Graham Hoyes, Danial Hasan, Daniil Orekhov

ECE419 M3 Design Document

ECS Changes

To improve the reliability of sending admin messages, the process has been slightly modified. Each admin message is given a UUID. Previously, admin messages were sent on `/servers/<serverID>/admin`, and responses written to `/servers/<serverID>`. Now, admin responses are written to `/servers/<serverID>/<Message UUID>`. This helps eliminate race conditions when asynchronous admin messages arrive due to node failure.

The global `HashRing` metadata object and `ServerNode`s have been improved to maintain an actual circular doubly-linked list of nodes, to make determining predecessors and successors easier. However, the linked list pointers cannot be serialized when sending the hash ring in admin messages. We thus also store the nodes in an ordered list, which can be serialized with node pointers excluded (via the `transient` field keyword). On deserialization, the circular doubly-linked list is reconstructed.

Replication Procedure

Controllers

As per the milestone specifications, replication of a hash range is handled by the controller node responsible for said hash range. Assuming a controller is already running, for `PUT` requests of any kind (including `UPDATE` and `DELETE`) the controller will perform the regular `PUT` operation, in addition to appending the action to a write log file. After the `PUT` operation has been performed and logged to the write log, the controller node will update all of its replicators.

As this replication scheme must comply with eventual consistency, the replication updates from controllers can be performed asynchronously. To this end, a new `KVDataSender` thread is created for sending the write log to each replicator for asynchronous updates. `KVDataSender` will acquire the read lock for the write log file, and first writes a line to the output stream with the controller server's name to identify the incoming write log to the replicator. Subsequently, the `KVDataSender` streams the write log file contents to the replicator and upon a barrier being reached by all sender threads, the write log file is deleted. The write log itself consists of the key-value pair encoded as JSON, along with an optional prefix of `P` (put), `U` (update), `D` (delete) which denotes the type of `PUT` action to perform on the replicated data.

In cases where the controller must move data to another server due to an ECS `MOVE_DATA` action, the data that is sent to another server (and hence no longer resides on the controller) will be appended to the write log with the delete prefix and subsequently sent to all replicators using separate sender threads. When the controller receives data from another server as per the `RECEIVE_DATA` action, the controller will receive the data and store it into a temporary file. This temporary file is first merged into the controller's own storage file and then subsequently sent to the replicators for them to merge into their replicated data.

Replicators

Replicators are passive and are notified by their respective controllers to update their replicated versions of the controller's data. To achieve this, on initialization every server requests an open port on its machine which is used by a passive `KVDataListener` thread to listen for receipt of replication data. This port is included in the server metadata that is sent to the ECS and propagated to all servers. When a controller connects to the port, a new `KVDataReceiver` instance is created and forked into a new thread to receive the incoming replication updates via a write log file. The thread will read the first line of the incoming socket stream to identify which server is asking it to replicate data, then stores the remaining data from input stream into a temporary file. This temporary file is then passed to the `replicateData` method in `KVSimpleStore`.

When receiving a write log file, the `KVSimpleStore` will first check if the server that has sent the replicated data is already a controller of this replicator. If the controller is new, a new file is created to store the replicated data and the incoming data is copied into it. Otherwise, the controller already has replicated data on this replicator and the incoming write log is parsed to update the replicated data. Write actions with: a) `PUT` prefixes are appended to the replicated file, b) `UPDATE` prefixes search the file for the relevant key and update its value, c) `DELETE` prefixes search the file for the relevant key and delete the key-value pair, and d) no prefix denote a batch of new data that is appended to the replicated file (in cases of data that has been moved to an existing controller).

Adding Servers

If a server node is added to the system, a metadata update is triggered to all existing nodes in the service:

- **Controllers**: check to see if they have a new replicator and if so, initialize that replicator with their current data.
- **Replicators**: check to see if they have a new controller and if so, delete the data from their old controllers to avoid future conflicts.

The new server itself will receive an updated version of the hashring and uses a synchronized object to wait until it has received data from its successor before initializing its replicators.

Removing Servers

If a server node is removed from the system, a metadata update is triggered to all existing nodes in the service:

- **Controllers**: check to see if they have a new replicator and if so, initialize that replicator with their current data.
- **Replicators**: check to see if they have a new controller or if a current controller has been removed (in cases of fewer than 3 nodes) and if so, delete the data from their old controller to avoid future conflicts.

