Milestone 4: Distributed Features Utilizing Write-Ahead-Log

Team Info

- Xuan Chen (1002344732)
- Tian Tian (1002556138)

Design Overview

The team implemented a Write-Ahead-Log Mechanism to track all relevant client requests. Each client request is logged with its process ID (distinguished by the client port number), request information (command, key, value) and timestamp. This mechanism helps the servers monitor previously received requests and process these requests more efficiently. For example, the log files can be used to rollback to a previous state if the current requests fail. In addition, the log files help the server to avoid redundant work by checking the previously logged client request.

With the implemented WAL, the team is able to design a more competent Fault Tolerance system and a simple Weak Consistency method. The idea is to communicate the latest requests between the relevant servers upon a failure point or a synchronization point, and utilize the request information to reach an agreement across the entire network.

Client Auto Retry Mechanism

In our M4 design, we treat each PUT writes including the two REPLICATE writes as a single transaction. Therefore, the client will have a higher chance of receiving a PUT_ERROR response due to the aggregated chances of faults due to three nodes. Therefore, the client will automatically initiate a retry of the PUT request with a maximum of five tries. This will work together with our WAL design to serve for fault tolerance.

Moreover, to serve for the WAL implementations, each client message will come together with a message id(i.e. a client's timestamp). Thus, each client message can be uniquely identified by all the server nodes together with the client process's ID. Note that a retry will hold the same message id as the initial attempt.

Uses of Write-Ahead-Log

When a PUT request comes to a primary or a PUT_REPLICATE comes to a replica, it will be logged first with a unique local sequence number(i.e. assigned by the server), together with the message id and client ID, before it actually updates in the database. Note that the lsn will only be updated in an incremental manner. Once the write has been finished in the node's database, the node will update its *lastWrittenLsn*.

Fault Tolerance #1: Rollback Mechanism for Replica Failures

In our design, the PUT and its corresponding replication writes is treated as one single transaction, managed by the primary node. Therefore, if any of the three writes(i.e., one PUT and two PUT_REPLICATE) fail, the primary server will identify this exception error and rollbacks to its previous value for this key. All the replicas will also receive the previous value as another PUT_REPLICATE message so that the change is consistent across the database. As a result, the client will receive a PUT_ERROR for this request. Therefore, this implementation will tolerate a crash of replicas during one PUT transaction.

Fault Tolerance #2: Primary Crash after Replication has been logged on the database

If the primary crashes after it has sent the replication messages to its replicas, the client will initiate a retry for this request. Therefore it will direct this retry request either to one of the replicas. When the two replicas receive this request, it will check its own WAL so that it can find out it has already been logged and written to disk. Therefore, the replica will not make a duplicate write.

Fault Tolerance #3: Replica Change right before a PUT request

If one replica fails just before a PUT request, the new replica will remain in WRITE_LOCKED state as it is performing data migration. Therefore, the newly coming PUT request will be forced to the replica by the primary. The primary node will resend the PUT_REPLICATE once it detects a hash tree update from the zookeeper. The primary node identifies which replica to resend the message by looking at its WAL to compare the lsn synchronized with each replica with its own largest value of lsn.

Fault Tolerance #4: Redundant Requests

If the user sends a redundant PUT request to the server, the primary node will check this request against the latest log in its WAL to avoid unnecessary work. Since our design has additional fault tolerance mechanisms to ensure that the logged requests in the WAL has successfully completed, this check can safely bypass a request if it is deemed redundant.

Weak Consistency

Since our WAL contains requests from all clients for all servers. A simple Weak Consistency can be implemented utilizing these information. Upon a synchronization point (client-input), the ECS will pool the latest requests from each server using the ZooKeeper. These requests will then be replayed in an orderly fashion to establish a convergence. The team decides to replay all requests from each server to ensure each server has the chance to replay its request as a primary node.

