# Milestone 3 Report

## Distributed Systems, ECE419

March 22 2020

Jackson Rigby (1002423763)
Sofia Tijanic (1002202243)
Michael Vu (1002473272)

# System Level Overview

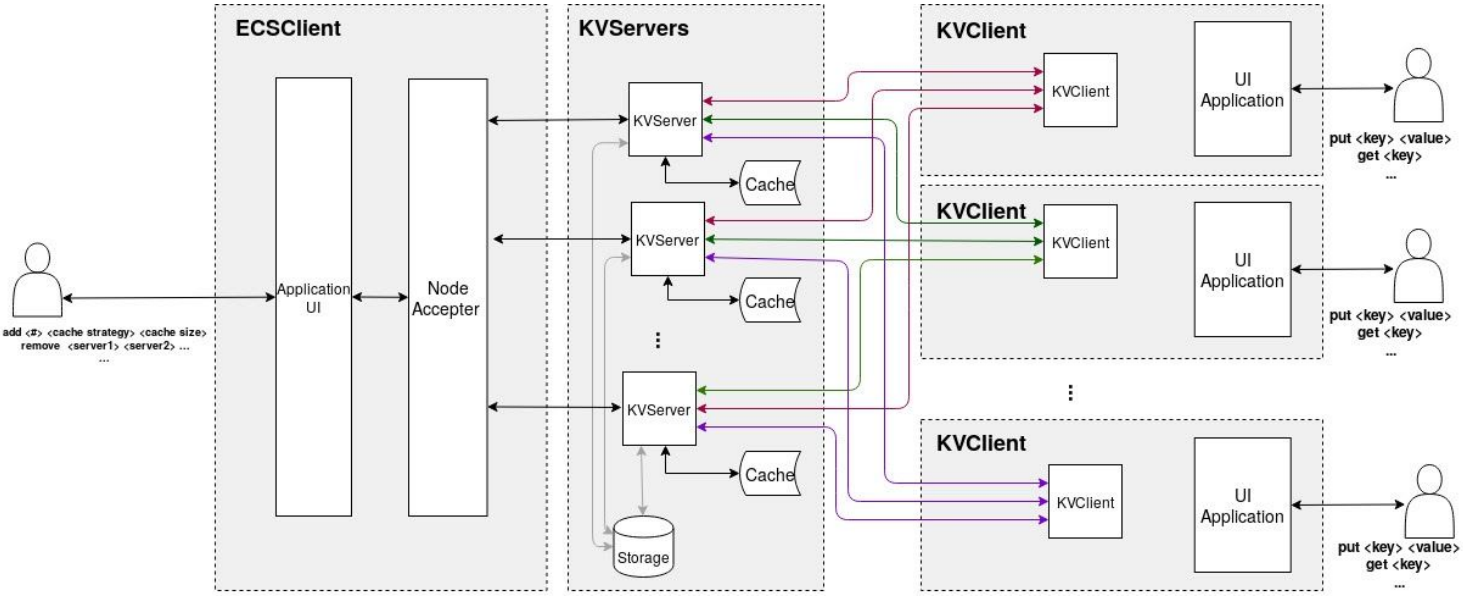


*Figure 1: System Design*

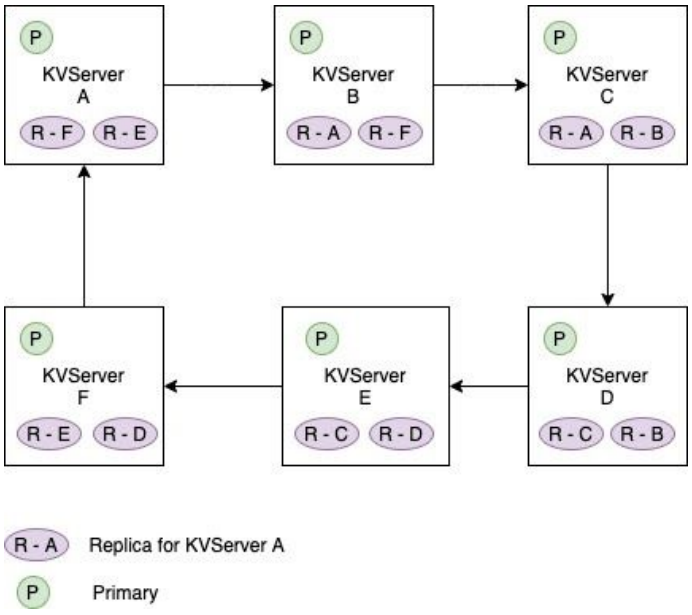


*Figure 2: Server Replication Design*

# Design Overview

## Replication Mechanism

The replication mechanism implemented in this lab is an example of *active* Replication, where a primary server communicates with the client, and then passes changes to its replicas internally. Each KVServer acts as a primary server for its own metadata set, as well as a replica for the 2 preceding servers in the hash ring. The replication is maintained upon server removal, and new servers being added, by re-broadcasting new metadata and transferring relevant data to new replicas. In the case when only 1 server is present, it acts as only its own primary. In the case with only 2 servers, they each act as a primary for themselves and a replica for each other. With 3 or more servers, the full replication mechanism of 2 replicas per server can be implemented. Correct replication is maintained by the primary server for a given key value pair. When a change in server configuration is detected, a server will give a new replica server all of the data it is the primary for. If a change in configuration means that data is no longer supposed to be stored as a replica by a certain server, that server will detect this and purge the data from its storage.

### Get Request

Get requests are served by any of the servers holding the requested data, meaning the primary or any of its replicas.

### Put Request

On put requests, the primary server stores the new value, passes the value to its replicas, and then returns an acknowledgement to the client. This is *eager* replication.

## Data Transfers

### Server Remove

When a server is removed, the hash range needs to be recalculated for all remaining servers. The ECS is responsible for this. Once the new metadata exists, the ECS evokes a data transfer between the server being removed, and the server that now encompasses the removed server's hash range.

### Server Add

When a new server is added, the ECS re-calculates the metadata for all servers. It then evokes a data transfer between the server previously responsible for the new server's data and the new server. This transfer also read/write locks the transfering server. Once the transfer is complete, the new metadata is broadcast to all servers.

### Server Kill

When a server is killed, the ECS re-calculates metadata for all servers. The killed server's data does not get re-distributed to other servers because it is killed, not removed. The ECS will do any data transfers necessary between live servers, and then broadcast the new metadata.