Dying Servers

Detection of failing servers is explained in detail below, here we assume server has died and the ECS has detected it and broadcasted a metadata update to all servers:

- **Controllers**: check to see if they have a new replicator and if so, initialize that replicator with their current data.
- **Replicators**: check to see if they have a new controller or if a current controller has been removed (in cases of fewer than 3 nodes) and if so, delete the data from their old controller to avoid future conflicts. **Additionally**: the replicators will check to see if the server that died was their immediate predecessor (and controller), in which case that replicator will merge the replicated data for the failed server into their own storage file and assume responsibility for the failed server's hash range and data.

Failure Detection and Handling

Failure detection is handled by ZooKeeper, while restoring data is handled by the KVServer which takes over the hash range of the failed node. When a server is created, it creates an ephemeral ZNode at `/heartbeats/<serverID>`. Prior to the ECS launching the KVServer, a watcher is set on this ZNode to detect when the server has been brought online. After server initialization is complete, another watcher, `HeartbeatDeathWatcher`, is set on the heartbeat path to watch for a `NodeDeleted` event. When the KVServer disconnects from ZooKeeper for any reason (intentional shutdown or node / network) failure, its heartbeat ZNode is deleted and the watcher is triggered.

To prevent triggering the heartbeat death watcher for intentional server shutdowns, a set of active nodes is maintained separate from the global `HashRing` metadata object. Whereas a node is only added to or removed from the global `HashRing` object (and other KVServers subsequently informed of its addition or deletion) upon completion of the startup or shutdown procedure, the active node set is updated immediately. The `HeartbeatDeathWatcher` first checks whether the node is in this set (as opposed to checking the `HashRing` object), and only proceeds if so. With this method, a node is removed from the active node set on intentional shutdown prior to the watcher being triggered, but before the `HashRing` update is complete. Thus when the node's heartbeat ZNode is deleted, the `HeartbeatDeathWatcher` will not treat the intentional shutdown as a node failure.

When actual node failure is detected by the watcher, it is removed from the `HashRing` object and a global metadata update made. Metadata updates now contain a node action that triggered the update, one of `ADDED`, `DELETED`, `DIED`, `STARTED`, or `STOPPED`, as well as which node changed to trigger the update. When receiving a metadata update with the `DIED` event, each node will determine if its immediate predecessor was the node that died. If so, that node (the successor) is now responsible for the hash range of the failed node, and it will merge the failed node's replicated data into the successor's main storage file, after which it will be able to serve the data without any loss (other than any keys not yet replicated due to eventual consistency).

Client Library

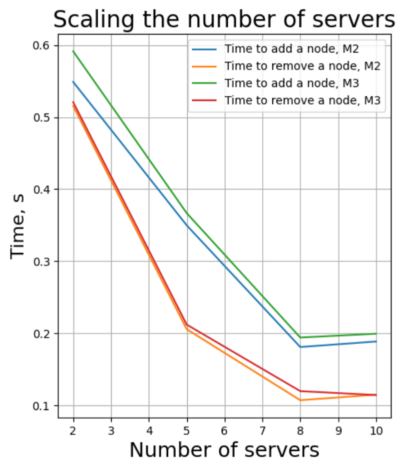
The client library has been extended to allow issuing read requests to replicators. As of milestone 2, the client library maintains a single open connection to a KVServer. At the time, if the server is not responsible for a key, the client will disconnect and connect to the correct server.

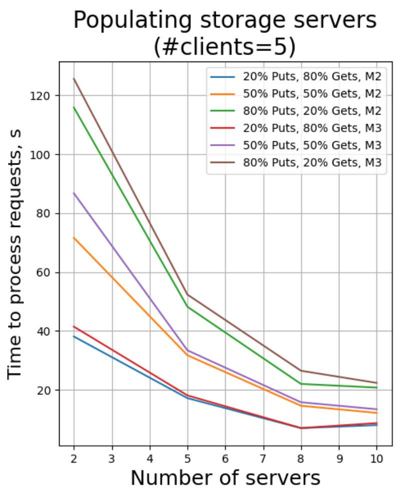
We continue to use only a single open connection. However, if the client is currently connected to a server which replicates the key associated with a request, read requests will be issued there rather than reconnecting to the controller. This results in a moderate performance improvement by not having to switch servers as frequently, but the effects are diminished as the number of nodes increases (and hence the likelihood of being connected to a replicator decreases).