Performance Evaluation

M4 performance is evaluated using the following metrics:

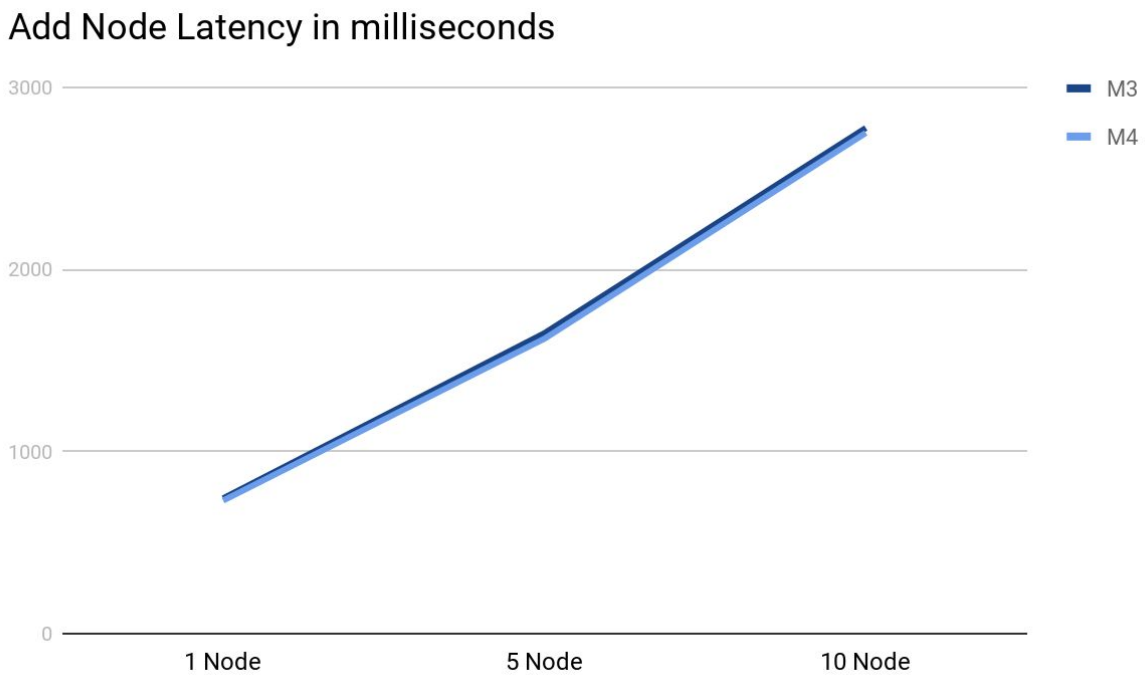
- The latency of add/remove of 1, 5, and 10 nodes/servers
- The latency of Redundant Put/Get using a custom data set

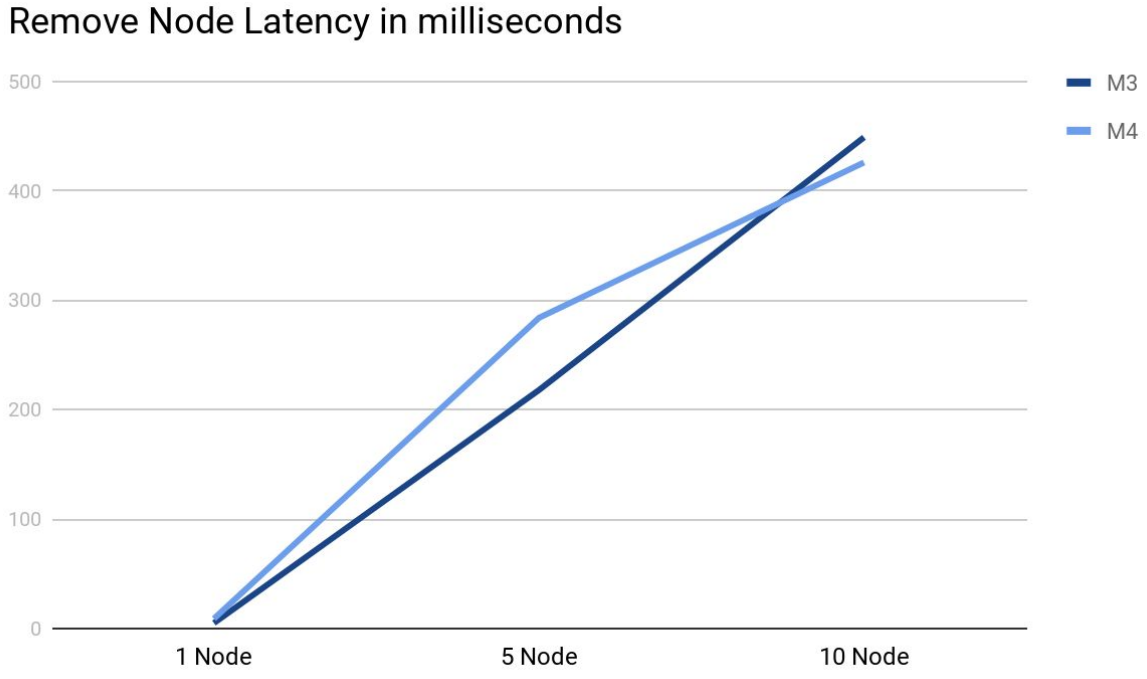
As seen in the appendix section below, performance for this milestone primarily focuses on redundant PUT/GET requests and is improved by 25% compared to M3. This is expected as the newly implemented WAL mechanism helps the program to process redundant requests more efficiently, and thus achieving higher throughput. Aside from this main difference, the two milestones share a similar trend in its performances for adding and removing nodes. This is also expected as the main framework (ECS, ZooKeeper, Hash Ring) of our design is unchanged since M3.

