

ECE419 Project: Milestone 1 - Design Document

Patricia Marukot (1002157499), Jing Yi Li (1002346730), Seongju Yun (1002275290)

Major Design Decisions

1) Replication Strategy: Eager Replication

- In order to satisfy the requirement for *eventual consistency*, we decided to use the eager replication strategy which uses the following flows:

Client PUT request:

1. Client PUT
2. KVServer replicates PUT to replicas
3. KVServer executes PUT locally only once all replicas returns successful PUT
 - a. If not all replicas returned success, KVServer must rollback on the successful replicas, in this case, KVServer would not execute PUT locally and return PUT_ERROR back to client

Client GET request: (same behaviour as M2)

If KVServer is primary/coordinator of object:

- Return data

If KVServer is replica of object:

Return NOT_RESPONSIBLE along with the metadata to redirect client to primary

Implementation details: Changes made to *KVServer*

Event	Procedure
ECS_TO_SERVER: Broadcast metadata Sent by the ECSClient to indicate that replicas may need to be updated	Check current replicas against new hash ring configuration If any replicas are no longer viable: <ul style="list-style-type: none">• Send request to “old” replica to delete data stored from this server• Send request to “new” replica to transfer all data (same as MOVE_DATA)
SERVER_TO_SERVER: Receive replication (a list - like move data)	Issued from a Primary to its replicas with updated data As the replica, have a file dedicated to replication storage (a secondary disk). Update the replica disk upon receiving this request.
Client PUT	<ul style="list-style-type: none">• Send request to replicas to transfer all data from primary to replica• Respond to client SUCCESS <p>Both primary and replicas should keep track of client requests (UUID) to prevent re-execution.</p> <p>If any of the replication requests to replicas fail:</p> <ul style="list-style-type: none">• Respond to client FAIL• ECSClient will provide an updated hash ring in the case that any nodes crash through Broadcast metadata <p>If current KVServer crashed / unresponsive:</p> <ul style="list-style-type: none">• Client has timeout mechanism and will retry request which will fail if primary has crashed• Client should try to connect to another available KVServer and retry request

2) Failure Detection and Recovery (ECSCClient)

Detection:

1 round of heartbeats periodically (e.g. every 45s) → if no response to ECS within timeout (5s), then consider server failed - proceed with recovery

- Implemented in *HeartbeatMonitor.java* which is run as a background thread in ECSCClient when the application is started - it periodically sends HEART_BEAT requests to servers participating in the service (statuses: IDLE_START, START, STOP)

Recovery:

- If any KVServer nodes fail, a new node should be started to replace the failed node - the state of this new node should be the same as the failed node (e.g. if failed node was IDLE_START, new node should be IDLE_START),
- All KVServer nodes should know about the failure via ECSCClient, and be able to update their replications such that they all retain 2 replicas which are both immediate successors to itself and have full data replications of itself.
- Their hash rings should also be updated to reflect the failure.

Implementation Details

Procedure for recovery:

Step 1: For each failed node:

- Add a new node in place of failed node
 - What happens if no more new nodes? → return, nothing we can do
- Who gets the failed node's data, transfer data from replica to that node
 - Be careful: the amount of data to transfer depends on where the new node is inserted (e.g. replicated data might not even go to the new node - see cases)
- After each iteration (per each failed node), all primary data should be **recovered** and in their rightful places on the hash ring

Step 2: Do global re-replication (all servers)

Before handling cases, new node must be:

- Started (initServer)

Failure Recovery Cases:

Define S_i as server i . $H(S_i)$ as the hash of server i .

Then:

If S_i failed, with new node S_n to be added, then:

Case 1 → $H(S_{i-1}) < H(S_n) \leq H(S_i)$

Case 2 → $H(S_i) < H(S_n) \leq H(S_{i+1})$

Case 3 → elsewhere

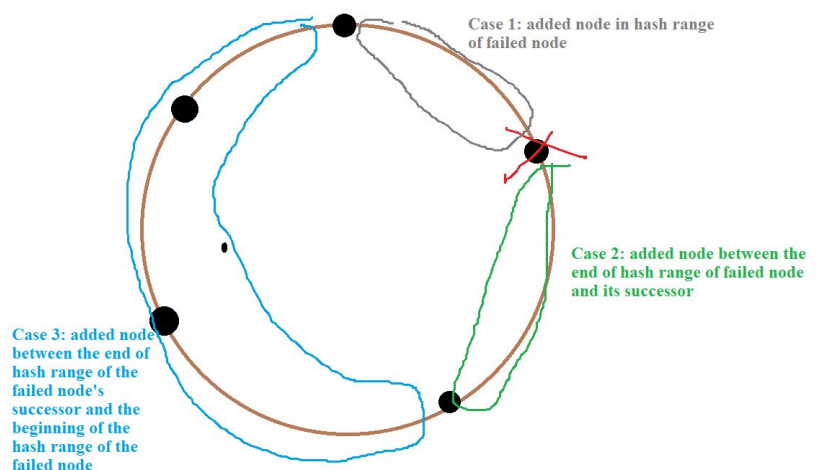
Define: Replica failure scheme:

