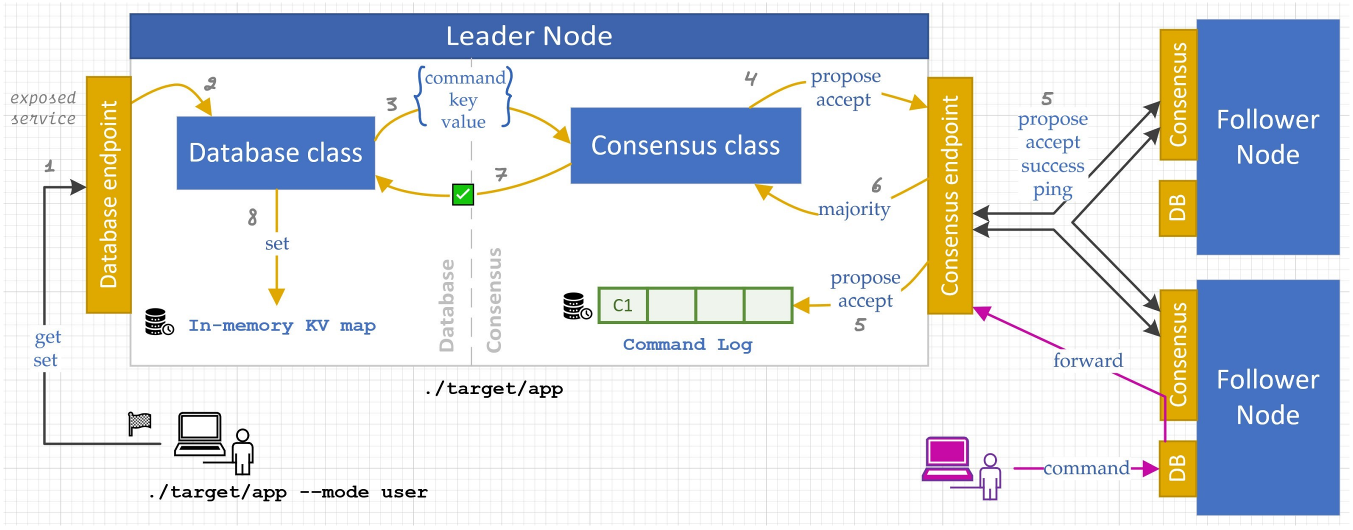


**Figure 1: Software Architecture** — a flow chart of the program modules and state datastructures involved during the consensus process.

sends the newly accepted key-value pair to all nodes in the cluster, such that, if a node recovers in the middle of consensus, it is informed of the new agreed upon value.

Building our implementation produces a single binary. Command line arguments are used to specify if the executable should run as a node in the cluster or as a user communicating with the cluster. If a replica is being started up, it will reach out to all nodes in the cluster, attempting to see if there is an accepted leader. If not, it will trigger an election, nominating itself to be the leader. If the cluster already has an agreed upon leader, the node that is just starting up will pull the leader's key-value store to ensure replication.

## 2 RESULTS

We have evaluated the implementation on clusters of different sizes, ranging from 5 to 50 nodes. Besides achieving consistency, our evaluation has revealed other noteworthy results, starting from point number 2 onwards.

### 2.1 Consistency

Having thoroughly tested the system's consistency [3, 6], which was the primary objective of this project. We have verified that the system maintains perfect consistency even when subjected to a script executing 1000 randomly generated SET operations *Figure 3*, we have also ran it locally on a cluster of 50 nodes and reached got consistent results (when comparing the DB snapshots of each participant node). As previously mentioned, the system maintains consistency even in the face of node failures and subsequent recovery. This is achieved through success RPC calls from the designated leader, as described in the original multipaxos algorithm.

The following tests were ran on the un-optimized version of our binary — compiled with `-O0 & -g` flags, in addition to debugging statements on *(due to limited time, we wanted to verify the output of our system while running the benchmarks).*

### 2.2 RPC Calls Count

*Figures 2 and 3* present a detailed count of incoming and outgoing RPC calls in a cluster of 3 nodes, revealing interesting observations:

(1) The SET operation requests are distributed equally among replicas 1 and 2, which is a desirable characteristic.
(2) Outgoing propose/accept calls are solely made by the designated leader, aligning with the system's design.
(3) Our analysis of the system's behavior during the initial election process reveals that the first replica to initiate propose/accept calls is elected as the leader, in accordance with the expected protocol.

Additionally, no RPC calls are observed after the first replica's calls.

### 2.3 Time to reach consensus for different cluster sizes

In this test we send `set` RPC calls, repeated for $10^4$ iterations, with the same key-value pair to the leader on different cluster sizes. Then, we repeated the test targeting a random follower node instead. For measuring the time in CPU cycles  regular time we used *google/benchmark* tool. Hardware used: `Intel Core i7-6700 CPUx64@4GHz, 2756 MHz` under `WSL2 environment`.