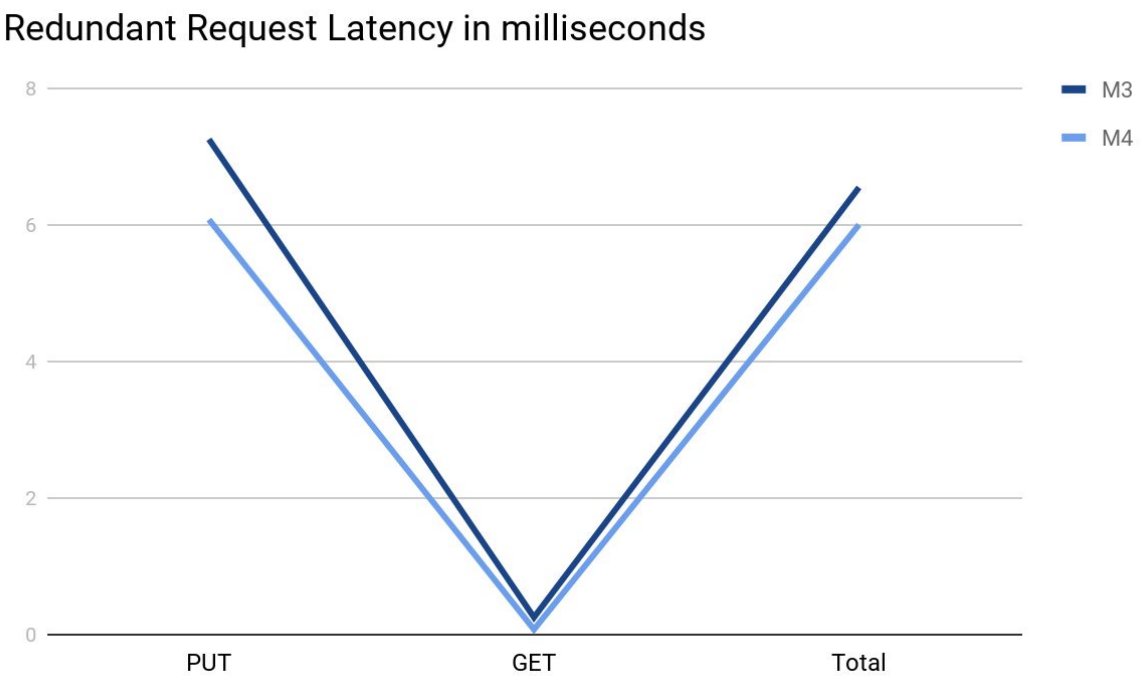
Video Link

Section	**Link**
Client Auto Retry Mechanism Uses of Write-Ahead-Log Fault Tolerance #1 Fault Tolerance #2 Fault Tolerance #3	https://drive.google.com/file/d/1LL6QJL38TAKPtXVQNuQ7Tm1Ol6152aEu/view?usp=sharing https://drive.google.com/file/d/1CiV-hLBCMWJgX-HHS12wlsImOy1PjniU/view?usp=sharing
Fault Tolerance #4 Weak Consistency	https://drive.google.com/file/d/1HKh1oBszoIQ8AvQrKJ8BZjWokctaD0X5/view?usp=sharing
Performance	https://drive.google.com/file/d/19m2lQWGzXKr83NxnvyunYjMz4Ym_DFGT/view?usp=sharing