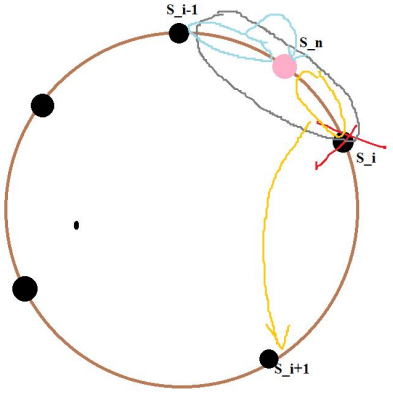
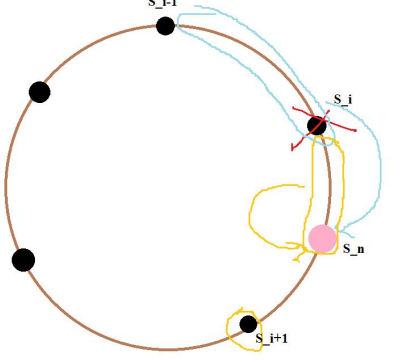
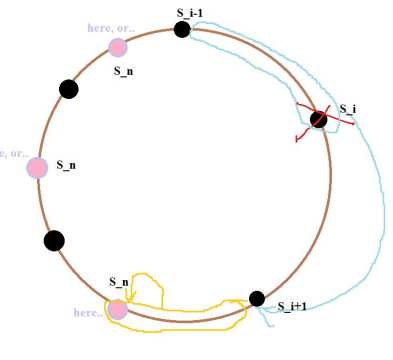
If one replica is unresponsive:

- Try the remaining replica

If both replicas are unresponsive:

- Fail



<p>Case 1</p> 	<p>From ANY replica of S_i, transfer:</p> <ul style="list-style-type: none"> • Data within range of $(H(S_{i-1}), H(S_n)] \rightarrow S_n$ <ul style="list-style-type: none"> ◦ Follow replica failure scheme • Data within range of $(H(S_n), H(S_i)] \rightarrow S_{i+1}$ <ul style="list-style-type: none"> ◦ Follow replica failure scheme ◦ If S_{i+1} is unresponsive, send to S_{i+2}, if S_{i+2} fails, send to S_{i+3}, and so on.. Until loops back to S_i (if this case, then fail) <p>Note: KVServer should be responsible for detecting of the movedata node is itself, in which case it would locally replicate (and not, say, send a message to itself)</p>
<p>Case 2</p> 	<p>From ANY replica of S_i, transfer:</p> <ul style="list-style-type: none"> • All data of S_i $(H(S_{i-1}), H(S_i)] \rightarrow S_n$ <ul style="list-style-type: none"> ◦ Follow replica failure scheme <p>From S_{i+1} (the successor of S_i), transfer:</p> <ul style="list-style-type: none"> • Data within range of $(H(S_i), H(S_n)] \rightarrow S_n$ • S_{i+1} must remove the transferred data: $(H(S_i), H(S_n)]$ from its cache & disk (as per <code>remove_node</code>) <ul style="list-style-type: none"> ◦ If S_{i+1} unresponsive, try from S_{i+1}'s replicas (follow replica failure scheme)
<p>Case 3</p> 	<p>From ANY replica of S_i, transfer:</p> <ul style="list-style-type: none"> • All data of S_i $(H(S_{i-1}), H(S_i)] \rightarrow S_{i+1}$ • Note: since S_{i+1} should be a replica of S_i, this transfer should be local - assuming S_{i+1} was chosen as the replica to transfer data <ul style="list-style-type: none"> ◦ If S_{i+1} is unresponsive, send to S_{i+2}, if S_{i+2} fails, send to S_{i+3}, and so on.. Until loops back to S_i (if this case, then fail) <p>Since S_n is not in the hash range of S_{i-1}, S_i, S_{i+1}, it gets none of the data from any of those nodes:</p> <ul style="list-style-type: none"> • S_{n+1} transfer data within range of $(H(S_{n-1}), H(S_n)] \rightarrow S_n$ <ul style="list-style-type: none"> ◦ If S_{n+1} is unresponsive, try from S_{n+1}'s replicas (follow replica failure scheme)

Note:

Add transition states for new nodes, because we add them to the ring right away, but they should not be picked up by `getReplicas` as the replicas to other nodes yet. Transition states allows us to differentiate between newly added nodes while maintaining information on their final state.