- Real Time := average wall-clock time that each iteration took to complete.
- CPU := average CPU time per iteration.

*Figure 4 and 5* display the results. It seems like *google/benchmark* doesn't accurately capture real time per operation (something we need to investigate), but the comparison with different cluster sizes is still valid.

Regarding the drop in CPU time for 15 nodes settings, it seems like we should repeat that part of the experiment. *Check the test under* `-NoValue-`;

### 2.4 Additional tests

In addition, to the above metrics we have set to test, but could not complete on time, the average election time for different cluster sizes and also the average recovery time in different settings.

## 3 TAKEAWAYS

This endeavor was both exciting and challenging. Although time-consuming, it was worthwhile. Doing it in C++ added some difficulties, but we also learned a lot. Utilizing wrapper classes, multi-threading, and C++ terms like unique pointer, shared pointer, move, etc. were all beneficial learning experiences. Implementing a complex algorithm like Paxos, creating boilerplate code for a large project, and assigning team members to different parts of the project proved to be fruitful challenges.

We found that, although our implementation correctly achieves consensus, there are some efficiency improvements that could be made. For instance, since our implementation only allows for one leader, the ping and propose phases are not necessarily needed to achieve consensus for elements in the database. The only situation in which concurrent proposals could be made is during leader election, in which case the aforementioned stages would be necessary.

## REFERENCES

[1] 2023. Distributed Replicated Database using Paxos. https://github.com/szn-cs/cs739-p2-replicated-database. (2023).

[2] Leslie Epstein. 2014. Paxos Lecture. YouTube video. (2014). https://www.youtube.com/watch?v=JEpsBg0AO6o

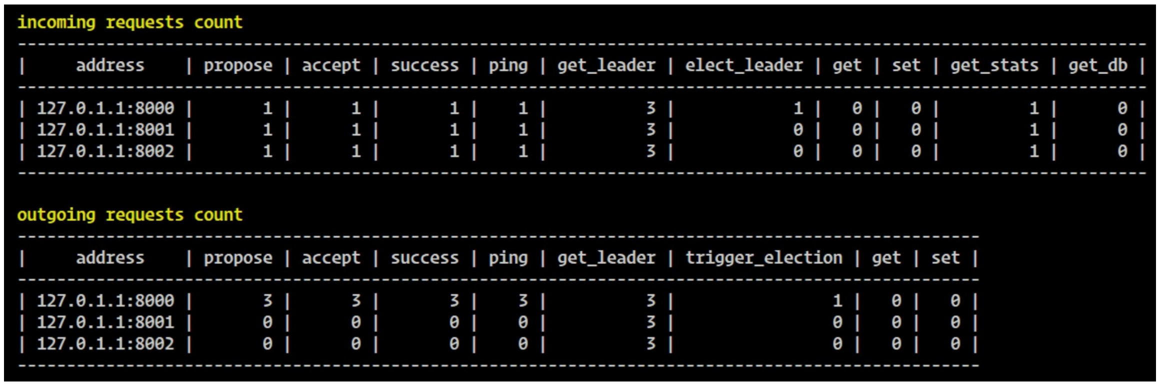
[3] Damian Gryski. 2023. Awesome Consensus. GitHub repository. (2023). https://github.com/dgryski/awesome-consensus

[4] Leslie Lamport. 2001. Paxos Made Simple.

[5] Leslie Lamport. 2002. Paxos Made Simple, Fast, and Byzantine. In *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002 (Studia Informatica Universalis)*, Alain Bui and Hacène Fouchal (Eds.), Vol. 3. Suger, Saint-Denis, rue Catulienne, France, 7–9.

[6] Andrey Satarin. 2023. List of tests for distributed system. Website. (2023). https://asatarin.github.io/testing-distributed-systems/