Appendix A: M2 & M3 Performance Comparison






Redundant Request Latency in milliseconds
8
6
4
2
0
PUT
GET
Total
M3
M4

Appendix B: Milestone 4 Performance Test

Test Name	**test01RedundancyPerformanceTest**
Status	Pass
Description	Test Key-Value pair replication across replicas

Appendix C: Regression Tests

M3

Test Name	**test01Replication**
Status	Pass
Description	Test Key-Value pair replication across replicas

Test Name	**test02ReplicationAfterAddNode**
Status	Pass
Description	Test Key-Value pair replication when adding new nodes

Test Name	**test03ReplicationAfterRemoveNode**
Status	Pass
Description	Test if predecessors become new replicas if the original replicas are removed

Test Name	**test04NodeFailureDetection**
Status	Pass
Description	Test if server crashes are identified by the ECS

Test Name	**test05NodeFailureRecovery**
Status	Pass
Description	Test crashed server can be recovered after a failure detection

Test Name	**test06ReplicaGet**
Status	Pass

Description	Test if replicas can successfully process GET command

Test Name	**test07ReplicaPut**
Status	Pass
Description	Test if replicas can correctly handle PUT command

Test Name	**test08HashRingAddingNode**
Status	Pass
Description	Test if the hash ring can identify the correct replicas when nodes are added

Test Name	**test09HashRingRemovingNode**
Status	Pass
Description	Test if the hash ring can identify the correct replicas when nodes are removed

Test Name	**test10Consistency**
Status	Pass
Description	Test if data changes such as PUT/PUT_UPDATE/DELETE that happened during a server crash are reflected in the recovered server side.

M2

Test Name	**test_createECS**
Status	Pass
Description	Test the creation of ECS

Test Name	**test_addNode**
Status	Pass
Description	Test adding nods to ECS

Test Name	**test_startNode**
Status	Failed
Description	Test if the previously added nodes can be started.

Test Name	**test_removeNode**
Status	Pass
Description	Test if all the nodes can be removed from ECS

Test Name	**test_removeNonExistNode**
Status	Pass
Description	Test if removing a non-existed node is handled

Test Name	**test_stop**
Status	Pass
Description	Testing if ECS can be stoped

Test Name	**test_shutdown**
Status	Pass
Description	Test if ECS can be shutdown

Test Name	**test_connection**
Status	Pass
Description	Test if clients can connect to ECS started servers

Test Name	**test_putGetData**
Status	Pass

Description	Test if the client can put and get data from ECS stared servers

Test Name	**test_addNodes**
Status	Pass
Description	Test if a node can be added to hash ring

Test Name	**test_getNode**
Status	Pass
Description	Test if the hash ring can return a node by name and hash

Test Name	**test_removeNode**
Status	Pass
Description	Test if a node on the hash ring can be deleted

M1

Test Name	**test_Set_Value**
Status	Pass
Description	Test set KEY VALUE pair function

Test Name	**test_Get_Value**
Status	Pass
Description	Test get VALUE with a given KEY function

Test Name	**test_Update_Value**
Status	Pass
Description	Test update VALUE with a given KEY function

Test Name	**test_Get_non_exist**
Status	Pass
Description	Test if the server responds correctly with a getting a non-exist KEY

Test Name	**test_Put_LongStr**
Status	Pass
Description	Test if the server responds correctly with putting a very long string

Test Name	**test_Delete_Pair**
Status	Pass
Description	Test delete KEY VALUE pair with a given KEY

Test Name	**test_Delete_non_exist**
Status	Pass
Description	Test if the server responds correctly when deleting a non-exist KEY

Test Name	**test_FIFO_func**
Status	Pass
Description	Test if the FIFO cache strategy is working

Test Name	**test_LRU_func**
Status	Pass
Description	Test if the LFU cache strategy is working

Test Name	**test_cache_perf**
Status	Pass
Description	Test the throughput and latency of the server with various cache size and number of PUT/GET requests

Test Name	**test_Connection**
Status	Pass
Description	Test if the client can make a connection with the server