ECSCClient collects all failed nodes, then for each failed node:

- Remove node from hash ring, set state to SHUTDOWN
- Call add_node

Broadcast replication request to all KVServers

3) Reconciliation mechanism for topology changes (ECSCClient)

Kept most of the same functionality from M2 with a few slight changes to ECSCClient -- add_node and remove_node functionalities.

- Whenever a node is added/removed, ECSCClient will broadcast the metadata update to all servers participating in the service (as described in Decision 1 Replication Strategy)
 - The server will then compare its current replicas to the new hash ring and send its data to its new replicas if any changes were detected

NOTE: To ensure that a coordinator *always* has 2 replicas, we do not allow the user to remove a node if only 3 nodes are participating in the service

add_node/s	Broadcast metadata , this will be received by KVServer's, who will each update their replicas according to the updated hash ring.
remove_node	Broadcast metadata to all KVServers on the hash ring at the end (STOP, START, IDLE_START) Do not allow remove_node if # servers on hash ring < 4 (minimum = 3)

Performance Evaluation

Not enough time to complete.

Test Cases

Most testing was done manually. Simple tests for basic functionality of individual modules were added as part of the JUnit Tests.

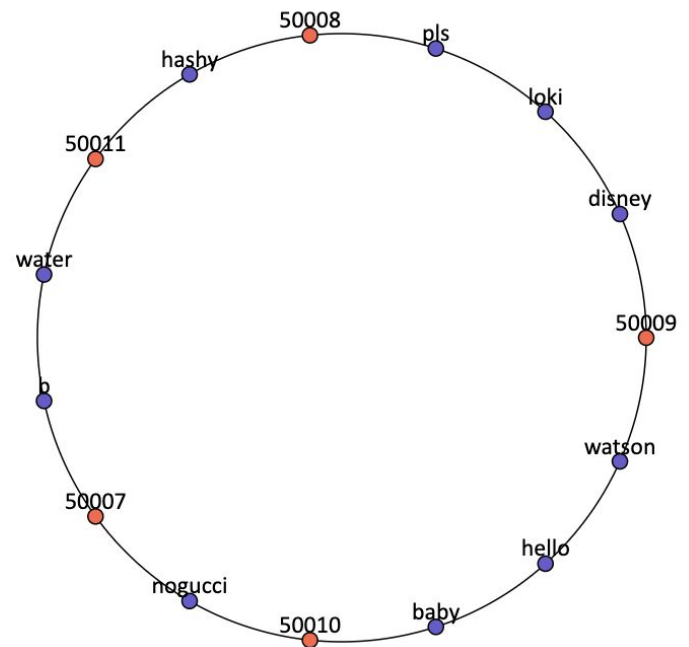
JUnit Tests

Test Steps	Expected Result
Case: Get Replicas Using a set of predefined nodes with known hash values, verify that <code>ring.getReplicas()</code> returns the correct values	Each server should have 2 replicas which are the 2 nodes immediately after the server on the hash ring.

Manual Tests

Server	Hash
localhost:50007	74b91dd7922e5c54f5b31c54d79038ac
localhost:50008	b87c6da95b39edc845134fb118f001aa
localhost:50009	29c3f3fdbabd1c3c026d1b3124bba64b
localhost:50010	6a4997ef87eaa5951bf2fd21ffab145a
localhost:50011	98ed9b6bce6feb0e4414523de8440558

KV pair	Hash
<hello, world>	5d41402abc4b2a76b9719d911017c592
<disney, land>	f240a4a08ef4d49a9b643168779d8491
<walt, disney>	fb1c9e05e53928d05f77f4eab0dc587c
<water, bottle>	9460370bb0ca1c98a779b1bcc6861c2c
<b, ts>	92eb5ffee6ae2fec3ad71c777531578f
<loki, watson>	e64cfa3fd59e32df57003c7401f48c99
<watson, loki>	4190908d675abc6c2e3931c01c92a6ca
<baby, bear>	6848d756da66e55b42f79c0728e351ad
<pls, help>	d89a8633204d02f952c89b8245f2287e
<hashy, oats>	99860a3a539808152e555b93c0aefdf4
<nogucci, gang>	6c2fedd51eab6a47bd3961058a4c0ed3