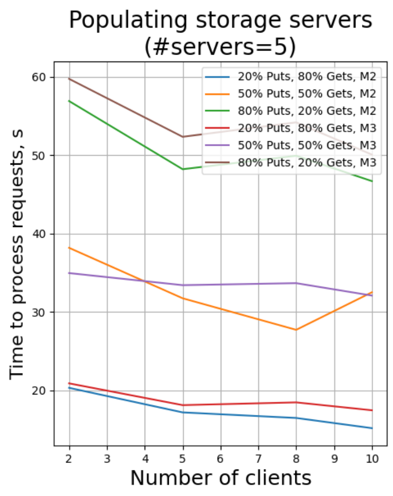
Note that due to eventual consistency, the client may retrieve a stale value (or possibly return a invalid key error) from a replica if the key has not yet been updated from the controller. Put requests must be issued to the controller, and hence the client will always reconnect to the controller if necessary for a put request.

As of milestone 2, the client library was resilient against KVServers being shut down by the ECS. This resiliency also extends to unexpected failures of KVServers, without any additional messaging required. If the connection to the current KVServer is terminated, the client will attempt to repeat the request on the next node in the hash ring (provided there are any left). The request will then be made to the first KVServer where a connection could be established. The KVServer will respond with an updated hash ring, which the client will store. If the first KVServer was not responsible for the key (as a coordinator for put requests, or coordinator/replicator for get requests), the request will be re-issued to the correct server in the updated hash ring.

Performance Evaluation







As with milestone 2, we first populate the servers with 500 entries to saturate the data storage, then perform 500 total GET and PUT operations on the service, varying the proportions of each. GET and PUT operations are interleaved, so as time progresses the database continues to saturate. No deletion is performed, so these numbers represent a worst case scenario for operations which are O(n). Once the database is populated, we also add and remove a node to test how the system scales up and down.

Overall, introducing replication leads to slightly degraded performance over M2. On the server side, replication is handled asynchronously after responding to the request, which has no impact on request latency. However, there are additional locks involved on the storage and replica files, which results in slightly higher latency.

On the client side, replication can be beneficial to performance since the client will not need to switch servers if it happens to be connected to a replica. However, the likelihood of this happening decreases with an increased number of servers - thus with our scheme, there is no performance gain on the client side from replication. Further, when the data stores and values are large, the time to read and return a value is far greater than the time taken to switch server connections, further diminishing the benefits. A possible improvement for get requests that cannot be served from the current server, is to pick a random server from the replicas and coordinators for the key to connect to, rather than always connecting to the coordinator. This would help distribute load on the servers and decrease latency, however once again the benefits of this are diminished as the number of nodes increases (and hence the fraction of nodes replicating a key decreases).

Appendix

Appendix A: Unit Tests

Unit tests were written in JUnit 4 style (the provided test cases were modified to conform to this when required).

Provided tests (unmodified, other than formatting):

- Connection tests
    - Valid connection
    - Unknown host
    - Invalid port
- KVStore Client Interaction tests
    - Successful PUT insert
    - Successful PUT update
    - PUT after a disconnect
    - Successful delete
    - Successful GET
    - GET with unset key

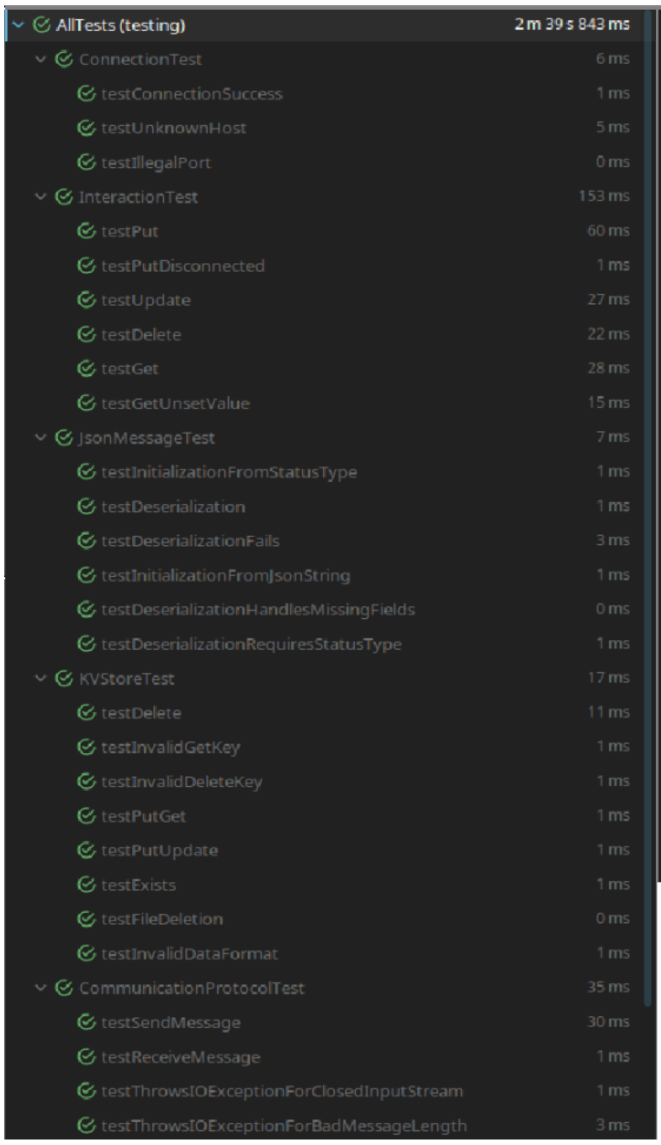
Additional tests (50 total, new M3 tests at the bottom):