## Failure Detection and Recovery among Servers

We used Zookeeper for server failure detection. Zookeeper keeps track of *znodes*, which are kvServers in our system. It uses a heart-beat mechanism to ping the servers on a consistent basis, and can detect that a server has died when it does not respond within a timeout window. Once Zookeeper has detected a server failure, *watchers* are used from ECS to receive updates and respond accordingly. In the case of a server failing, ECS must update the metadata, complete any data transfers based on new metadata, and broadcast the new metadata to all live servers. When a server fails, its data is lost, however due to replicas, this data will persist in the system, and the client will be able to retrieve any key-value pairs stored through the failed server previously.

## Storage Server Graceful Failure

Graceful failure means that the client is not affected by server failures. This is the case with our system, as the client does not interact with any one server directly, it simply sends requests to the storage system, and ECS handles the request and which kvServer to invoke. In the same way, when a server fails, ECS handles all of the data and metadata changes, such that the client is unaffected, and able to access all of the same data. When a server does fail, the ECS ensures that replicas receive all of their primary's data as well.

## Data Persistency on Shut Down

Throughout the lifetime of an instance of the server-client system, data is persistent on-disk, meaning that it is stored in multiple files, and persists through server failures, removals and additions. However, when the entire system is shut down, the system resolves to storing all data into one file, so that it is available upon system start-up. This implementation is such that as servers fail or are removed, their data is constantly being transferred to live servers until there is only 1 left. This one primary server holds all previous servers' data on disk, and it remains there

on system shut-down. When the system is started up again, the client has access to all of the data that was previously available. The old, persistent data and any new data, behaves in the same way, meaning that it is transferred on server additions and removals, and is copied to replicas. In this way, data always persists with our system.

# Notable Design Decisions

## Eager Replication

Eager replication was chosen as the method for updating replicas when the primary makes changes. This was done as a way of enabling the ECS to have full control over the replicas, and how and when they receive their updates. ECS uses an internal server-to-server data transfer method to serve individual puts that need to be updated between servers. This means that when a primary receives new data, ECS ensures that the replicas receive and accept the new data as well. Once the client receives a success response from the primary server, all replicas are up to date as well.
We recognized that this was a trade off with response time - updating all replicas each time an update to primary is made is more time costly than eventual consistency would be. For this project and system, we decided to use this implementation in order to fulfil ECF's role as a "management" hub for all the servers. Our alternative design would have been to respond to client requests from the primary only, and to invoke data transfer updates between servers on a regular timed interval, or when a certain number of requests are made.