#	Test Steps	Expected Result
1	<p>Successful recovery</p> <p>Setup: 5 nodes available to participate as specified in ECS.config file</p> <p>Steps:</p> <ol style="list-style-type: none"> 1. Add 4 nodes in hash ring (add_node) and start 2. Send PUT requests as specified in the table above 3. Kill node <p>Verify:</p> <ol style="list-style-type: none"> 1. New node (S_n) contains replicated data from range A - B 2. Successor node (S_{i+1}) contains replicated data from range B - C 	<p>TEST DATA</p> <p>Remove Node: 50011</p> <p>Pred: 50010</p> <p>Replicas: 50008, 50009</p> <p>EXPECTED</p> <p>S_n: 50007</p> <p>Stored data:</p> <ul style="list-style-type: none"> - nogucci <p>S_{i+1}: 50008</p> <p>Stored data:</p> <ul style="list-style-type: none"> - b - Water <p>STATUS: Requests look good, actual data transfer between servers must be implemented</p>
2	<p>Failure Scenario 1: One replica failed (the replica immediately following the deleted node on the ring)</p> <p>Steps:</p> <ol style="list-style-type: none"> 1. Add 4 nodes to hash ring and start <ol style="list-style-type: none"> a. 50011 b. 50010 c. 50009 d. 50008 2. PUT requests as specified in previous table 3. Kill node 50011 and one of its replicas (50008) - must delete both in the same heartbeat rest period <p>Verify:</p> <ol style="list-style-type: none"> 1. Same results as Case 1 but the second replica is used to transfer data (50009) 	<p>Other (active) replica is used to obtain lost data</p> <p>TEST DATA</p> <p>Remove Node: 50011</p> <p>Pred: 50010</p> <p>Replicas: 50008, 50009</p> <p>Replacement node: 50007</p> <p>Remove Node: 50008</p> <p>Pred: (@ time that it's recovery is handled) 50007</p> <p>Replicas: 50009, 50010</p> <p>No replacement node - no available nodes left</p> <p>EXPECTED DATA</p> <p>S_n: 50007</p> <p>Stored data:</p> <ul style="list-style-type: none"> - nogucci <p>S_{i+2}: 50009</p> <p>Stored data:</p> <ul style="list-style-type: none"> - b (from 50011) - water (from 50011) - hashy (from 50008)
3	Failure Scenario 2: Both replicas failed	<p>Verify that metadata with new node is broadcasted</p> <p>NOTE: in this case, the data that was stored on the failed node is lost to the system</p>

#	Test Steps	Expected Result	Result
1	<p>Successful Recovery</p> <p>Steps:</p> <ol style="list-style-type: none"> 1. Add 4 nodes to hash ring 2. PUT requests from table 3. Kill server 50010 (S_i) <p>Verify:</p> <ol style="list-style-type: none"> 1. New node contains replicated data from S_{i-1} to S_n <ol style="list-style-type: none"> a. $S_{i-1} \rightarrow S_i$ taken from S_{i+1} <i>replica disk</i> b. $S_i \rightarrow S_n$ taken from S_{i+1} <i>core disk</i> 	<p>TEST DATA</p> <p>Removed Node: 50010 Pred: 50009 Replicas: 50011, 50008</p> <p>EXPECTED DATA</p> <p>S_n: 50007 Stored data:</p> <ul style="list-style-type: none"> - nogucci (from 50011 <i>core</i>) - baby (50011 <i>rep</i>) - hello (50011 <i>rep</i>) - watson (50011 <i>rep</i>) 	PASS
2	<p>Failure Scenario 1: One replica failed (the replica immediately following the deleted node on the ring)</p> <p>Steps:</p> <ol style="list-style-type: none"> 4. Add 4 nodes to hash ring and start (need 2 non-started nodes to replace 2 failed nodes) <ol style="list-style-type: none"> a. 50011 b. 50010 c. 50009 d. 50008 5. PUT requests as specified in previous table 6. Kill node 50011 and one of its replicas (50008) - must delete both in the same heartbeat rest period <p>Verify:</p> <ol style="list-style-type: none"> 2. Same results as Case 1 but the second replica is used to transfer data (50009) 	<p>Other (active) replica is used to obtain lost data</p> <p>TEST DATA</p> <p>Remove Node: 50010 Pred: 50009 Replicas: 50011, 50008 Replacement node: 50007</p> <p>Remove Node: 50011 Pred: (@ time that it's recovery is handled) 50007 Replicas: 50008, 50009 No replacement node - no available nodes left</p> <p>EXPECTED DATA</p> <p>S_n: 50007 Stored data:</p> <ul style="list-style-type: none"> - nogucci (50011 replica) - baby (50010 rep) - hello (50010 rep) - watson (50010 rep) <p>S_{i+2}: 50008 Stored data:</p> <ul style="list-style-type: none"> - b (from 50011) - water (from 50011) 	PASS