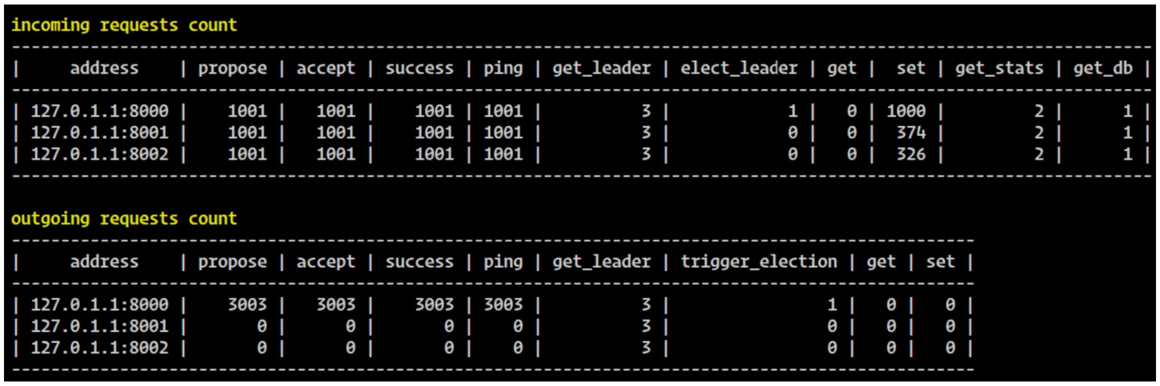
[7] Robbert Van Renesse. 2014. Paxos Made Moderately Complex. Webpage. (2014). https://harmony.cs.cornell.edu/docs/textbook/paxos/



```
incoming requests count
-----------------------------------------------------------------------------------------------------------------------------
|     address     | propose | accept | success | ping | get_leader | elect_leader | get | set | get_stats | get_db |
-----------------------------------------------------------------------------------------------------------------------------
| 127.0.1.1:8000 |       1 |      1 |       1 |    1 |          3 |            1 |   0 |   0 |         1 |      0 |
| 127.0.1.1:8001 |       1 |      1 |       1 |    1 |          3 |            0 |   0 |   0 |         1 |      0 |
| 127.0.1.1:8002 |       1 |      1 |       1 |    1 |          3 |            0 |   0 |   0 |         1 |      0 |
-----------------------------------------------------------------------------------------------------------------------------

outgoing requests count
-----------------------------------------------------------------------------------------------------------
|     address     | propose | accept | success | ping | get_leader | trigger_election | get | set |
-----------------------------------------------------------------------------------------------------------
| 127.0.1.1:8000 |       3 |      3 |       3 |    3 |          3 |                1 |   0 |   0 |
| 127.0.1.1:8001 |       0 |      0 |       0 |    0 |          3 |                0 |   0 |   0 |
| 127.0.1.1:8002 |       0 |      0 |       0 |    0 |          3 |                0 |   0 |   0 |
-----------------------------------------------------------------------------------------------------------
```

**Figure 2:** RPC calls count after an election in cluster of 3



```
incoming requests count
-----------------------------------------------------------------------------------------------------------------------------
|     address     | propose | accept | success | ping | get_leader | elect_leader | get |  set | get_stats | get_db |
-----------------------------------------------------------------------------------------------------------------------------
| 127.0.1.1:8000 |    1001 |   1001 |    1001 | 1001 |          3 |            1 |   0 | 1000 |         2 |      1 |
| 127.0.1.1:8001 |    1001 |   1001 |    1001 | 1001 |          3 |            0 |   0 |  374 |         2 |      1 |
| 127.0.1.1:8002 |    1001 |   1001 |    1001 | 1001 |          3 |            0 |   0 |  326 |         2 |      1 |
-----------------------------------------------------------------------------------------------------------------------------

outgoing requests count
-----------------------------------------------------------------------------------------------------------
|     address     | propose | accept | success | ping | get_leader | trigger_election | get | set |
-----------------------------------------------------------------------------------------------------------
| 127.0.1.1:8000 |    3003 |   3003 |    3003 | 3003 |          3 |                1 |   0 |   0 |
| 127.0.1.1:8001 |       0 |      0 |       0 |    0 |          3 |                0 |   0 |   0 |
| 127.0.1.1:8002 |       0 |      0 |       0 |    0 |          3 |                0 |   0 |   0 |
-----------------------------------------------------------------------------------------------------------
```