- JSON Message tests: Test the initialization, serialization, and deserialization of JSON messages
    - Initialization with only a status type
    - Deserialization from valid json
    - Throws DeserializationException for deserialization from invalid json
    - Initialization from a json string
    - Sets missing optional fields to null
    - Raises DeserializationException for missing status type
- Communication protocol tests: Test the byte communication protocol used by the TCP connections (JSON serialization tested separately, above)
    - Test message sending: Test that protocol version, message length, header unused space, and message payload bytestream are properly constructed
    - Test message receiving: Test that a message can be properly received from a valid communication bytestream
    - Test that IOException is thrown when input connection is closed
    - Test that IOException is thrown when message payload is shorter than indicated in header tag
- KVStore tests: Testing the persistent storage class for permanent storage of key-values
    - Test invalid keys for put/get/delete: Test that the storage throws an exception for keys that are invalid i.e. the key-value pair does not exist
    - Test put/get: Test that a put request for a new key-value pair is correctly stored and that a subsequent get request returns the correct value
    - Test put update: Test that a put request correctly updates an existing key-value pair
    - Test delete: Test that key-value pairs are deleted as intended
    - Test storage file deletion: Test that a correct exception is thrown if the storage file is moved/deleted

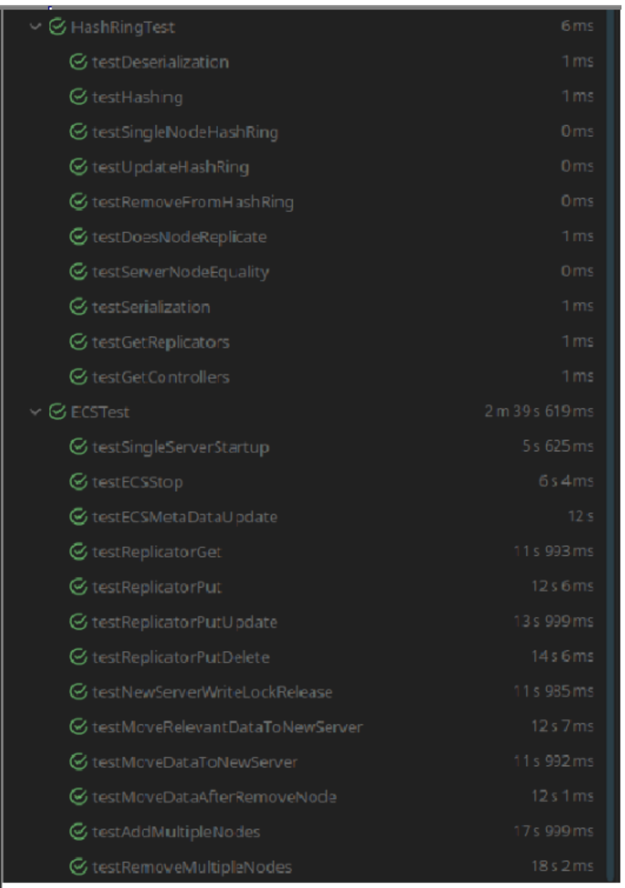
    - Test invalid data format: Test that an exception is thrown for invalid data formatting within the key-value persistent storage file
- Hash Ring tests: Testing actions on the global hash ring metadata object
    - Test single node hash ring: ensure that a single node is responsible for the entire hash ring
    - Test update hash ring: start hash ring with one node, ensure that adding another node splits the ring accordingly
    - Test remove from hash ring: start hash rings with multiple nodes, remove a node and ensure its successor adopts the removed node's range
    - Test hashing: ensure that nodes are responsible for keys in their range
    - **(new)** Test that the hash ring is aware of which nodes replicate with other nodes
    - **(new)** Test that server nodes can determine equality between each other
    - **(new)** Test that the hash ring can be serialized
    - **(new)** Test that the hash ring can be deserialized, and properly rebuilds the linked list
    - **(new)** Test getting replicators of a node
    - **(new)** Test getting controllers of a node
- ECS Tests: Test the ECS functionality of managing the hash ring, starting/stopping nodes, adding/removing nodes
    - Test single server startup: launch, setup, and start a single server and ensure that a client can connect to it
    - Test ECS stop server: ensure that ECS can stop a launched server, and that the server responds with a SERVER_STOPPED message to clients
    - Test metadata updates: ensure the ECS updates metadata for all servers when adding a new node
    - Test add node write lock release: test that newly added nodes to an already instantiated ECS are writable to after they have been started
    - Test moving of data: test that data in the respective hash range is moved to new nodes
    - Test data is moved after removing a nodes: ensure that data is moved to successor node in hash ring after a node is removed
    - Test adding multiple nodes: ensure that multiple nodes can be added and that data is moved accordingly
    - Test removal of multiple nodes: ensure that multiple nodes are removed and their data is moved to correct successors
    - **(new)** Test replicator get: Test that get requests can be served by a replicator
    - **(new)** Test replicator put: Test that a replicator eventually adds a new key-value pair put on its controller
    - **(new)** Test replicator put update: Test that a replicator eventually updates a key-value pair that was updated on its controller
    - **(new)** Test replicator delete: Test that a replicator eventually deleted a key-value pair that was deleted on its controller.
    - **(new)** Test move relevant data: Test that when moving data, only the data the new server is responsible for is sent