## Data Persistency

In M2, we had data persistency within an instance of ECS using virtual memory; we have now implemented on-disk persistency both during an instance of ECS and between shut-down and startup. We implemented on-disk persistency by creating unique files for each live server, and transferring data between these files as a part of replication and shut down. On shut down, all of the data is transferred to one file, which is saved and persists beyond server shut down. When the system is shut down, and the last node is killed, a file is saved with the name of the last living server, so that on startup, the first server knows where to retrieve data from. This was a big improvement in our system during this milestone, as supporting data persistency is a big demand in storage servers, and something that we prioritised in this milestone.

# Performance Report

*Table 2: Performance Report*

| cache_type | n_server_nodes | n_clients | throughput_p_s |
|---|---|---|---|
| FIFO | 1 | 1 | 391.79 |
| FIFO | 1 | 10 | 544.02 |
| FIFO | 1 | 100 | 520.41 |
| FIFO | 2 | 1 | 178.56 |
| FIFO | 2 | 10 | 257.89 |
| FIFO | 2 | 100 | 239.85 |
| FIFO | 10 | 1 | 149.41 |
| FIFO | 10 | 10 | 158.50 |
| FIFO | 10 | 100 | 145.32 |
| LRU | 1 | 1 | 579.72 |
| LRU | 1 | 10 | 605.02 |
| LRU | 1 | 100 | 516.24 |
| LRU | 2 | 1 | 243.96 |
| LRU | 2 | 10 | 252.94 |
| LRU | 2 | 100 | 238.65 |
| LRU | 10 | 1 | 146.88 |
| LRU | 10 | 10 | 167.12 |
| LRU | 10 | 100 | 165.05 |

### Notes

All 100 cache size. Nodes are distributed across eecg lab machines. All clients are running on the same machine in the eecg lab.

### Observations

When the number of server nodes is 1, LRU performs better than FIFO with more clients, and FIFO performs better than LRU with fewer clients.. If the number of server nodes is not 1, LRU and FIFO perform relatively similarly.

When the number of server nodes increases, the throughput of the system decreases. This is because we do eager replication, which incurs overhead and slows down the responses to each client request.

# Appendix A - Test Cases

| Test Name | M3IntegrationTesting |
|---|---|
| Objective | Test client-server system as a whole, with a focus on new functionality. Replicates real run-time functionality, and tests specific features of the system individually. |
| Description | **Server Functionality Tests:**<br><br>*testBasic*: Create an ECS server, add 3 kv servers, performs simple put and get requests. Asserts that basic functionality, behaviour, response types and data persistency works.<br><br>*testRemovalTransferBasic*: Creates 4 zk servers, adds 10 entries, and confirms that the entries still exist after removing a server.<br><br>*testAddingTransferBasic*: Creates 4 zk servers, adds 10 entries, and confirms that the entries still exist after adding a 5th server.<br><br>*testSendToKilledServerA*: Creates 4 zk servers, kills 3rd server. Performs 10 put and get requests, asserts correct behaviour among remaining live servers.<br><br>testSendToKilledServerB: Creates 4 zk servers, kills 3rd and 4th server. Performs 10 put and get requests, asserts correct behaviour among remaining live servers.<br><br>testAddingAndRemovingTransferBasic: Combination of previous add and remove server tests. Creates 4 zk servers, adds 10 entries, confirms that the entries still exist after adding a 5th server. Removes 2 servers, confirms that entries still exist after removal.<br><br>**Replication Tests:**<br><br>*testReplicationWithThreeServers*: Replication Case: 3 servers. Creates 3 servers. Tests that all 3 servers contain the same information, as they are all replicas of the primary, and should hold the same data. Asserts that all the primary, replica 1 and replica 2 can serve get requests for the primary's data, for all primaries and replicas.<br><br>*testReplicationWithFiveServers*: Replication Case: 3+ servers. Creates 5 servers. Tests that all servers have 2 replicas that contain the correct information. Asserts that get requests can be fulfilled by any primary or its |

| | |
|---|---|
| | replica.<br><br>*testReplicationWithTwoServers*: Replication Case: 2 servers. Creates 2 servers. Tests that all each server has 1 replica, for a total of 2 primarys and two replica 1s. No second replicas in this case.<br><br>testReplicationWithOneServer: Replication Case: 1 server. Creates 1 server. Tests that server functions properly with put and get requests. There are no replicas in this case. |
| Result | All tests pass. |