**Figure 3:** RPC calls count after testing-1000-random-ops in cluster of 3

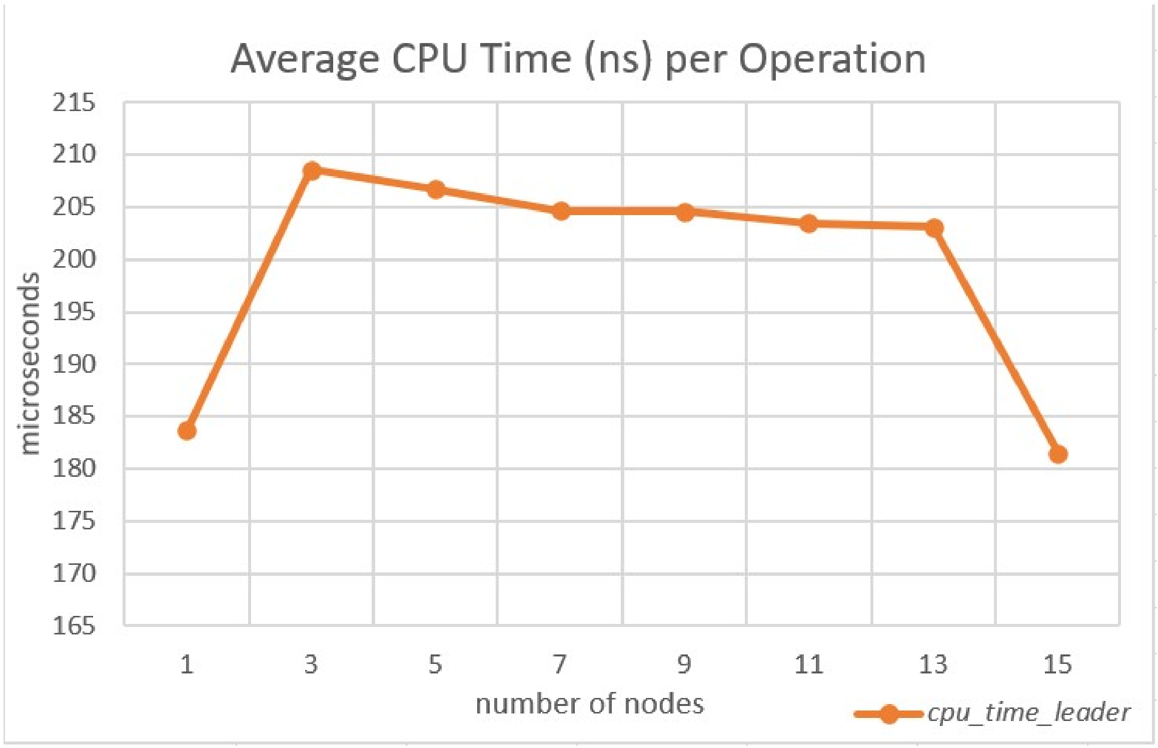


**Figure 4:** Average CPU Time per Operation

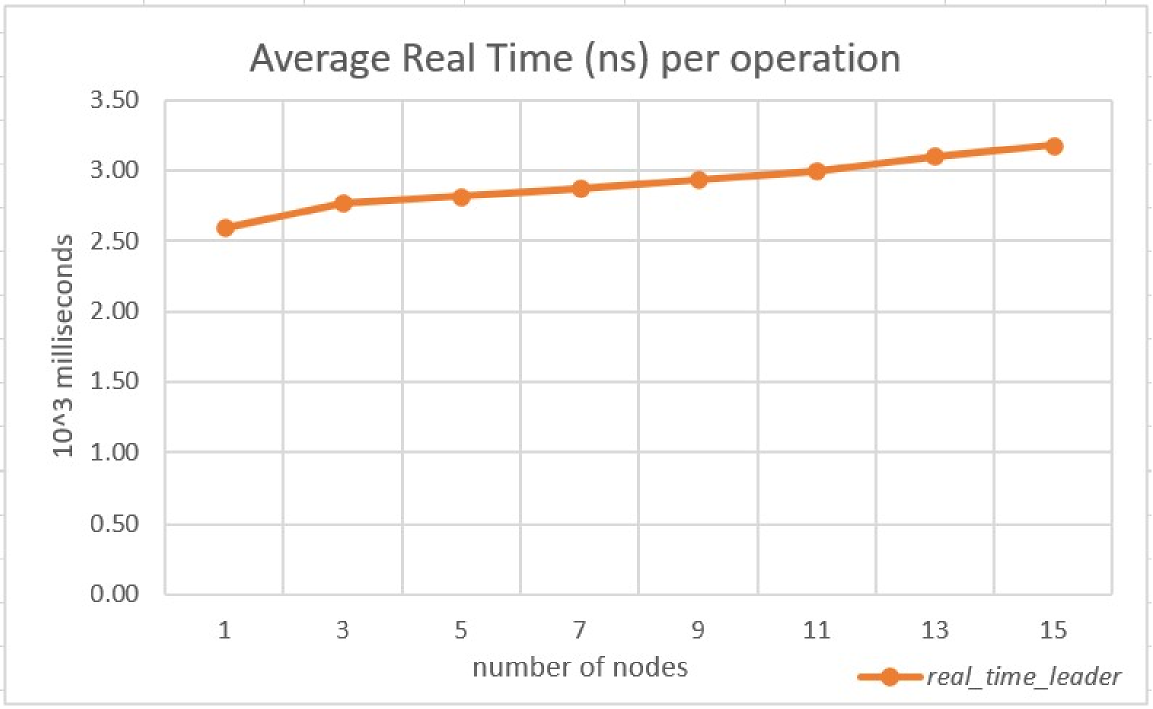


**Figure 5:** Average Real Time per Operation