Graham Hoyes, Danial Hasan, Daniil Orekhov

Proof of tests passing:



```
AllTests (testing)                                2 m 39 s 843 ms
  ConnectionTest                                            6 ms
    testConnectionSuccess                                   1 ms
    testUnknownHost                                         5 ms
    testIllegalPort                                         0 ms
  InteractionTest                                         153 ms
    testPut                                                60 ms
    testPutDisconnected                                     1 ms
    testUpdate                                             27 ms
    testDelete                                             22 ms
    testGet                                                28 ms
    testGetUnsetValue                                      15 ms
  JsonMessageTest                                           7 ms
    testInitializationFromStatusType                        1 ms
    testDeserialization                                     1 ms
    testDeserializationFails                                3 ms
    testInitializationFromJsonString                        1 ms
    testDeserializationHandlesMissingFields                 0 ms
    testDeserializationRequiresStatusType                   1 ms
  KVStoreTest                                              17 ms
    testDelete                                             11 ms
    testInvalidGetKey                                       1 ms
    testInvalidDeleteKey                                    1 ms
    testPutGet                                              1 ms
    testPutUpdate                                           1 ms
    testExists                                              1 ms
    testFileDeletion                                        0 ms
    testInvalidDataFormat                                   1 ms
  CommunicationProtocolTest                                35 ms
    testSendMessage                                        30 ms
    testReceiveMessage                                      1 ms
    testThrowsIOExceptionForClosedInputStream               1 ms
    testThrowsIOExceptionForBadMessageLength                3 ms
```



```
  HashRingTest                                              6 ms
    testDeserialization                                     1 ms
    testHashing                                             1 ms
    testSingleNodeHashRing                                  0 ms
    testUpdateHashRing                                      0 ms
    testRemoveFromHashRing                                  0 ms
    testDoesNodeReplicate                                   1 ms
    testServerNodeEquality                                  0 ms
    testSerialization                                       1 ms
    testGetReplicators                                      1 ms
    testGetControllers                                      1 ms
  ECSTest                                         2 m 39 s 619 ms
    testSingleServerStartup                          5 s 625 ms
    testECSStop                                        6 s 4 ms
    testECSMetaDataUpdate                                  12 s
    testReplicatorGet                               11 s 993 ms
    testReplicatorPut                                 12 s 6 ms
    testReplicatorPutUpdate                         13 s 999 ms
    testReplicatorPutDelete                           14 s 6 ms
    testNewServerWriteLockRelease                   11 s 985 ms
    testMoveRelevantDataToNewServer                   12 s 7 ms
    testMoveDataToNewServer                         11 s 992 ms
    testMoveDataAfterRemoveNode                       12 s 1 ms
    testAddMultipleNodes                            17 s 999 ms
    testRemoveMultipleNodes                           18 s 2 ms
```