| | |
|---|---|
| Test Name | KVAdminMessageTest |
| Objective | KVAdminMessage is a protocol used for communication between KVServer and ECS. |
| Description | KVAdminMessageTest is comprised of the following tests:<br><br>testConstructorValid: Creates a new KVAdminMessage; asserts that the message status type and payload is correctly initialized on creation. Serializes and then deserializes the KVAdminMessage and asserts that it remains consistent through (de)serialization.<br>Repeats the above test with a null payload to ensure that functionality remains consistent. |
| Result | All tests pass. |

| | |
|---|---|
| Test Name | MetaDataTest |
| Objective | Test the MetaData calculation and distribution of hashes based on server name and host names. |
| Description | MetaDataTest is comprised of the following independent tests:<br><br>*testHashValue*: Creates hash values from strings, and asserts that they are created correctly. Serializes and deserializes hash values, and asserts that they are still the correct values.<br><br>*testHashRange*: Creates new hash ranges from hash values. Asserts that the ranges are created correctly, and that hash values within and outside of the hash range are correctly identified. Serializes and deserializes hash values, and asserts that they are still the correct values and range. |

| | |
|---|---|
| | *testMetaData*: Creates a new metadata entry from a server name, host, port, and hash range. Asserts that the metadata entry contains the correct information. Serializes and deserializes the metadata, and asserts that it is still correct.<br><br>testMetaDataSet: Creates a metadata set from a collection of ServerInfo objects. Asserts that the set contains the correct hash for each server by asserting that the server name associated with each hash matches that stored in the metadata set. |
| Result | All tests pass. |

| | |
|---|---|
| Test Name | CacheTest |
| Objective | Test the Cache class to ensure that the basic functions of put, get, remove and clear work correctly for all cache replacement policies. |
| Description | CacheTest is comprised of the following independent tests:<br><br>*testPutGet:* Inserts 6 key-value pairs into a 6 element cache and checks each key to ensure that all values are correctly stored<br><br>*testDelete:* Inserts 3 key-value pairs into a 3 element cache and deletes each one individually, checking to ensure correctness between deletes.<br><br>*testClear:* Inserts 3 key-value pairs into a 3 element cache, calls the clear function, and checks to make sure the cache no longer contains any inserted values<br><br>*testFIFO:* Inserts a series of 200 key-value pairs into a 10 element cache using a FIFO replacement policy and checks to make sure the cache contains the correct values each time.<br><br>*testLRU:* Inserts 3 key-value pairs into a 3 element cache using an LRU replacement policy and a series of PUTs and GETs are called. Correctness of the cache is checked throughout.<br><br>*testLFU:* Inserts 3 key-value pairs into a 3 element cache using an LFU replacement policy and a series of PUTs and GETs are called. Correctness of the cache is checked throughout. |
| Result | All tests pass. |

| Test Name | KVMessageTest |
|---|---|
| Objective | Ensure that KVMessage objects are valid. Check that constructors behave correctly, and the errors are caught for invalid arguments. |
| Description | *testConstructorValid:* Creates KVMessage objects, and ensures that the object's getters return the correct values.<br>*testConstructorExceptions:* Creates KVMessage objects with invalid arguments, and ensures that exceptions are thrown. |
| Result | All tests pass. |

| Test Name | SerializationTest |
|---|---|
| Objective | Ensure that our custom serialization/deserialization protocol behaves correctly. Tests consistent behaviour, and handles errors correctly.. |
| Description | *testBasic:* Uses serializer to encode an int, string, int, and byte into a byte array. Uses deserializer on the byte array to recover the int, string, int, and byte, and checks that the values are unchanged. Once there is no more information left in the byte array, attempts to read another byte, and makes sure that a *DeserializationException* is thrown. |
| Result | All tests pass, Serializer and Deserializer are stable. |

| Test Name | ServerStoreSmartTest |
|---|---|
| Objective | Test ServerStore class, which is responsible for communicating between KVServer and disk and cache. Test that correct cache/disk logic is implemented, and that all response types are consistent and correct. |
| Description | ServerStoreSmartTest is comprised of the following independent tests:<br><br>*testBasic*: get non-existant value, put several new values, replace existing values, delete existing values, get existing values. Assert that responses are all correct for each operation.<br><br>testConsistency: put new key-value pairs into disk and cache. Update values (on disk and cache). check that updates are consistent on disk and cache. Delete 1/3 of the values, and check that deletes are consistent on server and cache.<br><br>testPersistency:  Put new key-value pairs onto disk & cache. Delete cache, disk and serverStore object, and then create new ones. Check that all |

| | |
|---|---|
| | original key-value pairs are still on disk/cache. |
| Result | All tests pass. |